Overview of Compilers and Language Translation

1

Programming Languages

- Serve as a means of communication among people as well as between people and machines
- Provide a framework for formulating the software solution to a problem
- Can enhance or inhibit creativity
- Influence the ways we think about software design by making some program structures easier to describe than others (e.g., recursion in Fortran)

2

Programming Languages (continued)

"Language is an instrument of human reason, and not merely a medium for the expression of thought."
– George Boole

"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems."
– Bertrand Russell

3

Role of Programming Languages

- Machine independence
- Portability
- Reuse
- Abstraction
- Communication of ideas
- Productivity
- Reliability (error detection)

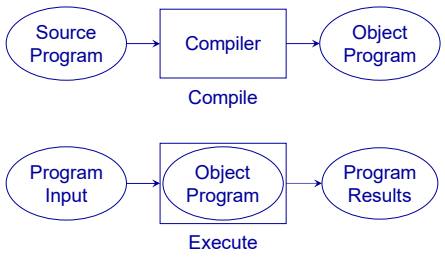
4

Translators and Compilers

- In the context of programming languages, a **translator** is a program that accepts as input text written in one language (called the source language) and converts it into a semantically equivalent representation in a second language (called the target or object language).
- If the source language is a high-level language (HLL) and the target language is a low-level language (LLL), then the translator is called a **compiler**.

5

Simplified View of Compile/Execute Cycle



6

Language Versus Its Implementation

Language

- Identifier may have an arbitrary number of characters
- Integer types with arbitrary number of digits
- Precision of floating-point types is not specified

Implementation

- May restrict the number of significant characters
- Can restrict valid range of integer types
- Precision of floating point types is (usually) determined by the machine

7

Role of a Compiler

- A compiler must first verify that the source program is valid with respect to the source language definition.
- If the source program is valid, the compiler must produce a **semantically equivalent and reasonably efficient** machine language program for the target computer.
- If the source program is not valid, the compiler must provide reasonable feedback to the programmer as to the nature and location of any errors. Feedback on possible multiple errors is usually desirable.

8

Other Language Processors

- Assembler
  - translates symbolic assembly language to machine code
- High-level language translator (a.k.a., transpiler)
  - e.g., C++ to C or TypeScript to JavaScript
- Interpreter (more on this topic in subsequent slides)
- Testing/Re-engineering tools
- Macro preprocessors
- Disassemblers
- Decompilers

9

Integrated Development Environment (IDE)

- Syntax-directed editor
- Source code formatter
- Error reporting
- Refactoring
- Source level debugger
- Run time profiler

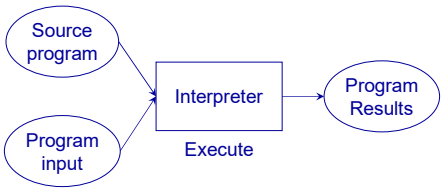
10

Interpreter

- Translates/executes source program instructions immediately (e.g., one line at a time)
- Does not analyze and translate the entire program before starting to run – translation is performed every time the program is run
- Source program is basically treated as another form of input data to the interpreter
  - Control resides in interpreter, not in user program.
  - User program is passive rather than active.
- Some interpreters perform elementary syntactic translation (e.g., compress keywords into single byte operation codes).

11

Simplified View of an Interpreter



12

Examples of Interpreters

- BASIC and Lisp language interpreters
- Read-Eval-Print Loop (REPL) for programming languages
  - e.g., JShell for Java or kotlinc-jvm for Kotlin
- Java Virtual Machine (JVM)
  Java is compiled to an intermediate, low-level form (Java byte code) that gets interpreted by the JVM
- Operating system command interpreter
  - various Unix shells (sh, csh, bash, etc.)
    - can also run shell scripts
  - Windows command prompt
    - can also run batch files
- SQL interpreter (interactive database query)

13

Compilers Versus Interpreters

- Compilation
  - two step process (compile, execute)
  - better error detection
  - compiled program runs faster
- Interpretation
  - one step process (execute)
  - provides rapid feedback to user
  - good for prototyping and highly interactive systems
  - performance penalty

14

Emulators

- An **emulator** or **virtual machine** is an interpreter for a machine instruction set. The machine being "emulated" may be real or hypothetical.
- Similar to real machines, emulators typically use an instruction pointer (program counter) and a fetch-decode-execute cycle.
- Running a program on an emulator is functionally equivalent to running the program directly on the machine, but the program will experience some performance degradation on the emulator.

