CHAPTER TWO

Language Processors

In this book we shall study two particularly important kinds of language processor: translators (particularly compilers) and interpreters. In this chapter we start by reviewing the basic ideas of translation and interpretation, which will already be familiar to most readers. Then we build on these basic ideas to explore the more sophisticated ways in which language processors can be used. A language processor is itself a program, and thus can be processed (translated or interpreted) in just the same way as an ordinary program. The ultimate development of this idea is bootstrapping, whereby a language processor is used to process itself!

In this chapter we view translators and interpreters as 'black boxes'; we concentrate on what they do rather than how they do it. In subsequent chapters we shall look inside them to see how they work.

2.1 Translators and compilers

A ***translator*** is a program that accepts any text expressed in one language (the translator's ***source language***), and generates a semantically-equivalent text expressed in another language (its ***target language***).

Example 2.1 Translators

Here are some diverse examples of translators:

(a) A Chinese-into-English translator: This is a program that translates Chinese texts into English. The source and target languages of this translator are both natural languages.

Natural-language translation is an advanced topic, related to artificial intelligence, and well beyond the scope of this textbook. We shall restrict our attention to translators whose source and target languages are programming languages.

(b) A Java-into-C translator: This is a program that translates Java programs into C. The source language is Java, and the target language is C.

(c) A Java-into-x86[1] compiler: This is a program that translates Java programs into x86 machine code. The source language is Java, and the target language is x86 machine code.

(d) An x86 assembler: This is a program that translates x86 assembly-language programs into x86 machine code. The source language is x86 assembly language, and the target language is x86 machine code.

□

An ***assembler*** translates from an assembly language into the corresponding machine code. An example is the x86 assembler of Example 2.1(d). Typically, an assembler generates one machine-code instruction per source instruction.

A ***compiler*** translates from a high-level language into a low-level language. An example is the Java-into-x86 compiler of Example 2.1(c). Typically, a compiler generates several machine-code instructions per source command.

Assemblers and compilers are the most important kinds of programming language translator, but not the only kinds. We sometimes come across *high-level translators* whose source and target languages are both high-level languages, such as the Java-into-C translator of Example 2.1(b). A *disassembler* translates a machine code into the corresponding assembly language. A *decompiler* translates a low-level language into a high-level language. (See Exercise 2.1.)

Here the translated texts are themselves programs. The source language text is called the ***source program***, and the target language text is called the ***object program***.

Before performing any translation, a compiler checks that the source text really is a well-formed program of the source language. (Otherwise it generates error reports.) These checks take into account the *syntax* and the *contextual constraints* of the source language. Assuming that the source program is indeed well-formed, the compiler goes on to generate an object program that is semantically equivalent to the source program, i.e., that will have exactly the desired effect when run. Generation of the object program takes into account the *semantics* of the source and target languages.

Translators, and other language processors, are programs that manipulate programs. Several languages are involved: not only the source language and the target language, but also the language in which the translator is itself expressed! The latter is called the *implementation language*.

To help avoid confusion, we shall use ***tombstone diagrams*** to represent ordinary programs and language processors, and to express manipulations of programs by language processors. We shall use one form of tombstone to represent an ordinary program, and distinctive forms of tombstone to represent translators and interpreters.

---

[1] We use the term x86 to refer to the family of processors represented by the Intel 80386 processor and its successors.

An ordinary program is represented by a round-topped tombstone, as shown in Figure 2.1. The head of the tombstone names the program *P*. The base of the tombstone names the implementation language *L*, i.e., the language in which the program is expressed.

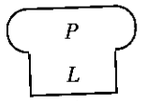


Figure 2.1 Tombstone representing a program *P* expressed in language *L*.

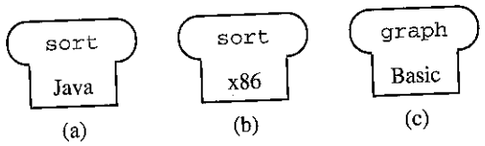
Example 2.2 Tombstone diagrams representing programs

The following diagrams show how we represent:

(a) A program named `sort` expressed in Java.

(b) A program named `sort` expressed in x86 machine code. (By convention, we abbreviate 'x86 machine code' to 'x86'.)

(c) A program named `graph` expressed in Basic.



□

Programs run on machines. A machine that executes machine code *M* is represented by a pentagon inside which *M* is named, as shown in Figure 2.2.

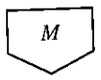


Figure 2.2 Tombstone representing a machine *M*.

