Programming Language Translation Lecture 4

Karen Bradshaw

Chapter 2, pp. 19--24

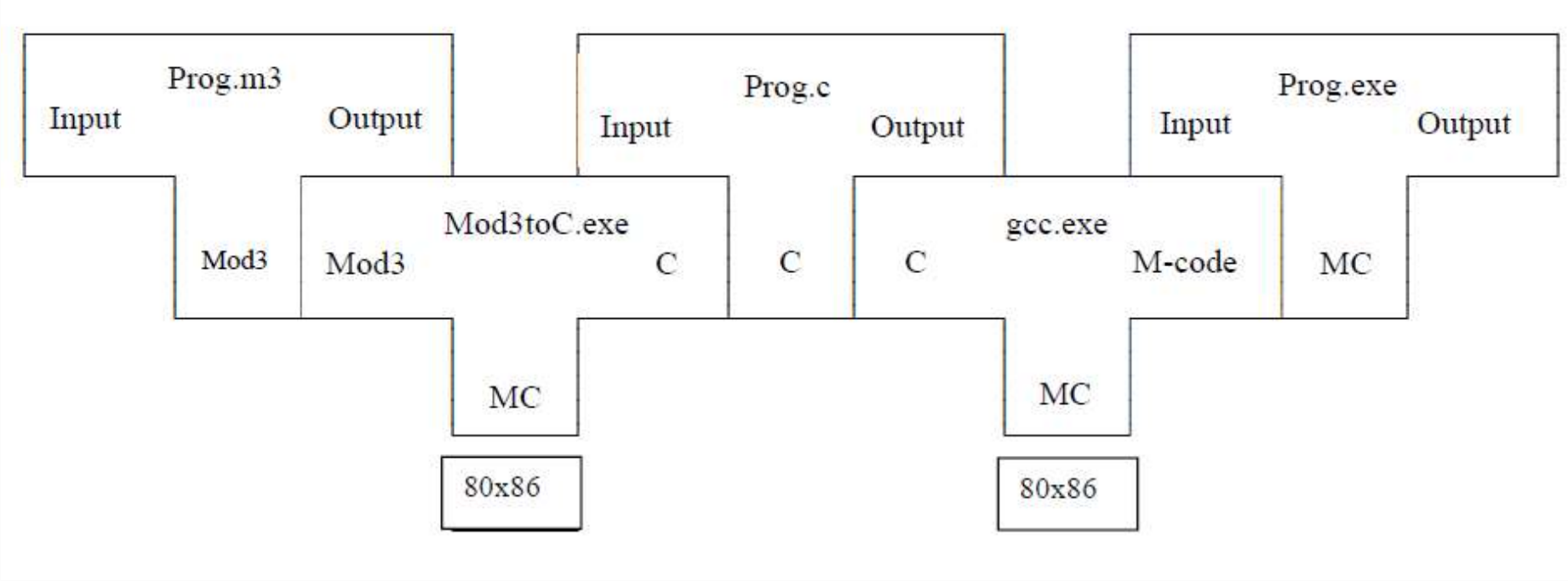
Multi-pass translation

- ***Advantages***
  - Space saving
  - More features possible (optimization, error handling)
  - Team development

