Languages and Compilers
(SProg og Oversættere)

Lecture 2
Tombstone Diagrams

Bent Thomsen

Department of Computer Science

Aalborg University

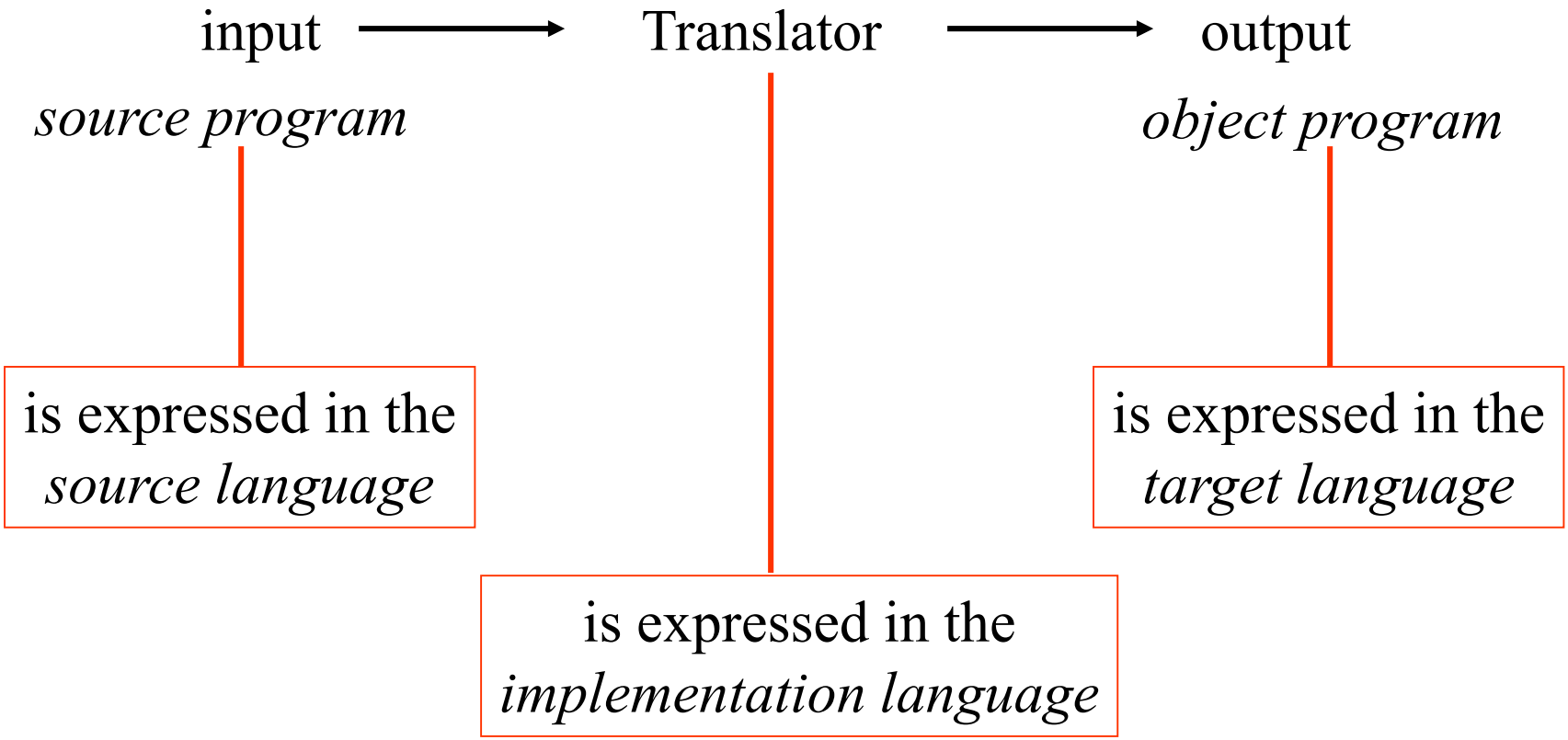
With acknowledgement to Norm Hutchinson whose slides this lecture is based on.

Learning goals

- Knowledge of compilers and interpreters as programs
- Knowledge of tombstone diagrams
- Introduction to Cross compilation
- Introduction to Two stage compiling
- Reasoning about Portability
- Introduction to bootstrapping

Terminology

Q: Which programming languages play a role in this picture?



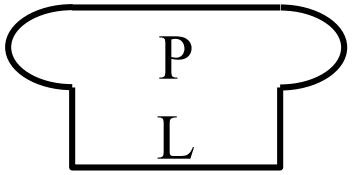
A: All of them!

Tombstone Diagrams

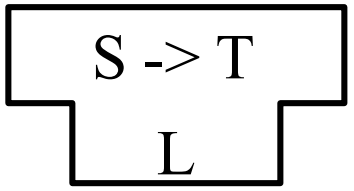
What are they?

- diagrams consisting out of a set of "puzzle pieces" we can use to reason about language processors and programs
- different kinds of pieces
- combination rules (not all diagrams are "well formed")

Program P implemented in L

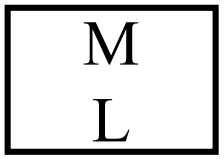


Translator implemented in L

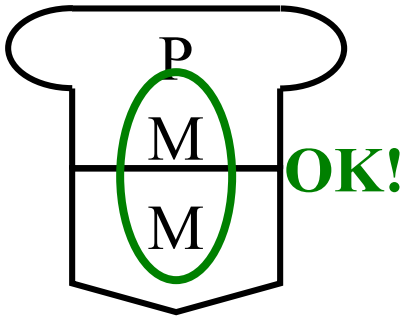


Machine implemented in hardware

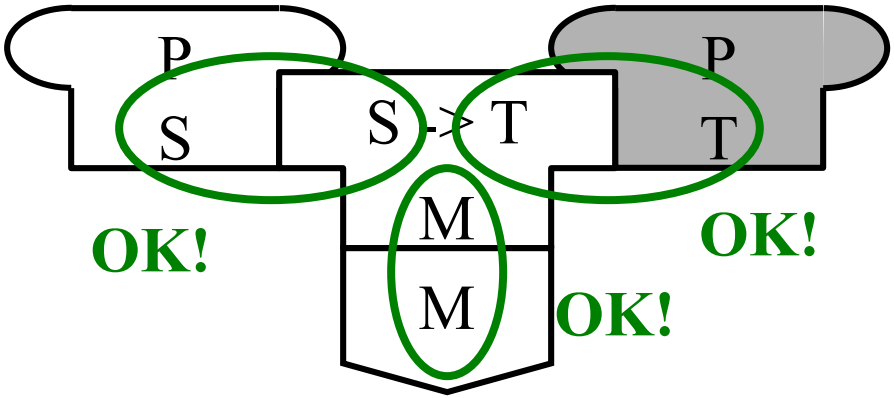
Language interpreter in L



Tombstone diagrams: Combination rules



OK!

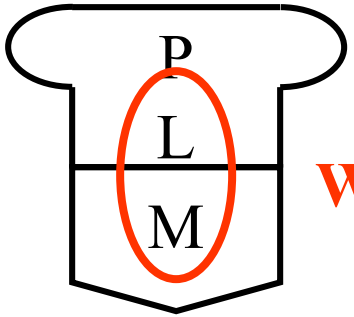


OK!

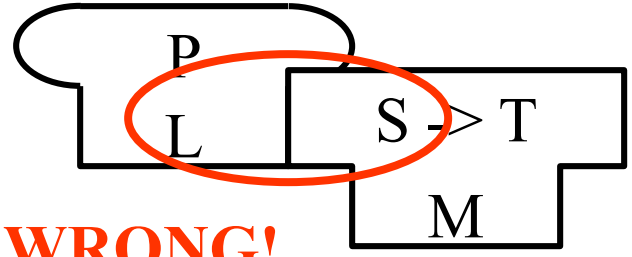
OK!

OK!

OK!



