CSCI-C311 Programming Languages

Compilation and Interpretation

Dr. Hang Dinh

1

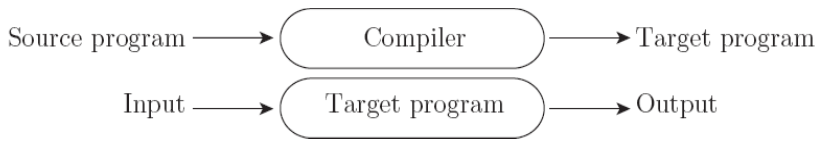
Outline and Reading

- After this lecture, you will learn
  - Compilation vs. Interpretation
  - Implementation strategies
- Reading
  - Scott 4e - Section 1.4

2

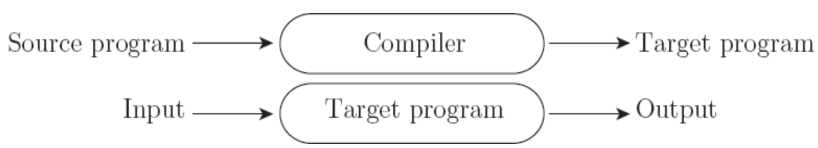
Pure Compilation

- Compilation and execution of a program at the highest level of abstraction:



  - The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away.
  - At some arbitrary later time, the user tells the operating system to run the target program.

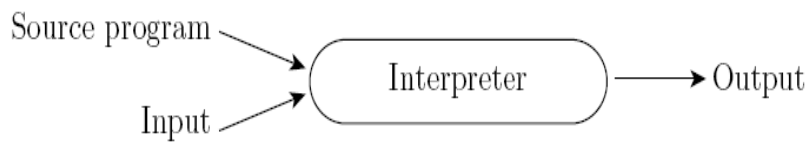
Pure Compilation



  - The compiler is the locus of control during compilation,
  - The target program is the locus of control during its own execution.
- The compiler itself is a machine language program
  - presumably created by compiling some other high-level program.
  - When written to a file in a format understood by the operating system, machine language is commonly known as *object code*.

Pure Interpretation

- Interpretation: alternative style of implementation for high-level language

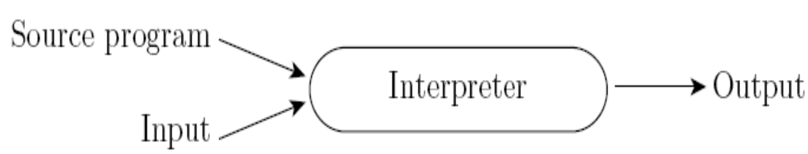


  - Interpreter stays around for the execution of the program
  - Interpreter is the locus of control during execution

5

Pure Interpretation

- Interpretation:



- Interpreter implements a virtual machine
  - whose "machine language" is the high-level programming language.
  - The interpreter reads statements in that language more or less one statement at a time, executing them as it goes along.

6

Compilation vs. Interpretation

- Compilation vs. interpretation
  - not opposites
  - not a clear-cut distinction
- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)
- Compilation
  - Better performance

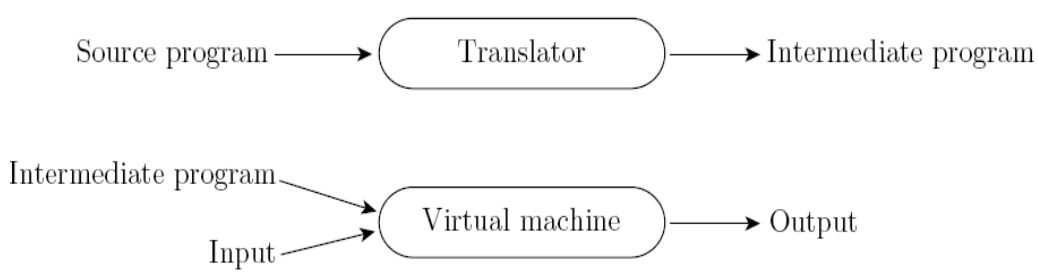
- Consider a loop that accesses variable x in every iteration

