



# Unit – 1

# BOOTSTRAPPING

Mrs. RUCHI SHARMA

[ruchi.sharma@bkbiet.ac.in](mailto:ruchi.sharma@bkbiet.ac.in)

# Language Translation



- A *programming language processor* is any system that manipulates programs expressed in a PL
- A *source* program in some source language is *translated* into an *object* program in some *target* language
- Translators are *assemblers* or *compilers*
- An *assembler* translates from assembly language to machine language
- A *compiler* translates from a high-level language into a low-level language the compiler is written in its *implementation* language
- An *interpreter* is a program that accepts a source program and runs it immediately
- An *interpretive compiler* translates a source program into an intermediate language, and the resulting object program is then executed by an interpreter

# Example of Language Translators

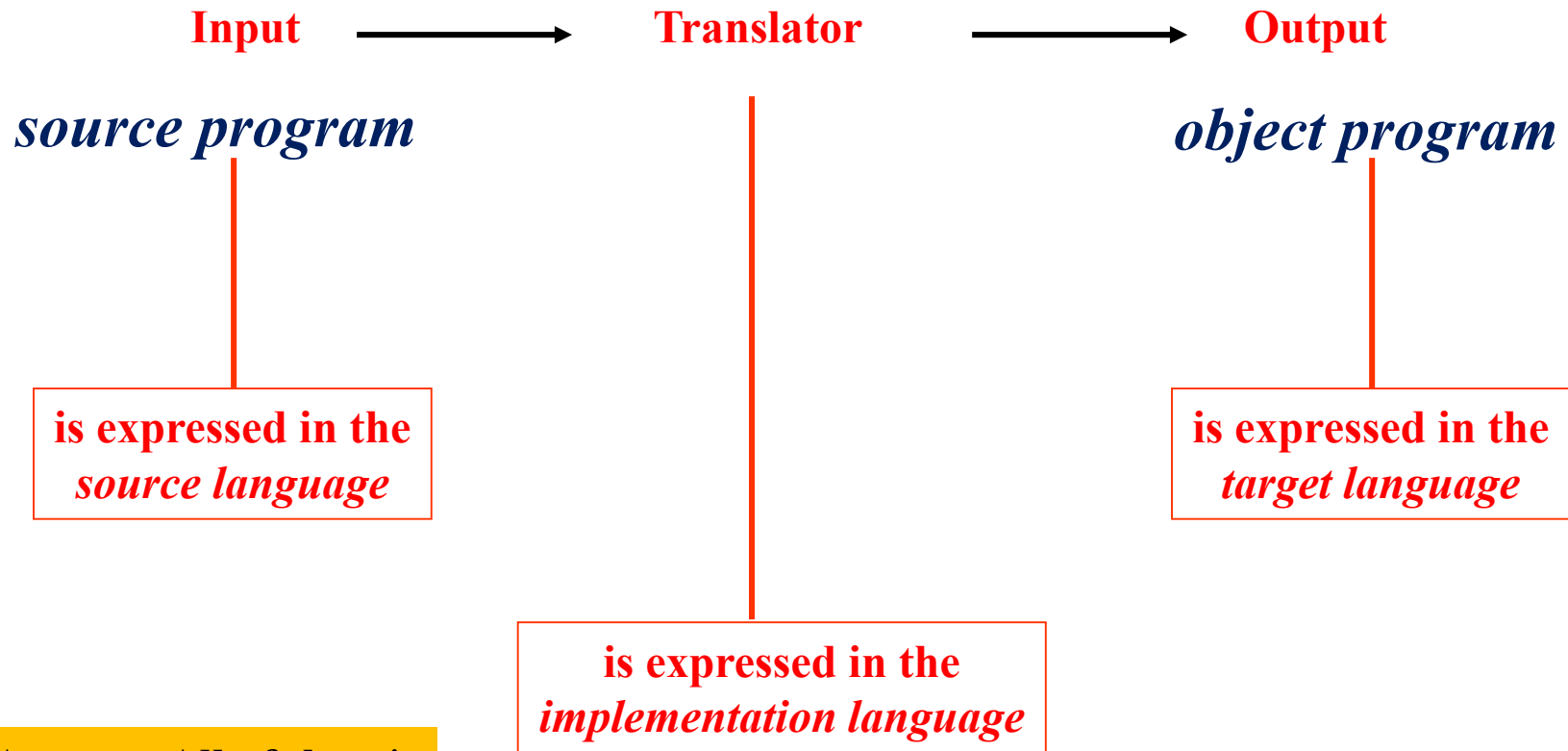


- Compilers for Fortran, COBOL, C, C++
- Interpretive compilers for Pascal (P-Code), Prolog (Warren Abstract Machine) and Java (Java Virtual Machine)
- Interpreters for APL, Scheme, Haskell, Python, and (early) LISP



# Terminology

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



**Answer: All of them!**

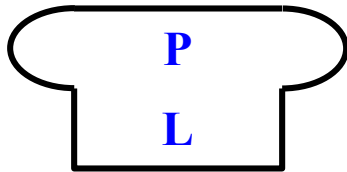
# Tombstone Diagrams(T-Diagram)



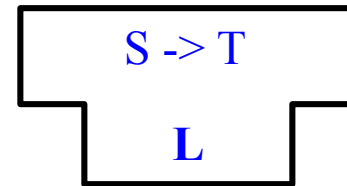
## What are they?

- diagrams consisting out of a set of “puzzle pieces” we can use to reason about language processors and programs
- different kinds of pieces
  - the base of the piece always contains the implementation language
- **Combination rules** (not all diagrams are “well formed”)

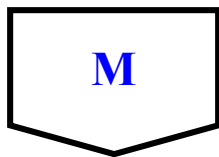
### Program P implemented in L



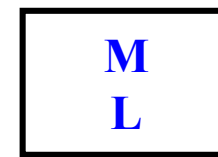
### Translator implemented in L



### Machine implemented in hardware

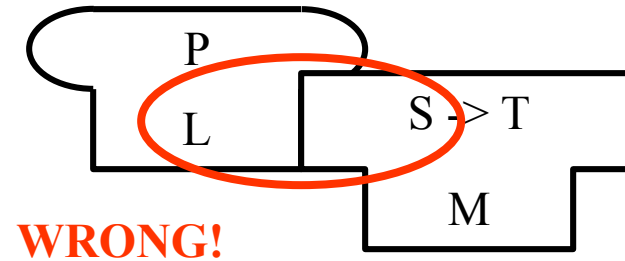
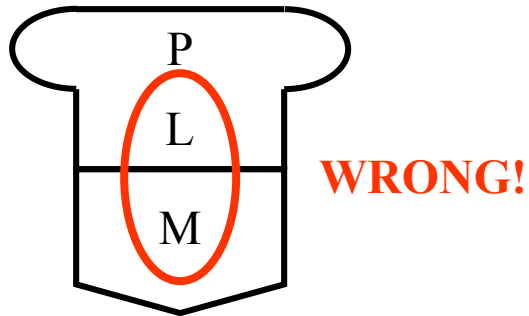
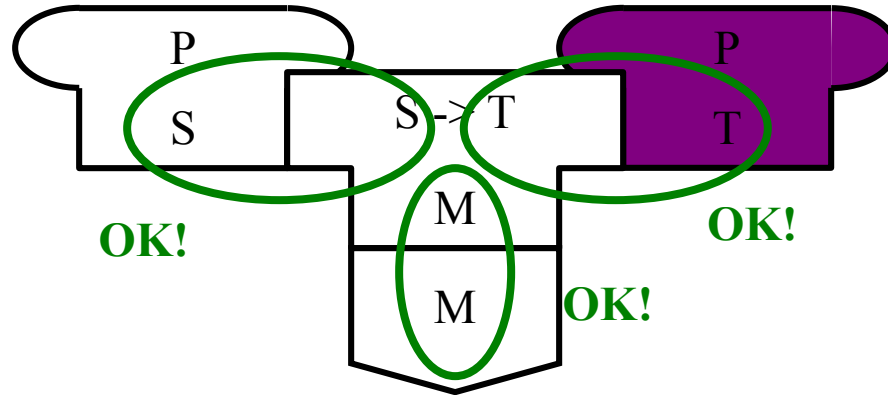
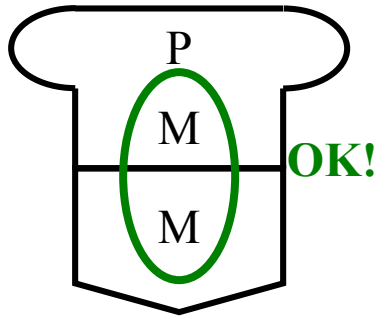


### Language interpreter in L





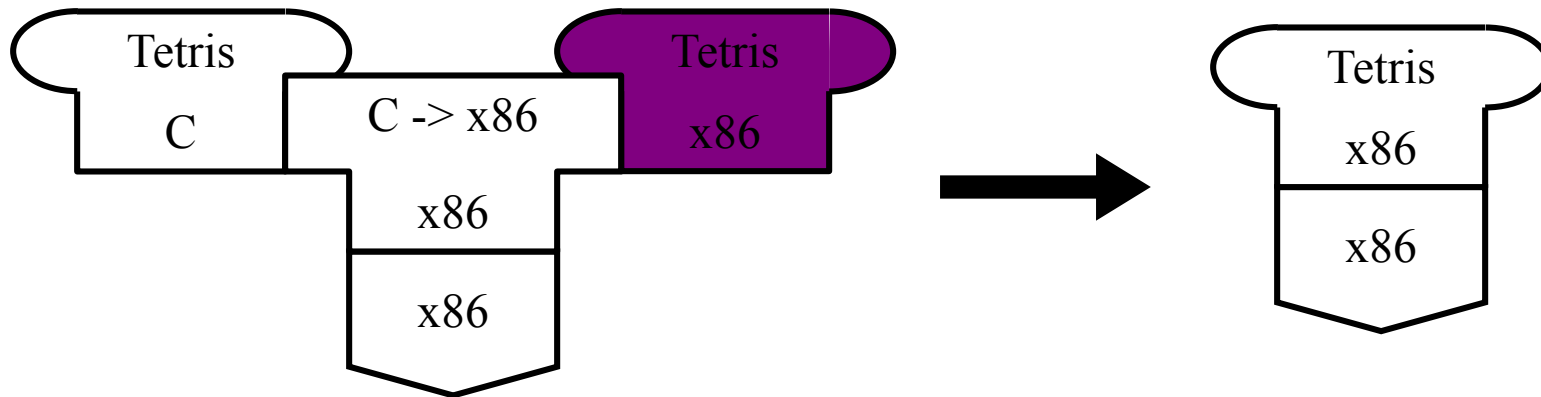
# Tombstone diagrams: Combination rules





# Compilation

**Example:** Compilation of C programs on an x86 machine



# Incremental Compiler



**Incremental Compiler** is a compiler that generates code for a statement, or group of statements, which is independent of the code generated for other statements.