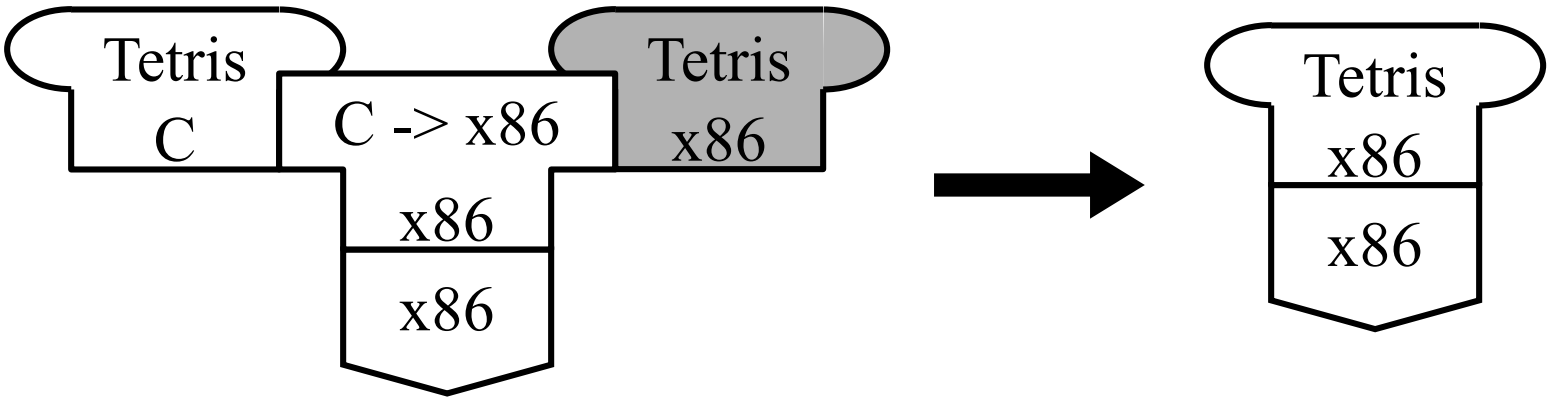
WRONG!



WRONG!

Compilation

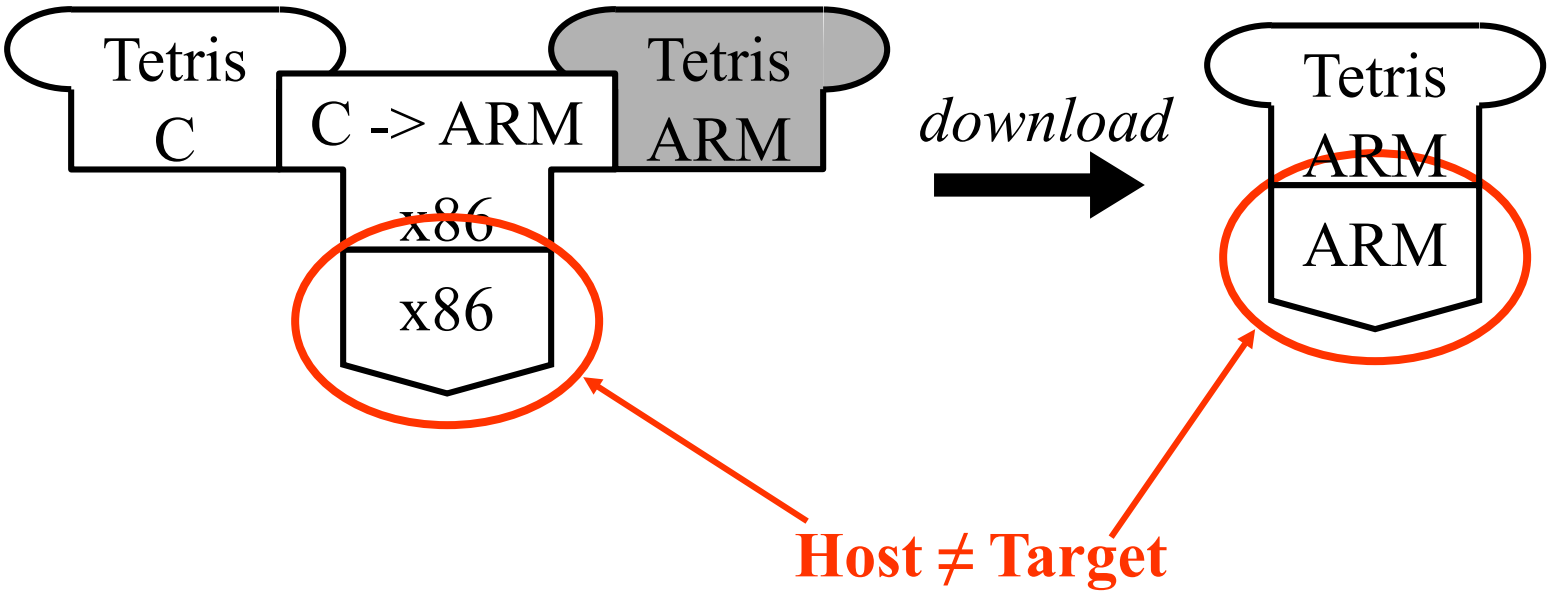
Example: Compilation of C programs on an x86 machine



Cross compilation

Example: A C "cross compiler" from x86 to ARM

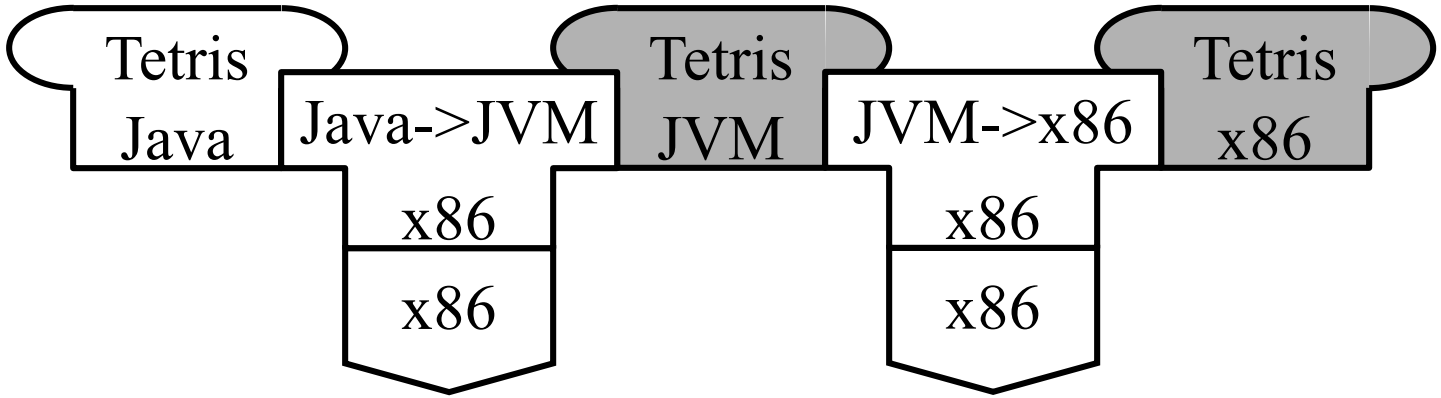
A *cross compiler* is a compiler which runs on one machine (the *host machine*) but emits code for another machine (the *target machine*).



Q: Are cross compilers useful? Why would/could we use them?