  - Compilation: access x via its fixed location
  - Interpreter looks x up in a table to find its location

7

Mixing Compilation and Interpretation

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



8

Compiled v.s. Interpreted

- Confusing distinction:
  - If initial translator is simple → the language is "interpreted"
  - If initial translator is complicated → the language is "compiled"
- Why confusing?
  - It's possible for a compiler (complicated translator) to produce code that is then executed by a complicated virtual machine (interpreter),
  - Example: Java
- Hallmarks of compilation: a language is *compiled* if
  - the translator analyzes it thoroughly, and
  - the intermediate program does not bear a strong resemblance to the source

9

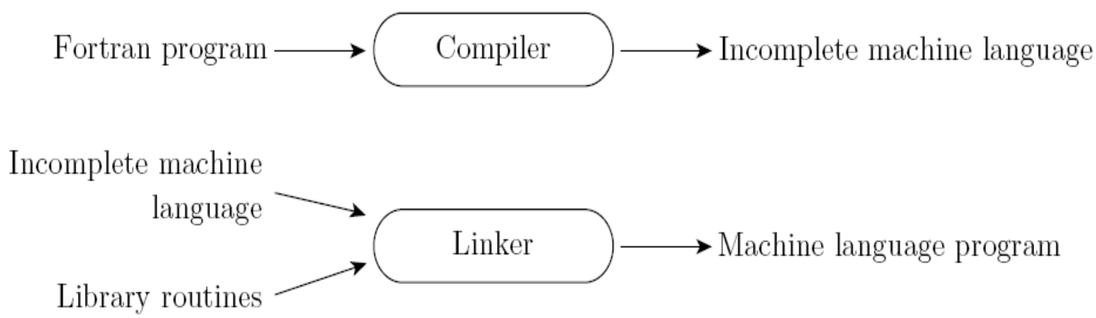
Implementation Strategies

- In practice, there's a broad spectrum of implementation strategies of mixing compilation and interpretation.
- Implementation Strategy: Preprocessing
  - Removes comments and white space
  - Groups characters into *tokens* (keywords, identifiers, numbers, symbols)
  - Expands abbreviations in the style of a macro assembler
  - Identifies higher-level syntactic structures (loops, subroutines)

11

Implementation Strategy: Library of Routines and Linking

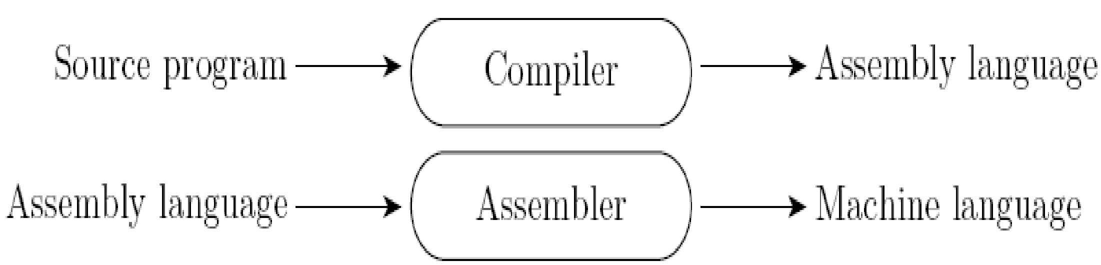
- Compiler uses a *linker* program to merge the appropriate *library* of subroutines into the final program
  - Examples of subroutines: math functions (sin, cos, log, etc.), I/O



