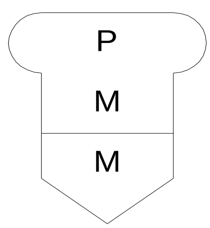
CST8152 – Compilers
Article #5
Tombstone Diagrams

Tombstone diagrams are a very convenient graphical representation of software, hardware (machines), processing of software by machines, and manipulation of programs by other programs (compilers).

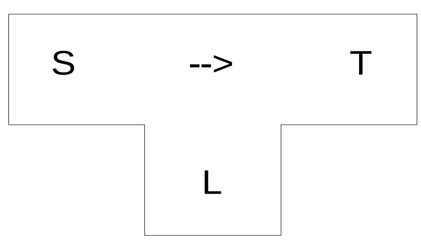
1. **Machines.** Programs run on machines executing some machine code. Pentagon-shape tombstone represents a machine executing machine code *M*. For example, *x86* is a machine executing Intel machine code.

2. **Programs.** Round-top (mushroom) tombstone represents a program ***P*** expressed in language ***L***. For example, a program **Find *written*** in Java and ***expressed*** in machine language (machine code) code *x86*.

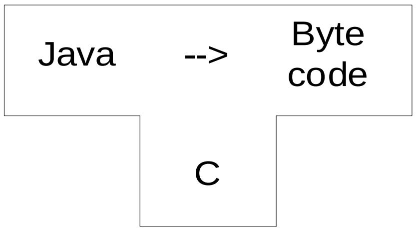
Programs always run on some machine. To express this, a program is put on top of machine. The language the program is expressed and the machine language must be the same.

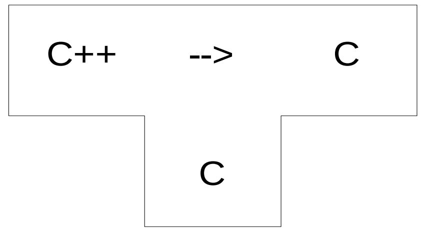


3. **Translators (Compilers).** T-shaped (T-shirt) tombstone represents translators (compilers). The head of the tombstone shows the translator's source language ***S*** and the target language ***T*** ; the arrow indicates the direction of the translation. The base of the tombstone shows the translator's implementation language.

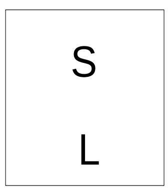


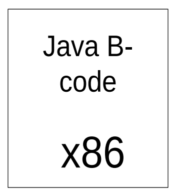
For example: a Java-into-JVM code (Byte-code) compiler written in C, and a C++-into-C compiler written in C (C++ C-front compiler).

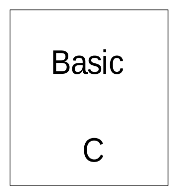




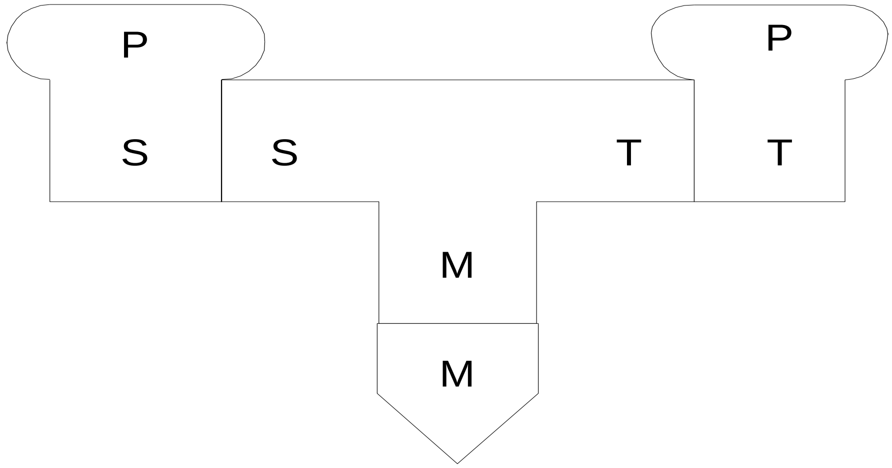
4. **Interpreters**. An interpreter is a program that accepts a program written in some source language, and in most cases, runs it immediately without translating the entire source program into target program (machine code). An interpreter is represented by a rectangular tombstone. The head of the stone indicates the interpreter's source language ***S***; the base shows the implementation language ***L***. For example: Java virtual machine (Java interpreter) expressed in x86, and BASIC interpreter written in C.