- **Need of Incremental Compiler :**  
Much of a programmer's time is spent in an edit-compile-debug workflow as following.
- you make a small change (often in a single module or even function).
- you let the compiler translate the code into a binary, and finally.
- you run the program or a bunch of unit tests in order to see results of the change.
- This will increase the time of compilation at every step of change. But this can be overcome with the concept of incremental compiler (save the previous compilation and compile the modified part of the code).



# How can this concept



- We have already heard that computing something incrementally means updating only those parts of the computation's output that need to be adapted in response to a given change in the computation's inputs.
- One basic strategy we can employ to achieve this is to view one big computation (like compiling a program) as a composite of many smaller, interrelated computations that build upon each other.
- Each of those smaller computations will yield an intermediate result that can be cached and hopefully re-used in a later iteration, sparing us the need to re-compute that particular intermediate result again.

**Code --> Compilation**

**New code --> New Compilation**

**(Previous Compilation + Code + Changes) → (Previous compilation of modified part)**

# Feature of Incremental compiler

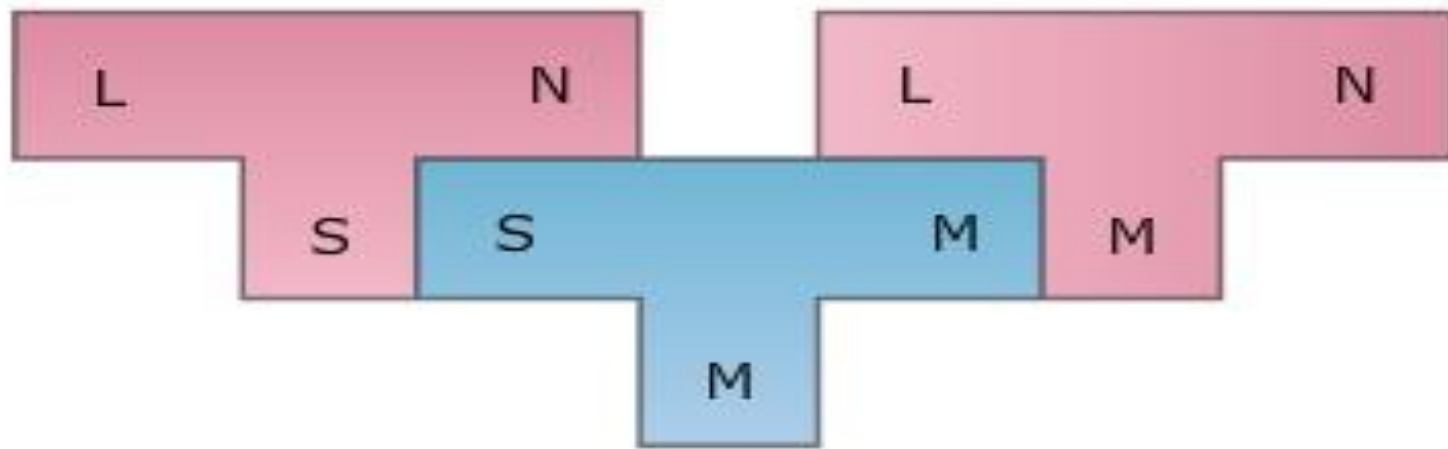


- During program development process modifications to Source program can cause recompilation of whole source text. This overhead is reduced in incremental compiler.
- Run time errors can be patched up just like compile time errors by keeping program modification as it is.
- The compilation process is faster.
- The memory used is less.
- Handling batch programs become very flexible using Incremental compiler.
- In incremental compiler program structure table is maintained for memory allocation of target code. When a new statement is compiled the new entry for it is created in program structure table. Thus memory allocation required for incremental compiler need not be contiguous.
- Helps in tracking the dependencies on source program.

# Cross Compiler



1. A compiler, that run on one machine and produce the target code for another machine. Such a compiler is called cross compiler
2. In image source language L, the target language N gets generated which runs on machine M



**Cross Compiler**

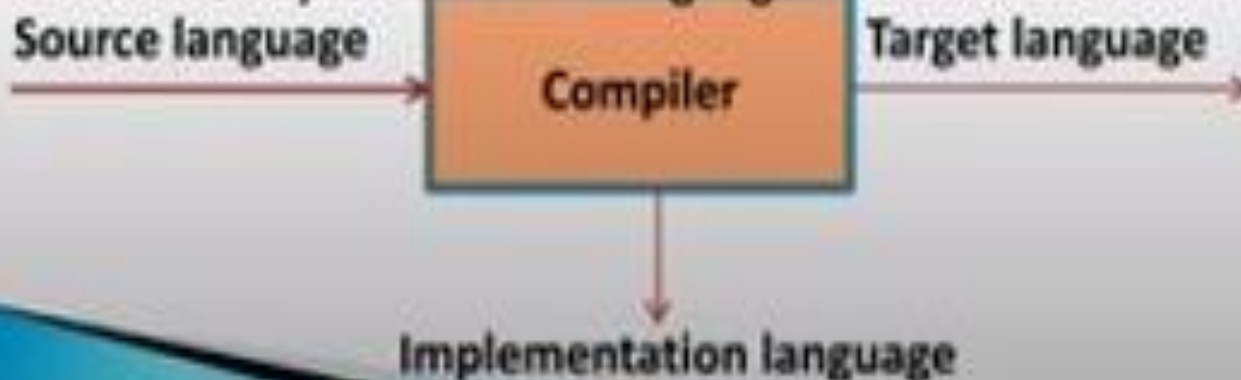
# Cross Compiler

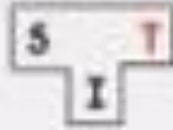


- A cross Compiler that runs on one machine and produces object code for another machine . For example a compiler runs on window 7 PC but generates code that runs on Anroid smartphone is a cross complier

- The cross-compiler is used to implement the compiler, which is characterized by three languages:

1. The source language
2. The target language
3. The implementation language



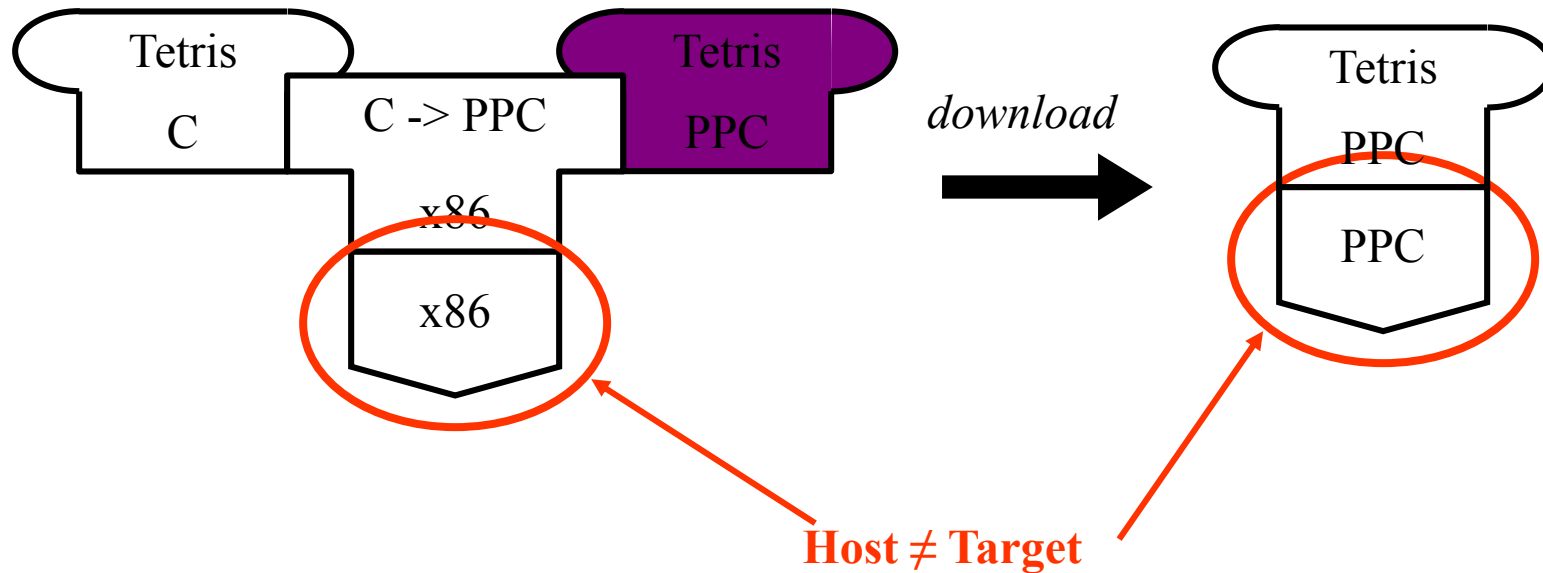
- This is represented by a T-diagram as:  

- In textual form this can be represented as  $S_I T$



# Cross compilation

Example: A C “cross compiler” from x86 to PPC

A *cross compiler* is a compiler that 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?



# Types of Cross Compiler

## 1. Native Compiler :

Native compiler are compilers that generates code for the same Platform on which it runs. It converts high language into computer's native language.

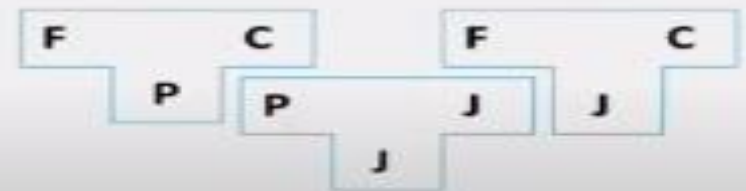
- For example Turbo C or GCC compiler

## 2. Cross compiler :

A Cross compiler is a compiler that generates executable code for a platform other than one on which the compiler is running.

- For example a compiler that running on Linux/x86 box is building a program which will run on a separate Arduino/ARM.

- Write a cross compiler for language **L** in implementation language **S** to generate code for machine **N**.
- Existing compiler for **S** runs on different machine **M** and generates code for **M**.
- When compiler **L<sub>S</sub>N** is run through **S<sub>M</sub>M** we get compiler **L<sub>M</sub>N**.



