3 Compilers and interpreters

- Compilers and other translators
- Interpreters
- Tombstone diagrams
- Real vs virtual machines
- Interpretive compilers
- Just-in-time compilers
- Portable compilers
- Bootstrapping

University of Glasgow

Translators *(1)*

- An ***S* → *T* translator** accepts code expressed in one language *S*, and translates it to equivalent code expressed in another language *T*:
  - *S* is the *source language*
  - *T* is the *target language*.
- Examples of translators:
  - compilers
  - assemblers
  - high-level translators
  - decompilers.

Translators *(2)*

- A **compiler** translates high-level PL code to low-level code. E.g.:
  - Java → JVM
  - C → x86as
  - C → x86

using "x86as" as shorthand for x86 assembly code

using "x86" as shorthand for x86 machine code

- An **assembler** translates assembly language code to the corresponding machine code. E.g.:
  - x86as → x86

Translators *(3)*

- A **high-level translator** translates code in one PL to code in another PL. E.g.:
  – Java → C
- A **decompiler** translates low-level code to high-level PL code. E.g.:
  – JVM → Java

University of Glasgow

Interpreters *(1)*

- An ***S* interpreter** accepts code expressed in language *S*, and *immediately* executes that code.
- An interpreter works by *fetching*, *analysing*, and *executing* one instruction at a time.
  - If an instruction is fetched repeatedly, it will be analysed repeatedly. This is time-consuming unless instructions have very simple formats.

Interpreters *(2)*

- Interpreting a program is slower than executing native machine code:
  - Interpreting a high-level language is ~ 100 times slower.
  - Interpreting an intermediate-level language (such as JVM code) is ~ 10 times slower.
- On the other hand, interpreting a program cuts out compile-time.

Interpreters *(3)*

- Interpretation is sensible when (e.g.):
  - a user is entering instructions interactively, and wishes to see the results of each instruction before entering the next one
  - the program is to be used once then discarded (so execution speed is unimportant)
  - each instruction will be executed only once or a few times
  - the instructions have very simple formats
  - the program code is required to be highly portable.

Example of interpreters *(1)*

- Basic interpreter:
  - A Basic program is a sequence of simple commands linked by unconditional and conditional jumps.
  - The Basic interpreter fetches, parses, and executes one simple command at a time.
- JVM interpreter:
  - A JVM program consists of "bytecodes".
  - The interpreter fetches, decodes, and executes one bytecode at a time.
  - *Note:* The JVM interpreter is available stand-alone (`java`) or as a component of a web browser.

Example of interpreters *(2)*

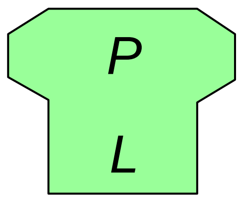
- Unix command language interpreter (*shell*):
  - The user enters one command at a time.
  - The shell reads the command, parses it to determine the command name and argument(s), and executes it.

Compilers vs interpreters

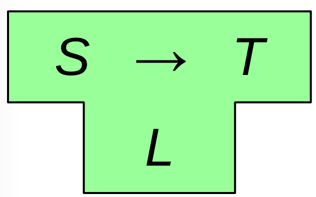
- Do not confuse compilers and interpreters.
- A compiler translates source code to object code.
  - It *does not execute* the source or object code.
- An interpreter executes source code one instruction at a time.
  - It *does not translate* the source code.

Tombstone diagrams

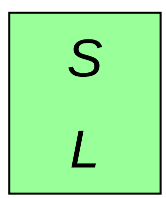
- Software:



an ordinary program *P*, expressed in language *L*

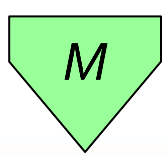


an *S* → *T* translator, expressed in language *L*



an *S* interpreter, expressed in language *L*

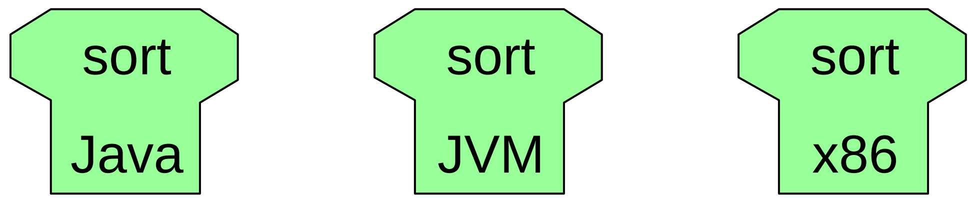
- Hardware:



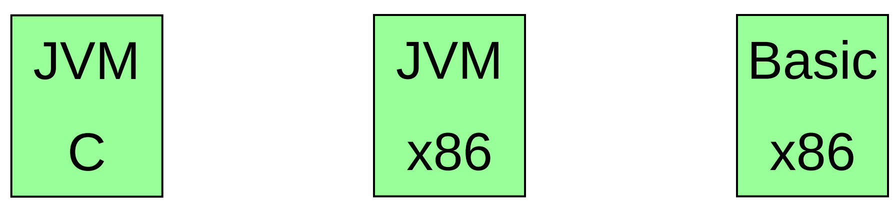
a machine *M* (which can only execute *M* ’s machine code)

Examples: tombstones *(1)*

- Ordinary programs:

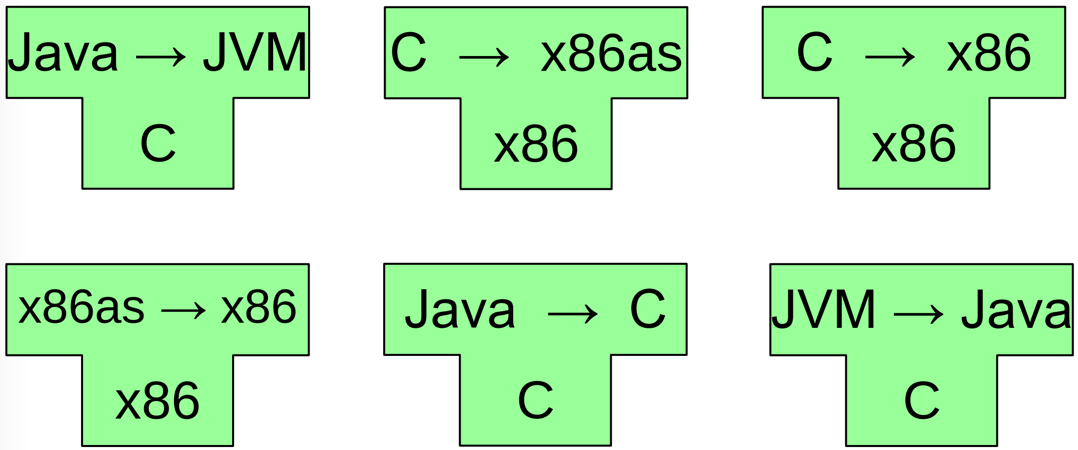


- Interpreters:



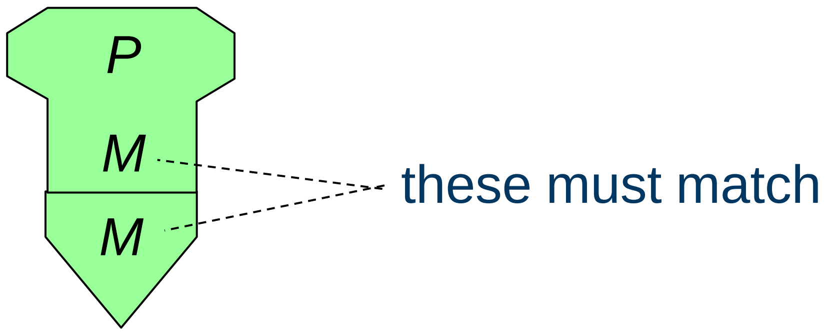
Examples: tombstones *(2)*

- Translators:



Tombstone diagrams: running programs

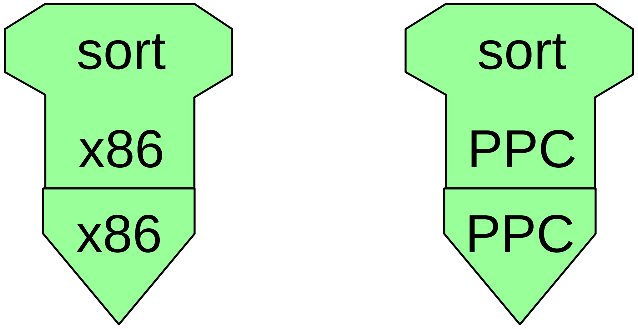
- Given a program *P* expressed in *M* machine code, we can run *P* on machine *M*:



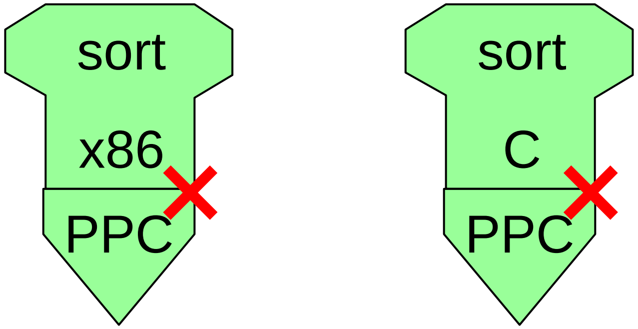
- Here "*M* " denotes both the machine itself and its machine code.

Examples: running ordinary programs

- Possible:



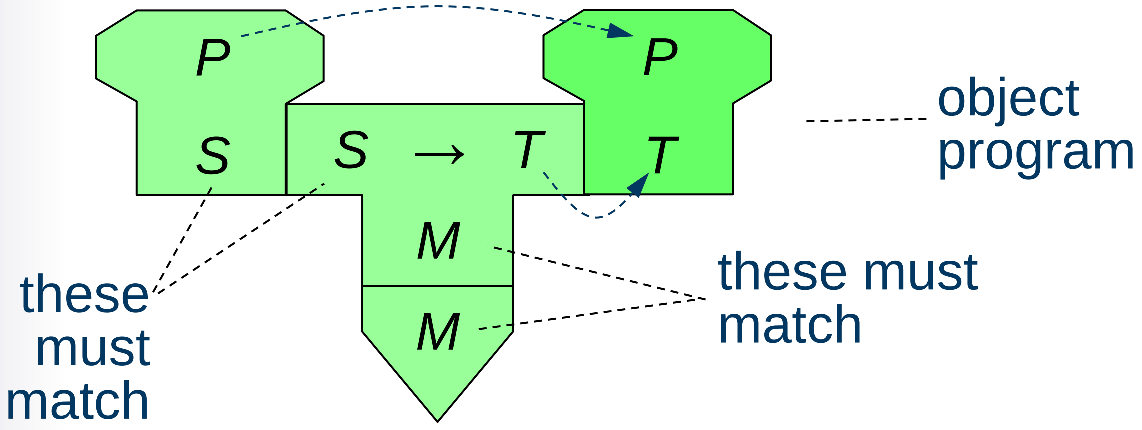
- Impossible:



Tombstone diagrams: translation

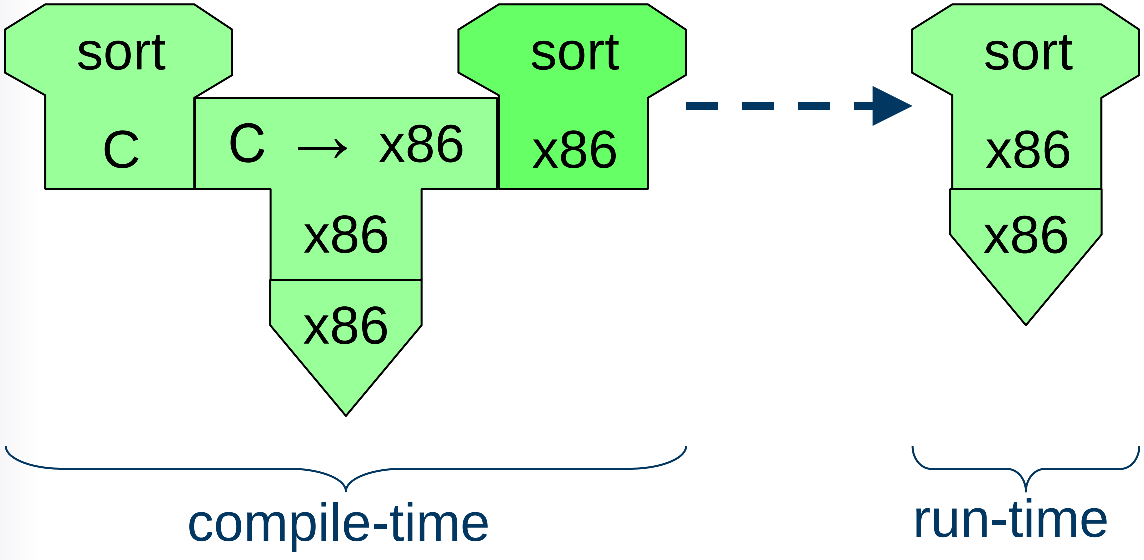
- Given:
  - an $S \rightarrow T$ translator, expressed in M machine code
  - a program P, expressed in language S

we can translate P to language T:



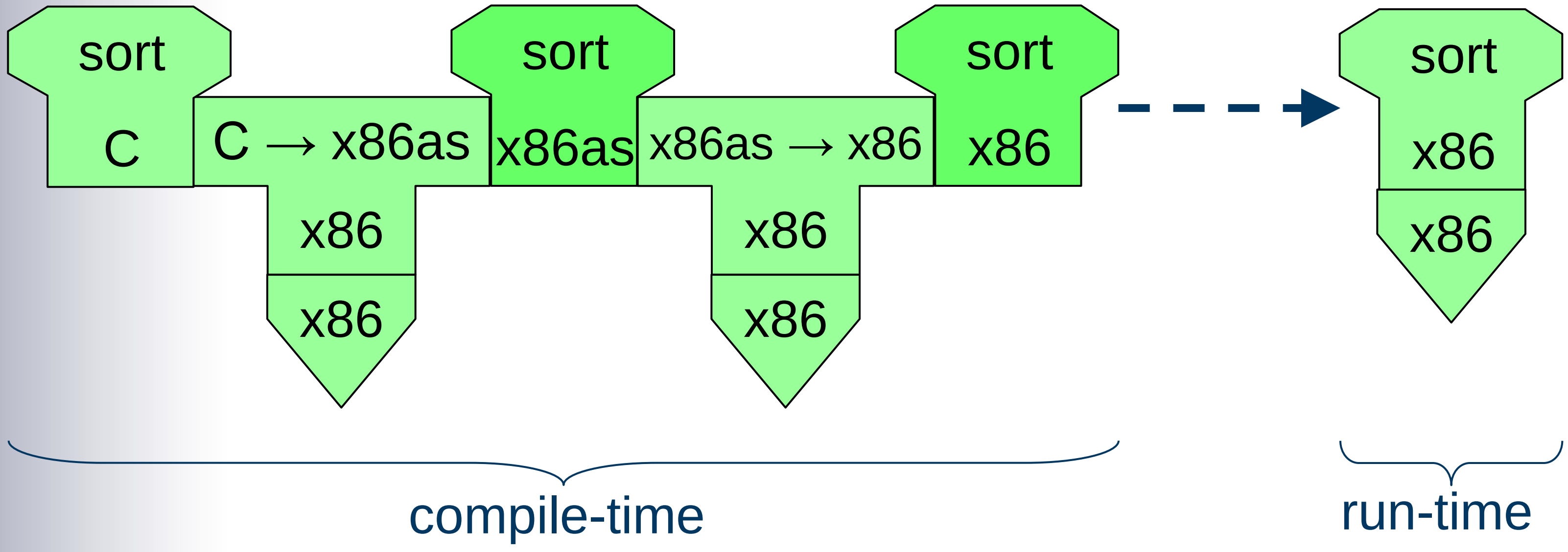
Example: compiling a program

- Given a C → x86 compiler, we can use it to compile a C program into x86 machine code. Later we can run the object program on an x86:



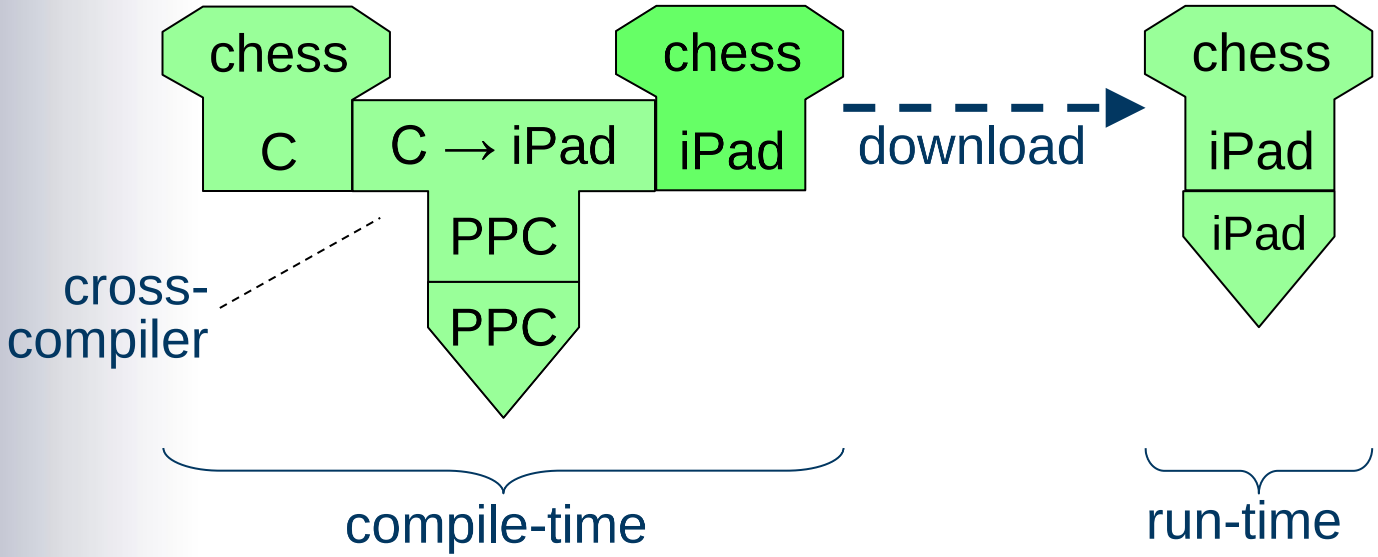
Example: compiling a program in stages

- Given a C → x86as compiler and an x86 assembler, we can use them to compile a C program into x86 machine code, in 2 stages. Later we can run the object program on an x86:



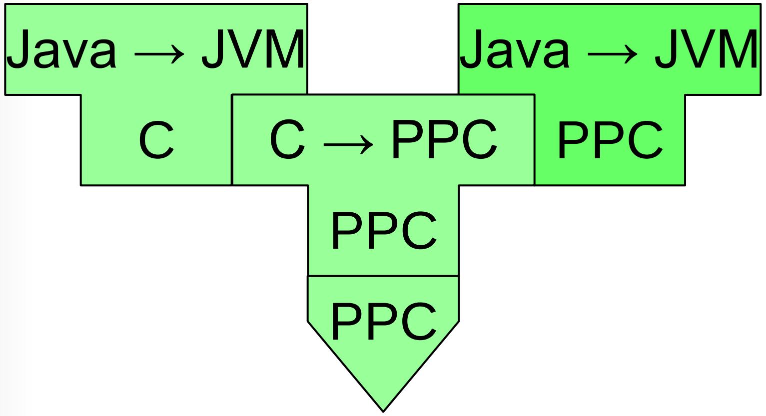
Example: cross-compiling a program

- Given a C → iPad compiler running on a PPC, we can use it to compile a C program into iPad machine code, then download the object program to an iPad. Later we can run the object program on the iPad:



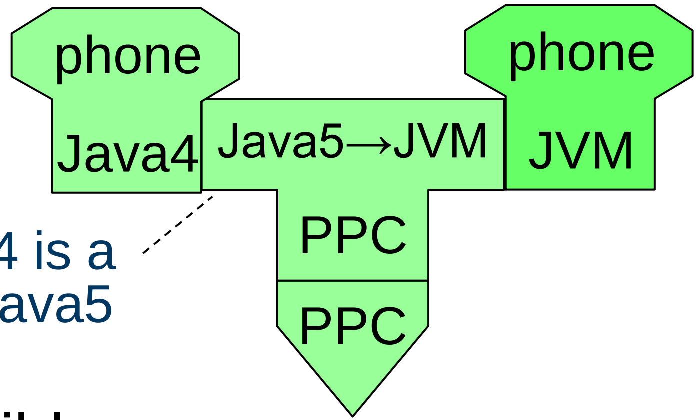
Example: compiling a compiler

- Given a C → PPC compiler, we can use it to compile *any* C program into PPC machine code.
- In particular, we can compile a compiler expressed in C:



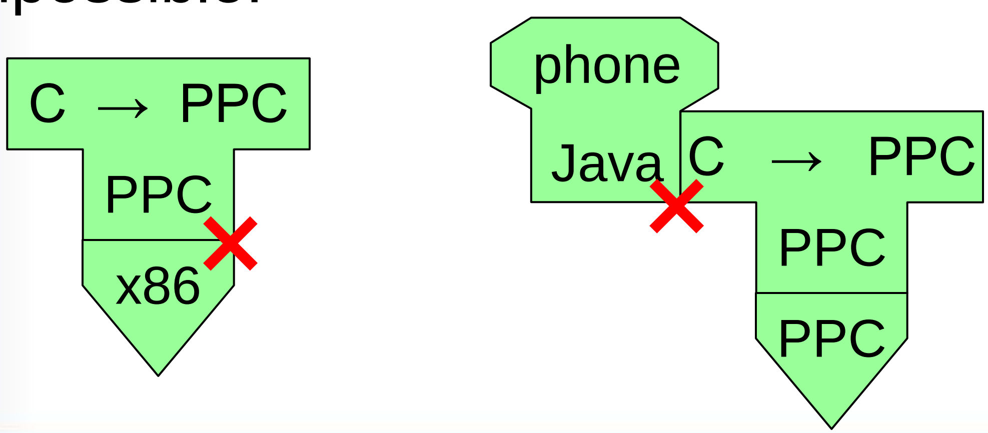
Examples: what can and can't be done

- Possible:



OK – Java4 is a subset of Java5

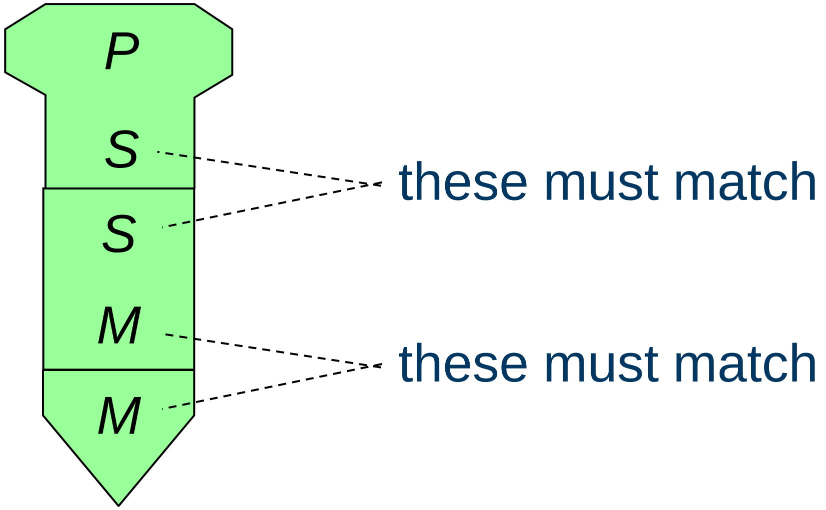
- Impossible:



Tombstone diagrams: interpretation

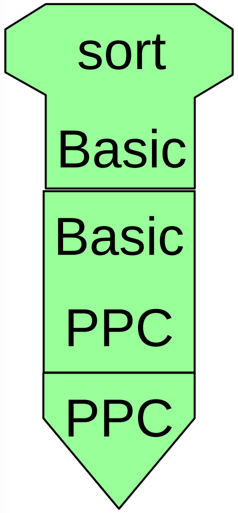
- Given:
  - an *S* interpreter, expressed in *M* machine code
  - a program *P*, expressed in language *S*

we can interpret *P*:

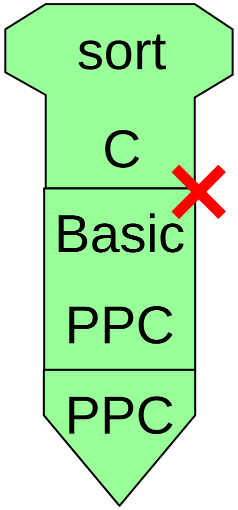


Examples: interpreting ordinary programs

- Possible:



- Impossible:



Real machines vs virtual machines

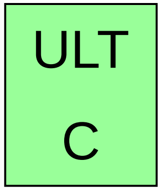
- A **real machine** is one whose machine code is executed by hardware.
- A **virtual machine** (or **abstract machine**) is one whose “machine code” is executed by an interpreter.

Example: hardware emulation *(1)*

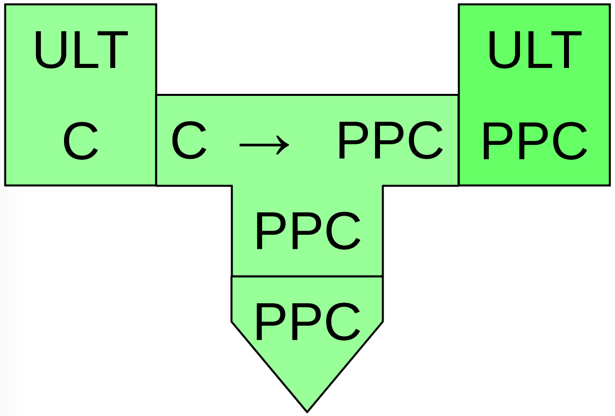
- Suppose we have designed the architecture and instruction set of a new machine, ULT.
- A hardware prototype of ULT will be expensive to build and modify.

Example: hardware emulation *(2)*

- Instead, first write an interpreter for ULT machine code (an **emulator**), expressed in (say) C:

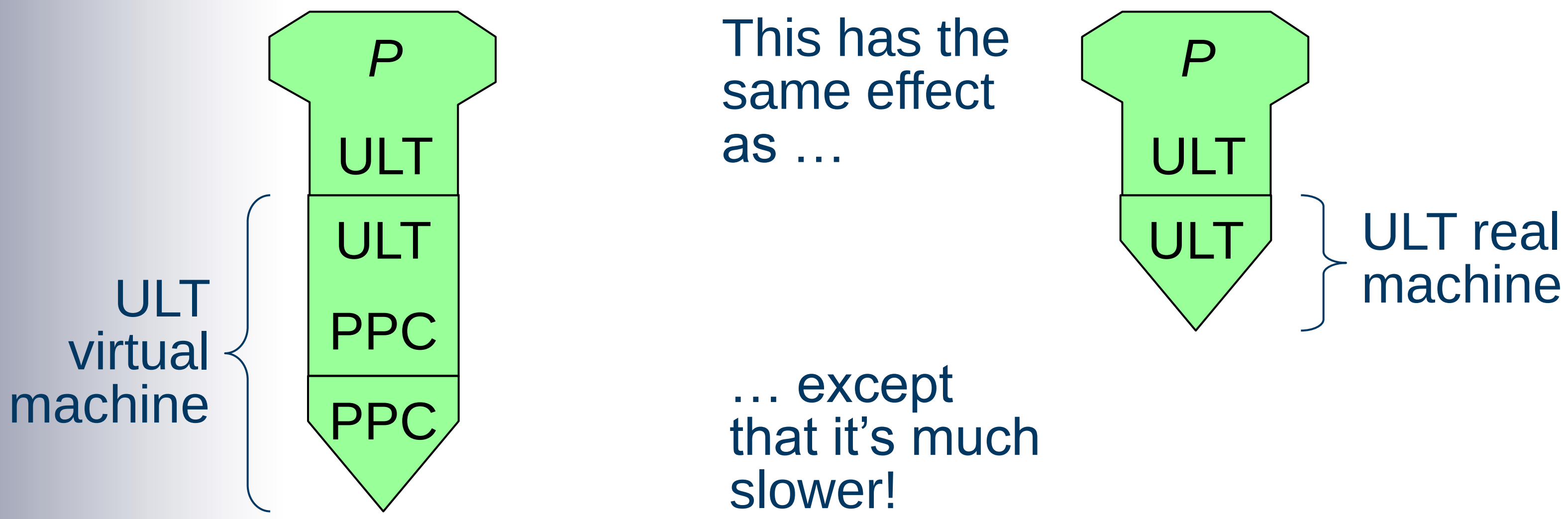


- Then compile it on a real machine, say PPC:



Example: hardware emulation *(3)*

- Now use the emulator to execute test programs *P* expressed in ULT machine-code:



Interpretive compilers *(1)*

- A compiler takes quite a long time to translate the source program to native machine code, but subsequent execution is fast.
- An interpreter starts executing the source program immediately, but execution is slow.
- An **interpretive compiler** is a good compromise. It translates the source program into virtual machine (VM) code, which is subsequently interpreted.

Interpretive compilers *(2)*

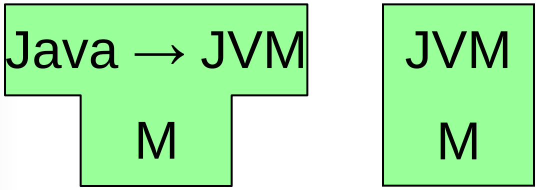
- An interpretive compiler combines fast translation with moderately fast execution, provided that:
  - the VM code is intermediate-level (lower-level than the source language, higher-level than native machine code)
  - translation from the source language to VM code is easy and fast
  - the VM instructions have simple formats (so can be analysed quickly by an interpreter).
- An interpretive compiler is well suited for use during program development.
  - But a compiler generating native machine code or assembly code is better suited for production use.

Example: JDK *(1)*

- **JDK** (Java Development Kit) provides an interpretive compiler for Java.
- This is based on the **JVM** (Java Virtual Machine), a virtual machine designed specifically for running Java programs:
  - JVM provides powerful instructions that implement object creation, method calls, array indexing, etc.
  - JVM instructions (often called “bytecodes”) are similar in format to native machine code: opcode + operand.
  - Interpretation of JVM code is “only” ~ 10 times slower than execution of native machine code.

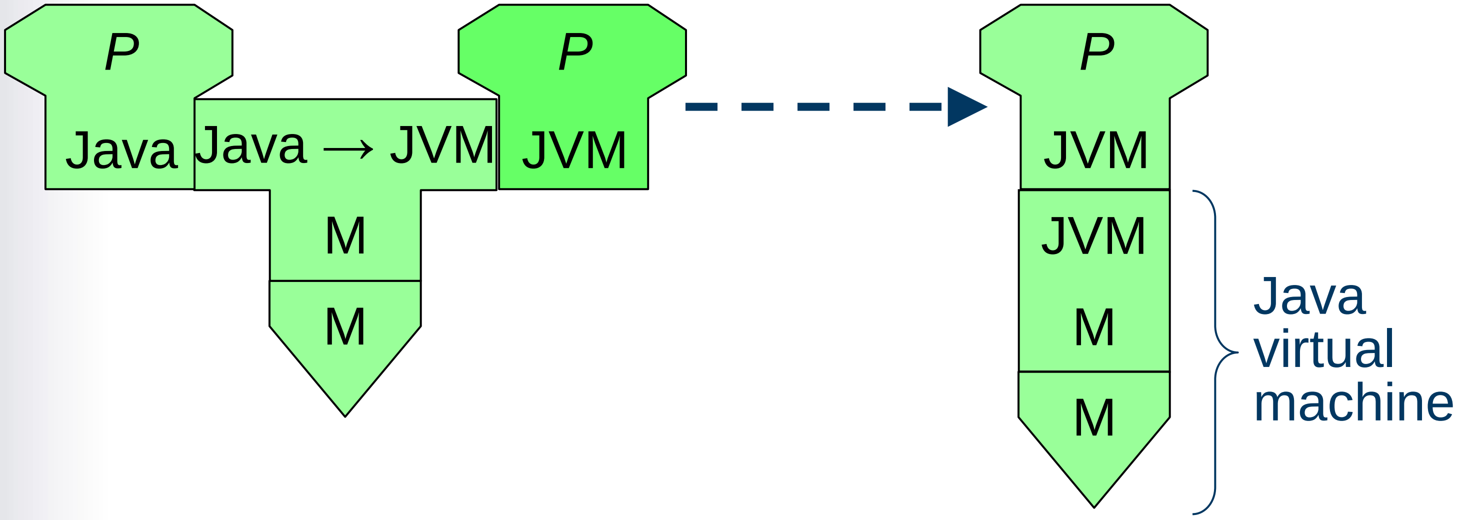
Example: JDK *(2)*

- JDK comprises a Java → JVM compiler and a JVM interpreter.
- Once JDK has been installed on a real machine *M*, we have:



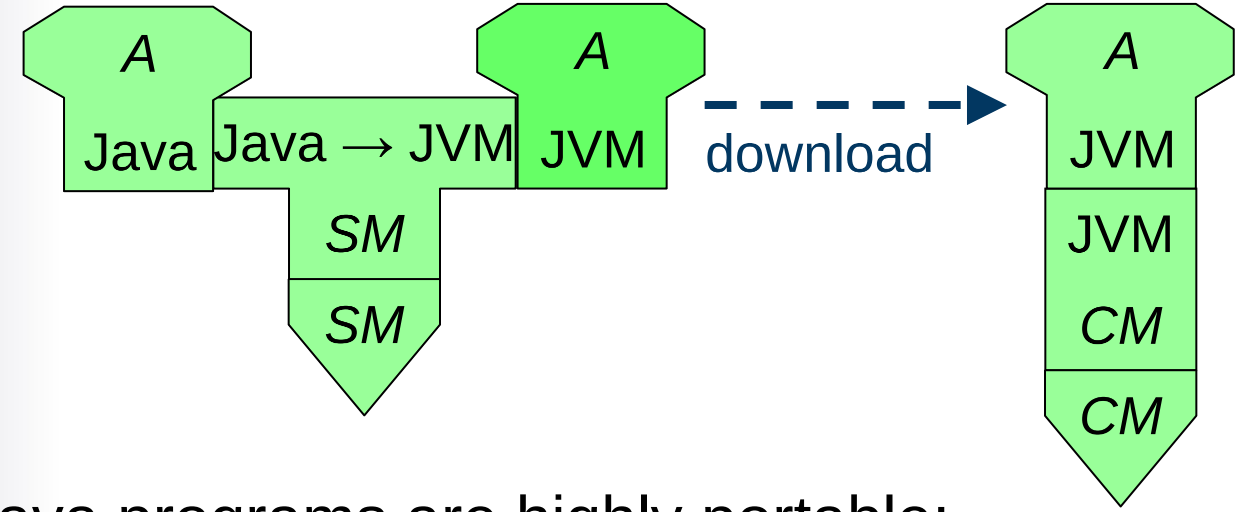
Example: JDK *(3)*

- A Java source program *P* is translated to JVM code. Later the object program is interpreted:



Example: JDK *(4)*

- A Java applet *A* is translated to JVM code on a server machine *SM*, where it is stored. Later the object program is downloaded on demand to a client machine *CM*, where it is interpreted:



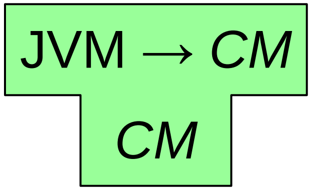
- Java programs are highly portable: “write once, run anywhere”.

Just-in-time compilers

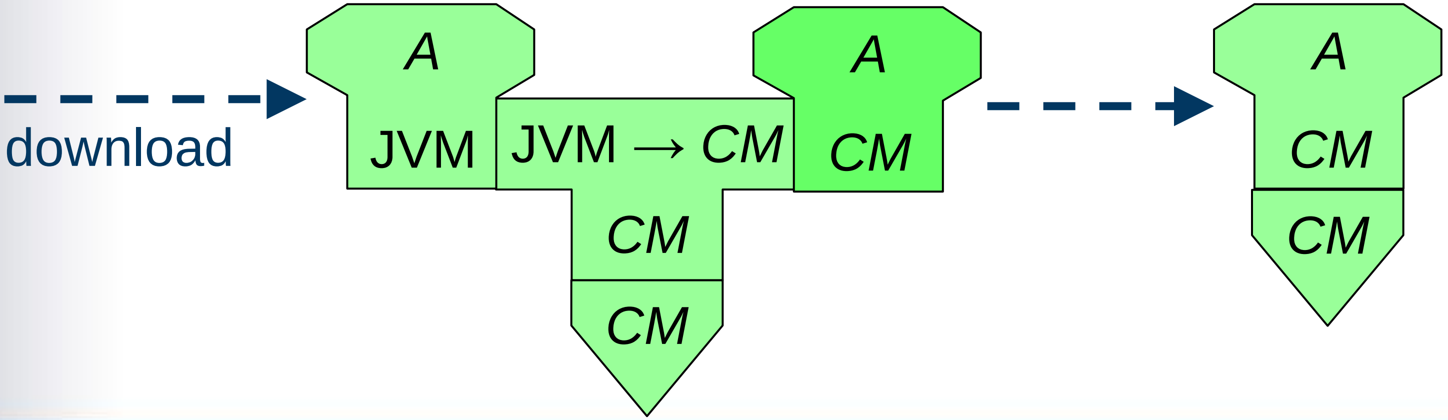
- A **just-in-time** (**JIT**) **compiler** translates virtual machine code to native machine code *just prior to execution*.
- This enables applets to be stored on a server in a portable form, but run at full speed on client machines.

Example: a Java JIT compiler

- A Java JIT compiler translates JVM code to client machine code:



- A JVM applet A is downloaded on demand from the server to a client machine *CM*, compiled to *CM* machine code, and then immediately run:



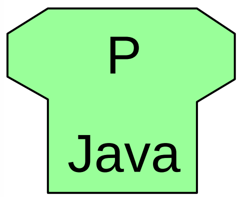
University of Glasgow

Selective JIT compilers

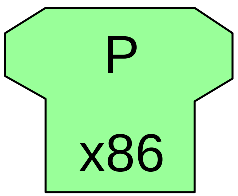
- More usually, a Java JIT compiler translates JVM code *selectively*:
  - The interpreter and JIT compiler work together.
  - The interpreter is instrumented to count method calls.
  - When the interpreter discovers that a method is “hot” (called frequently), it tells the JIT compiler to translate that particular method into native code.
- Selective Java JIT compilers are integrated into web browsers.

Portable compilers

- A program is **portable** if it can be made to run on different machines with minimal change:



is portable

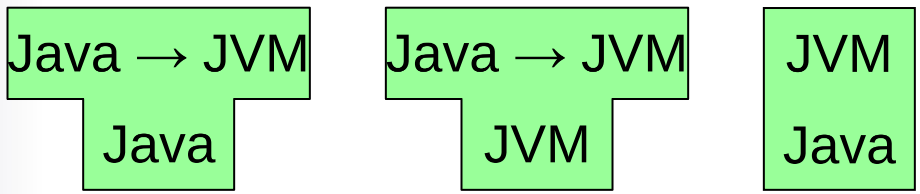


is not

- A compiler that generates native machine code is unportable in a special sense. If it must be changed to target a different machine, its code generator (≈ half the compiler) must be replaced.
- However, a compiler that generates suitable virtual machine code can be portable.

Example: portable compiler kit *(1)*

- A portable compiler kit for Java:

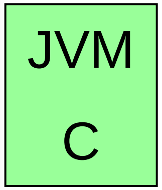


- Let’s install this kit on machine *M*.
- We face a chicken-and-egg situation:
  - We can’t run the JVM interpreter until we have a running Java compiler.
  - We can’t run the Java compiler until we have a running JVM interpreter.

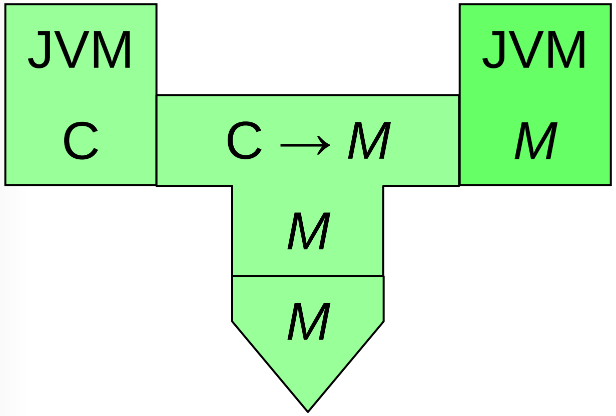
Example: portable compiler kit *(2)*

- To progress, first rewrite the JVM interpreter in (say) C:

~ 1 week's work

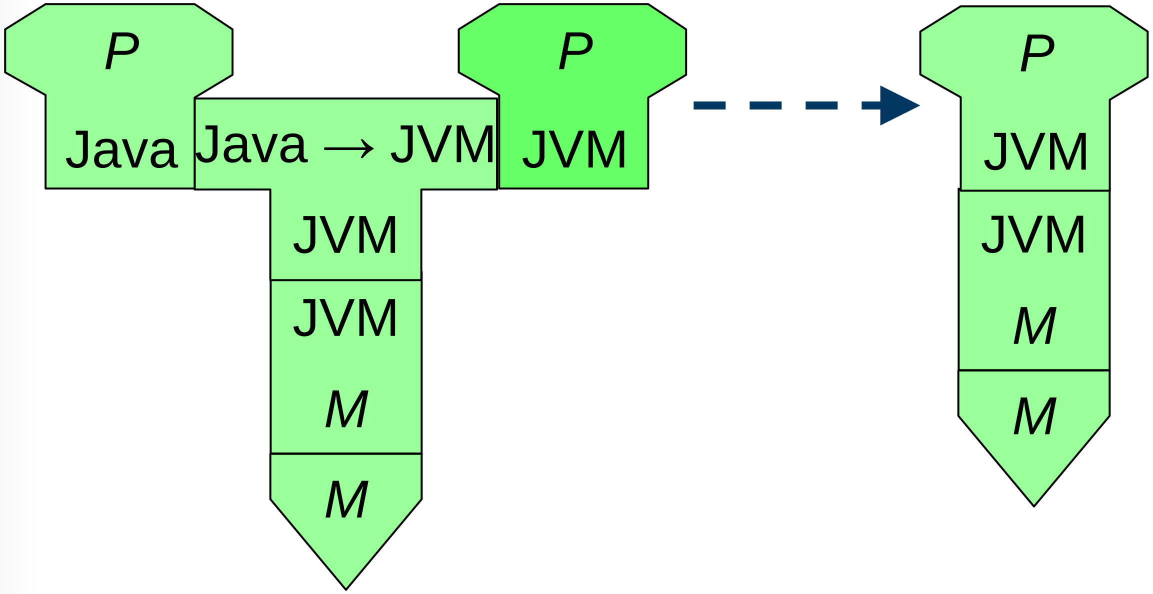


- Then compile the JVM interpreter on *M*:



Example: portable compiler kit *(3)*

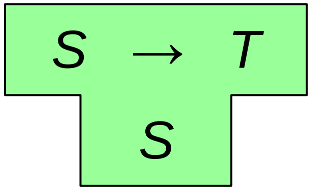
- Now we have an interpretive compiler, similar to the one we met before, except that the compiler itself must be interpreted:



- This compiler is very slow. However, it can be improved by bootstrapping.

Bootstrapping

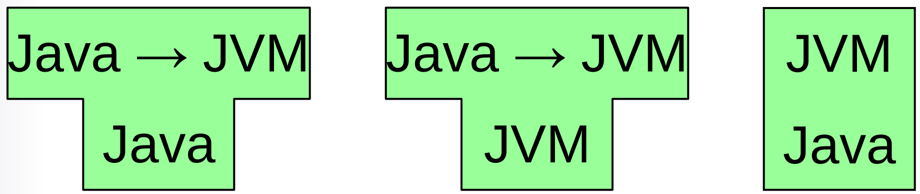
- Consider an $S \rightarrow T$ translator expressed in its own source language S:



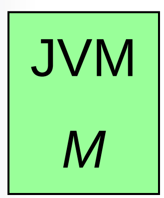
- Such a translator can be used to translate itself! This is called **bootstrapping**.
- Bootstrapping is a useful tool for improving an existing compiler:
  - making it compile faster
  - making it generate faster object code.
- In particular, we can bootstrap a portable compiler to make a true compiler, by translating virtual machine code to native machine code.

Example: bootstrapping *(1)*

- Take the Java portable compiler kit:



and the interpreter we generated from it:

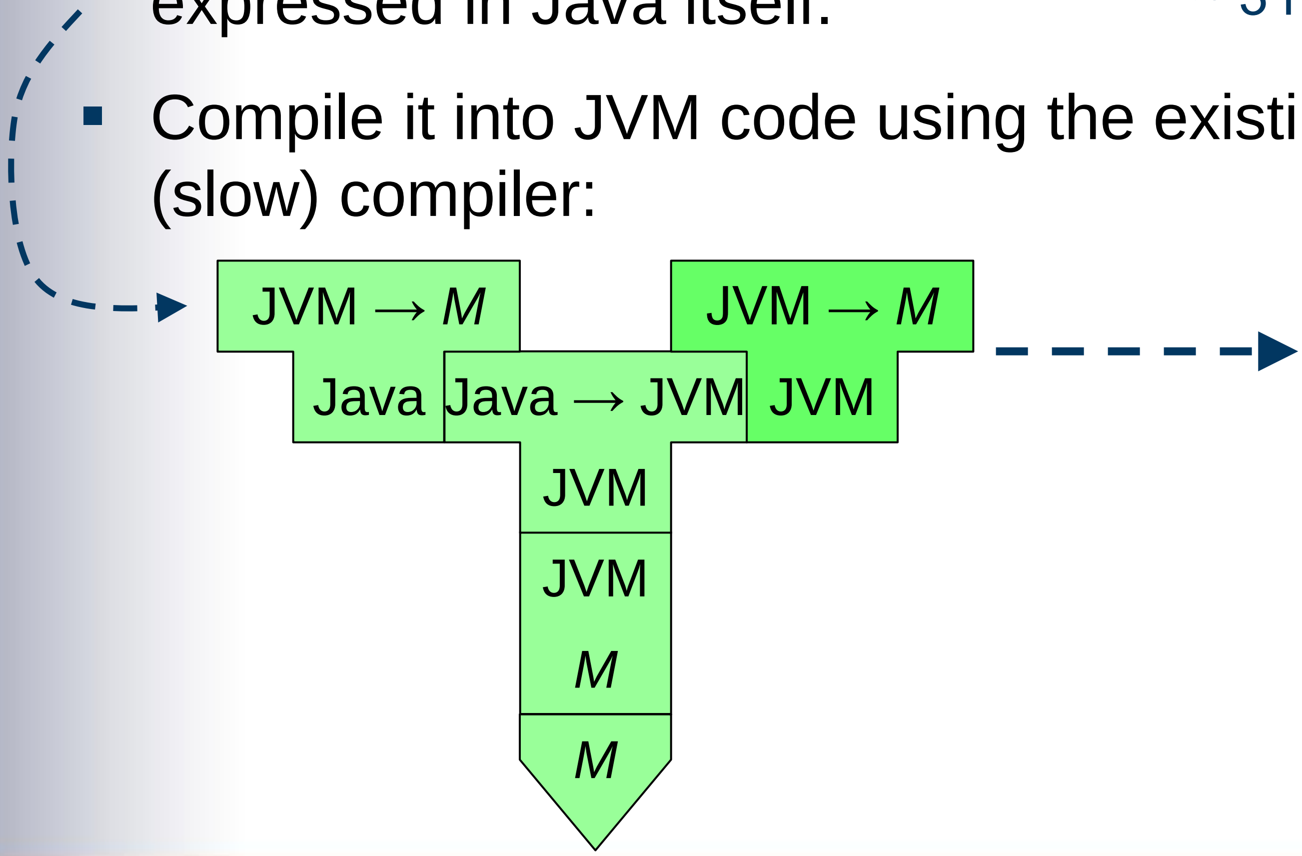


Example: bootstrapping *(2)*

- Write a JVM → *M* translator, expressed in Java itself.

