Compilation, Interpretation and Translation Models

Compilation Basics

- Translation is any action converting some source to some target

Figure 1 Translation

- Translation can be
  - Interpretation
    - The target is not explicit, it is transient, or just execution
  - Generation
    - The target is persistent
    - Compiler is generator where source
- Modern multi-file modular compilation

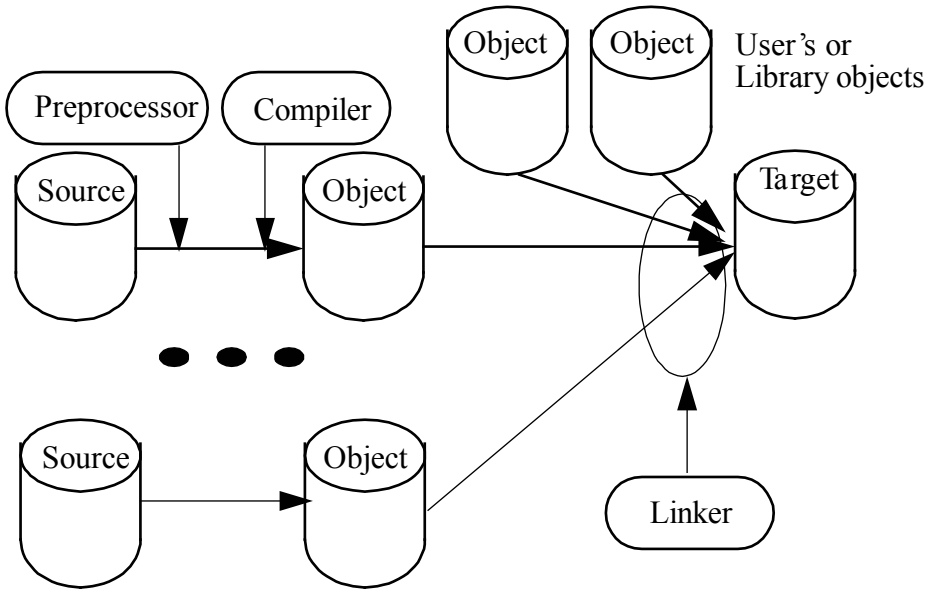

Figure 2 Modern compilation including preprocessor and linker.

- Compiler
  - **(Actual compiler) compiles only 1 source at a time into *object file***
  - The entire process is also referred to as compilation (of a project as opposed to a file)
    - Options can be used to stop the entire process or change properties
      - -E to stop after preprocessor
      - -c to stop after object is created
      - Static liking vs. dynamic linking
      - -o to rename target
      - The target is referred to as *executable*
  - *Preprocessor* does only textual substitutions and still produces text file that can be saved
  - Object file is not executable
    - *unresolved external references*
    - *relocatable* format
  - *Linker* resolves external references
    - Produces still relocatable load module
  - *Loader* creates true (absolute addresses) machine code
    - not really (*paging*)

Compilation Stages/Modules

- Compilation progresses in stages across three dimensions
  1. **Machine independent** processing -> **Machine dependent** processing
     - Can overlap in more optimizing compilers
     - If separate, there are Front End and Back End joined by some intermediate internal representation
  2. Analysis -> Synthesis
  3. **Front end** -> **Back end** (in modular compilers)
- The text source is temporarily represented differently (internal representation) to finish the processing
  - Virtual assembly instructions, Tree

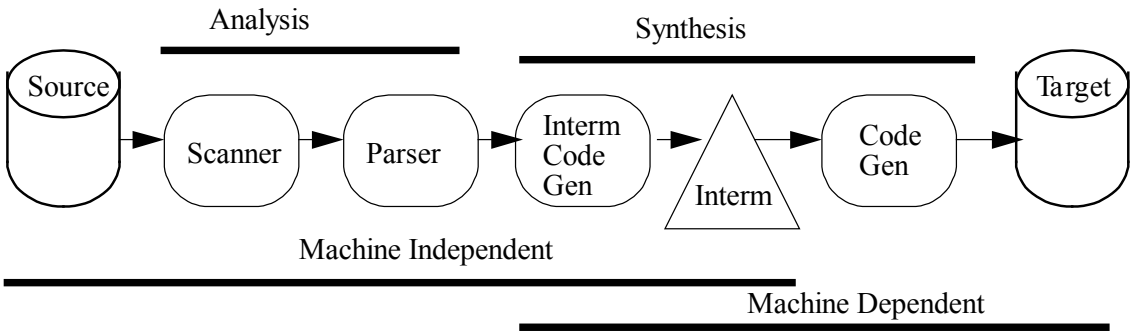

Figure 3 Stages in modern compiler. Machine independent processing can overlap with dependent or not.