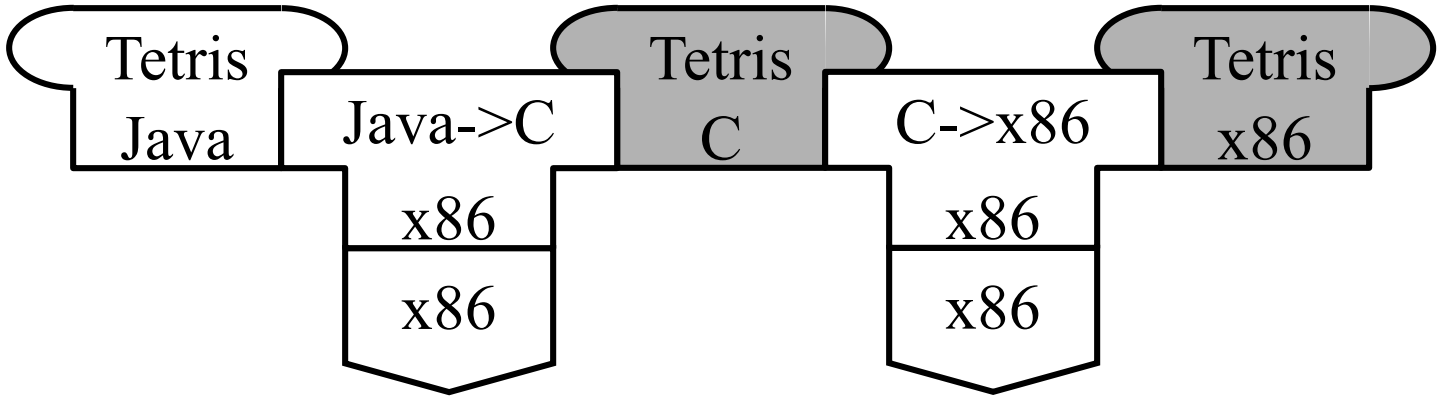
Two Stage Compilation

A *two-stage translator* is a composition of two translators. The output of the first translator is provided as input to the second translator.



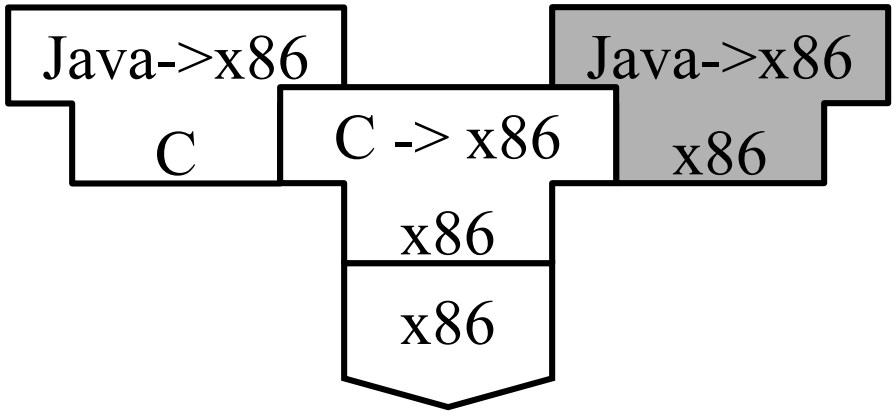
Two Stage Compilation (via C)

A *two-stage translator* is a composition of two translators. The output of the first translator is provided as input to the second translator.



Compiling a Compiler

Observation: A compiler is a program!
Therefore it can be provided as input to a language processor.
Example: compiling a compiler.

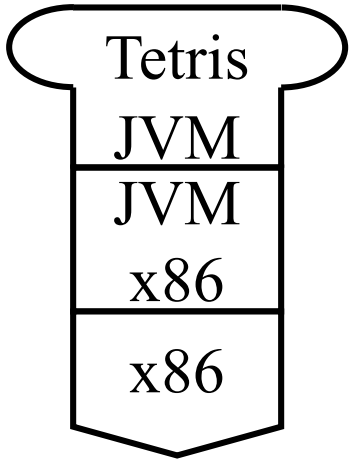


Interpreters

An *interpreter* is a language processor implemented in software, i.e. as a program.

Terminology: *abstract (or virtual) machine* versus *real machine*

Example: The Java Virtual Machine

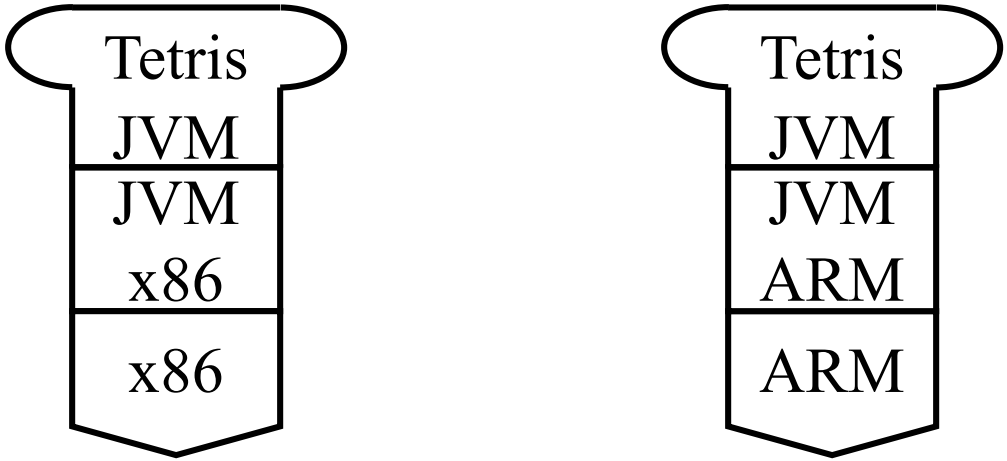


Q: Why are abstract machines useful?

Interpreters

Q: Why are abstract machines useful?

1) Abstract machines provide better platform independence

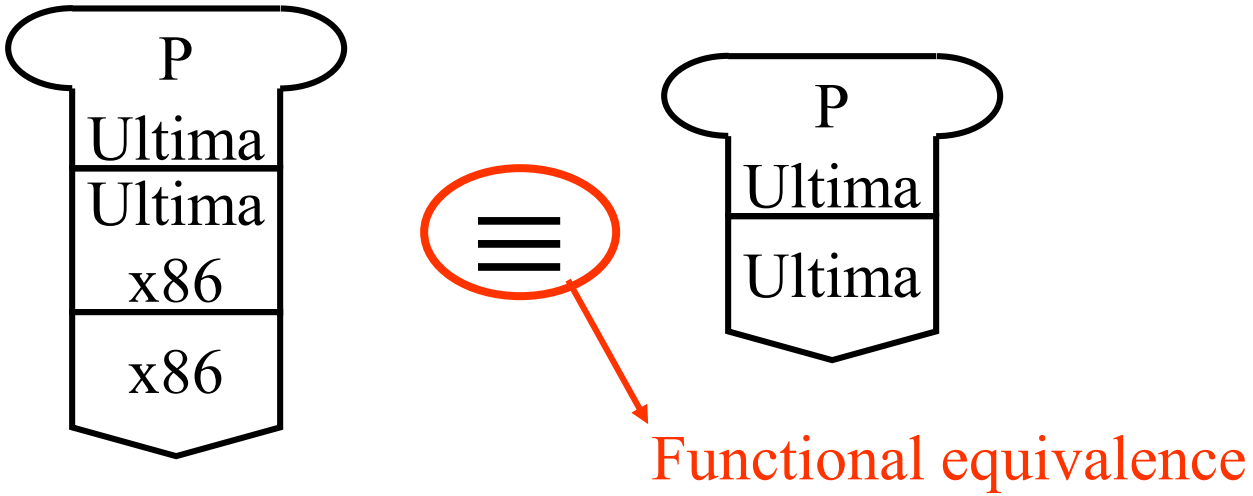


Interpreters

Q: Why are abstract machines useful?

2) Abstract machines are useful for testing and debugging.

Example: Testing the "Ultima" processor using *hardware emulation*



Note: we don't have to implement Ultima emulator in x86 we can use a high-level language and compile it.

Interpreters versus Compilers

Q: What are the tradeoffs between compilation and interpretation?

Compilers typically offer more advantages when

- programs are deployed in a production setting
- programs are "repetitive"
- the instructions of the programming language are complex

Interpreters typically are a better choice when

- we are in a development/testing/debugging stage
- programs are run once and then discarded
- the instructions of the language are simple
- the execution speed is overshadowed by other factors
  - e.g. on a web server where communications costs are much higher than execution speed

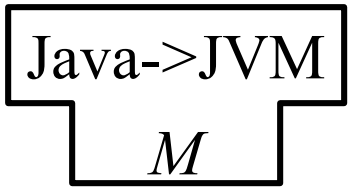
Interpretive Compilers

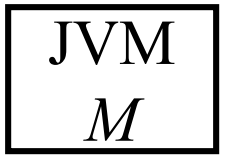
Why?
A tradeoff between fast(er) compilation and a reasonable runtime performance.