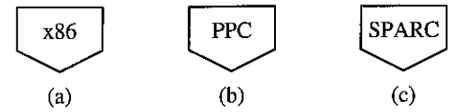
Example 2.3 Tombstone diagrams representing machines

The following diagrams show how we represent:

(a) An x86 machine.

(b) A Power PC (PPC) machine.

(c) A SPARC machine.



□

A program can run on a machine only if it is expressed in the appropriate machine code. Consider running a program *P* (expressed in machine code *M*) on machine *M*. We represent this by putting the *P* tombstone on top of the *M* pentagon, as shown in Figure 2.3.

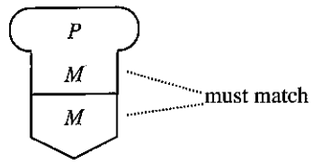


Figure 2.3 Running program *P* on machine *M*.

Example 2.4 Tombstone diagrams representing program execution

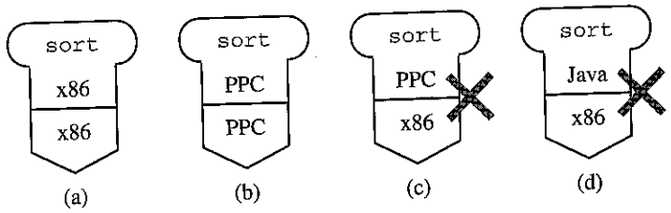
The following diagrams show how we represent:

(a) Running program `sort` (expressed in x86 machine code) on an x86 machine.

(b) Running program `sort` (expressed in PPC machine code) on a PPC machine.

(c) Attempting to run program `sort` (expressed in PPC machine code) on an x86 machine. Of course, this will not work; the diagram clearly shows that the machine code in which the program is expressed does not match the machine on which we are attempting to run the program.

(d) Attempting to run program `sort` (expressed in Java) on an x86 machine. This will not work either; a program expressed in a high-level language cannot run immediately on any machine. (It must first be translated into machine code.)



□

We have now introduced the elementary forms of tombstone. There are also distinctive forms of tombstone to represent different kinds of language processor. A translator is represented by a T-shaped tombstone, as shown in Figure 2.4. The head of the tombstone names the translator's source language S and target language T, separated by an arrow. The base of the tombstone names the translator's implementation language L.[2]

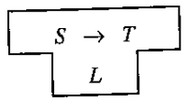


Figure 2.4 Tombstone representing an S-into-T translator expressed in language L.

Example 2.5 Tombstones representing translators

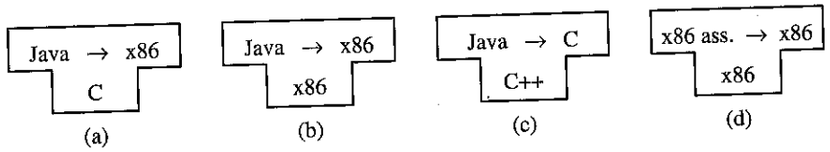
The following diagrams show how we represent:

(a) A Java-into-x86 compiler, expressed in C.

(b) A Java-into-x86 compiler, expressed in x86 machine code.

(c) A Java-into-C translator, expressed in C++.

(d) An x86 assembler, which translates from x86 assembly language into x86 machine code, and is itself expressed in x86 machine code.



□

[2] Although we use tombstones of different shapes to represent ordinary programs, translators, and interpreters, the base of a tombstone always names the implementation language. Compare Figures 2.1, 2.4, and 2.6.

An S-into-T translator is itself a program, and can run on machine M only if it is expressed in machine code M. When the translator runs, it translates a source program P, expressed in the source language S, to an equivalent object program P, expressed in the target language T. This is shown in Figure 2.5. (The object program is shaded gray, to emphasize that it is newly generated, unlike the translator and source program, which must be given at the start.)

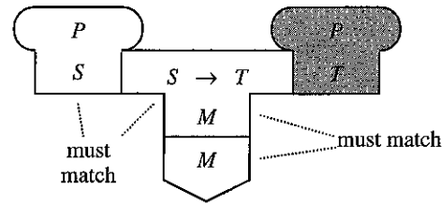
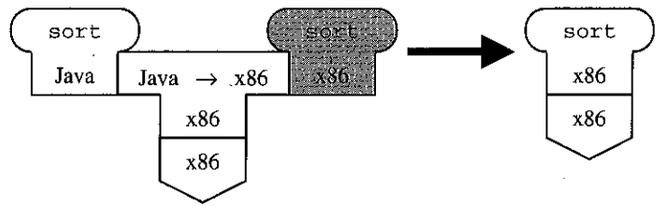


Figure 2.5 Translating a source program P expressed in language S to an object program expressed in language T, using an S-into-T translator running on machine M.