- ***Scanner*** (*lexical analyzer*) performs translation from characters to tokens
- ***Parser*** checks *syntax* (grammar) then generates *intermediate representation*
- Processes on the intermediate representation
    - Static semantics
    - Code generation
    - Storage allocation
- *Optimization* can be performed across all stages but primarily in/after code generation
- The intermediate representation separates
    - *Front end* – usually machine independent, but language specific
    - *Back end* – usually machine dependent, but language independent
      sometime interpreted even if front end compiled
- Advantages of modularization (into modules and/or ends)
    - Reusability
    - Independent development
    - Separation of concerns
    - Lower development cost
    - Faster development
- Disadvantage of modularization
    - Translation speed
    - Optimization

Tombstone Notation for Translation

- Program (source language)
- Compiler vs. Interpreter
- Machine (machine language)
- **Translator source is also a program**

Figure 4 Program P in language L, and translator as program: translating from S to T, written in language L (S and L could be the same).

Figure 5 Machine executing machine language M, and execution of program P on that machine.

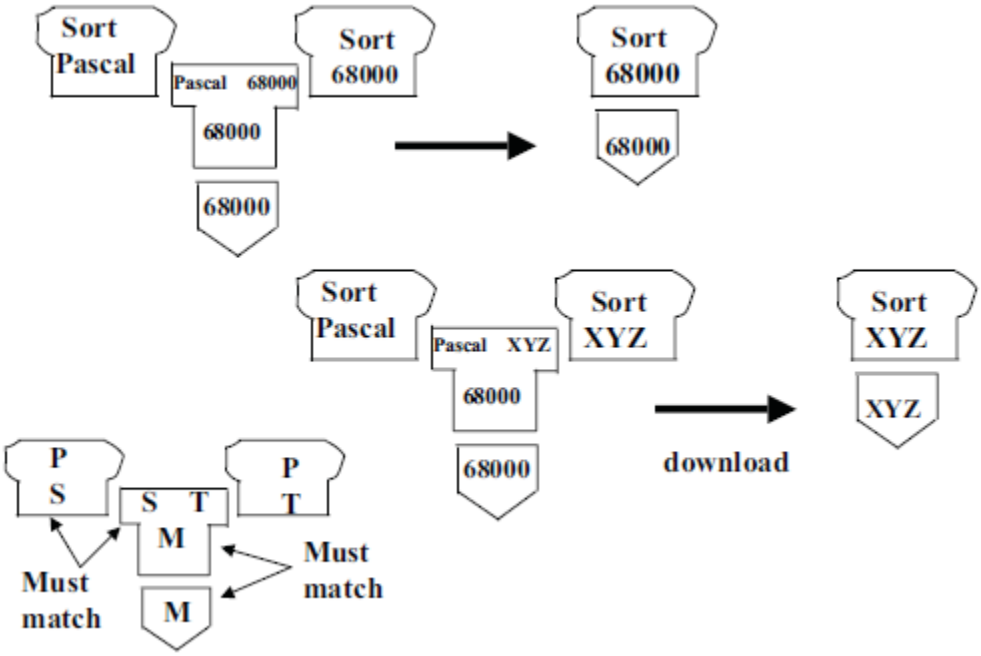

Figure 6 Examples of execution environments including translations: *host* and *cross*.

- Execution environment
  - Anything that needs to be provided for execution of a given source
  - In compiler, this is
    - Compilation to executable
    - Execution of executable
- *Host translation*
  - Translation and execution is on the same machine
  - Compiler is written in the same language as the target
- *Cross translation*
  - Compiler is written in different language than its target language
    - Target can be machine language or another HLL language

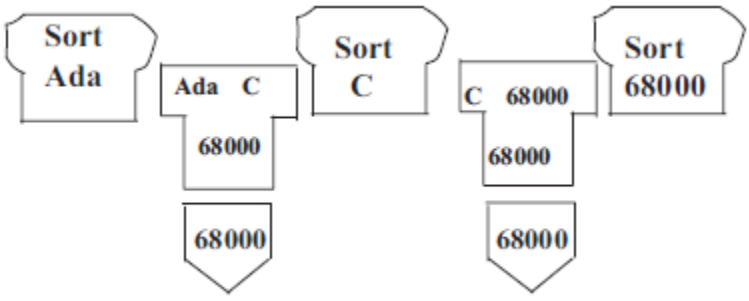

Figure 7 Example of cross translation to HLL language.

