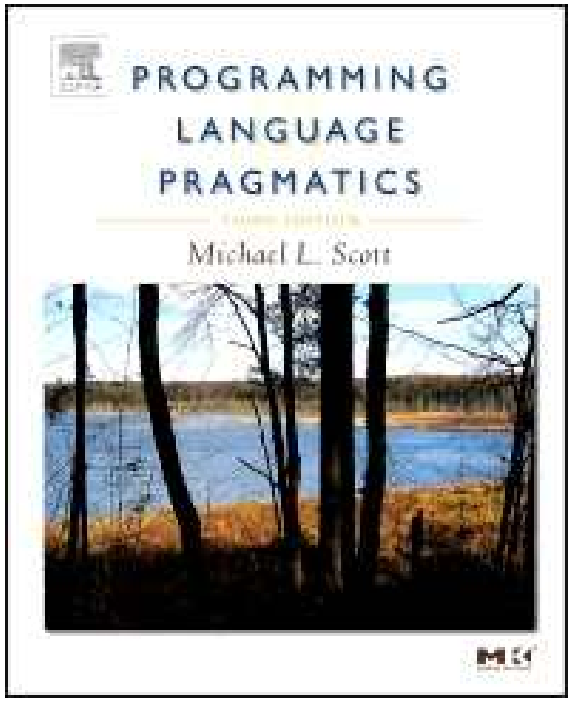
Programming Languages
Lecture02 – Compilation

남 범 석

bnam@skku.edu

Chapter 01



Language Groups

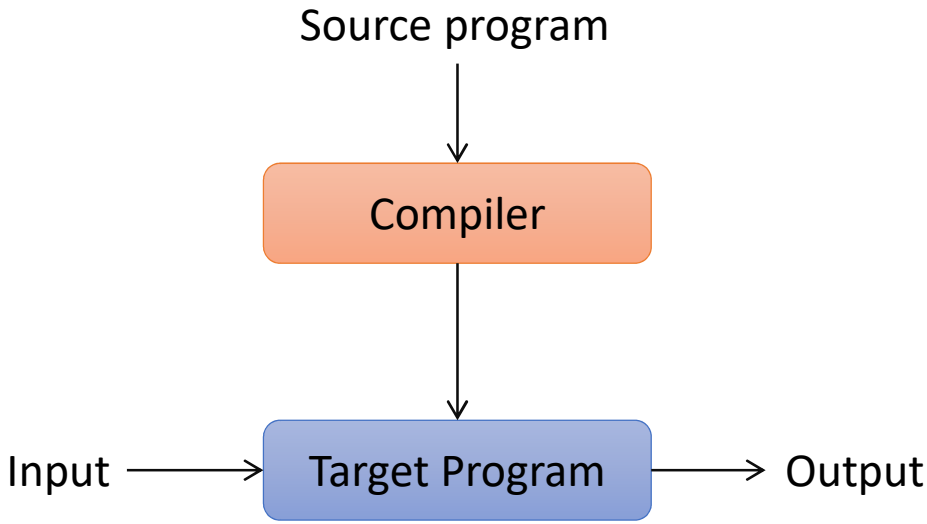
- Imperative
  - von Neumann (Fortran, Pascal, Basic, C)
  - Object-oriented (Smalltalk, Eiffel, C++)
  - Scripting languages (Perl, Python, JavaScript, PHP)
- Declarative
  - Functional (Scheme, ML, pure Lisp, FP)
  - Logic, constraint-based (Prolog, VisiCalc, RPG)

Why so many programming languages?