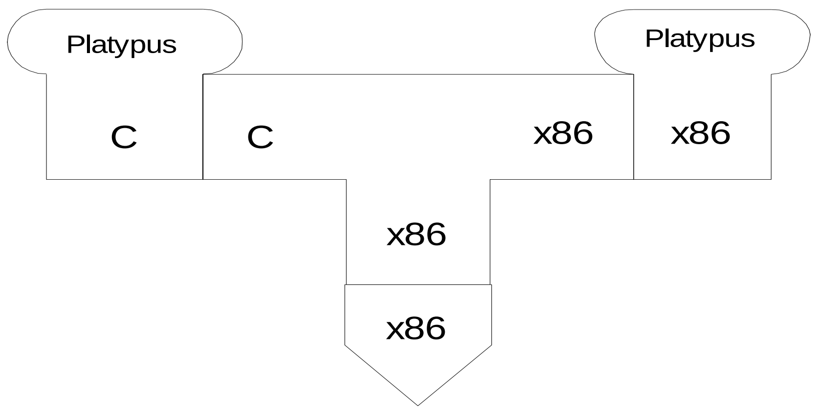




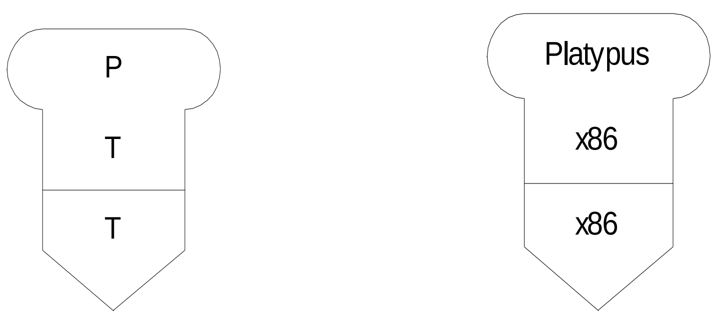
5. **Compilation and execution.** When you develop programs with a compiled language (like C, C++, and Java), the compilation of the program and the execution of the compiled program are two separate and distinctive steps. First, you need a compiler, which translates language ***S*** into language ***T*** and it is expressed in language ***M***. Since the compiler is a program itself, it must run on some machine that understands language ***M***. The compiler translates the program ***P*** written in language ***S*** into the same program ***P*** but expressed in language ***T***. The compiler runs on some machine that understands the language ***M***.
If ***T*** and ***M*** are different languages the compiler is called ***cross-compiler***. A cross-compiler is a compiler that runs on one machine (the host machine) but generates code for different machine (target machine). Example: Compiler that runs on SPARC machine but generates code for Intel machine.



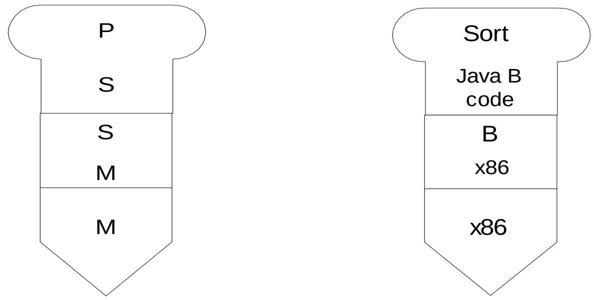
Example:



Once the program is compiled, it can be run on machine, which understands the same language the program is expressed in.



6. **Interpretation.** Interpretation is one-step process. The program is analyzed and executed one or several statements at a time. Interpreters do not produce target code.
   An ***interpretive compiler*** is a combination of compiler and interpreter. It translates the source program into an intermediate language usually for some ***virtual*** machine. After that this code is run by an interpreter of the intermediate language. Java is a perfect example.

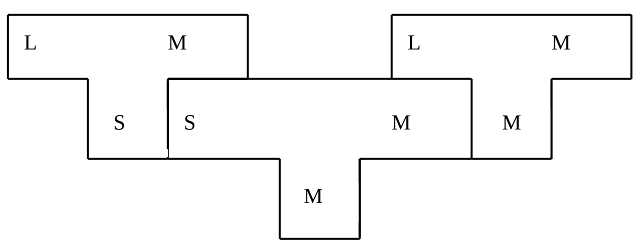


7. **Bootstrapping.** Is it possible to write a compiler using the same language the compiler is supposed to compile from? For example, to write a C compiler in C. The answer is "yes", and the process of writing such a compiler is called ***bootstrapping***. The idea is simple: First compiler is written (in assembler or other language) that can compile a subset of the language. This compiler and the subset are used to write a compiler for the full language. The method is called ***full bootstrap*** because the whole compiler is to be written from scratch. The method can be applied ones or several times for richer and richer subsets of the language until a compiler for the full language is implemented. If a compiler for the language already exist but we want to write a compiler for a different machine using the same language the compiler compiles, the process is named ***half bootstrap*** since roughly half of the compiler must be modified.

Example: Full Bootstrapping – (see section 11.2 of the textbook)

1. Write in machine code a compiler which compiles a subset S of a language L in machine code M to run on the same machine: → $S_M M$
2. Write compiler for a fuller version L of the language using the subset S: → $L_S M$
3. Use $S_M M$ to translate $L_S M$: →$L_M M$

   $L_S M + S_M M = [L_M M]_0$



4. Now with the fuller version of the language, write a better compiler and use $L_M M$ to compile it:

   $L_L M + [L_M M]_0 = [L_M M]_1$

5. and again…

$L_LM + [L_MM]_1 = [L_MM]_2$

the compiler must consistently compile itself