- Translator source is also a program that must be compiled if in the source format

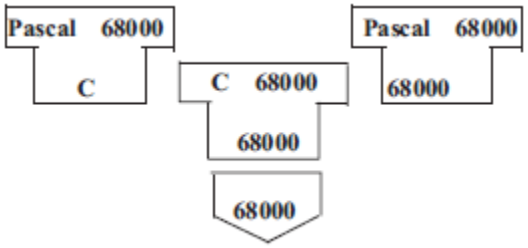

Figure 8 Pascal compiler: source written in C compiled to host executable.

Interpreters

- Interpreters, same as compilers, are both
  - Programs that have source and need to be translated (probably compiled) to have executables
  - Translators (in executable form)

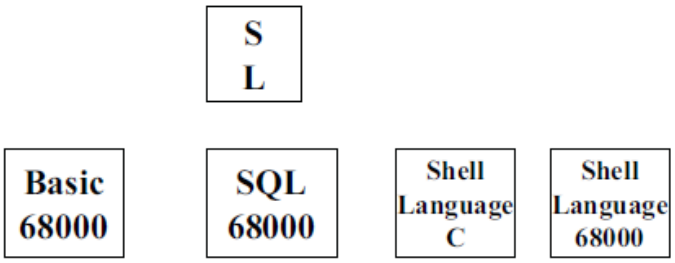

Figure 9. Interpreters. At top is interpreter for language S written in language L.

- You must have the proper interpreter for a given program and you must have the proper executable for it.

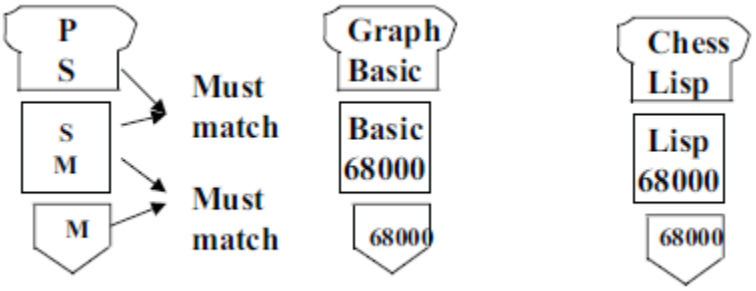

Figure 10. Interpreters in use.

- Interpreters lead to abstract machines

Real and Abstract Machines

- A *real machine* is one executing in hardware its designated language (machine language)
    - Micro-coded machines are still considered real
- An *abstract machine* is one executing some language through a software interpreter
    - The language of execution can be thought of as *abstract machine language*

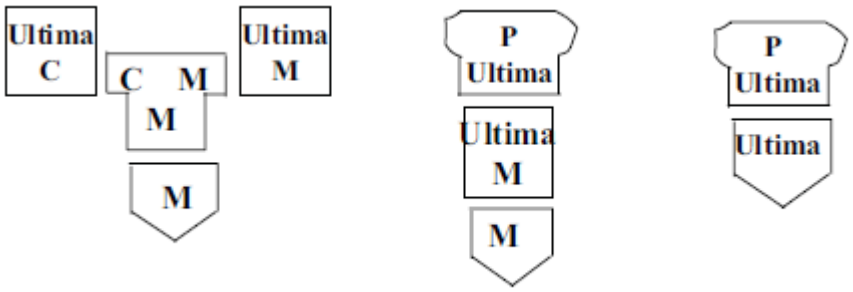

Figure 11. An interpreter is first compiled and then used to create an abstract machine running the ultima machine language.

Interpretive Compilers

- The standard compiler as in Figure 3 with the back end replaced with an interpreter
- Hybrid of compilation and interpretation made up of 2 stages
    - Front is compiler
    - Back is interpreter
- Tradeoffs in interpretive compiler
    - Uses some low level intermediate representation between the two stages
    - Slower to execute but faster to compile
    - Cheaper to produce
        - The same reconfigurable process as with the two stages of a compiler
        - Front compiler is language specific but machine independent
        - Back interpreter is language independent but machine specific
- Examples
    - Java
        - Front compiler and back JVM, intermediate representation is java bytecode
        - Many advantages
    - Pascal
        - Older but the same principles, intermediate representation is *p-code*

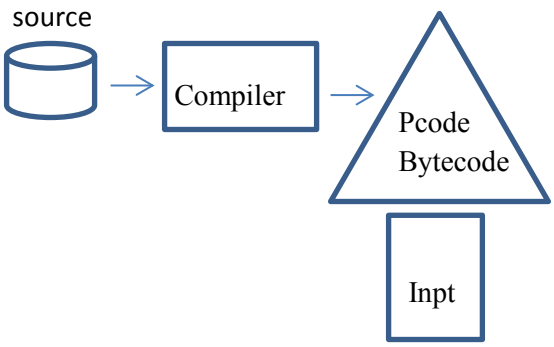

Figure 12. Interpretive compiler. Front end is machine independent but language specific compiler. Back end is machine dependent but language independent virtual machine (interpreter). The back end depends only on the intermediate representation (p-code in Pascal, bytecode in Java) and not on the original source.