- Evolution
  - We've learned better ways of doing things over time
- Socio-economic factors
  - Proprietary interests, commercial advantage
- Orientation toward special purposes
- Orientation toward special hardware
- Diverse ideas about what is pleasant to use

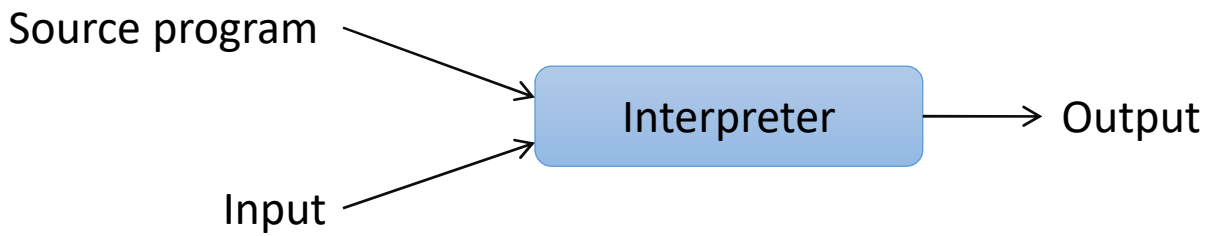
Pure Compilation

- Compiler
  - translates the high-level source program into an equivalent target program (typically in a machine language)
  - not used while running program



Pure Interpretation

- Interpreter
  - stays around for the execution of the program
  - the center of control during execution



Compilation vs. Interpretation (1)

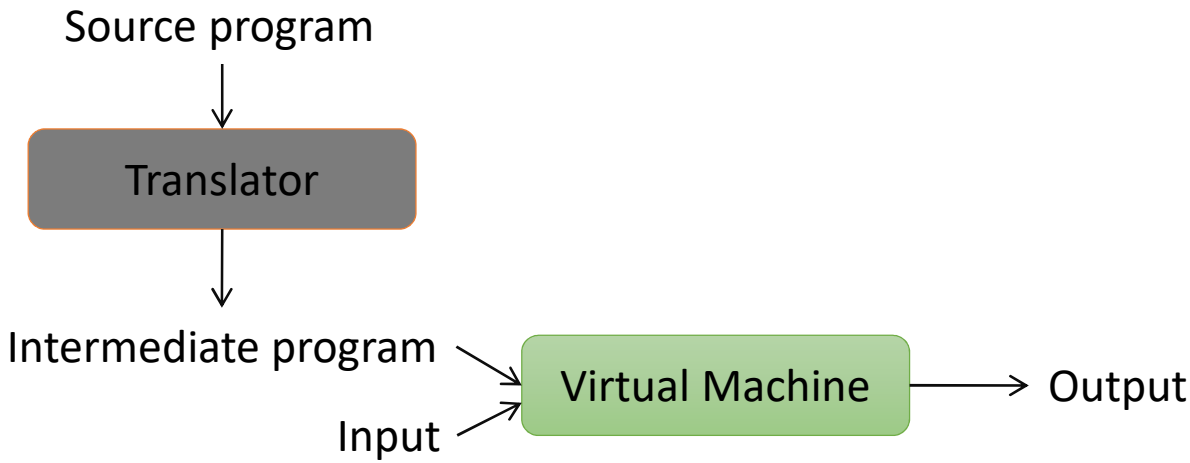
- Compilation vs. interpretation
  - Not opposites
  - Not a clear-cut distinction

- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)

- Compilation
  - Better performance

Compilation vs. Interpretation (2)

- Common case
  - Compilation or simple pre-processing, followed by interpretation
- Most language implementations include a mixture of both compilation and interpretation



Compilation vs. Interpretation (3)

- Note that compilation does NOT have to produce machine language for a hardware

- Compilation
  - is translation from one language into another,
  - with full analysis of the meaning of the input

- Compilation
  - entails semantic understanding of what is being processed,
  - but a pre-processing does not

Compilation vs. Interpretation (4)

- Most use "virtual instructions"
  - set operations in Pascal
  - string manipulation in Basic