15

Emulators
(continued)

- A real machine can be viewed as an interpreter implemented in hardware. Conversely, an emulator can be viewed as a machine implemented in software.

16

Interpretive Compilers

- An **interpretive compiler** is a combination of a compiler and a low-level interpreter (emulator). The compiler translates programs to the instruction set interpreted by the emulator, and the emulator is used to run the compiled program.
- Example – Oracle/Sun Java Development Kit
  - javac is a compiler
  - java is an emulator for the Java Virtual Machine (JVM)
- An interpretive compiler usually provides fast compilation with reasonable performance.

17

Just-In-Time Compiler

- A **Just-In-Time (JIT) Compiler** is a compiler that converts program source code into native machine code just before the program is run.
- Java provides a just-in-time compiler with the JVM that translates Java bytecode into native machine code. Use of the JIT compiler is optional.
- The translation for a method is performed when the method is first called.
- Performance improvements can be significant for methods that are executed repeatedly.

18

Writing a Compiler

Writing a compiler involves 3 languages

- Source language
  - input to the compiler
  - e.g., compiler for C++, compiler for Java, or compiler for CPRL
- Implementation language
  - the language that the compiler is written in
  - e.g., C++, Kotlin, or Java
- Target language
  - output of the compiler
  - e.g., assembly language or machine language (possibly for a virtual computer)

 Slide 19

19

Tombstone Diagrams

- Program P expressed in language L (may be a machine language)

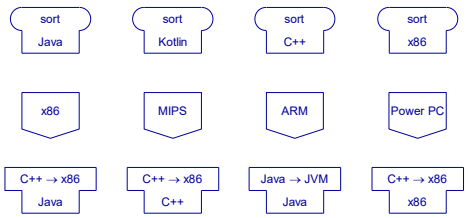
- Machine M

- S-to-T translator expressed in language L. (L could be a machine language.)

 Slide 20

20

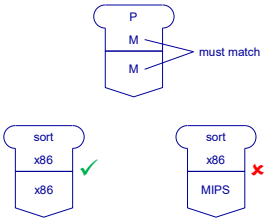
Examples: Tombstone Diagrams



 Slide 21

21

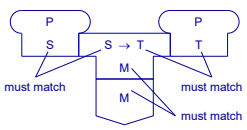
Running Program P on Machine M



 Slide 22

22

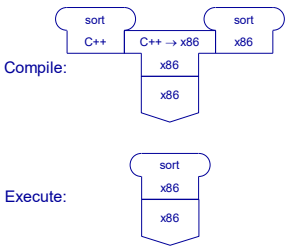
Compiling a Program



 Slide 23

23

Example: Compiling and Executing a Program

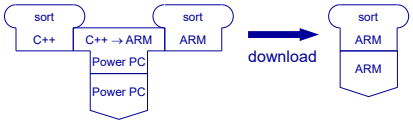


 Slide 24

24

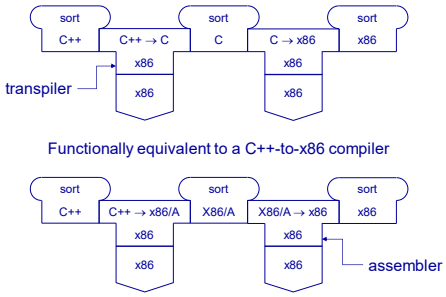
Cross-Compiler

- A **cross-compiler** runs on one machine and produces target code for a different machine.
- The output of a cross-compiler must be downloaded to the target machine for execution.
- Commonly used for mobile and embedded systems



25

Two-stage Compiler



Functionally equivalent to a C++-to-x86 compiler

26

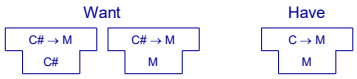
Using the Source Language as the Implementation Language

- It is common to write a compiler in the language being compiled; e.g., writing a C++ compiler in C++.
- Advantages
  - The compiler itself provides a non-trivial test of the language being compiled.
  - Only one language needs to be learned by compiler developers.
  - Only one compiler needs to be maintained.
  - If changes are made in the compiler to improve performance, then recompiling the compiler will improve compiler performance.
- For a new programming language, how do we write a compiler in that language? (chicken and egg problem)

27

Bootstrapping a Compiler

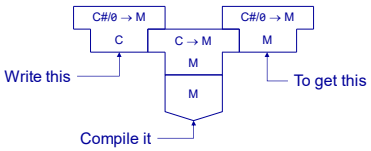
Problem: Suppose that we want to build a compiler for a programming language, say C#, that will run on machine M, and assume that we already have a compiler for a different language, say C, that runs on M. Furthermore, we desire that the source code for the C# compiler be C#.