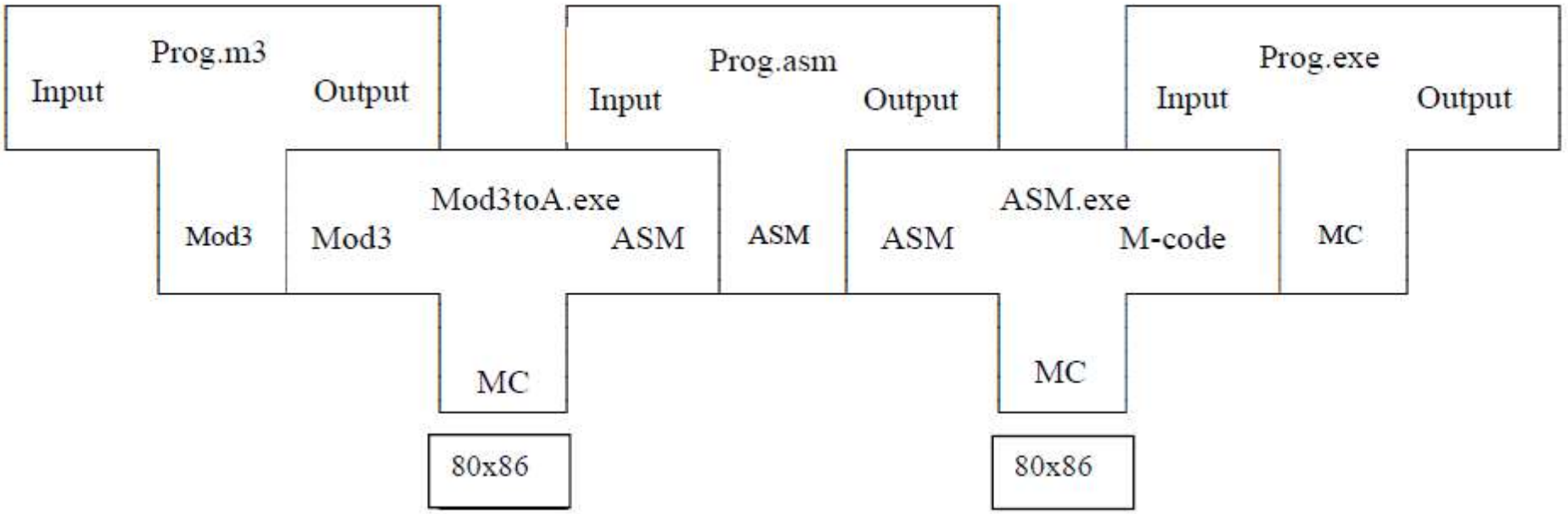
- ***Disadvantages***
  - Slow
  - Complicated to write

- Many modern languages such as Pascal have been designed to make single-pass compilation easy
- Some early compilers made many passes (up to 30!)

Compilation via another high-level language



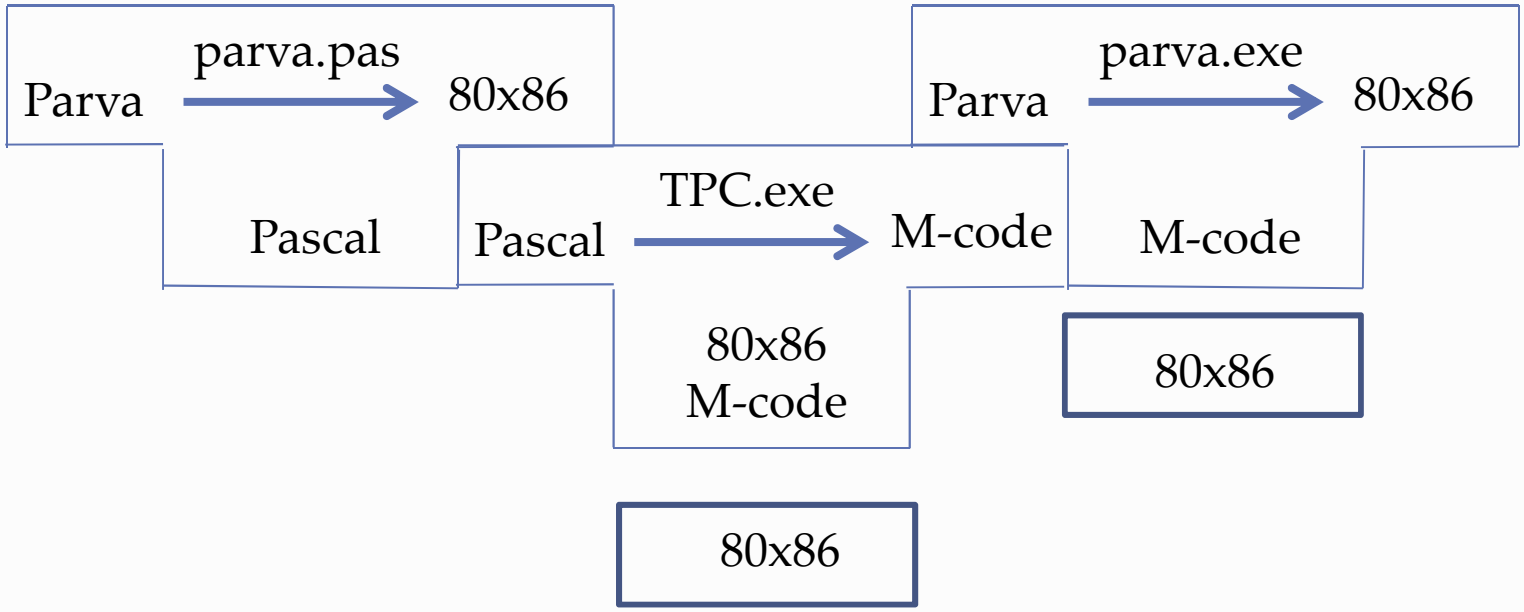
Compilation via assembler-level language