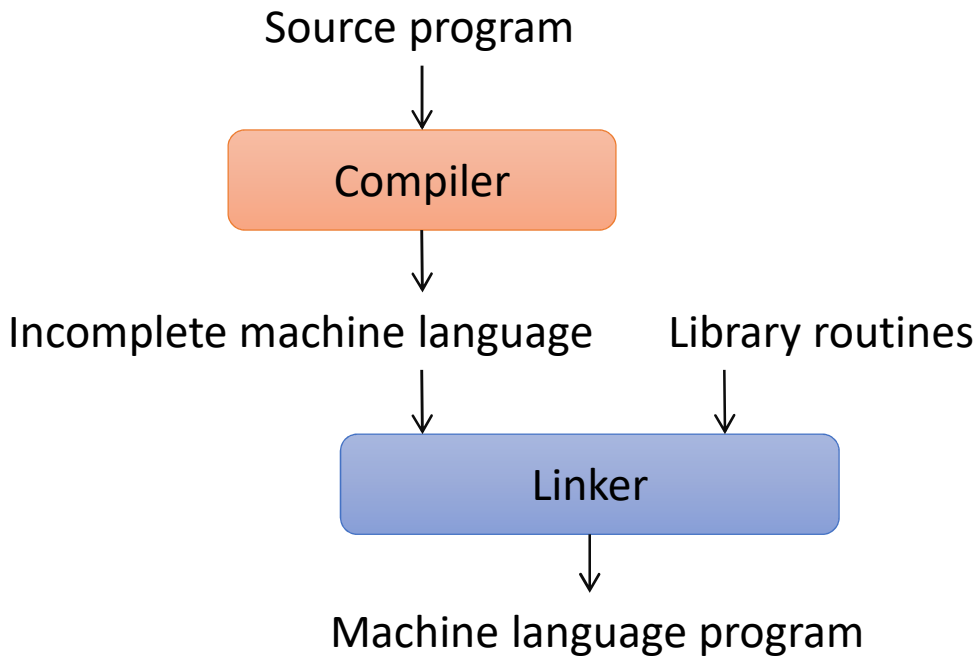
- Some compilers produce only virtual instructions
  - e.g., Java bytecode, Microsoft CIL, Pascal P-code

Implementation Strategies (1)

- Preprocessor
  - Removes comments and white spaces
  - Groups characters into tokens
    - e.g., keywords, identifiers, numbers, symbols
  - Expands abbreviations in the style of a macro assembler
  - Identifies higher-level syntactic structures
    - e.g., loops, subroutines

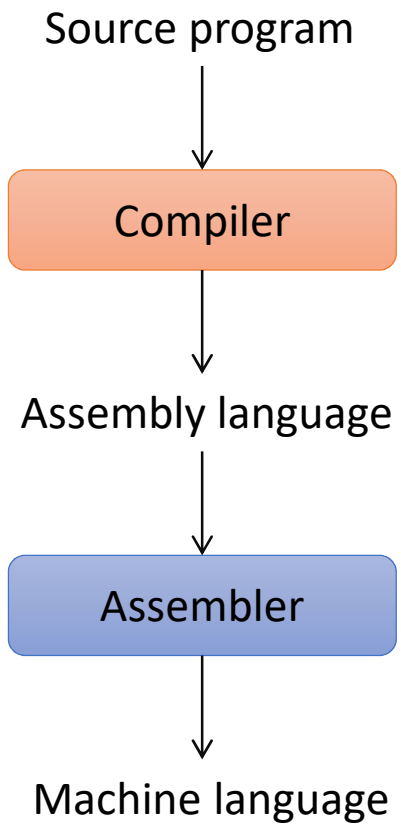
Implementation Strategies (2)

- Library of Routines and Linking
  - Compiler uses a linker program to merge the appropriate library of subroutines into the final program
  - e.g., math functions such as sin, cos, log, etc



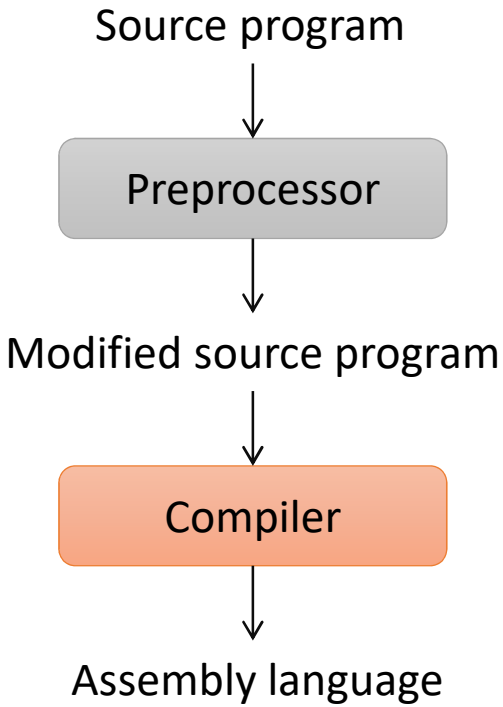
Implementation Strategies (3)

