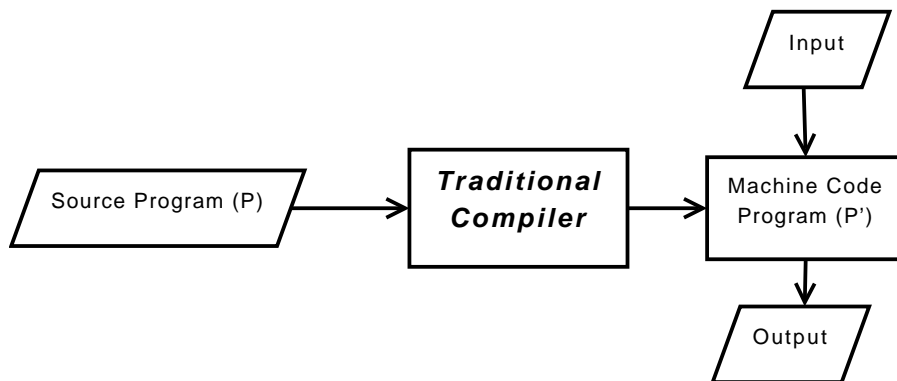


Compilers and Interpreters

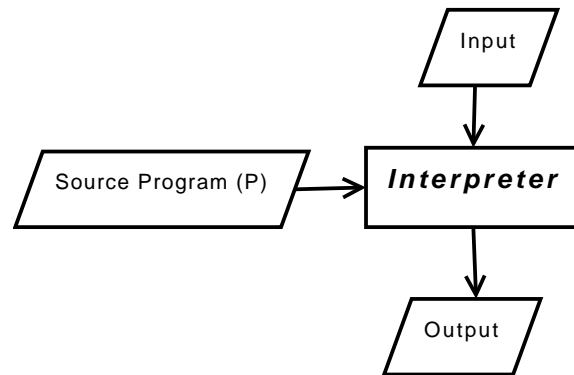
Most programs are written in high-level programming languages that are not directly implemented by any real hardware machine. So it is not possible to simply dump the source code onto a computer and expect it to run. There are two basic approaches to bridge the gap between high-level programming languages and machine code: *compilation* and *interpretation*.

Traditional Compilation. With compilation, some source program P written in some source language L is translated into an equivalent program P' in some target language L' . The program that does this translation is called a *compiler*. Typically, the source language L is some high-level programming language (e.g., C), and the target language L' is the machine code of some hardware machine. The output program P' would then be an executable that can run directly on the hardware. In this way, P' is both the output of the compiler program and a program itself. After compilation P' can be executed (independently of the compiler) to process input and produce output:



Interpretation. An alternative approach to program execution is interpretation. An interpreter is a program that executes another program P written in some programming language L by simulating L and directly performing the tasks required by the statements in P while P is running. With an interpreter, there is no independent executable output P' . The interpreter itself is the only program that runs directly on L' . It knows how to simulate the execution of each kind of statement in L , and so it can be used to perform the tasks for each of the statements in P as they would be performed on L .

For a non-interactive program P , the interpreter can be seen as taking two inputs (P and the input for P) and directly producing an output. This output must be the same as P would produce if it were executing directly on L :



Comparison of Compilation and Interpretation. Compilations happens once (each time the source is modified) whereas interpretation continues throughout every execution of a program. So for long running programs, compilation is almost always faster. In addition, many compilers expend effort to analyze programs and rearrange and ‘optimize’ the code so that it runs even faster. (A good interpreter might execute a program about $20\times$ slower than the compiled and optimized machine code for the same program.)

On the other hand, this translation and optimization takes time, so if you are developing a program, you might get faster turnaround time with a interpreter. In addition, with a optimizing compiler the program that gets executed does not always perform operations in the order that they appear in the program, whereas interpretation executes each statement as it appears in the program. Thus, interpreted programs are often easier to debug.