Example 2.6 Compilation

The following diagram represents compilation of a Java program on an x86 machine. Using the Java-into-x86 compiler, we translate the source program `sort` to an equivalent object program, expressed in x86 machine code. Since the compiler is itself expressed in x86 machine code, the compiler will run on an x86 machine.



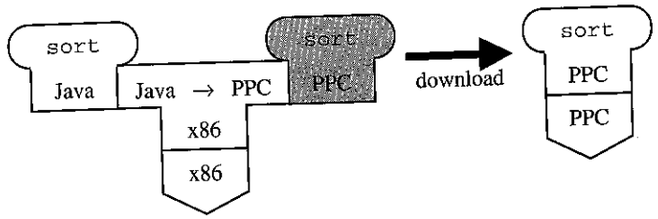
The second stage of the diagram shows the object program being run, also on an x86 machine.

□

A *cross-compiler* is a compiler that runs on one machine (the *host machine*) but generates code for a dissimilar machine (the *target machine*). The object program must be generated on the host machine but downloaded to the target machine to be run. A cross-compiler is a useful tool if the target machine has too little memory to accommodate the compiler, or if the target machine is ill-equipped with program development aids. (Compilers tend to be large programs, needing a good programming environment to develop, and needing ample memory to run.)

Example 2.7 Cross-compilation

The following diagram represents cross-compilation of a Java program to enable it to run on a Power PC microprocessor. Using a Java-into-PPC cross-compiler, we translate the source program `sort` to an equivalent object program, expressed in PPC machine code. Since the compiler is itself expressed in x86 machine code, the compiler runs on an x86 machine.



The second stage of the diagram shows the object program being run on a PPC machine, having been downloaded from the x86.

□

The behavior of a translator can be summarized by a few simple rules, which are clearly evident in Figure 2.5:

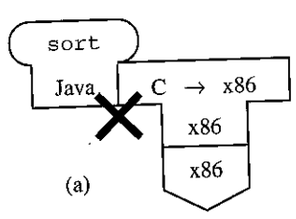
- A translator (like any other program) can run on a machine M only if it is expressed in machine code M.
- The source program must be expressed in the translator's source language S.
- The object program is expressed in the translator's target language T.
- The object program is semantically equivalent to the source program. (We emphasize this by giving the source and object programs the same name.)

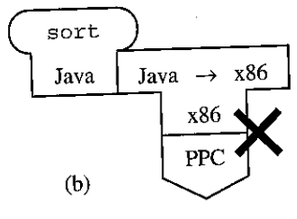
Example 2.8 Illegal translator interactions

The following tombstone diagrams illustrate what we *cannot* do with a translator:

(a) A C compiler cannot translate a Java source program.

(b) A translator expressed in x86 machine code cannot run on a PPC machine.





Similarly, it should be clear that a translator expressed in C or Java cannot run on any machine. (It must first be translated into machine code.)

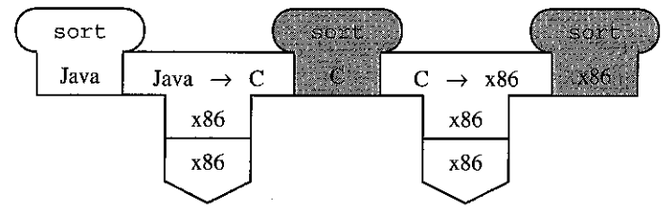
□

A *two-stage translator* is a composition of two translators. If we have an S-into-T translator and a T-into-U translator, we can compose them to make a two-stage S-into-U translator. The source language S is translated to the target language U not directly, but via an intermediate language T.

We can easily generalize this idea to multiple stages. An *n-stage translator* is a composition of n translators, and involves n–1 intermediate languages.

Example 2.9 Two-stage compilation

Given a Java-into-C translator and a C-into-x86 compiler, we can compose them to make a two-stage Java-into-x86 compiler, as shown below. The Java source program is translated into C, which is then compiled into x86 machine code.



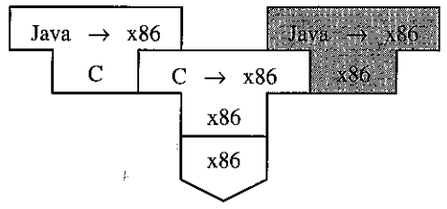
The two-stage compiler is functionally equivalent to a Java-into-x86 compiler.

□

A translator is itself a program, expressed in some language. As such, it can be translated into another language.