- Post-compilation Assembly
  - Facilitates debugging
    - assembly languages easier for human than machine binaries
  - Isolates the compiler from changes in the format of machine language files
    - only assembler must be changed, and shared by many compilers



Implementation Strategies (4)

- The C Preprocessor (conditional compilation)
  - allows several versions of a program to be built from the same source



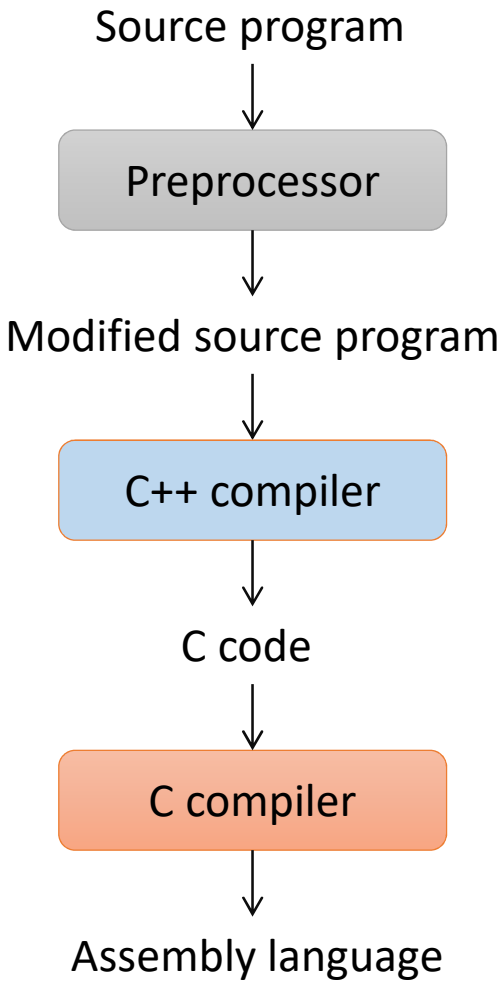

```
#if _X86_
  A=...
#elif _ARM_
  B=...
#else
  C=...
#end
```