How?
Use an "intermediate language"

- more high-level than machine code => easier to compile to
- more low-level than source language => easy to implement as an interpreter

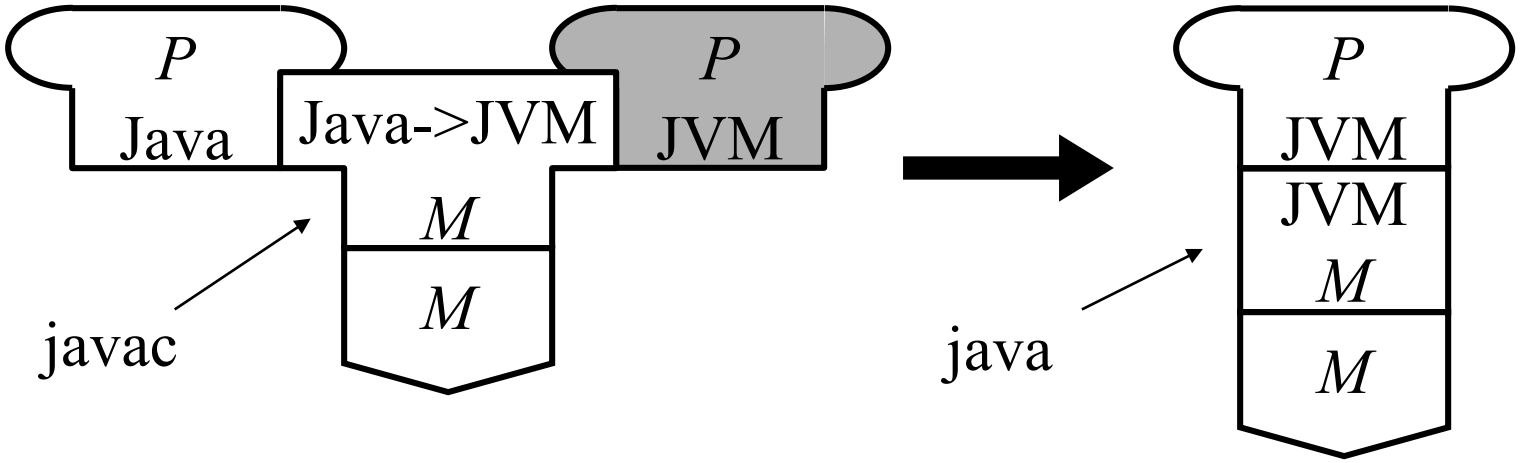
Example: A "Java Development Kit" for machine *M*





Interpretive Compilers

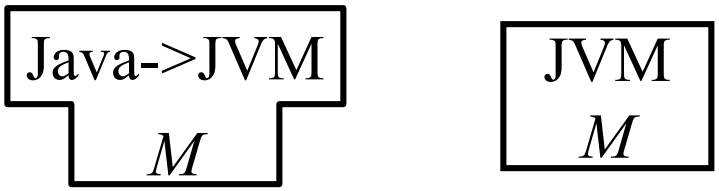
Example: Here is how we use our "Java Development Kit" to run a Java program *P*



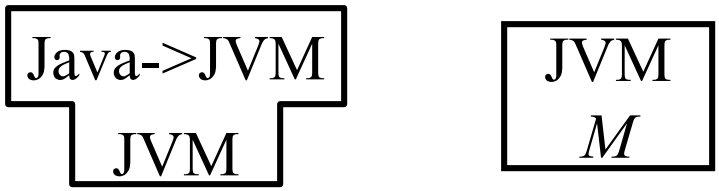
Portable Compilers

Example: Two different "Java Development Kits"

Kit 1:



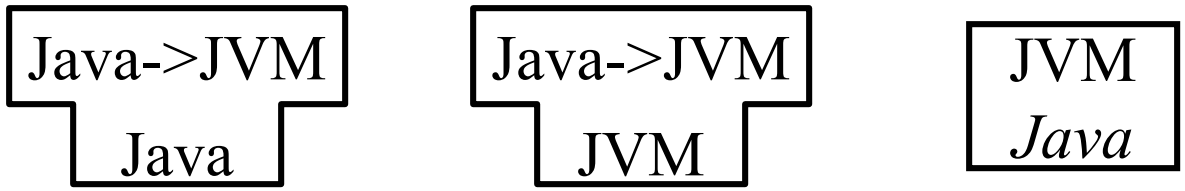
Kit 2:



Q: Which one is "more portable"?

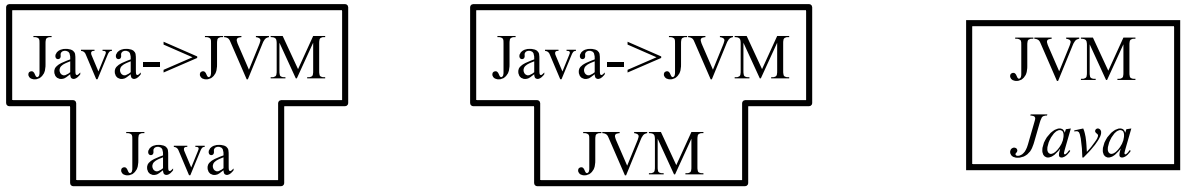
Example: a “portable” compiler kit

Portable Compiler Kit:

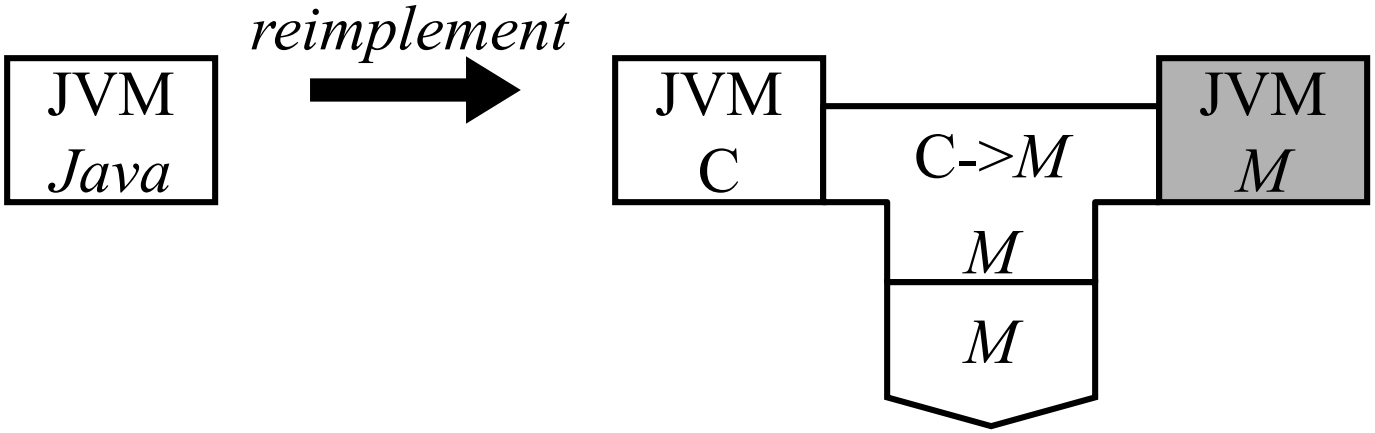


Q: Suppose we want to run this kit on some machine *M*. How could we go about realizing that goal? (with the least amount of effort)