F-Fortran, P-Pascal, J-Java

# Difference between Native Compiler and Cross Compiler



NATIVE COMPILER	CROSS COMPILER
<ul style="list-style-type: none"><li>• Translates program for same hardware/platform/machine on it is running.</li></ul>	<ul style="list-style-type: none"><li>• Translates program for different hardware/platform/machine other than the platform which it is running.</li></ul>
<ul style="list-style-type: none"><li>• It is used to build programs for same system/machine &amp; OS it is installed.</li></ul>	<ul style="list-style-type: none"><li>• It is used to build programs for other system/machine like AVR/ARM.</li></ul>
<ul style="list-style-type: none"><li>• It is dependent on System/machine and OS</li></ul>	<ul style="list-style-type: none"><li>• It is independent of System/machine and OS</li></ul>
<ul style="list-style-type: none"><li>• It can generate executable file like .exe</li></ul>	<ul style="list-style-type: none"><li>• It can generate raw code .hex</li></ul>

# Single pass, Two pass, and Multi pass Compilers



- A **Compiler pass** refers to the traversal of a compiler through the entire program. Compiler pass are two types: Single Pass Compiler, and Two Pass Compiler *or* Multi Pass Compiler.
- Single pass
- Two pass or
- Multi pass Compilers

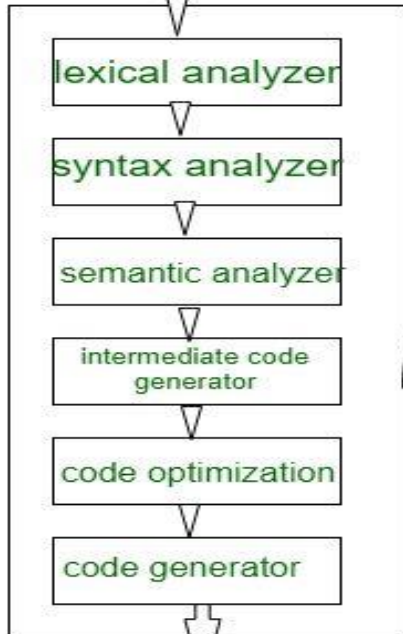




# Single Pass Compiler:

- If we combine or group all the phases of compiler design in a **single** module known as single pass compiler

High level language



low level language

Single Pass Compiler

© guru99.com

Source Code



Compiler



Target Code

all phases are in  
a single module

**In single pass Compiler source code directly transforms into machine code. For example, Pascal language**

# Single Pass Compiler:



- In above diagram there are all 6 phases are grouped in a single module, some points of single pass compiler is as:
- A one pass/single pass compiler is that type of compiler that passes through the part of each compilation unit exactly once.
- Single pass compiler is faster and smaller than the multi pass compiler.
- As a disadvantage of single pass compiler is that it is less efficient in comparison with multipass compiler.
- Single pass compiler is one that processes the input *exactly once*, so going directly from lexical analysis to code generator, and then going back for the next read.
- **Note:** Single pass compiler almost never done, early **Pascal compiler** did this as an introduction.

# Problems with single pass compiler:



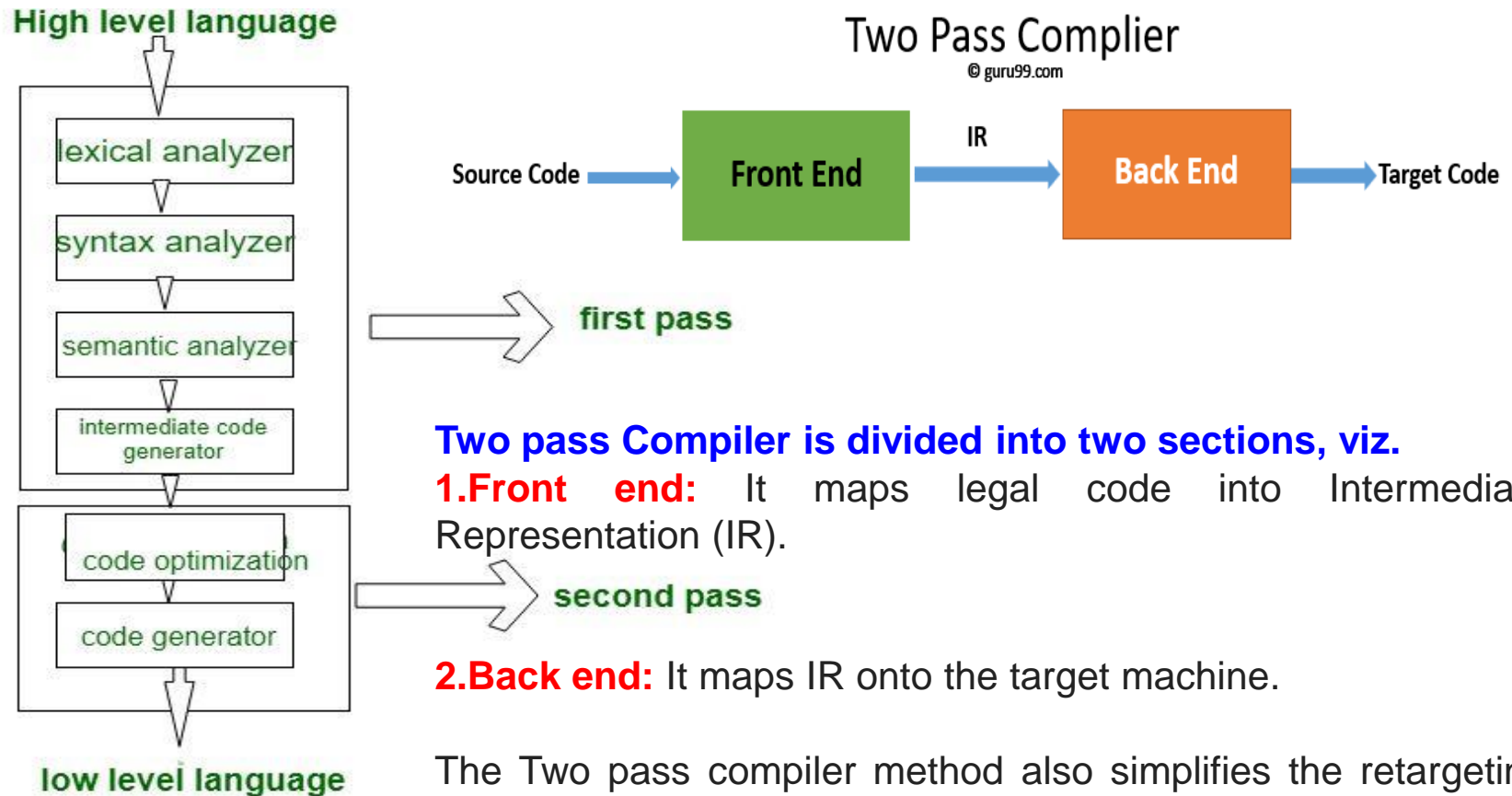
## Problems with single pass compiler are as follows

- We can not optimize very well due to the context of expressions are limited.
- As we can't backup and process, it again so grammar should be limited or simplified.
- Command interpreters such as *bash/sh/tcsh* can be considered as Single pass compiler, but they also execute entry as soon as they are processed.

## 2. Two Pass compiler or Multi Pass compiler:



- A Two pass/multi-pass Compiler is a type of compiler that processes the *source code* or abstract syntax tree of a program multiple times. In mul-tipass Compiler we divide phases in two pass as:



# First Pass



- **First Pass:** is refers as
  - *Front end*
  - *Analytic part*
  - *Platform independent*
- In first pass the included phases are **as Lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator** are work as front end and analytic part means all phases analyze the High level language and convert them **three address code**
- First pass is platform independent because the output of first pass is as three address code which is useful for every system and the requirement is to change the **code optimization and code generator phase** which are comes to the second pass.

# Second Pass



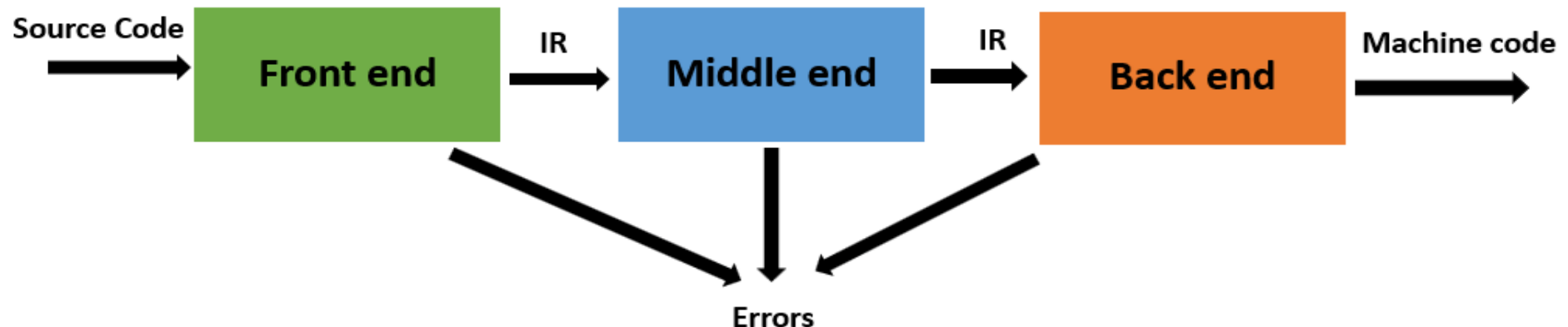
- **Second Pass:** is refers as
  - **Back end**
  - **Synthesis Part**
  - **Platform Dependent**
- In second Pass the included phases are as **Code optimization and Code generator** are work as back end and the synthesis part refers to taking input as three address code and convert them into Low level language/assembly language .
- Second pass is platform **dependent because final stage of a typical compiler converts the intermediate representation of program into an executable set of instructions which is dependent on the system.**

# Multi-pass Compiler

- The multi-pass compiler processes the source code or syntax tree of a program several times.
- It divided a large program into multiple small programs and process them.
- It develops multiple intermediate codes.
- All of these multi-pass take the output of the previous phase as an input. So it requires less memory. It is also known as '**Wide Compiler**'.