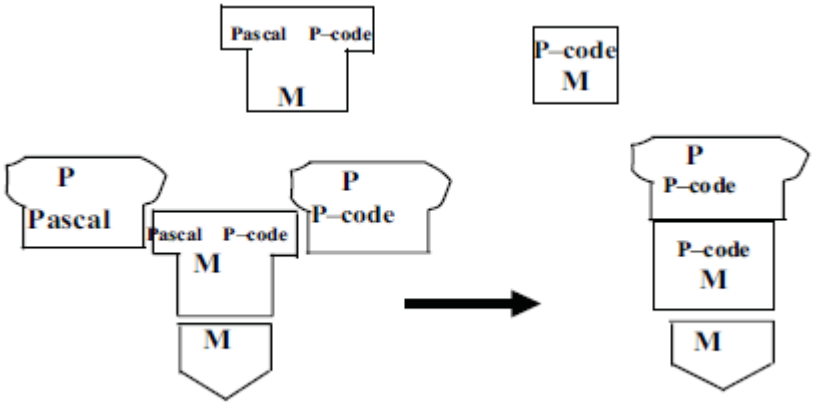

Figure 13. Pascal interpretive compiler in action.

- Interpretive compilers are simplest examples of *portable compilers*
  - Developed in interchangeable/reconfigurable modules
  - For example, the same JVM on Windows is the back interpreter for all compiled java programs and possible compiled programs from other language

Example: Improving properties of Pascal Interpretive Compiler *(skip)*

Suppose we have the parts as of Fig. 14.

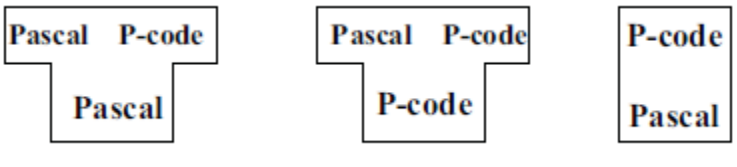

Figure 14. Suppose these are the starting pieces: front compiler source in Pascal and compiled in p-code, and a p-code interpreter source in Pascal.

How can we accomplish execution of a Pascal program?

Solution 1. Build interpreter(s) for the sources above (compiler source and interpreter source), then run the interpretive compiler

- Interpreter for P-code is sufficient
  - could to Pascal interpreter but harder to build and not really needed
- inefficient through abstract machines on both sides front end and back end

Solution 2. Improve efficiency by

1. Removing interpreters and using real machines, and possibly then
2. by substituting the back interpreter with real compiler
   - combining both sides into one compiler

Example Solution 1

1. Currently available compiler are sources: 2 sources for the compiler front end and 1 source for the interpreter back end as in Figure 14
   Suppose we have Pascal program P to execute
   - At present, the program cannot be executed yet
2. Looking back at what we have in Figure 14, we can try either of the following
   - Find a host Pascal compiler and compile the first source compiler and then the interpreter
   - Find a cross compiler from Pascal to another language assuming we also have a compiler for the other language
   - Create virtual machine (interpreter) for either Pascal or p-code.
     Here suppose we write an interpreter for P-code, in C, and compile with an available C host compiler on the same machine to create a P-code virtual machine.

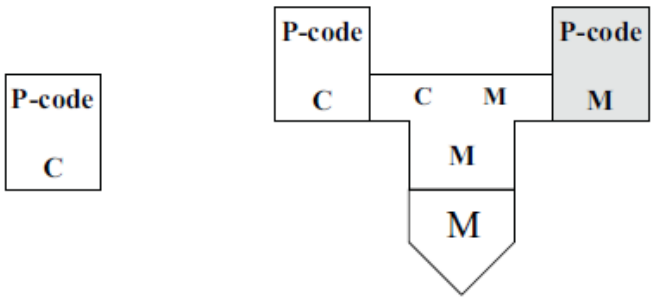

Figure 15. A newly implemented p-code interpreter, in C, compiled with a C compiler.