12

Implementation Strategy: Post-compilation Assembly

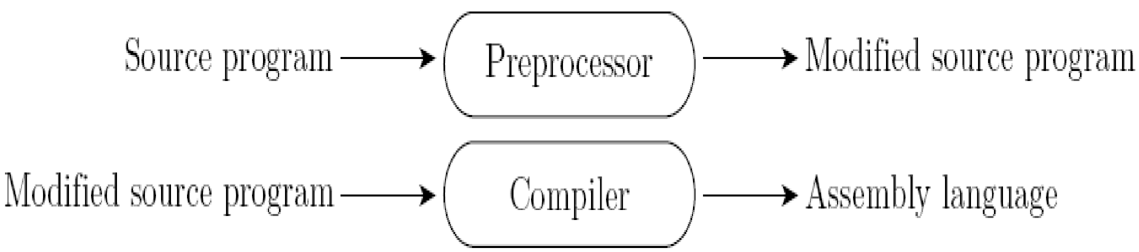
- Facilitates debugging (assembly language easier for people to read)
- Isolates the compiler from mandated changes in the format of machine language files
  - Only assembler must be changed, and it is shared by many compilers



13

Implementation Strategy: The C Preprocessor

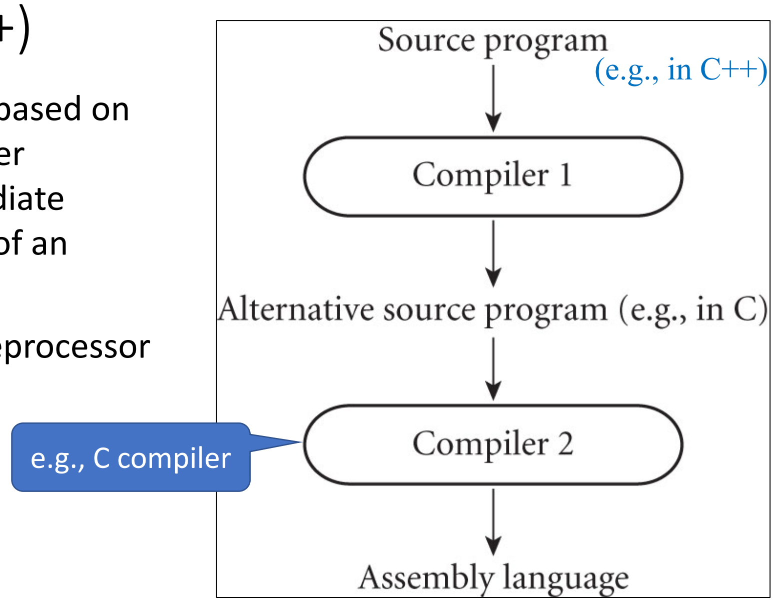
- Preprocessor first removes comments and expands macros
- *Conditional Compilation*: Preprocessor can delete portions of code,
  - which allows several versions of a program to be built from the same source



14

Implementation Strategy: 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.
- Compiler 1 is not a preprocessor



16

Compilers v.s. Preprocessors

Compilers

- attempt to "understand" their source
- hide further steps when errors occur

Preprocessors

- do not attempt to understand the source
- often let errors through

17

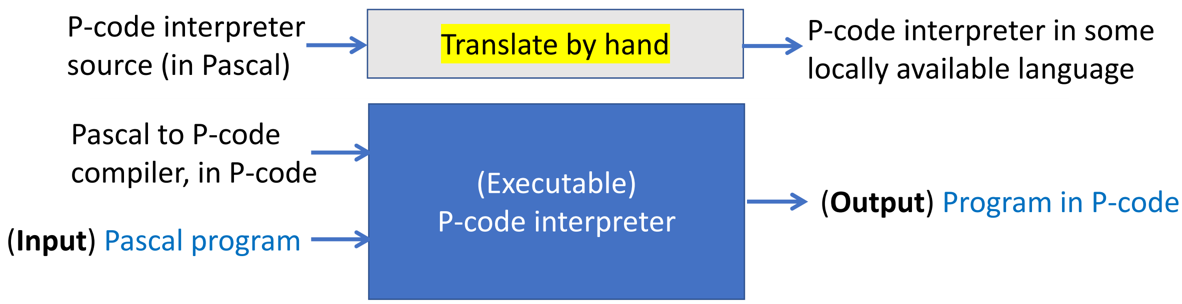
Implementation Strategy: *Bootstrapping*