Compiler Generation

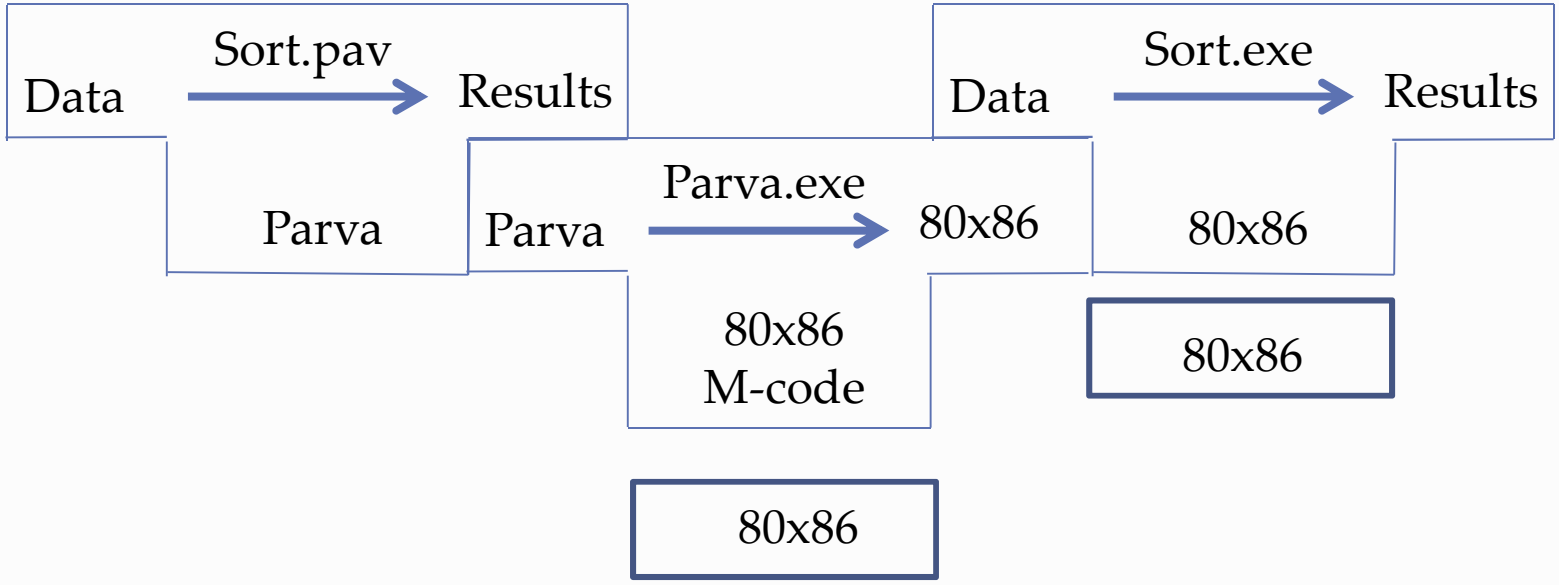
- Once we have one compiler we can use it to construct other programs, even programs that are themselves compilers
- We write the compiler for our dream language (Parva) in Pascal, and then compile this version of the Parva compiler with the Pascal compiler

This version cannot run directly Equivalent version can be executed directly



Compiler Generator (2)

- After creating the Parva compiler executable, we are in a position to compile Parva programs and execute them:



Interpreters

- ***Compilers*** usually
  - analyse and translate an entire program before it is executed
  - produce machine code that runs at the full speed of a machine

- Some applications require that part of an application be executed without analysing all of it
- Typically (but not always) these are interactive

- For example
  - Command line interpreters
  - Spreadsheets
  - Database queries

- Such systems often make use of an ***interpreter***

Interpreters (2)

Definition:

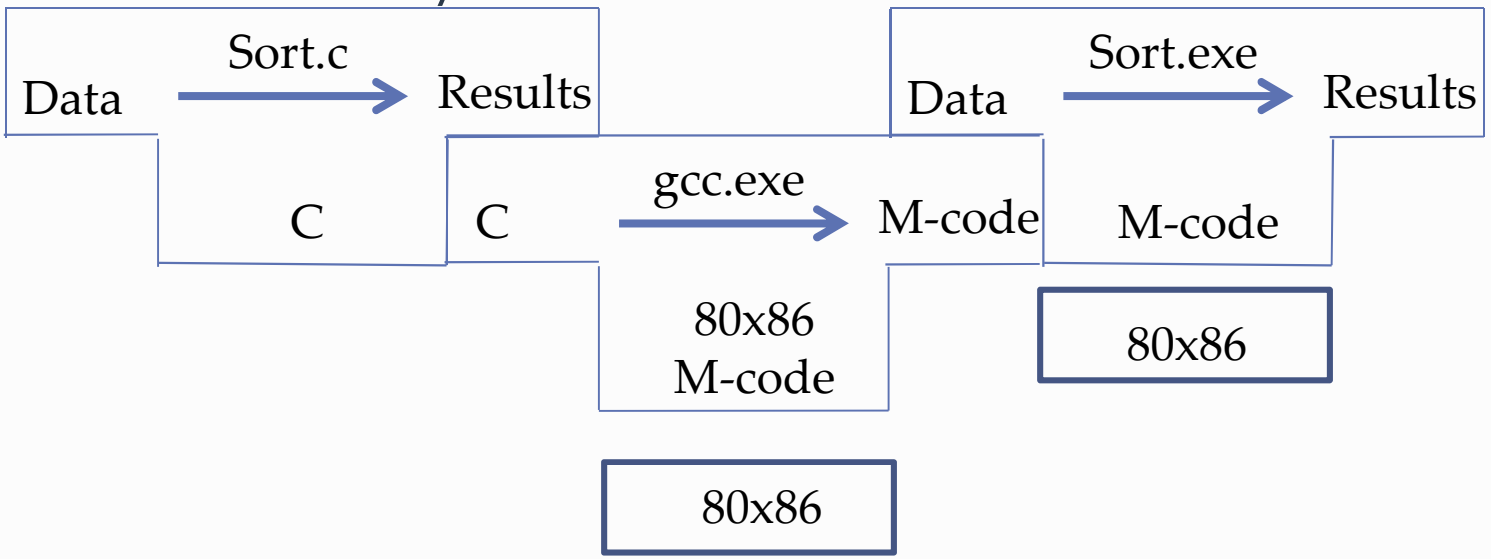
- a translator that accepts instructions (usually very simple ones) one-by- one, and then analyses and executes each of these one-by-one

- ***Interpreters*** are usually
  - Fairly easy to develop
  - Rather slow when used to solve large problems
  - User friendly

- Some entire programming languages have been developed around this idea -- original BASIC is the most obvious example – also Python

Full compiler

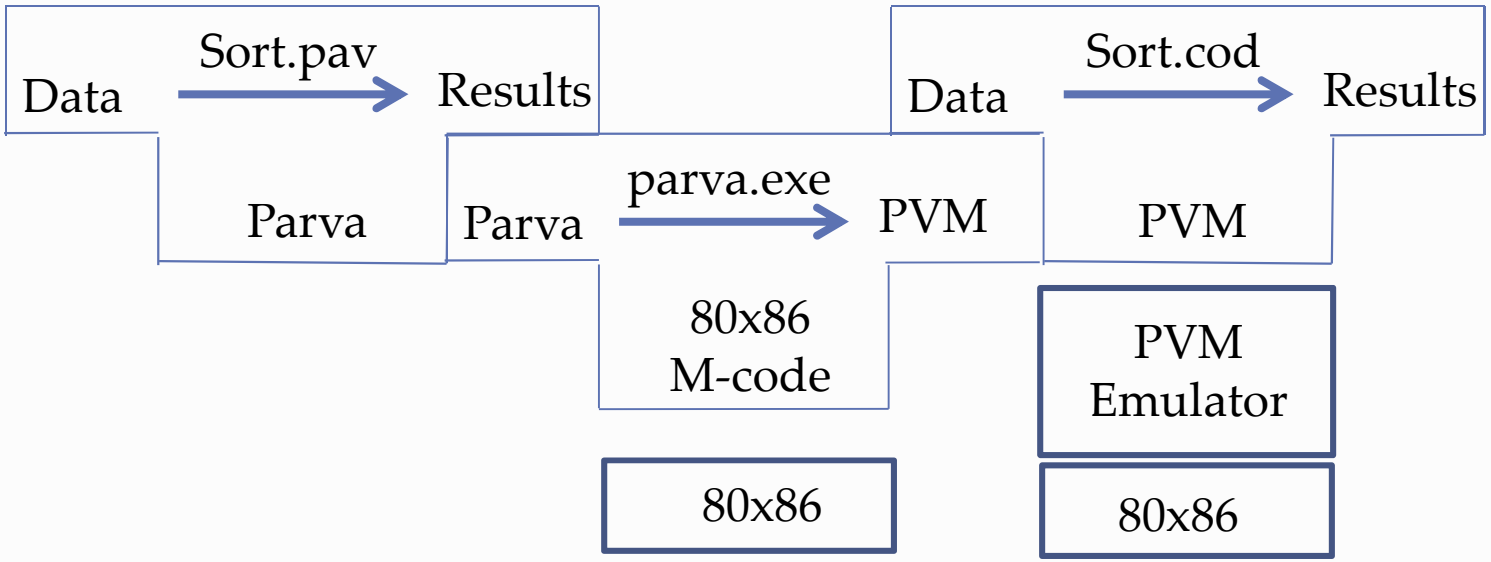
- Full compiler translates program to code that can be run directly