Example: a "portable" compiler kit

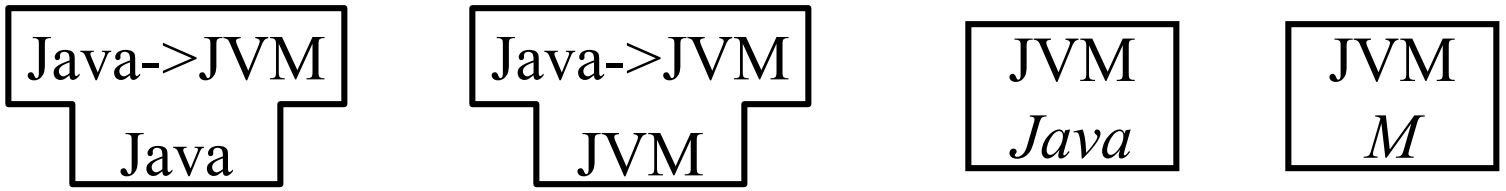


Q: Suppose we want to run this kit on some machine *M*. How could we go about realizing that goal? (with the least amount of effort)

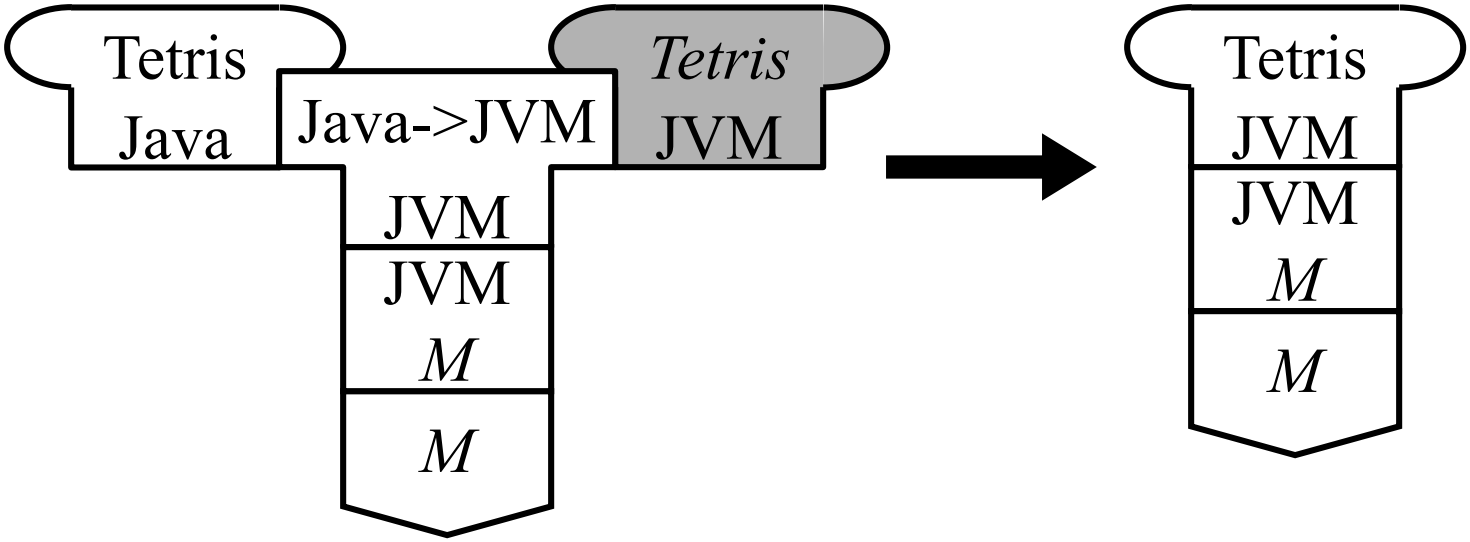


Example: a “portable” compiler kit

This is what we have now:

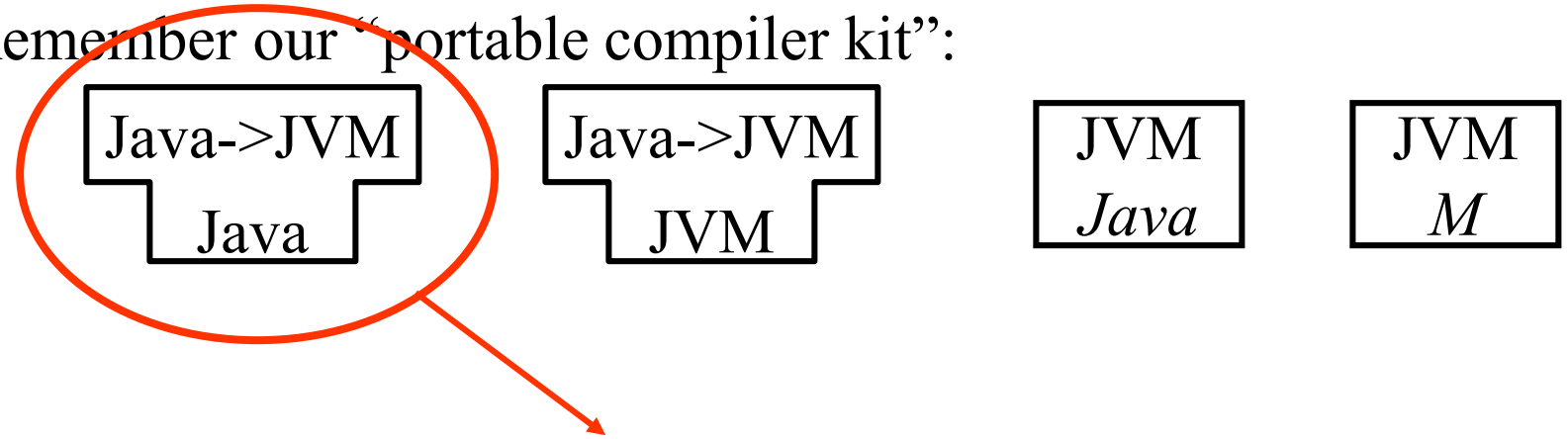


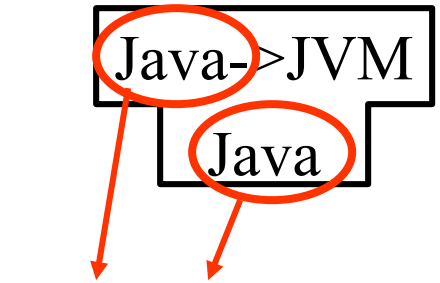
Now, how do we run our Tetris program?



Bootstrapping

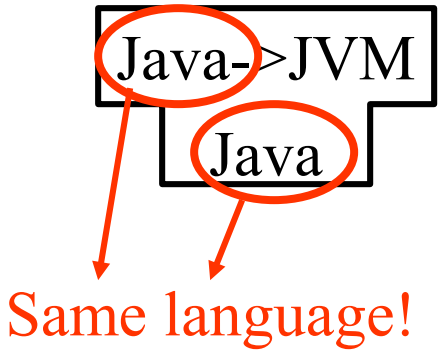
Remember our "portable compiler kit":





Q: What can we do with a compiler written in itself? Is that useful at all?

Bootstrapping



Q: What can we do with a compiler written in itself? Is that useful at all?

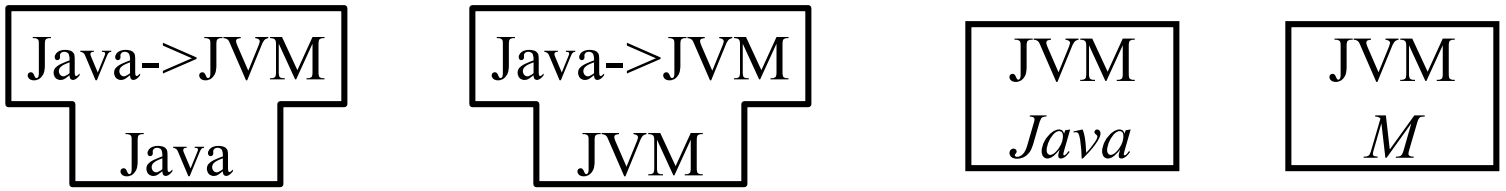
- By implementing the compiler in (a subset of) its own language, we become less dependent on the target platform => more portable implementation.
- But… "chicken and egg problem"? How do to get around that?