   - Now the Pascal program P can be executed. But the execution environment is very slow as seen in Figure 16
     - It is 2 stage interpretive compiler as shown in Figure 12
       - the first stage is compiler running through an interpreter
       - the second stage is a standard interpreter

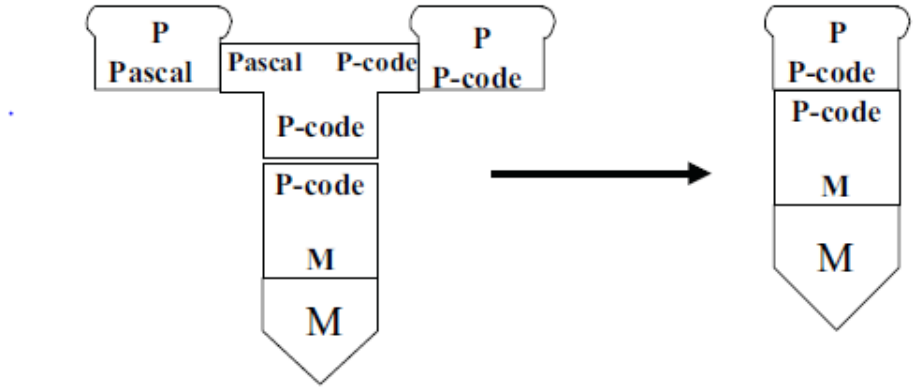

Figure 16. Execution environment for the Pascal program P.

Example Solution 2

3. Because both stages suffer through the same interpretation (p-code) we can get rid of both interpreters if we can get rid of both virtual machines
    - The compiler will be cross-compiler Pascal to P-code but must be in M to run on real machine M
    - The interpreter can be replaced with host compiler P-code to M to run on real machine M
    - Of course a better solution is to change the compiler to be a host compiler and thus generate M code directly in the front end above, but it would be cheaper to build P-code to M instead of Pascal to M (P-code is lower level)
        - It is bootstrapping
    - Shown in Figure 17
        - The new compiler is written in Pascal and then compiled into p-code version
        - The p-code version is compiled through itself
            - *bootstrapping* (compiling through itself in this case)
        - One of the original sources is now recompiled through this new compiler
        - Now we still have 2-stage compiler with p-code as the intermediate representation
            - Shown in Figure 17 and Figure 18
        - Could continue to result in 1-stage compiler if desired but the two stages can be integrated so that user has no knowledge of the 2 stages and sees this compiler now as a single stage compiler

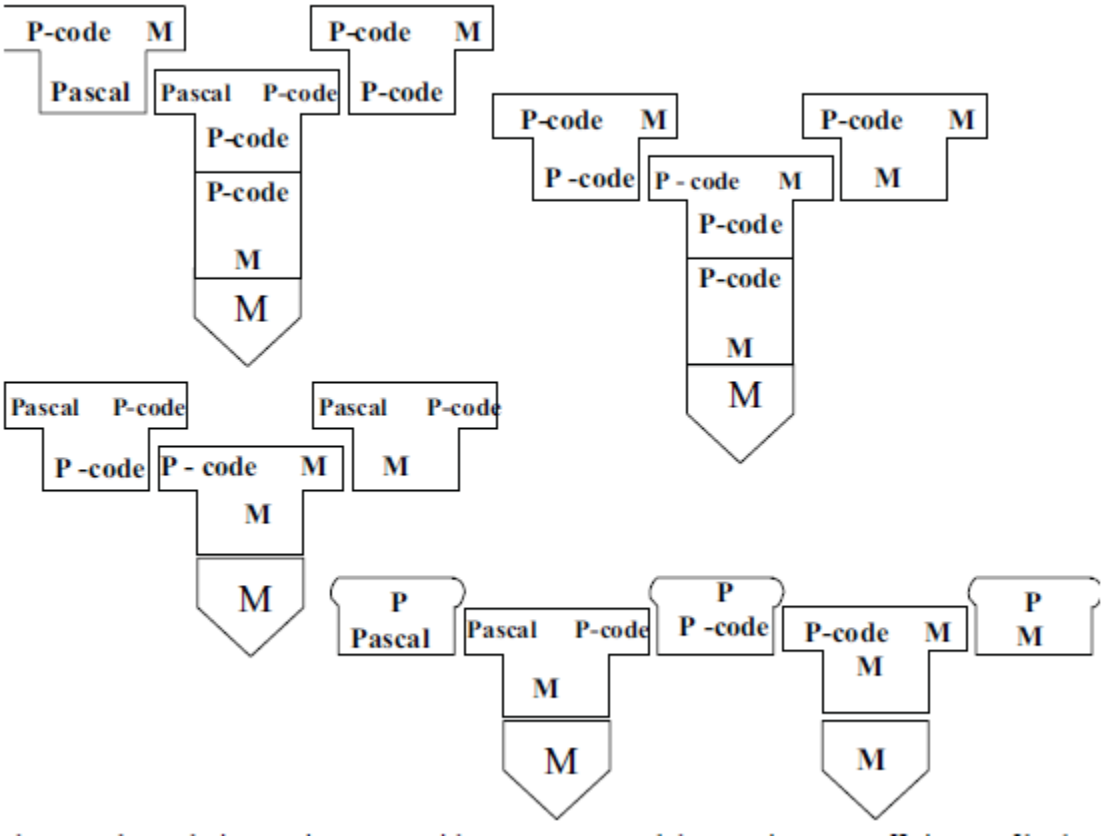

Figure 17. Improving efficiency of the previous interpretive compiler by moving both stages to real machine. This is an example of *bootstrapping*.