## Multi Pass Compiler

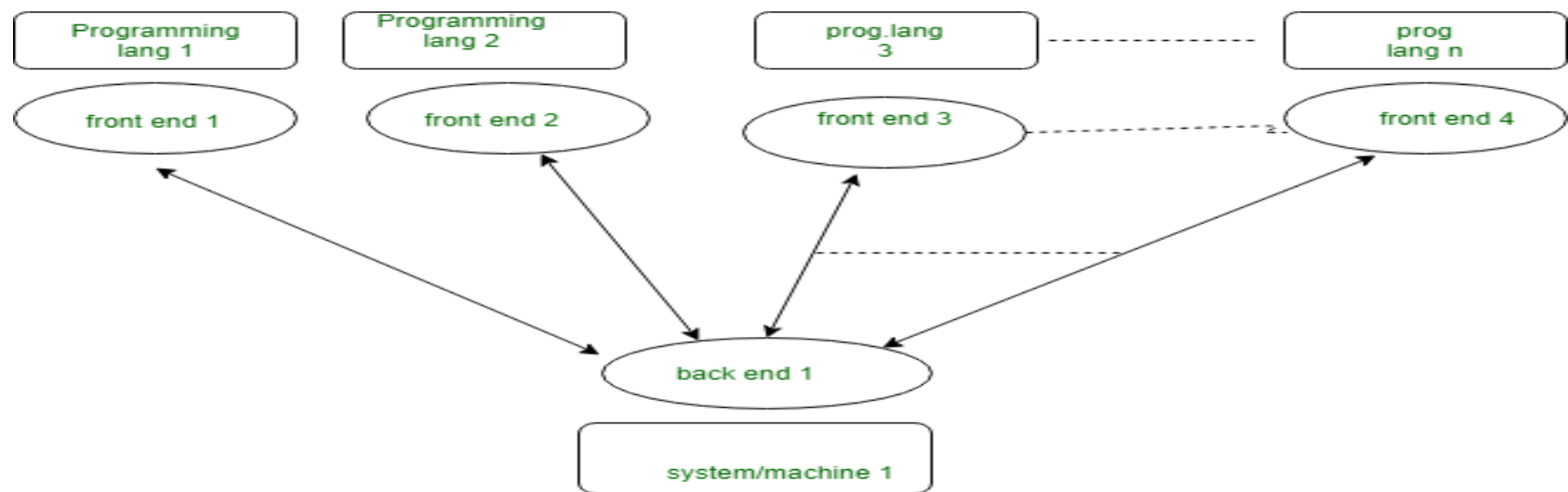
© guru99.com



# Multi-pass Compiler



- **With multi-pass Compiler we can solve these 2 basic problems:**
- If we want to design a compiler for different programming language for same machine. In this case for each programming language there is requirement of making Front end/first pass for each of them and only one Back end/second pass as:

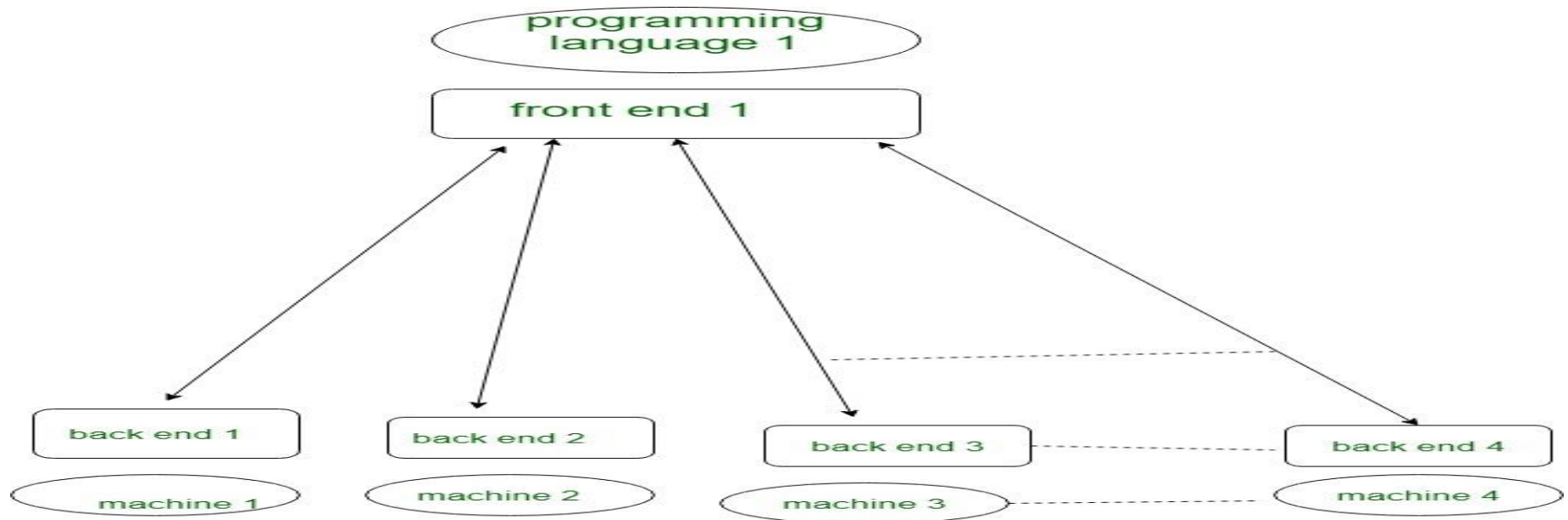




# Multi-pass Compiler



- If we want to design a compiler for same programming language for different machine/system. In this case we make different Back end for different Machine/system and make only one **Front end** for same programming language as:



# Bootstrapping



- The process of writing a compiler (or Assembler) in the target programming language which has to be compiled is known as "Bootstrapping"
- A compiler leads to a self-hosting compiler by applying Bootstrapping technique
- Many compilers for many programming languages are bootstrapped
- Examples: BASIC, ALGOL, C, PASCAL, JAVA, PYTHON, etc.

# Bootstrapping

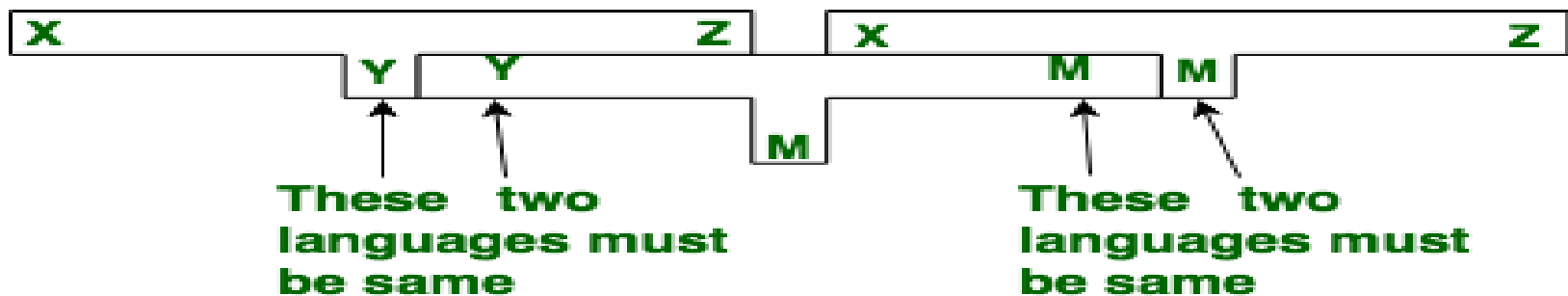


- **Bootstrapping** is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.
- Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages. to clearly understand the **Bootstrapping** technique consider a following scenario.

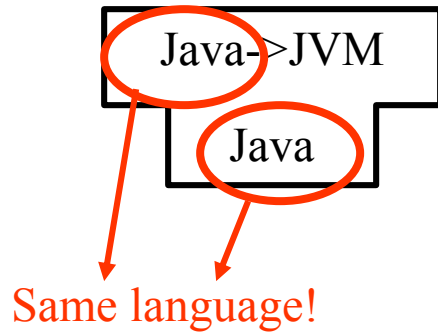
# Example:



- Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.
- We can create compiler of many different forms. Now we will generate.



# Bootstrapping



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

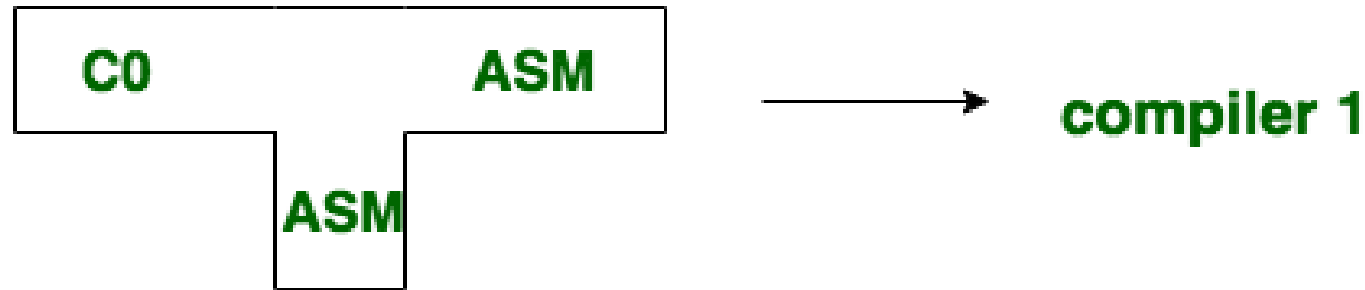
- By implementing the compiler in (a subset of) its own language, we become less dependent on the target platform => more portable implementation.
- But... **“chicken and egg problem”**? How do to get around that?  
**=> BOOTSTRAPPING**: requires some work to make the first “egg”.

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

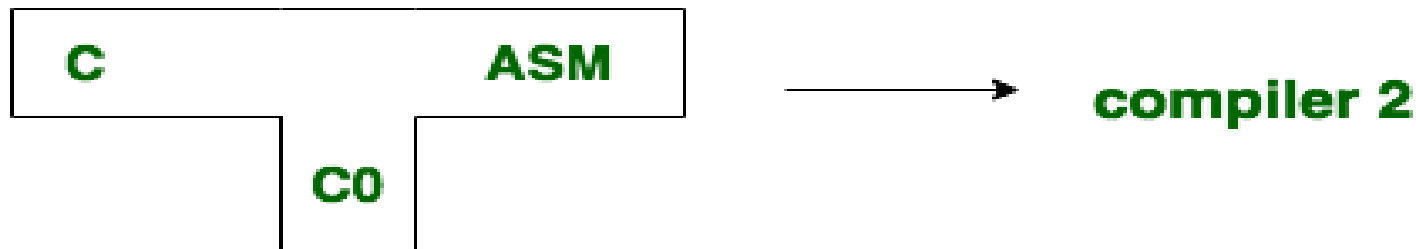
# Steps of Bootstrapping



- **Step-1:** First we write a compiler for a small of C in assembly language



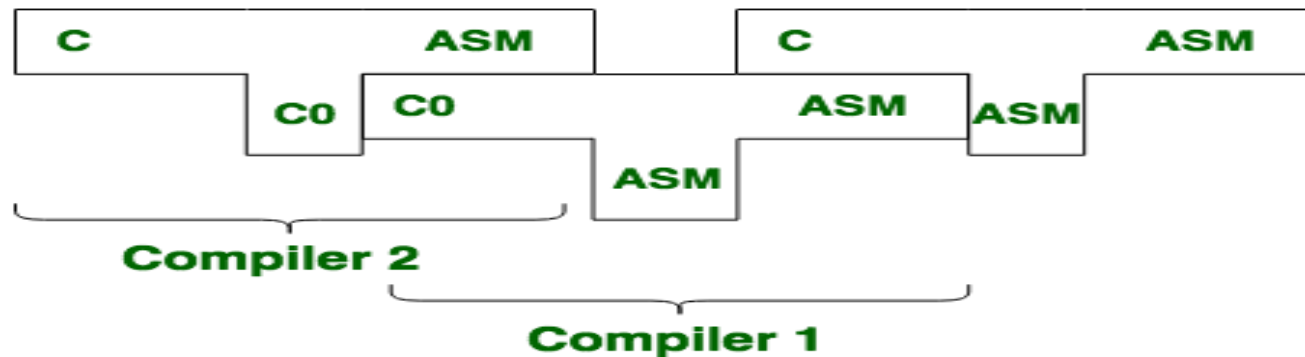
- **Step-2:** Then using with small subset of C i.e. C0, for the source language c the compiler is written.