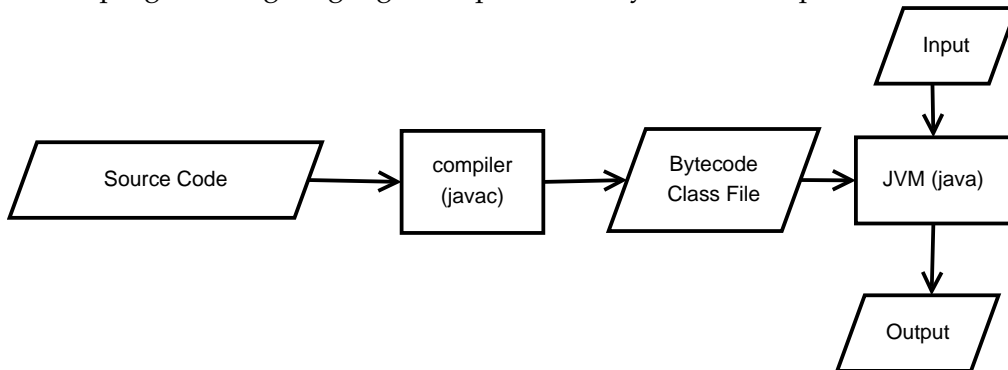
Another issue is the amount of runtime overhead required to implement various features. Languages like Lisp, SmallTalk, and most ‘scripting languages’ tend to require a lot of decisions and actions to take place in the background that are not explicitly mentioned in the code. One example of this we’ve already discussed is garbage collection, but other example include the way variable names are mapped to their contents, complex built-in data types, dynamic loading of new program code, and reflection. In general the runtime performance overhead of these features is the same whether the program is compiled or interpreted, so in more ‘dynamic’ languages where lots of these decisions occur at runtime, interpretation will not be significantly slower than compilation. These languages are typically implemented with an interpreter and are sometimes called “interpreted languages” even though a compiler could be written for them. On the other hand, languages that can be analyzed easily and for which many decisions can be decided during compilation (e.g., Fortran, C, C++) benefit from compilation; these are often called “compiled languages” even though an interpreter could be written for them.

Virtual Machines

Modern execution environments, however, blur the distinction between compilers and interpreters. On the one hand, modern compilers process provide an ability to separately compile various program modules, and operating systems provide facilities for dynamically loading and linking the modules together when the program starts or even as the program runs. On the other hand, interpreters have evolved into complex virtual machines that can improve performance by selectively compiling parts of the program that it is executing. In addition, a hybrid compilation/interpretation process is often used where the source program is first compiled to an intermediate bytecode, which is then interpreted and/or compiled by a virtual machine.

As a concrete example, consider Java—or more specifically, Sun’s reference implementation of the Java, the Java platform standard edition. There are executable programs that are of interest: *javac* and *java*. *javac* stands for Java compiler but it does not produce executable machine language instead it translates from the Java source programming language into a bytecode for the Java Virtual Machine.¹ A separate stream of bytecode instructions is created for each method and the bytecode for all of the methods in a given class are collected into a Java ‘class file.’