=> BOOTSTRAPPING: requires some work to make the first "egg".

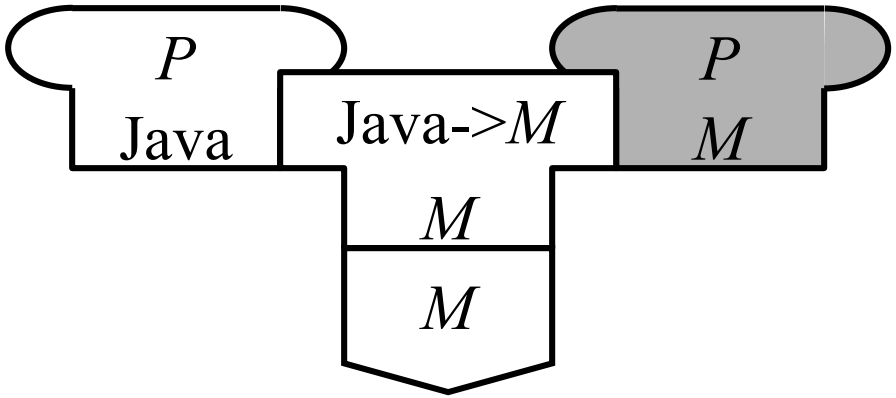
There are many possible variations on how to bootstrap a compiler written in its own language.

Bootstrapping an Interpretive Compiler to Generate *M* code

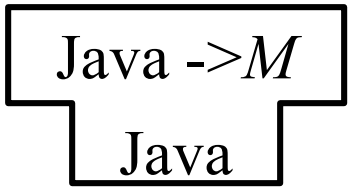
Our "portable compiler kit":

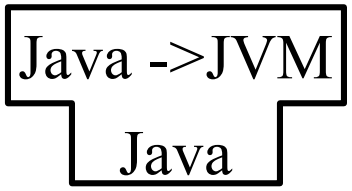


Goal: we want to get a "completely native" Java compiler on machine *M*

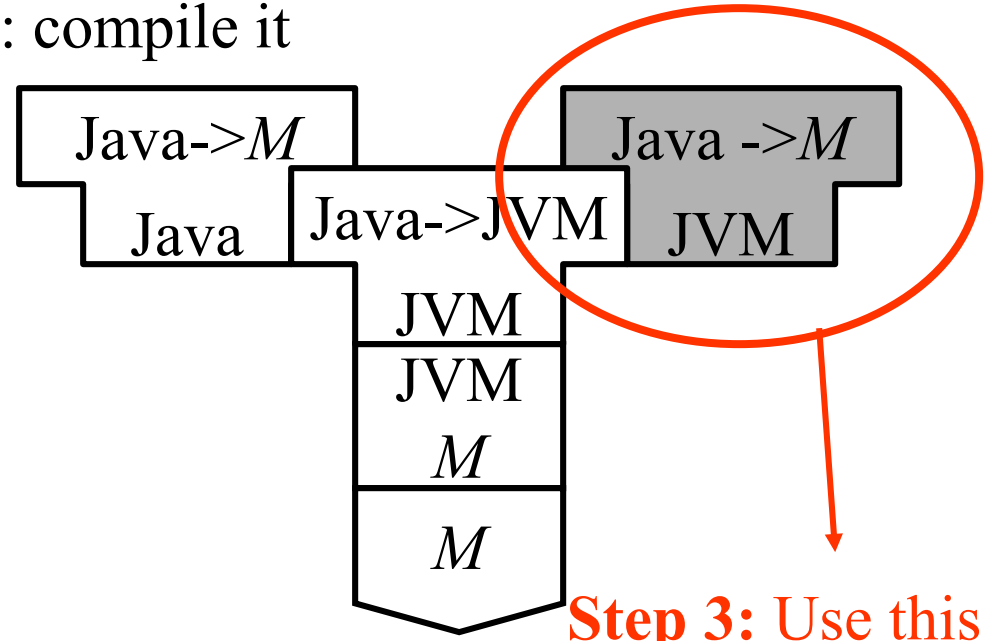


Bootstrapping an Interpretive Compiler to Generate *M* code (first approach)