# Steps of Bootstrapping



**Step-3:** Finally we compile the second compiler. using compiler 1 the compiler 2 is compiled.

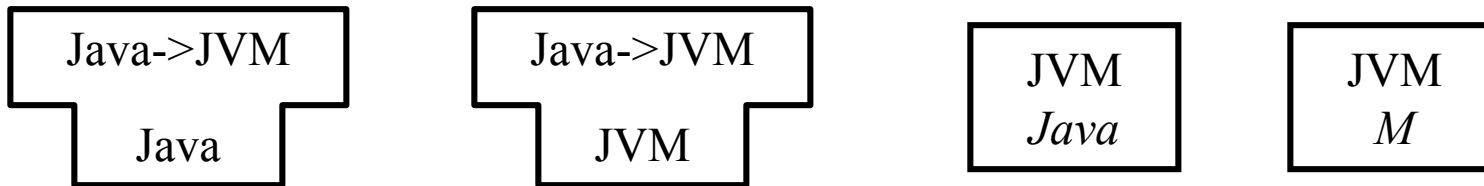


**Step-4:** Thus we get a compiler written in ASM which compiles C and generates code in ASM.

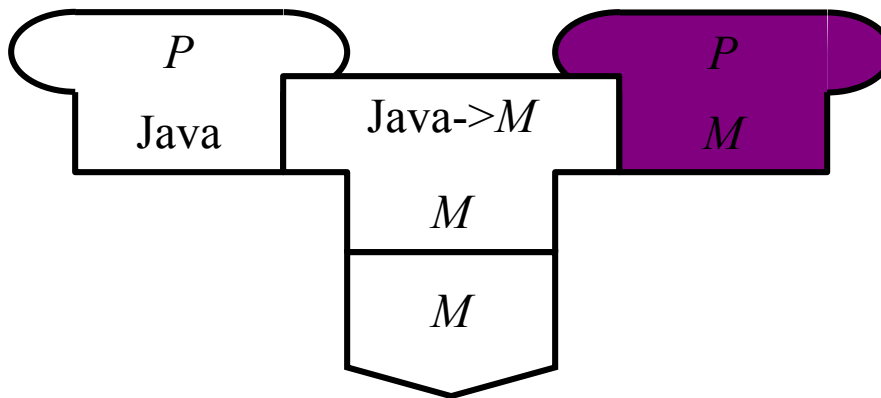
# Bootstrapping an Interpretive Compiler to Generate *M* code



Our “portable compiler kit”:



Goal we want to get a “completely native” Java compiler on machine *M*

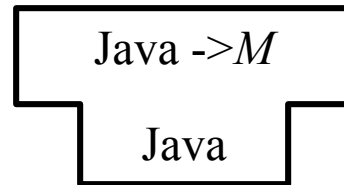




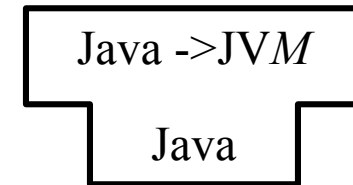
# Bootstrapping an Interpretive Compiler to Generate *M* code (first approach)



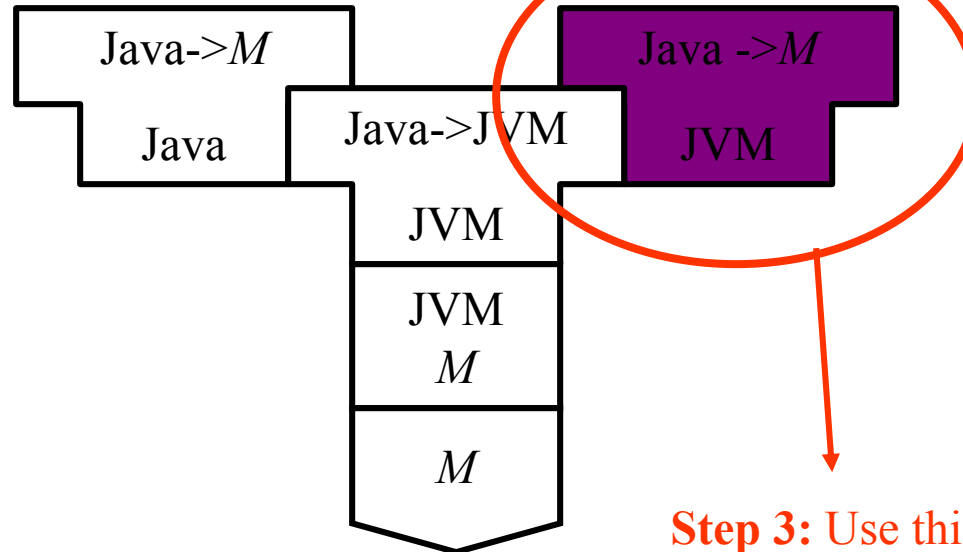
**Step 1:** implement



by rewriting



**Step 2:** compile it

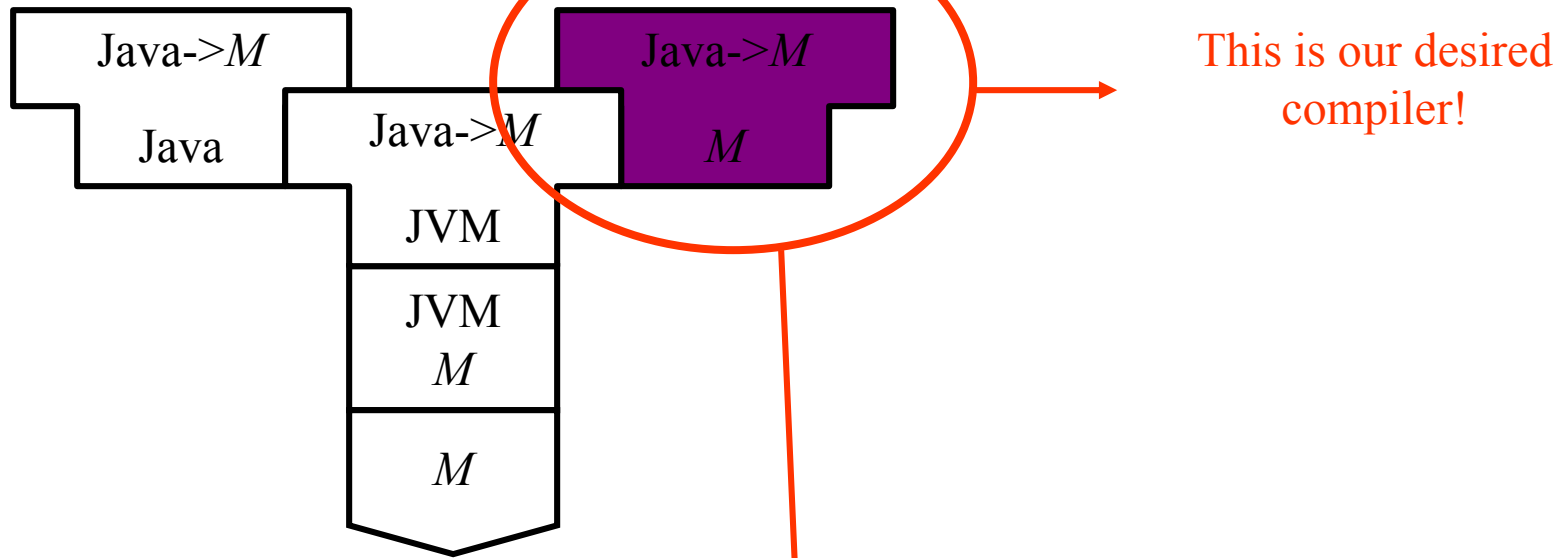


**Step 3:** Use this to compile again

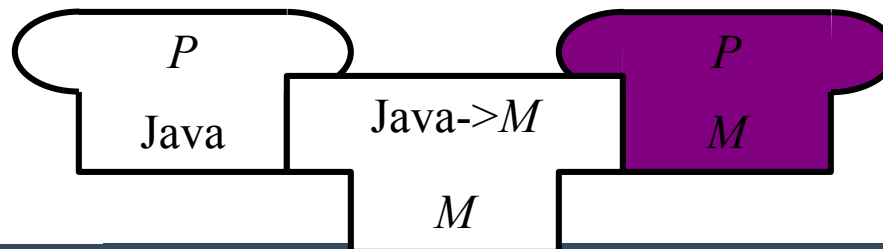
# Bootstrapping an Interpretive Compiler to Generate *M* code (first approach)