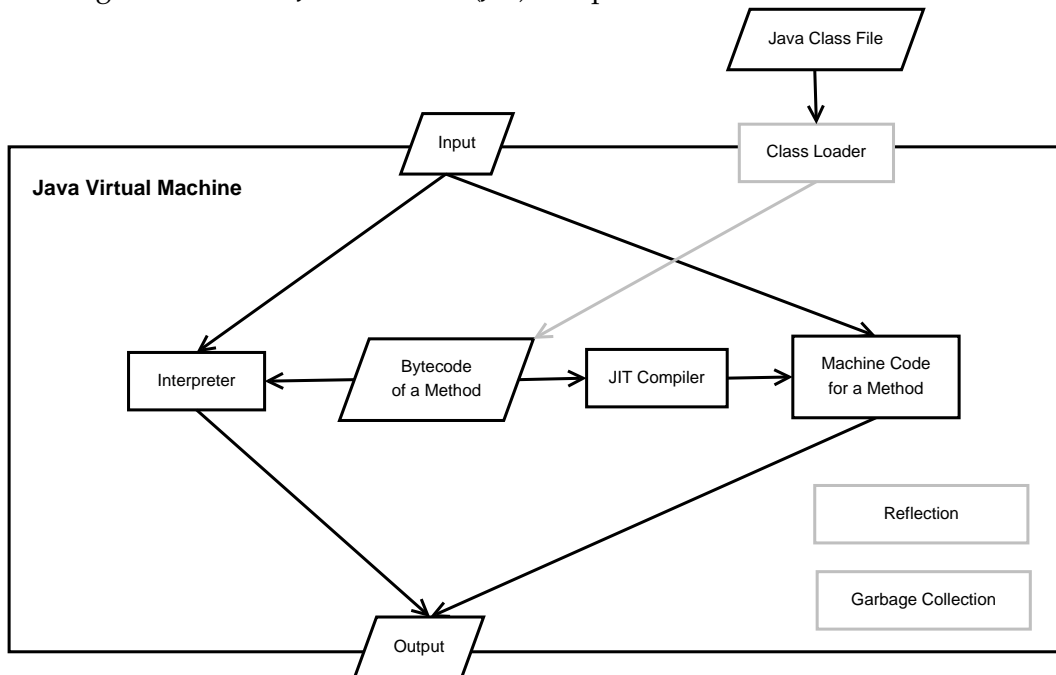
These ‘class files’ are the output of compilation, but they are also the input to the Java Virtual Machine (the *java* executable). From the outside the Java Virtual Machine (JVM) looks a lot like our standard interpreter but instead of having a source programming language as input, it has bytecode as input:



Internally, though things are even more complicated. The Java Virtual Machine loads class files into memory on a class-by-class basis but then executes bytecode stored in the classes on a method-by-method basis. It usually interprets the bytecode for a method the first few times the method is called. After a method is called many times, however, the virtual machine will decide to compile that method to native machine code and then execute that machine code directly.

¹Although, at one point, Sun did actually build a computer chip that could run Java Bytecode as its native machine code instruction set.

This process of compiling parts of the program as needed while the program is running is known as “Just-in-time” (JIT) compilation:



In addition to a compiler and/or interpreter, a Java Virtual Machine also includes other runtime facilities that support features such as dynamic class loading, reflection, and garbage collection.

Dynamic Loading and Eval

One of the more interesting facilities provided by the Java Virtual Machine is the ability to load class files into the virtual machine and link them dynamically to a running program. This dynamic class loading happens automatically the first time each class is used, but can also be explicitly requested by the running program to, for example, download a new plugin over the internet and integrate into the running program.

Slightly less flexible versions of dynamic loading and linking are provided for traditional compiled languages (e.g., C) by most modern operating systems (e.g., Linux) and through calls to system libraries (e.g., `dlopen`).

An even more flexible version of dynamic loading is available in many ‘interpreted languages.’ Lisp/Scheme provides an `eval` function. This function takes a list containing Lisp/Scheme code as an argument

```
(let ((code (cons '+ (cons 1 (cons 1 '())))))
  (cons code (eval code)))
```

will evaluate to: `((+ 1 1) . 2)`.

Similar statements/operators are provided by most Unix Shells ('...' and \$(...)) and by scripting languages such as Perl (eval "...");. These tend to use program source code stored in strings rather than as lists.

This facility is incredibly powerful and can be used in many of the same ways higher-order functions can be used. But it is also dangerous, since the code being incorporated by eval can be anything. Letting user input from an untrustworthy source into the an eval string is an easy way to introduce security vulnerability.

For more information: Mitchell, 4.1; Scott, Programming Language Pragmatics, 2nd ed.; Lindholm and Yellin, Java Virtual Machine Specification, 2nd ed.; dlopen man page