Step 1: implement 

by rewriting 

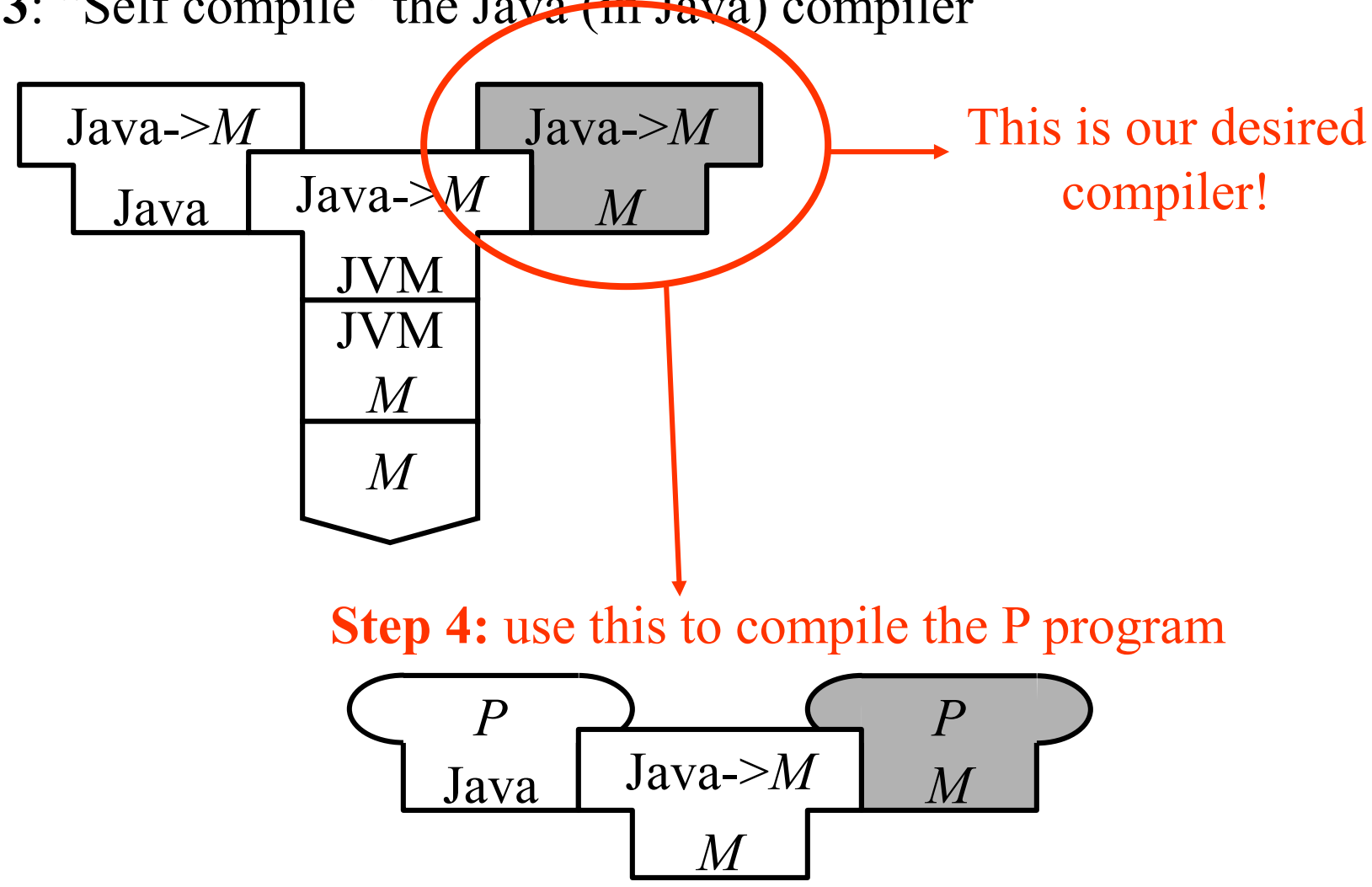
Step 2: compile it



Step 3: Use this to compile again

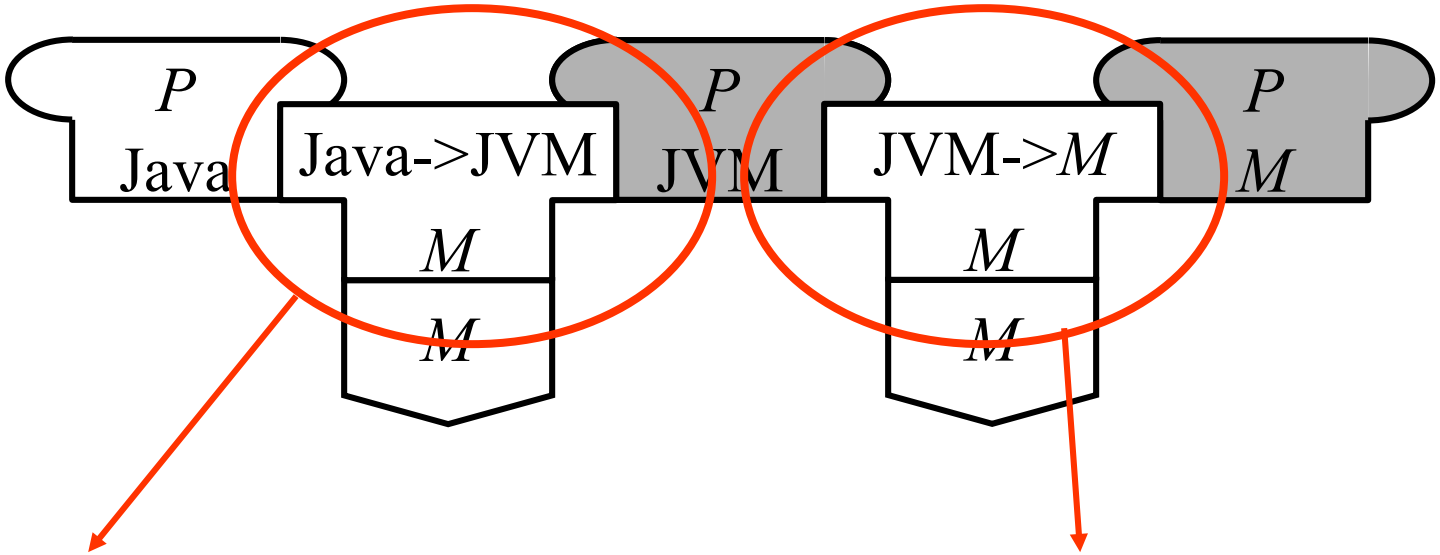
Bootstrapping an Interpretive Compiler to Generate *M* code (first approach)

Step 3: "Self compile" the Java (in Java) compiler

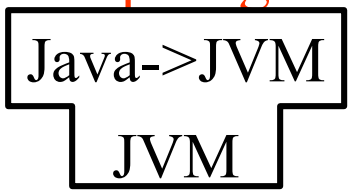


Bootstrapping an Interpretive Compiler to Generate *M* code (second approach)

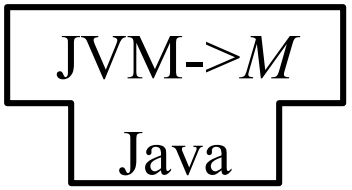
Idea: we will build a two-stage Java -> *M* compiler.



We will make this by compiling



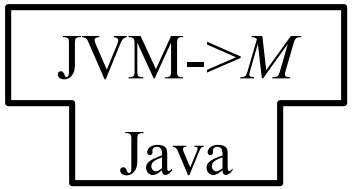
To get this we implement



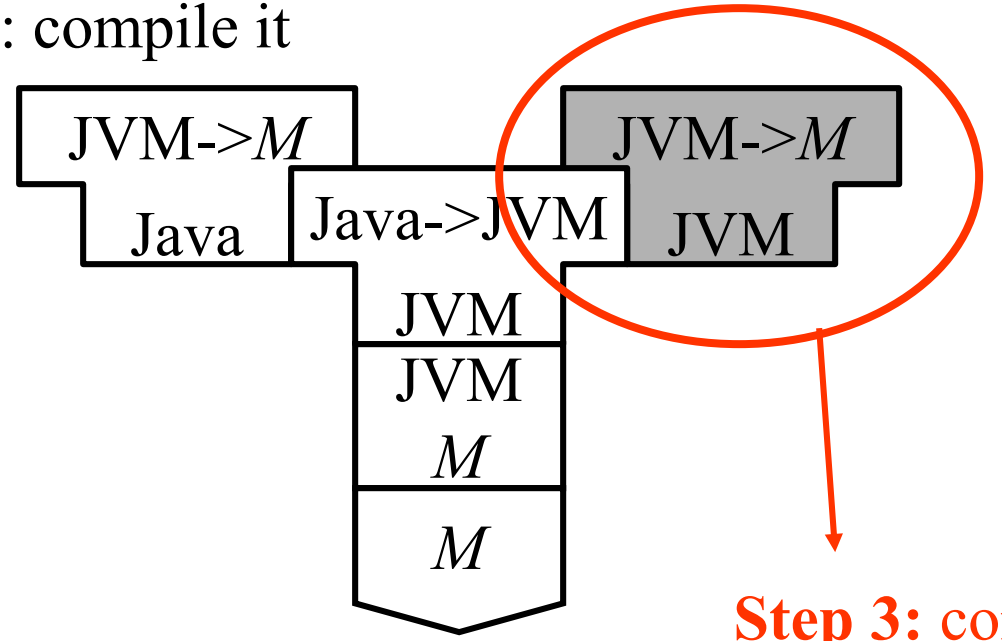
and compile it

Bootstrapping an Interpretive Compiler to Generate *M* code (second approach)

Step 1: implement



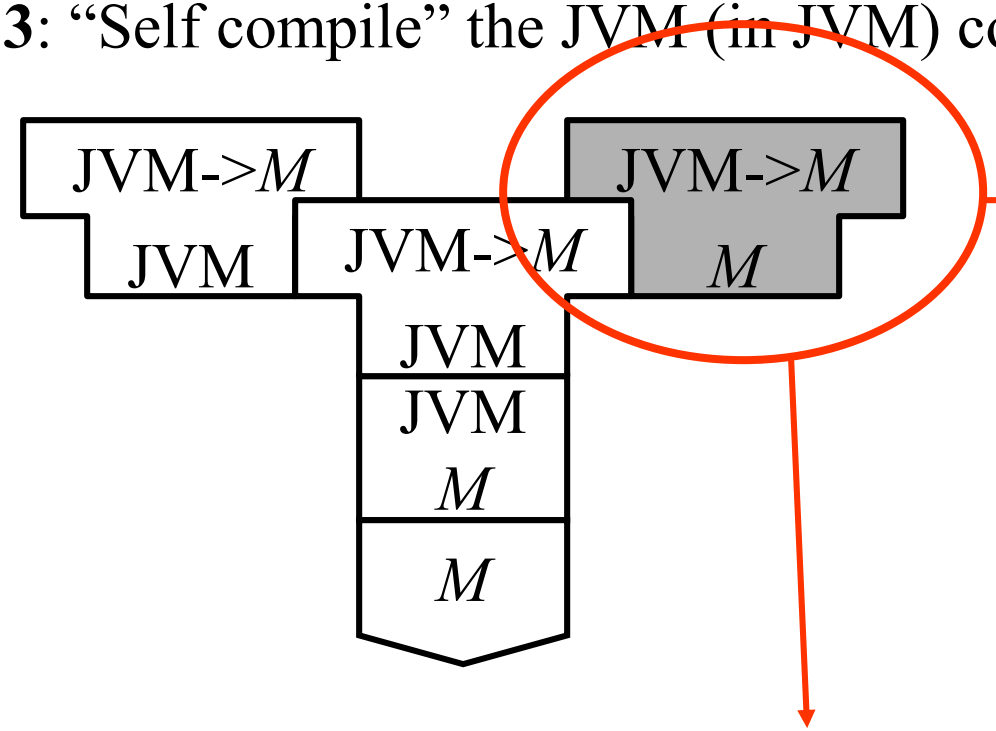
Step 2: compile it



Step 3: compile this

Bootstrapping an Interpretive Compiler to Generate *M* code (second approach)