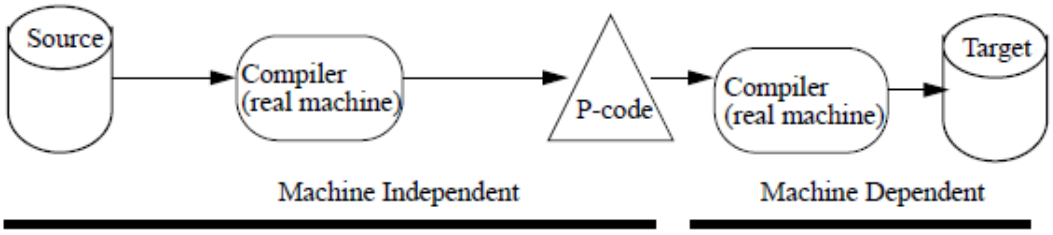

Figure 18. The resulting 2-stage compiler (not interpretive compiler any more).

Example: Improving Properties of Java Model

- Assume a Java model with front end compiler and back end interpreter and the intermediate representation as bytecode (BC). Compilation/execution of a source is illustrated in Figure 19. Also assume the source for the compiler is available as shown.

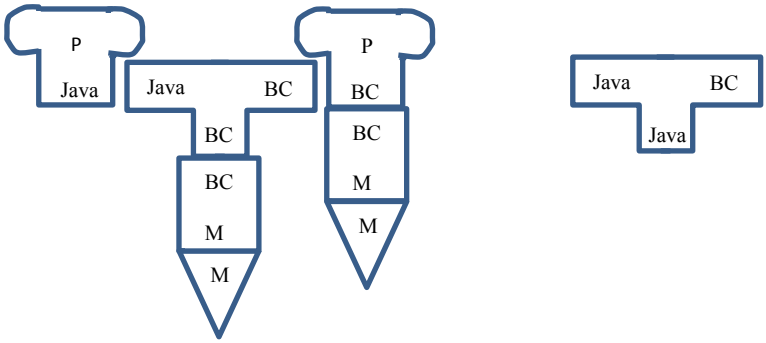

Figure 19. Java program compiled then executed. Java compiler runs on JVM interpreter, and so does the BC target.

- The objective is to speed up execution of program P. This can be accomplished here in two ways, and either one can also then be used to speed up the compiler through bootstrapping.
  - Change the compiler to generate machine language M directly
  - Replace the interpreter with a compiler from BC to machine language M

Change the compiler to generate machine language M directly

- Change the back end BE in the original source to generate machine language M instead of bytecode BC then recompile through the available compiler. We get faster target execution by removing the interpreter as in

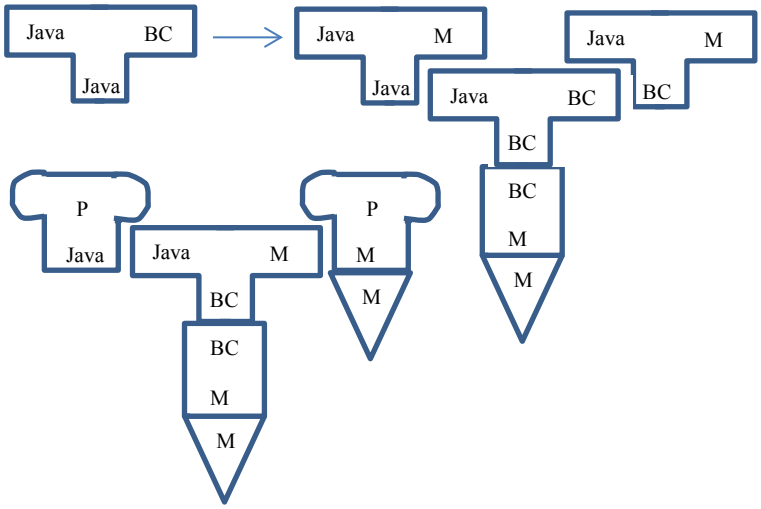

Figure 20. Modify the original javac source to generate machine language instead speeds up execution of targets.