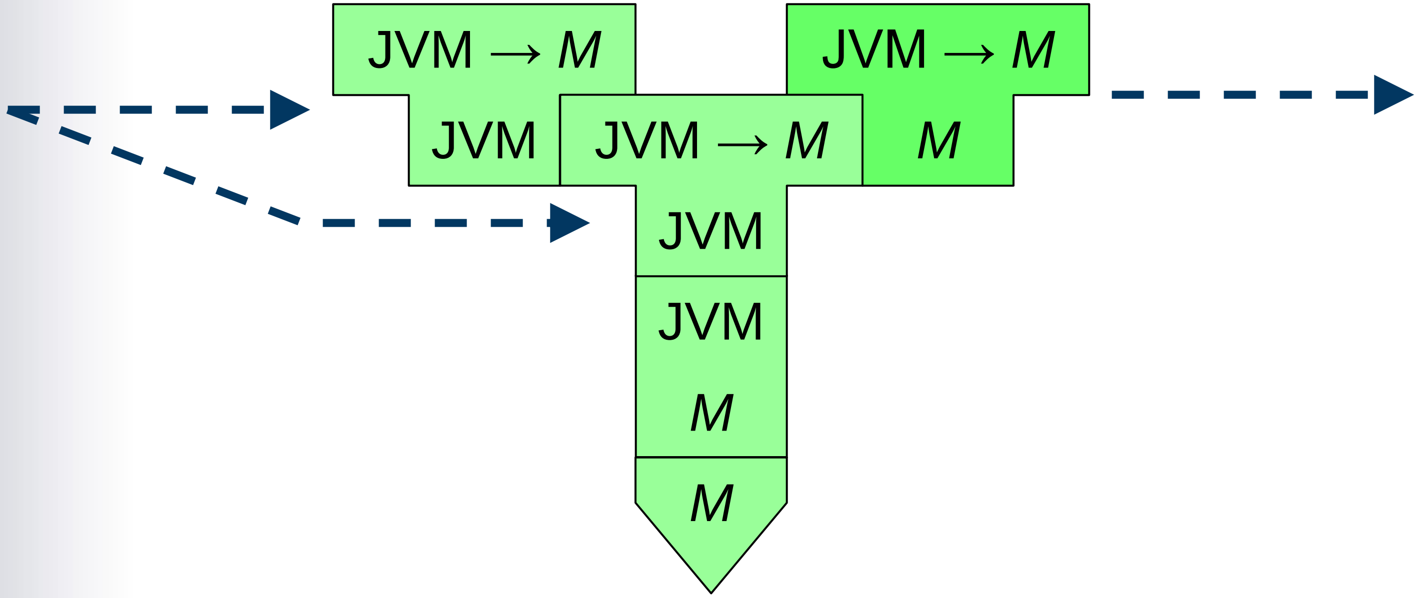
~ 3 months' work

- Compile it into JVM code using the existing (slow) compiler:



Example: bootstrapping *(3)*

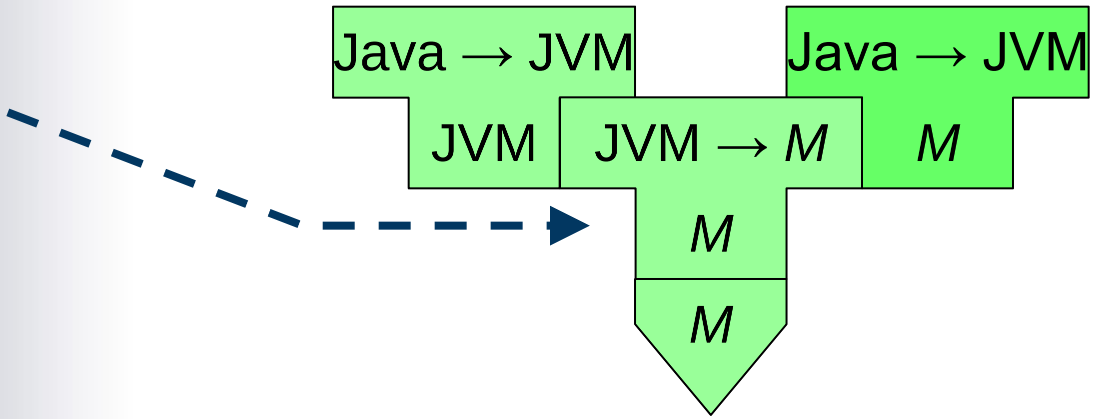
- Use this JVM → *M* translator to translate itself:



- This is the actual bootstrap. It generates a JVM → *M* translator, expressed in *M* machine code.

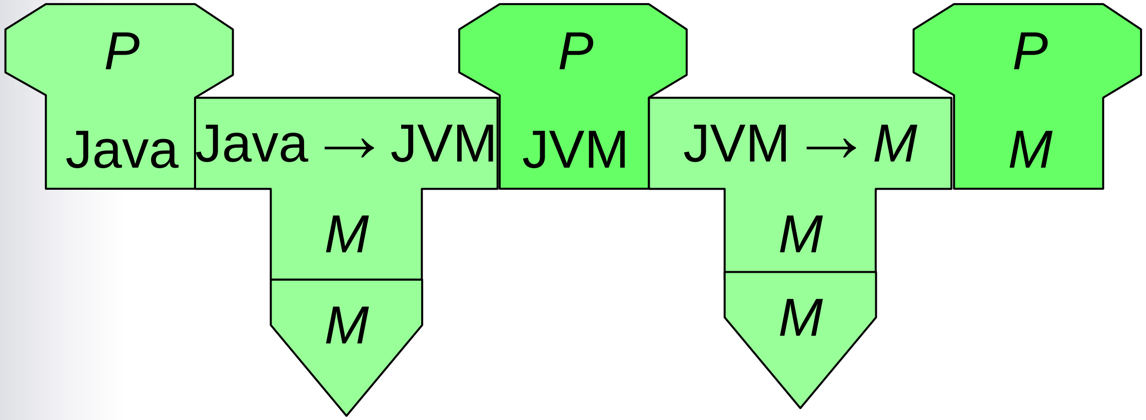
Example: bootstrapping *(4)*

- Finally, translate the Java → JVM compiler into *M* machine code:



Example: bootstrapping *(5)*

- Now we have a 2-stage Java → *M* compiler:



- This Java compiler is improved in two respects:
  - it compiles faster (being expressed in native machine code)
  - it generates faster object code (native machine code).