Step 3: "Self compile" the JVM (in JVM) compiler

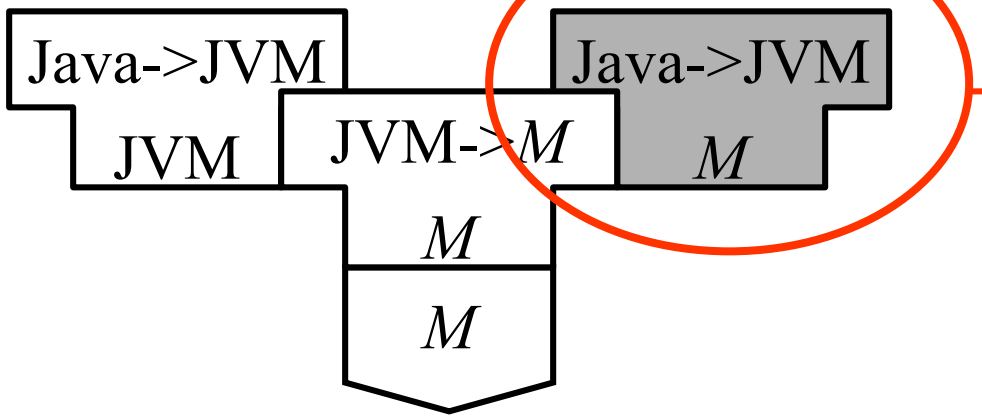


This is the second stage of our compiler!

Step 4: use this to compile the Java compiler

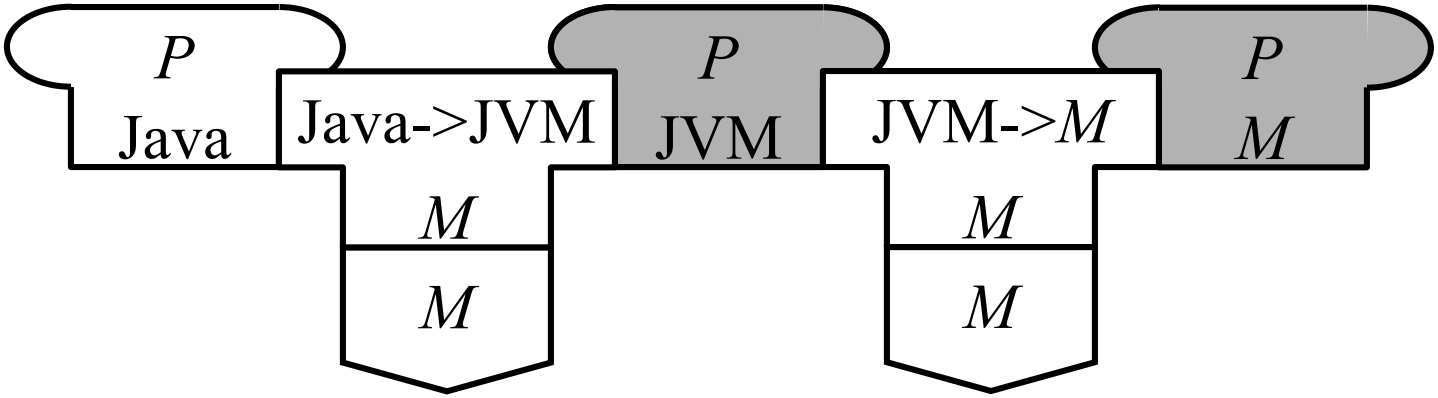
Bootstrapping an Interpretive Compiler to Generate *M* code

Step 4: Compile the Java->JVM compiler into machine code



The first stage of our compiler!

We are DONE!



Bootstrapping to Improve Efficiency

The efficiency of programs and compilers:

Efficiency of programs:

- memory usage
- runtime

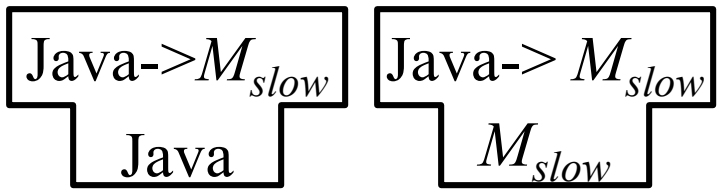
Efficiency of compilers:

- Efficiency of the compiler itself
- Efficiency of the emitted code

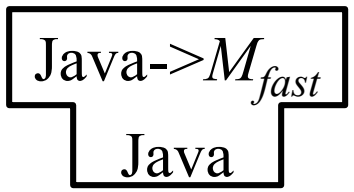
Idea: We start from a simple compiler (generating inefficient code) and develop more sophisticated versions of it. We can then use bootstrapping to improve performance of the compiler.

Bootstrapping to Improve Efficiency

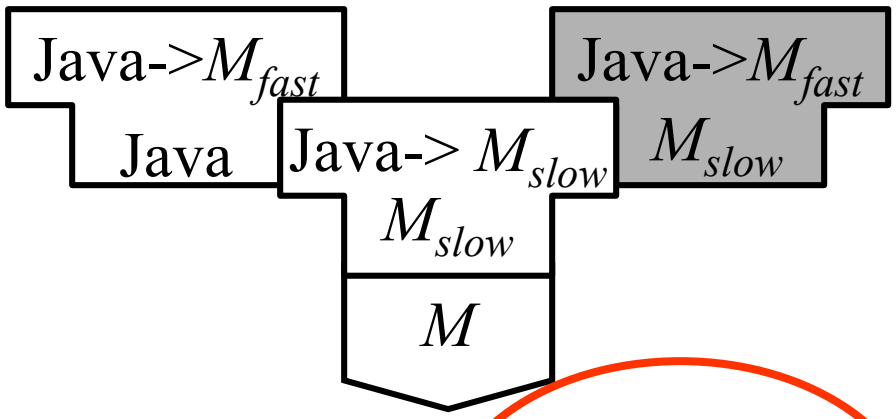
We have:



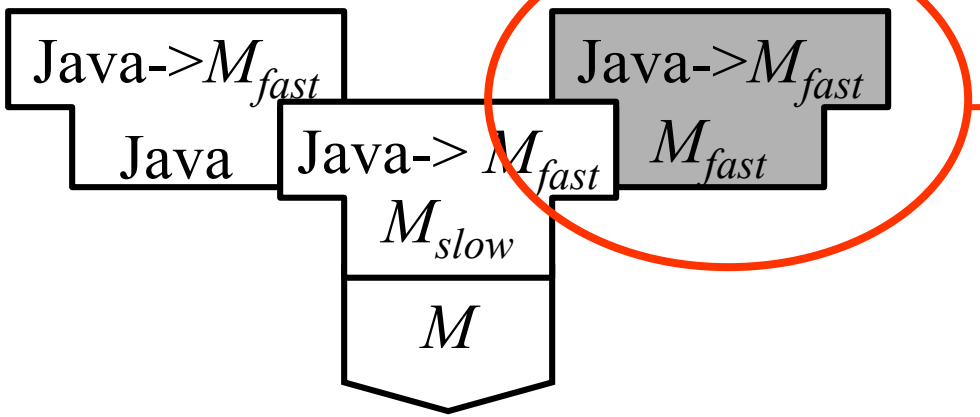
We implement:



Step 1



Step 2



Fast compiler that emits fast code!

Conclusion

- To write a good compiler you may be writing several simpler ones first
- You have to think about the source language, the target language and the implementation language.
- Strategies for implementing a compiler
  1. Write it in machine code
  2. Write it in a lower level language and compile it using an existing compiler
  3. Write it in the same language that it compiles and bootstrap
- The work of a compiler writer is never finished, there is always version 1.x and version 2.0 and …

AtoCC Demo