- Now recompile the modified source through bootstrapping to speed up the compiler too as in Figure 21.

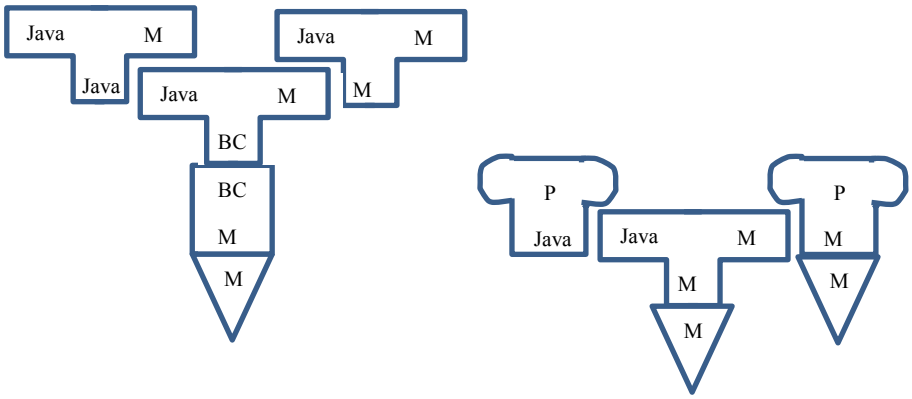

Figure 21. Bootstrapping the modified Java compiler. Now the compiler also runs fast.

Replace the interpreter with a compiler from BC to machine language M

- In this approach we introduce a low level compiler from BC to M so that bytecode targets can be subsequently compiled in native machine language M, as illustrated in Figure 22.

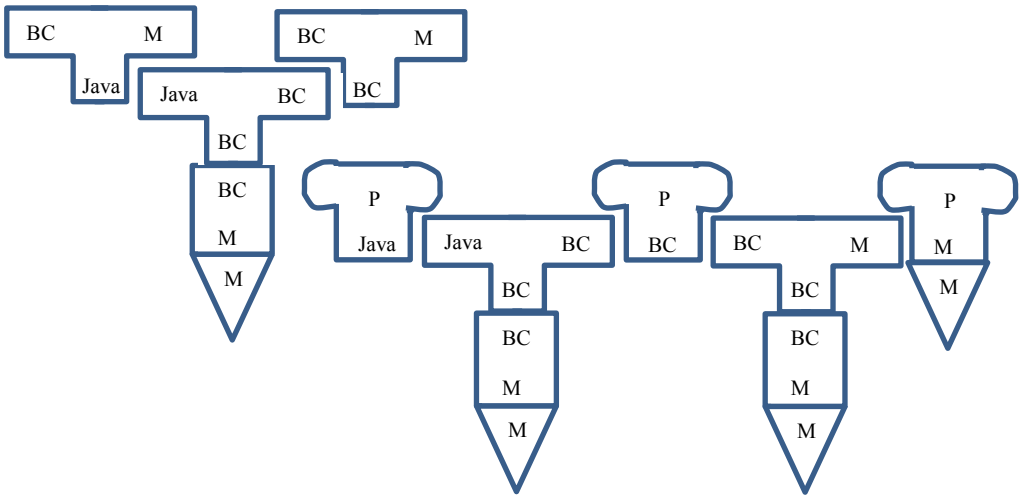

Figure 22. Replacing BC (JVM) interpreter with a compiler.

- The targets are now faster but the compilation (*2-stage*) is still slow.
  - Can be improved through bootstrapping since we have compiler to change BC to M, as in Figure 23.