**Step 3:** “Self compile” the Java (in Java) compiler



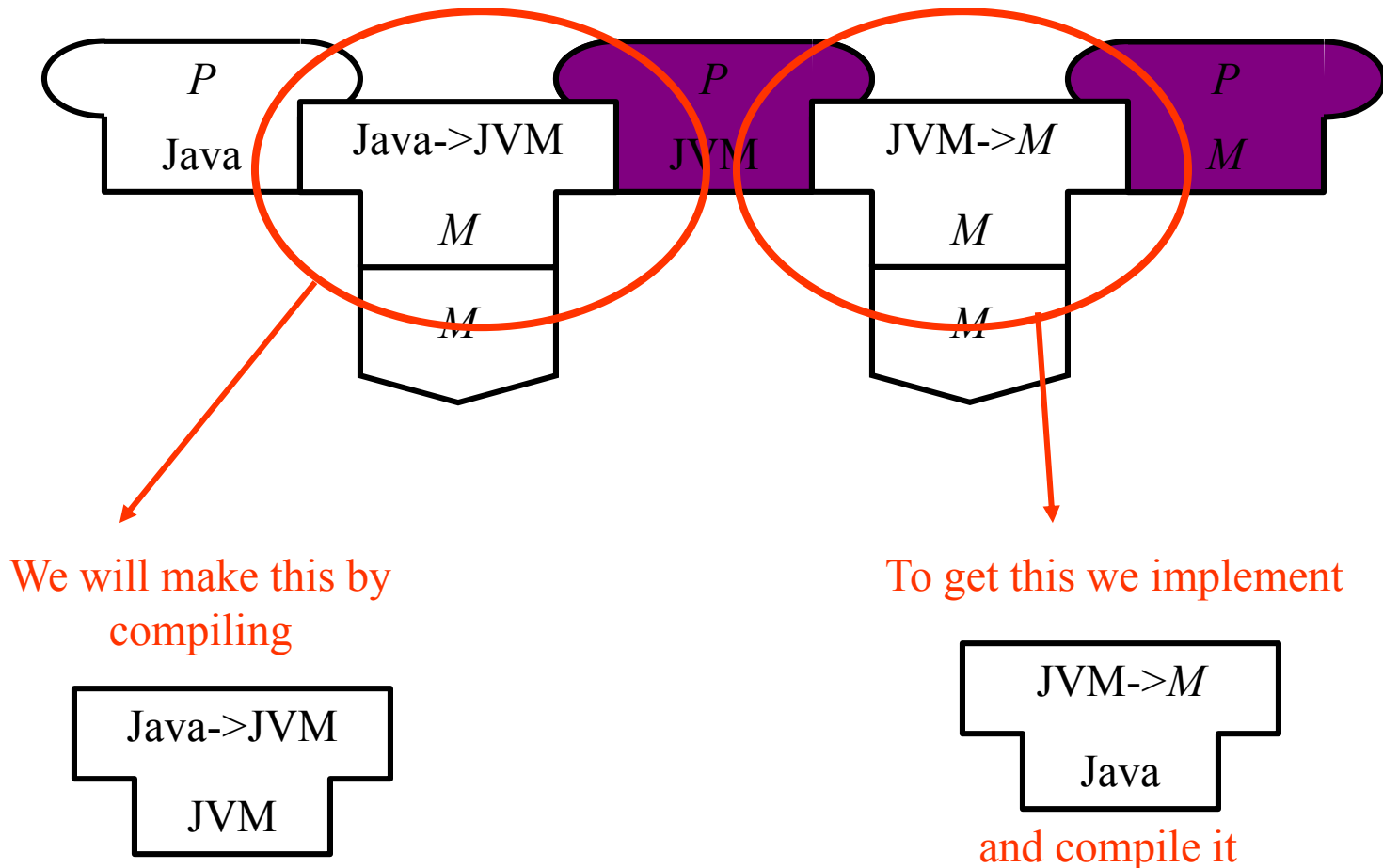
**Step 4:** use this to compile the *P* program



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



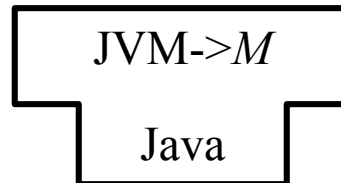
Idea: we will build a two-stage Java  $\rightarrow M$  compiler.



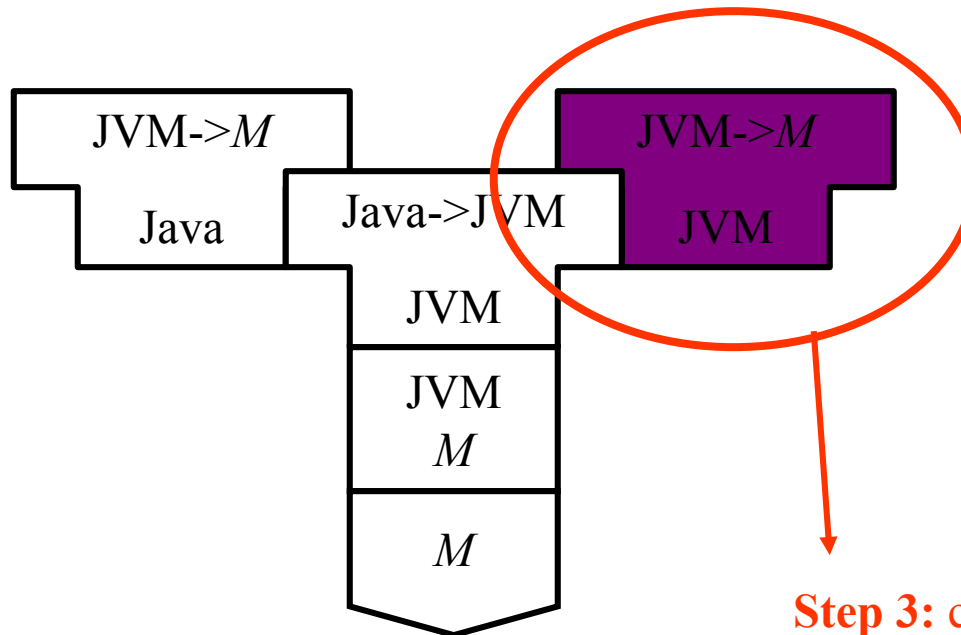
# Bootstrapping an Interpretive Compiler to Generate *M* code (second approach)



Step 1: implement



Step 2: compile it

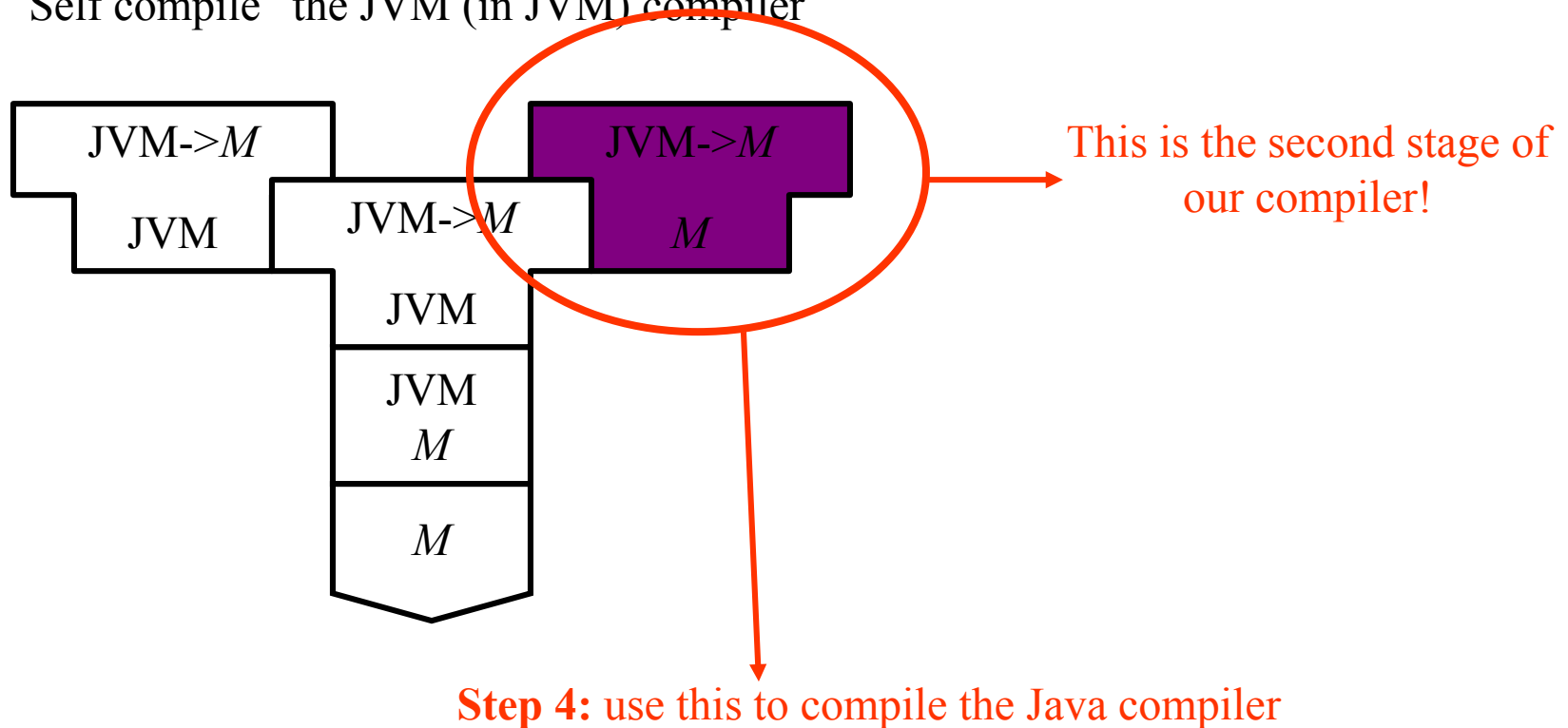


Step 3: compile this

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



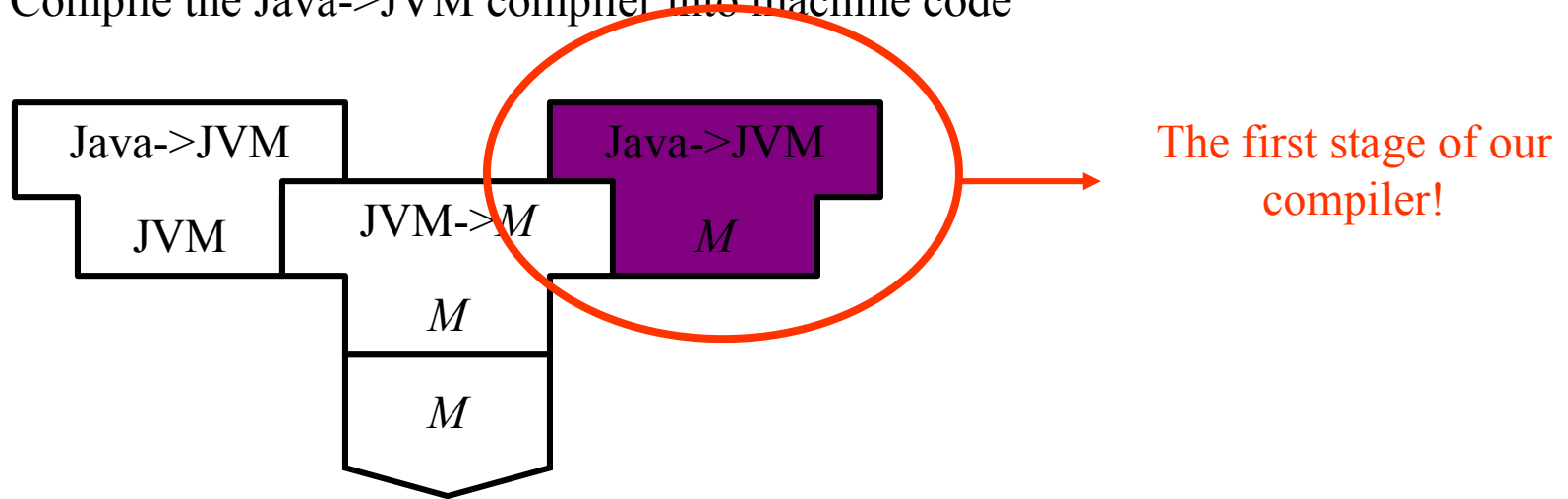
**Step 3:** “Self compile” the JVM (in JVM) compiler