28

Bootstrapping a Compiler: Step 1

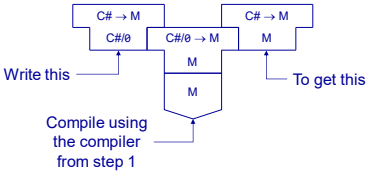
- Start by selecting a subset of C# (C#/0) that is sufficiently complete for writing a compiler.
- Write a compiler for C#/0 in C and compile it.



29

Bootstrapping a Compiler: Step 2

- Write the full compiler for C# in C#/0.
- Compile it using the compiler obtained from step 1



30

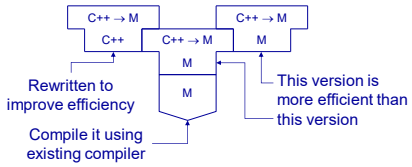
Efficiency

- Efficiency of a program
  - speed
  - use of memory
- Efficiency of a compiler
  - efficiency of the compiler itself
  - efficiency of the object code that it generates

31

Improving Efficiency of a Compiler

- Suppose you have a compiler for a language (say C++) written in that language.
- If you modify the compiler to improve efficiency of the generated object code, then you can recompile the compiler to obtain a more efficient compiler.



32

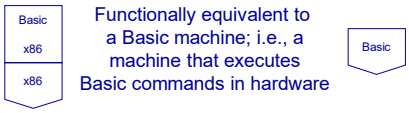
Tombstone Diagram for an Interpreter

- An interpreter for S expressed in language L (L could be a machine language)

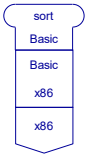
- Examples

33

Running an Interpreter

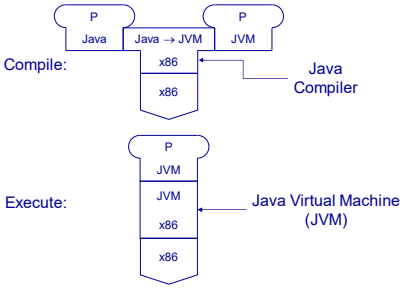


Example:



34

Writing/Executing a Java Program



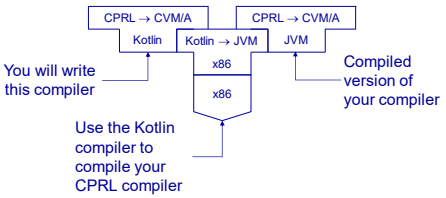
35

Compiler Project

- Source language: CPRL
- Target language: CVM/A, assembly language for the CPRL Virtual Machine (CVM)
- You will write a CPRL-to-CVM/A compiler in Kotlin.
- I will provide a CVM assembler.
- When you compile your compiler, you will have a CPRL-to-CVM/A compiler that runs on a Java virtual machine.

36

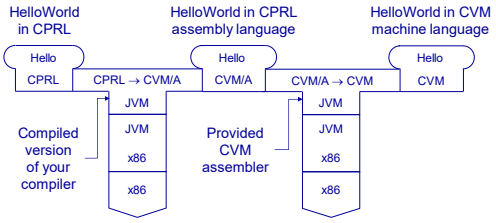
Compiler Project
(continued)



37

Compiler Project
(continued)

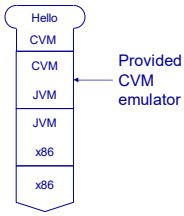
- Once your compiler is working, you can write test programs in CPRL and compile/assemble them.



38

Compiler Project
(continued)

- I will provide a CVM interpreter (emulator) that runs on the JVM. You can use the CVM interpreter to execute programs compiled using your compiler and assembled using the CVM assembler.



39