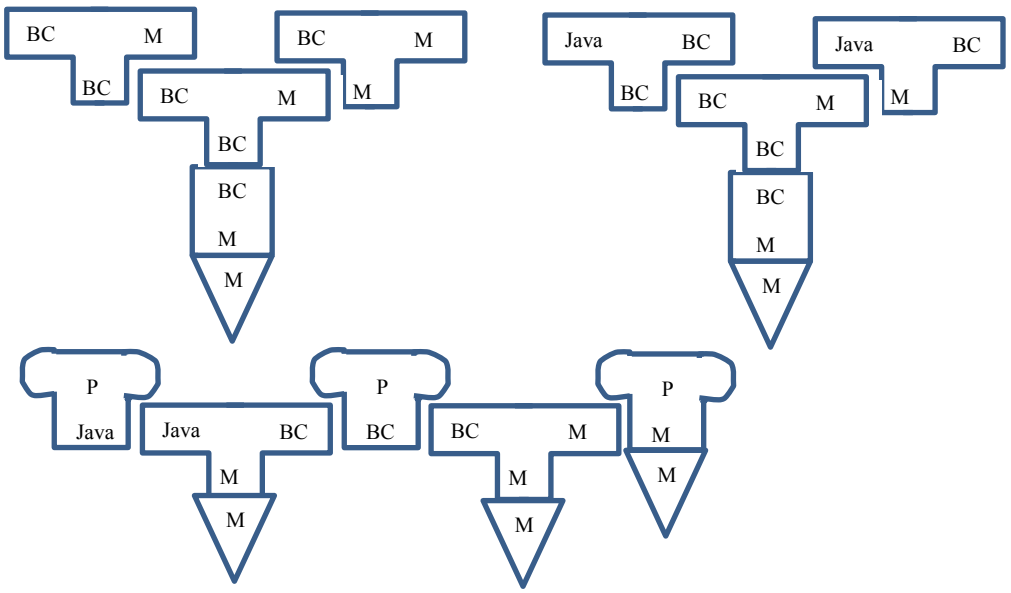

Figure 23. Bootstrapping to make both stage compilers run directly in machine language M.

Improving Runtime Properties

- When speaking of execution by compilation, we speak of 2 separate processes
  - Compilation of the source
  - Execution of the target
- Either or both can be inefficient and subject to improvement
  - Bootstrapping can improve them both
- Example:
  - Using version v1 results in slow compilation and slow execution (Mx indicated slow version for machine language M)
  - Version v2 improves execution of target and then can improve compilation speed if bootstrapped

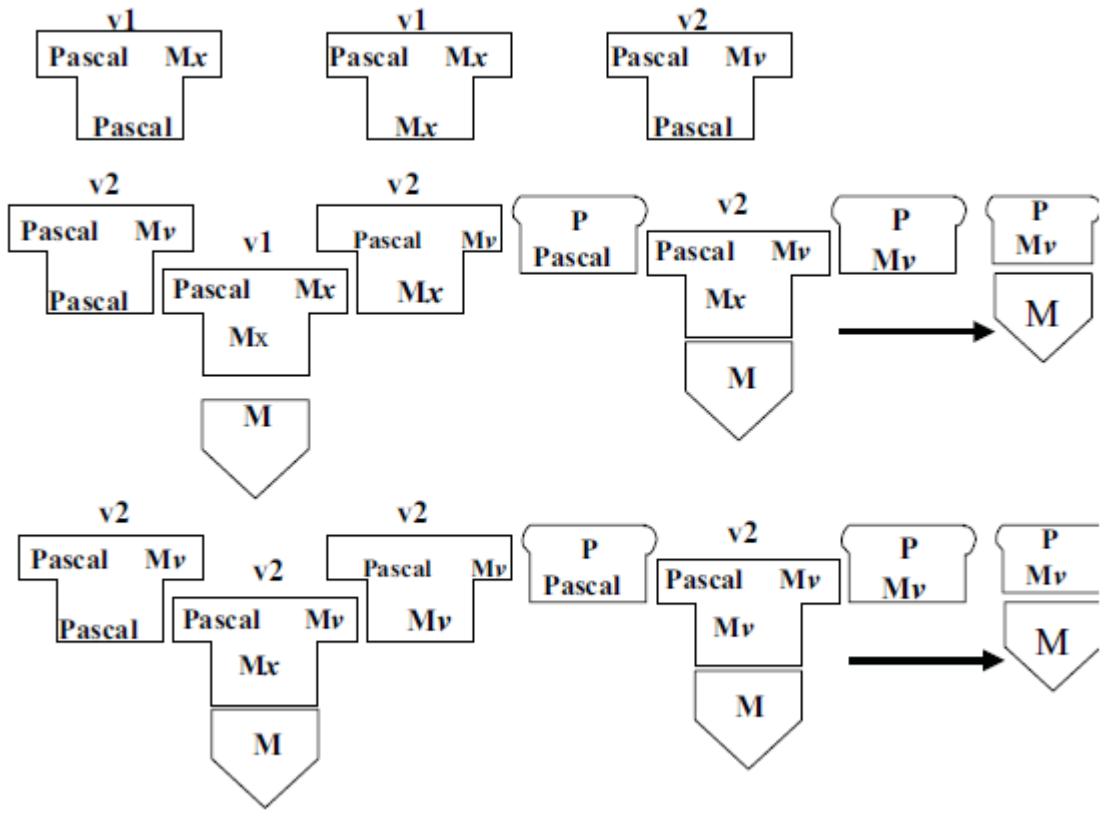

Figure 24. Improving compiler and execution properties (such as speed) through bootstrapping improves them both.