- But writing full compiler is difficult
- To prototype new languages or compilers for new machines, often use compilers that generate simple pseudo-machine code

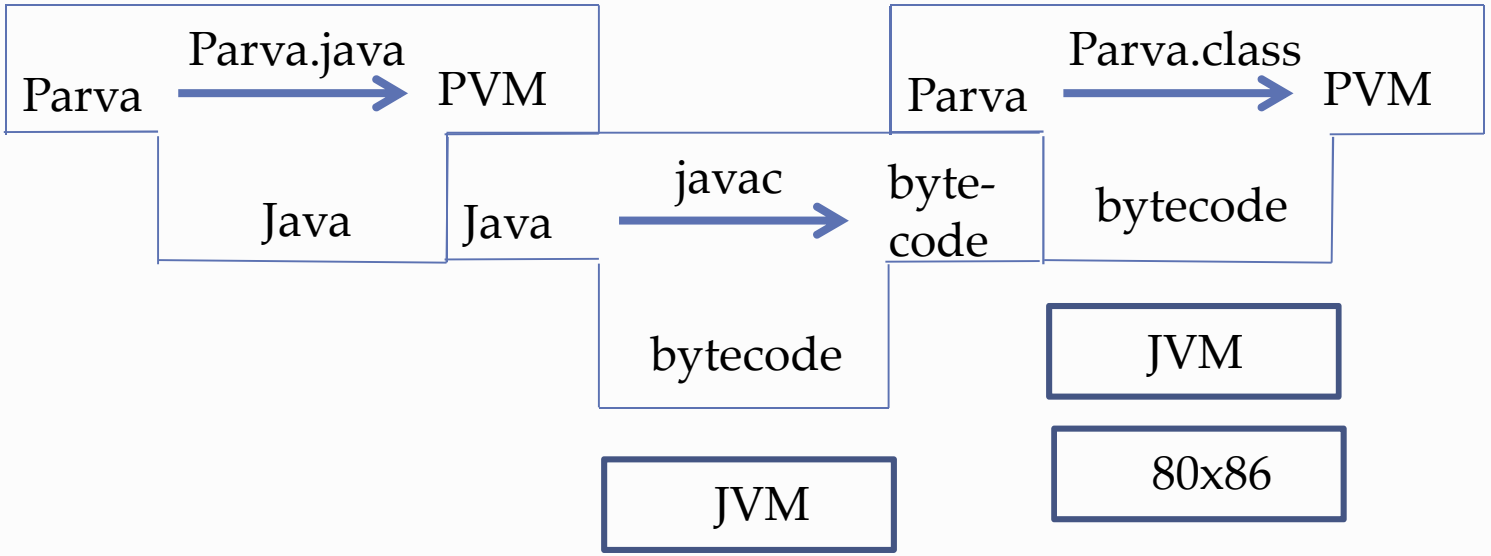
Interpretive compiler

- Pseudo-machine code executed by an interpreter that emulates the simple pseudo-machine