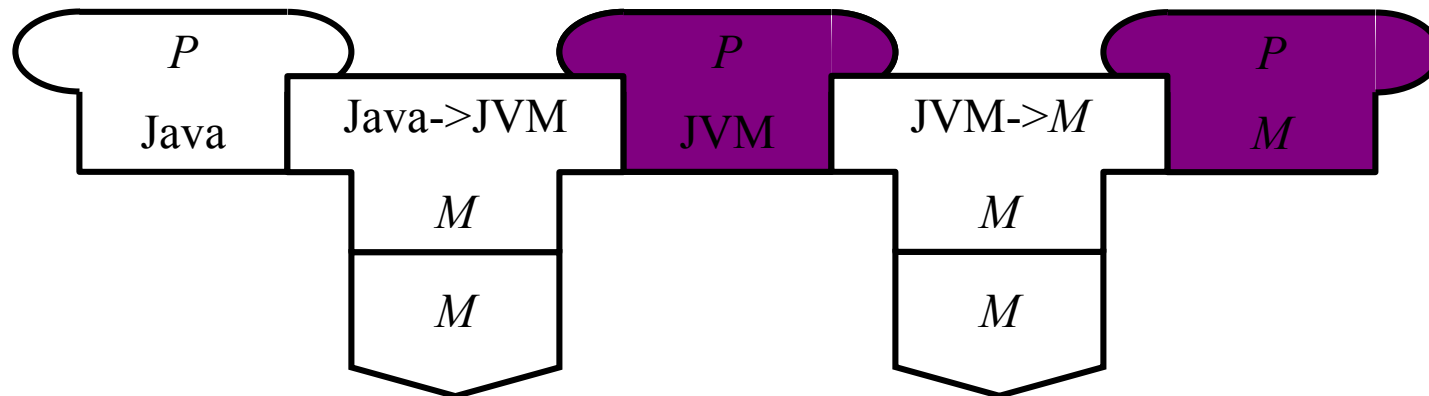
# Bootstrapping an Interpretive Compiler to Generate *M* code



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



**We are DONE!**



# Comparison of approaches to bootstrapping an interpretive compiler (portable compiler kit)



In approach one, we implement

Java  $\rightarrow M$

Java

by rewriting

Java  $\rightarrow \text{JVM}$

Java

In approach two, we implement

JVM  $\rightarrow M$

Java

by rewriting

Java  $\rightarrow \text{JVM}$

Java

In approach one, we obtain a one-stage compiler

Java  $\rightarrow M$

$M$

In approach two, we obtain a two-stage compiler

$P$

Java

Java  $\rightarrow \text{JVM}$

$M$

$M$

$P$

JVM

JVM  $\rightarrow M$

$M$

$M$

$P$

$M$

# Full Bootstrap



A full bootstrap is necessary when we are building a new compiler from scratch. One goal is to remove the dependence on a compiler for a different high-level language, even though such a compiler is very useful to start building the new compiler.

## Example:

We want to implement an Ada compiler for machine  $M$ . We don't currently have access to any Ada compiler (not on  $M$ , nor on any other machine).

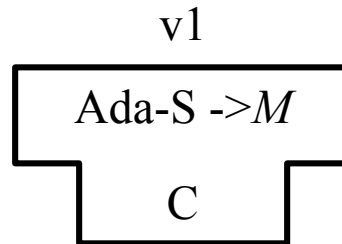
Idea: Ada is very large, so we will implement the compiler in a subset of Ada and bootstrap it from a compiler for a subset of Ada implemented in another language. (e.g. C)



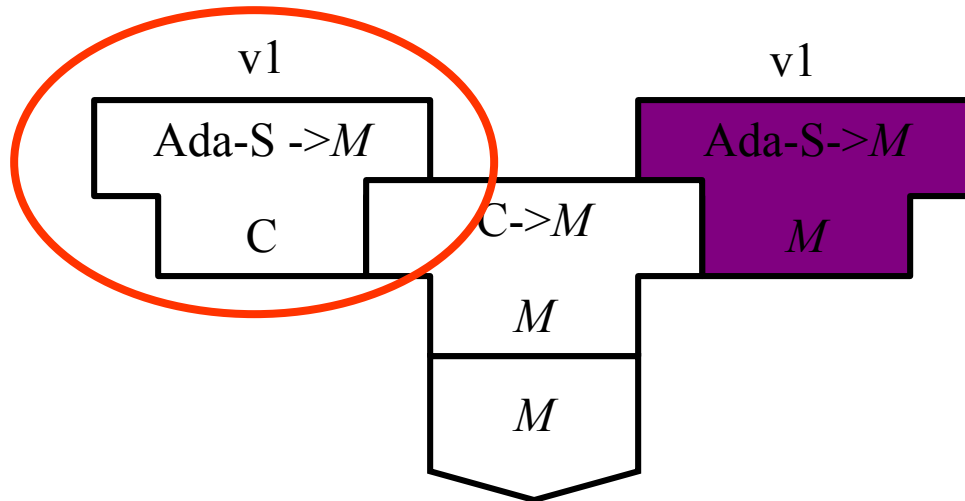
# Full Bootstrap



**Step 1a:** build a compiler (v1) for Ada-S in another language



**Step 1b:** Compile v1 compiler on *M*

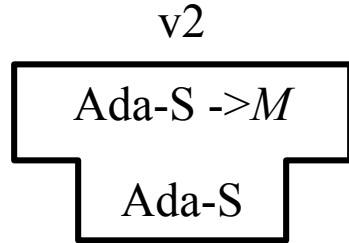


This compiler can be used for bootstrapping on machine *M* but we do not want to rely on it permanently, since it is written in *C*, and we do not want to depend on the existence of *C* compilers.

# Full Bootstrap

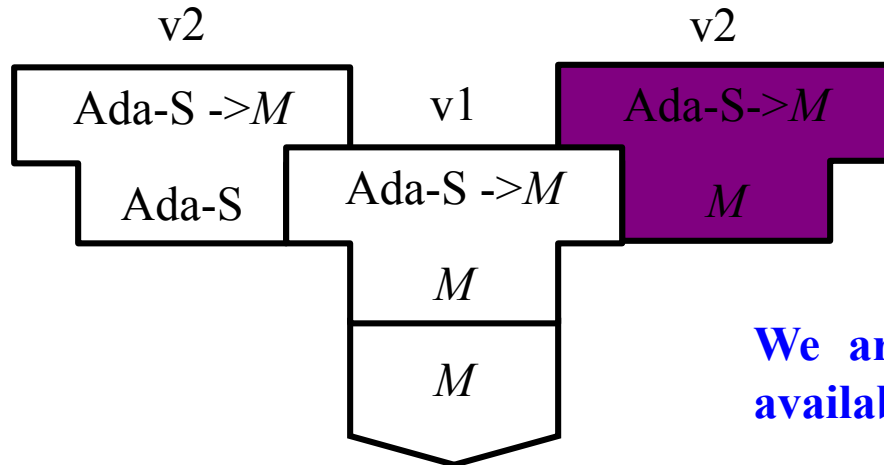


**Step 2a:** Implement v2 of Ada-S compiler in Ada-S



**Q:** Is it hard to rewrite the compiler in Ada-S?

**Step 2b:** Compile v2 compiler with v1 compiler

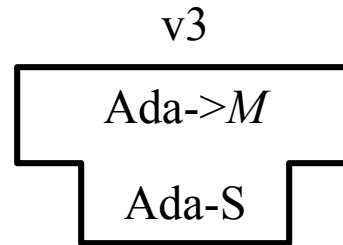


**We are now no longer dependent on the availability of a C compiler!**

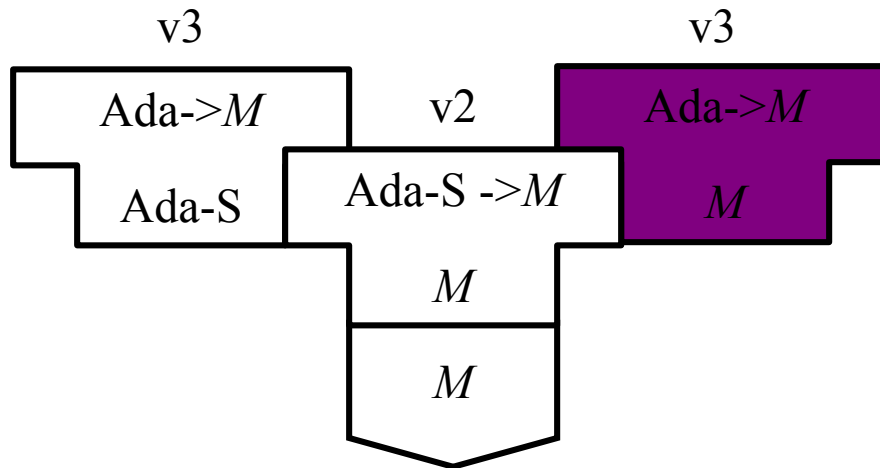
# Full Bootstrap



**Step 3a:** Build a full Ada compiler in Ada-S



**Step 3b:** Compile with v2 compiler



**From this point on we can maintain the compiler in Ada.**

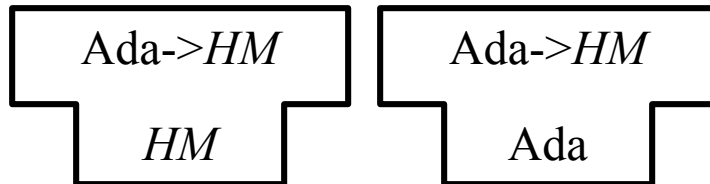
**Subsequent versions v4,v5,... of the compiler are written in the previous version of Ada**

# Half Bootstrap

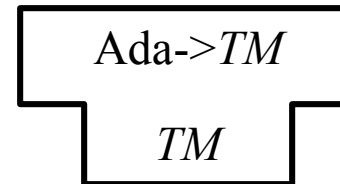


**Q:** What if we have access to an compiler for our language on a different host machine  $HM$  but want to develop one for target machine  $TM$  ?