- The chicken-and-egg question:
  - Many compilers are *self-hosting*, i.e., written in the language they compile
  - *How does one compile the* compiler *in the first place?*
- Answer: the *bootstrapping* technique
  - Start with a simple implementation (an interpreter)
  - Use a simple implementation to build progressively more sophisticated versions

18

Implementation Strategy: *Bootstrapping* Example

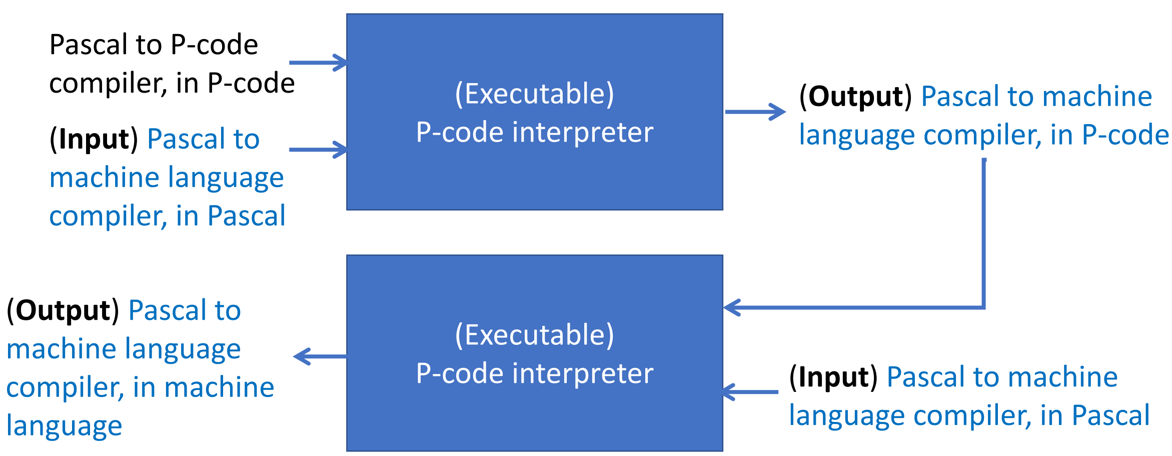
- Early Pascal compilers were built around a set of tools:
  - A Pascal to P-code compiler, written in Pascal
  - Same Pascal to P-code compiler, already translated into P-code
  - A P-code interpreter, written in Pascal



19

Implementation Strategy: *Bootstrapping* Example

- Faster implementation of Pascal compiler:



22

Implementation Strategy: Compilation of Interpreted Languages

- Some traditionally interpreted languages (e.g., Lisp, Prolog, Smalltalk) that permit a lot of *late binding* can have compilers.
- In general case, these compilers generate code
  - that that performs much of the work of an interpreter, or
  - that makes calls into a library that does that work instead
- In special cases, the compiler generates code that makes assumptions about decisions that won't be finalized until runtime.
  - If these assumptions are valid, the code runs very fast.
  - If not, a dynamic check will revert to the interpreter.

23

Implementation Strategy: Dynamic and Just-in-Time Compilation

- In some cases, a programming system may deliberately delay compilation until the last possible moment.
  - Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set.
  - The Java language definition defines a machine-independent intermediate form known as *byte code*, which is translated into machine code by a *just-in-time* compiler
  - The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

24

Implementation Strategy: Microcode

- Assembly-level instruction set is not implemented in hardware; it runs on an interpreter.
- Interpreter is written in low-level instructions (*microcode* or *firmware*), which are stored in read-only memory and executed by the hardware.

25

Compilation Term

- A compiler does not necessarily translate from a high-level programming language into machine language
- Unconventional compilers
  - Text formatters (e.g., Tex, LaTex)
  - Query language processors
  - Compilers for logic-level circuits
- Compilation is *translation* from one nontrivial language into another, with full analysis of the meaning of the input.

26