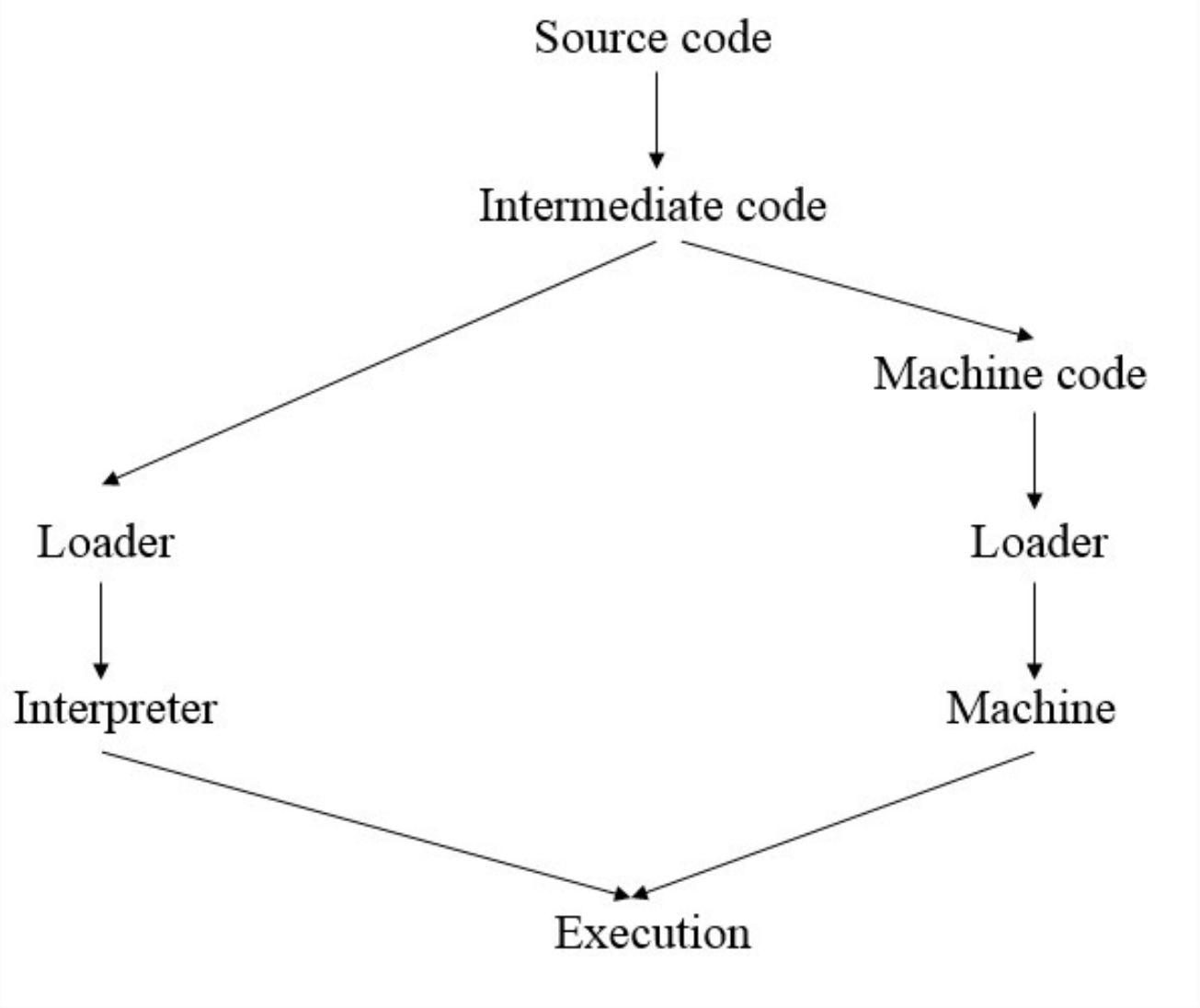


Interpretive compiler (2)

- An ***interpretive*** compiler can be developed in any convenient high-level language for which a compiler already exists (for example Java)



Pseudo-code and native code compilers compared



Advantages of interpretive compilers

- Easier to generate pseudo-code than real machine code
- Easily made user friendly
- Allows quick prototyping of new languages
- Allows for easy porting to new machines - interpreters are easy to write and to debug
- Can be made very portable by using a well-defined pseudo-code
- Can implement a whole range of languages easily on wide range of different machines
- Useful in connection with cross-translators which can be tested by simulating the target machine
- Pseudo-code is very compact (more so than real machine code). Hence interpretive compilers can be run on small machines
- Easy to cover in an introductory course!

Disadvantages of interpretive compilers

- Much slower "object code" than is the case with real machine code. Slow-down factors of 10 - 100 are fairly common.

Next lecture …

- Please read Chapter 3

Interpretive compilers (2)

- An ***interpretive*** compiler can be developed in any convenient high-level language for which a compiler already exists (for example C)