We have:



We want:



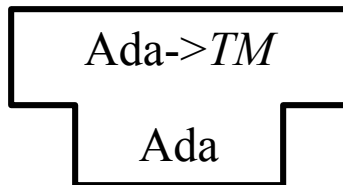
**Idea:** We can use cross compilation from  $HM$  to  $TM$  to bootstrap the  $TM$  compiler.

# Half Bootstrap

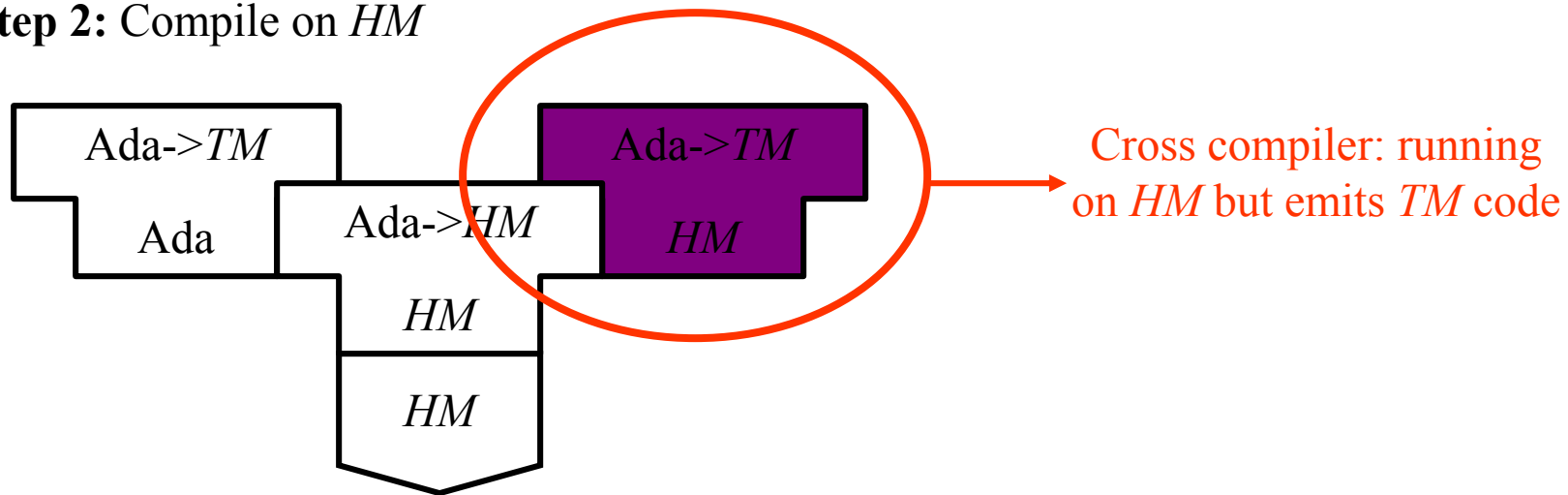


**Idea:** We can use cross compilation from  $HM$  to  $M$  to bootstrap the  $M$  compiler.

**Step 1:** Implement Ada- $\rightarrow$ TM compiler in Ada



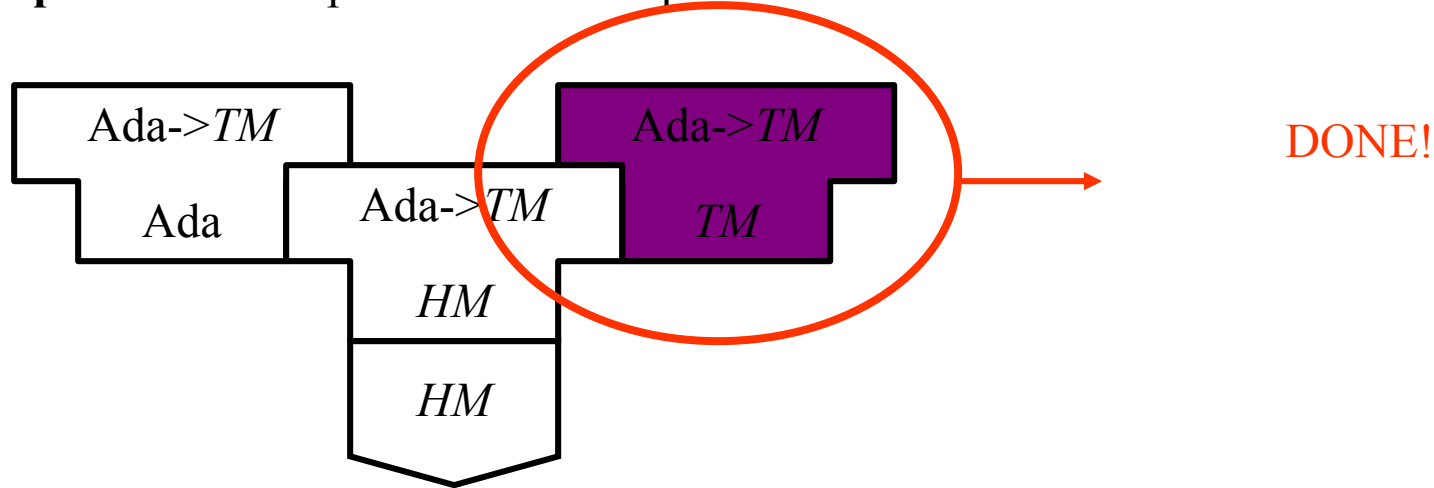
**Step 2:** Compile on  $HM$



# Half Bootstrap



**Step 3:** Cross compile our *TM* compiler.



**From now on we can develop subsequent versions of the compiler completely on *TM***

# Bootstrapping to Improve Efficiency



## The efficiency of programs and compilers:

Efficiency of programs:

- memory usage
- runtime

Efficiency of compilers:

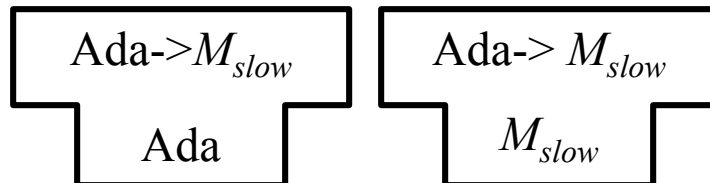
- Efficiency of the compiler itself
- Efficiency of the emitted code

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

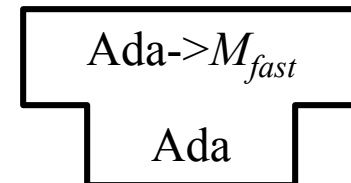
# Bootstrapping to Improve Efficiency



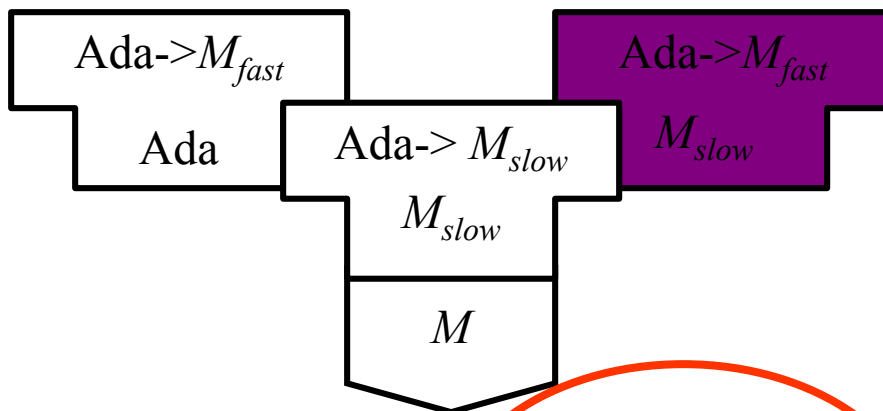
We have:



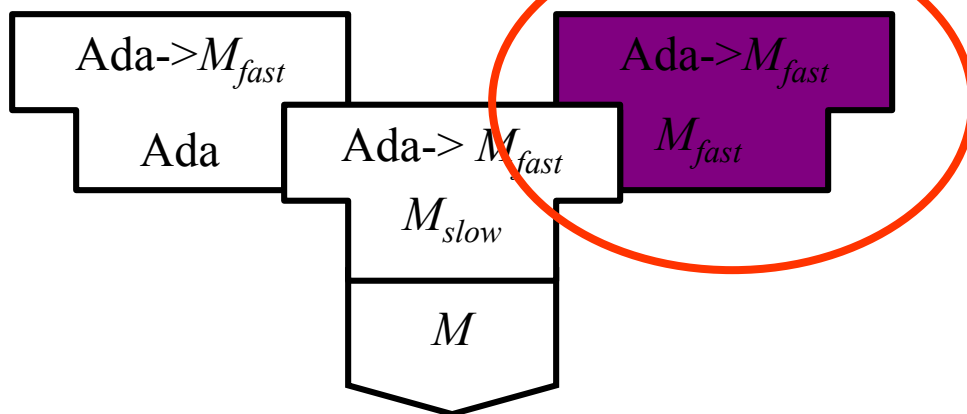
We implement:



Step 1



Step 2



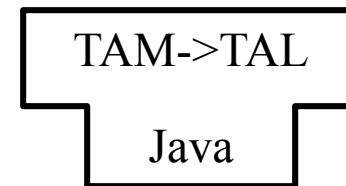
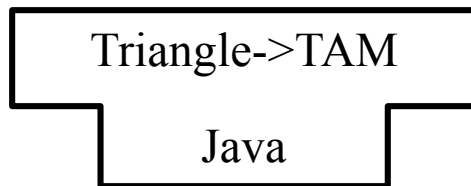
Fast compiler that emits fast code!



# The Triangle Language Processor



- The Triangle language processor includes a compiler, an interpreter, and a disassembler
- The compiler and interpreter together constitute an interpretive compiler
- TAM is an abstract machine
- TAL (Triangle Assembly Language) is an abstract version of the machine language of the TAM



# Advantages of Bootstrapping



- It is a non-trivial test of the language being compiled
- Compiler developers only need to know the language being compiled
- Compiler development can be done in the higher level language being compiled
- Improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself
- It is a comprehensive consistency check as it should be able to reproduce its own object code



# End of Bootstrapping