-D _X86_

```
A=...
```

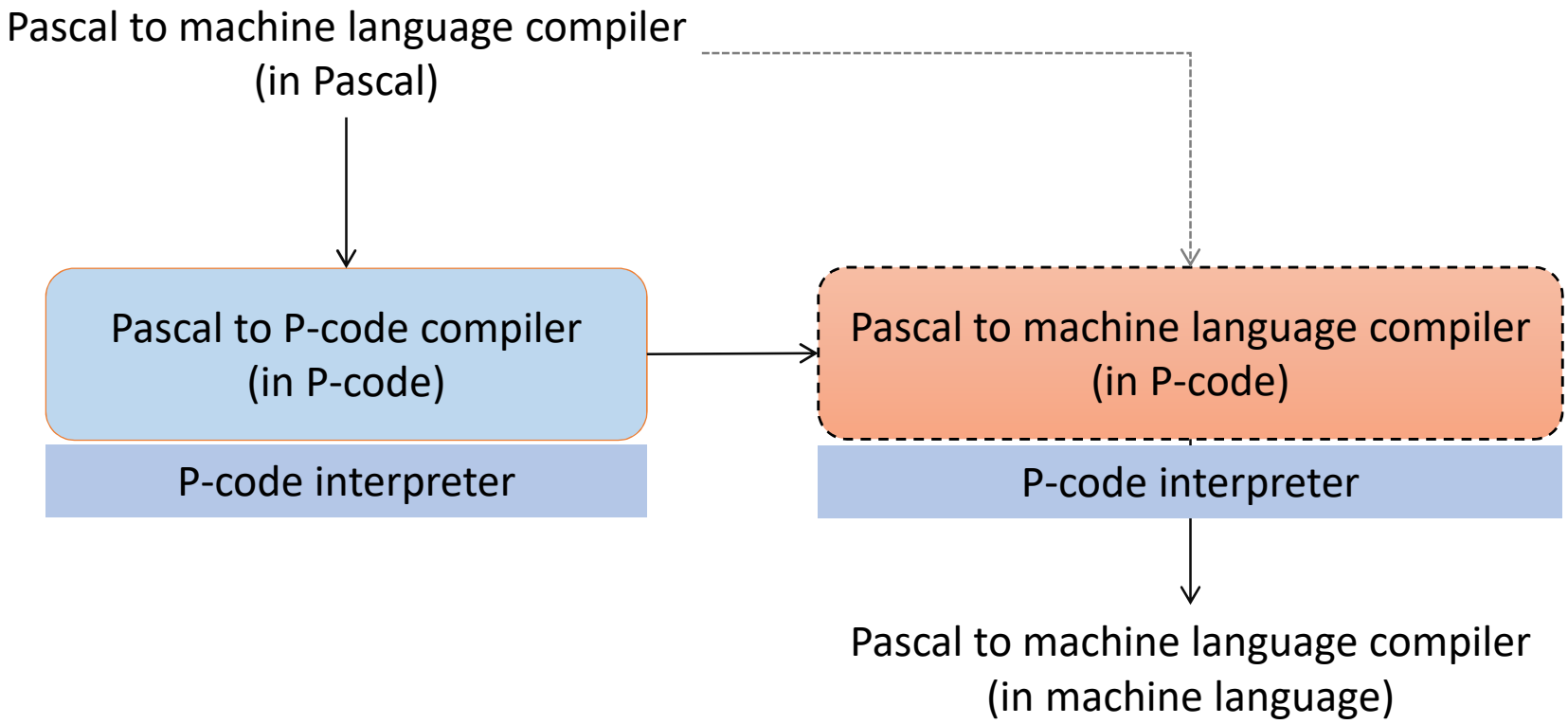
Implementation Strategies (5)

- Source-to-Source Translation (C++)
  - C++ implementations based on
  - the early AT&T compiler generated
  - an intermediate program in C,
  - instead of an assembly language:



Implementation Strategies (6)

- Bootstrapping
  - C compilers in C
  - Pascal compilers in Pascal



Implementation Strategies (7)

- Compilation of Interpreted Languages
  - Some features of IL are not finalized until runtime
    - Late binding
  - Generates code with assumptions on runtime decision
    - If these assumptions are valid, the code runs very fast
    - If not, a dynamic check will revert to the interpreter.

Implementation Strategies (8)

- Dynamic and Just-in-Time Compilation
  - Deliberately delay compilation until the last possible moment

- Dynamic compilation
  - Lisp or Prolog invoke the compiler on the fly,
    - Translate newly created source into machine language, or
    - Optimize the code for a particular input set

- Just-in-time (JIT) compilation
  - Java defines a machine-independent intermediate form (bytecode)
  - C# compiler produces Common Intermediate Language (CIL)
  - Bytecode, CIL are translated into machine code immediately prior to execution

Implementation Strategies (9)

- Microcode (firmware)
  - Assembly-level instruction set is not implemented in hardware, but runs on an interpreter
  - Interpreter is written in low-level instructions (microcode or firmware), which are stored in read-only memory (ROM) and executed by the hardware
  - Popular in machines before the mid 1980s

Compilation vs. Interpretation

- Compilers exist for some interpreted languages, but they aren't pure:
  - Selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source
  - Interpretation of parts of code, at least, is still necessary

- Unconventional compilers
  - Text formatters (TEX, troff)
  - Query language processors
  - Silicon compilers

Programming Environment Tools

- Tools

Type	Unix examples
Editors	vi, emacs
Pretty printers	cb, indent
Pre-processors (esp. macros)	cpp, m4, watfor
Debuggers	adb, sdb, dbx, gdb
Style checkers	lint, purify
Module management	make
Version management	sccs, rcs, cvs, subversion
Assemblers	as
Link editors, loaders	ld, ld-so
Perusal tools	more, less, od, nm
Program cross-reference	ctags