Example 2.10 Compiling a compiler

Suppose we have a Java-into-x86 compiler expressed in C. We cannot run this compiler at all, because it is not expressed in machine code. But we can treat it as an ordinary source program to be translated by a C-into-x86 compiler:



The object program is a Java-into-x86 compiler expressed in x86 machine code (shaded gray). We can now use this to compile Java programs, as illustrated in Example 2.6. □

More generally, all language processors are themselves programs, and as such can be manipulated by other language processors. For example, language processors can be translated (as in Example 2.10) or interpreted. We shall see the importance of this later in the chapter.

2.2 Interpreters

A compiler allows us to prepare a program to be run on a machine, by first translating the program into machine code. The program will then run at full machine speed. This method of working is not without disadvantages, however: the entire program must be translated before it can even start to run and produce results. In an interactive environment, *interpretation* is often a more attractive method of working. Thus we come to a new kind of language processor, an interpreter, that also allows us to run programs.

An ***interpreter*** is a program that accepts any program (the *source program*) expressed in a particular language (the *source language*), and runs that source program immediately.

An interpreter works by fetching, analyzing, and executing the source program instructions, *one at a time*. The source program starts to run and produce results as soon as the first instruction has been analyzed. The interpreter does *not* translate the source program into object code prior to execution.

Interpretation is sensible when most of the following circumstances exist:

- The programmer is working in interactive mode, and wishes to see the results of each instruction before entering the next instruction.
- The program is to be used once and then discarded (i.e., it is a 'throw-away' program), and therefore running speed is not very important.
- Each instruction is expected to be executed only once (or at least not very frequently).
- The instructions have simple formats, and thus can be analyzed easily and efficiently.

Interpretation is very slow. Interpretation of a source program, in a high-level language, can be up to 100 times slower than running an equivalent machine-code program. Therefore interpretation is not sensible when:

- The program is to be run in production mode, and therefore speed is important.
- The instructions are expected to be executed frequently.

- The instructions have complicated formats, and are therefore time-consuming to analyze. (This is the case in most high-level languages.)

Example 2.11 Interpreters

Here are some well-known examples of interpreters:

(a) A Basic interpreter: Basic has expressions and assignment commands like other high-level languages. But its control structures are low-level: a program is just a sequence of commands linked by conditional and unconditional jumps. A Basic interpreter fetches, analyzes, and executes one command at a time.

(b) A Lisp interpreter: Lisp is a very unusual language in that it assumes a common data structure (trees) for both code and data. Indeed, a Lisp program can manufacture new code at run-time! The Lisp program structure lends itself to interpretation. (See also Exercise 2.10.)

(c) The UNIX command language interpreter (*shell*): A UNIX user instructs the operating system by entering textual commands. The *shell* program reads each command, analyzes it to extract a command-name together with some arguments, and executes the command by means of a system call. The user can see the results of a command before entering the next one. The commands constitute a command language, and the *shell* is an interpreter for that command language.

(d) An SQL interpreter: SQL is a database query language. The user extracts information from the database by entering an SQL query, which is analyzed and executed immediately. This is done by an SQL interpreter within the database management system.

□

An interpreter is represented by a rectangular tombstone, as shown in Figure 2.6. The head of the tombstone names the interpreter's source language. The base of the tombstone (as usual) names the implementation language.

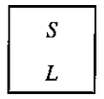


Figure 2.6 Tombstone representing an *S* interpreter expressed in language *L*.

Example 2.12 Tombstones representing interpreters

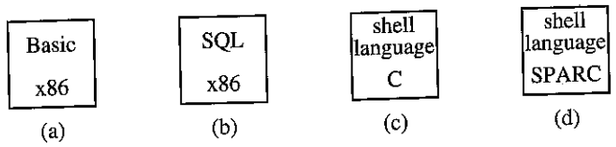
The following diagrams show how we represent:

(a) A Basic interpreter, expressed in x86 machine code.

(b) An SQL interpreter, expressed in x86 machine code.

(c) The UNIX shell (command language interpreter), expressed in C.

(d) The UNIX shell, expressed in SPARC machine code.



□

An *S* interpreter is itself a program, and can run on machine *M* only if it is expressed in machine code *M*. When the interpreter runs, it runs a source program *P*, which must be expressed in source language *S*. We say that *P* runs *on top of* the *S* interpreter. This is shown in Figure 2.7.

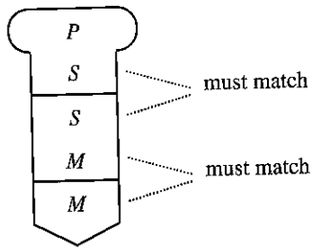


Figure 2.7 Interpreting a program *P* expressed in language *S*, using an *S* interpreter running on machine *M*.

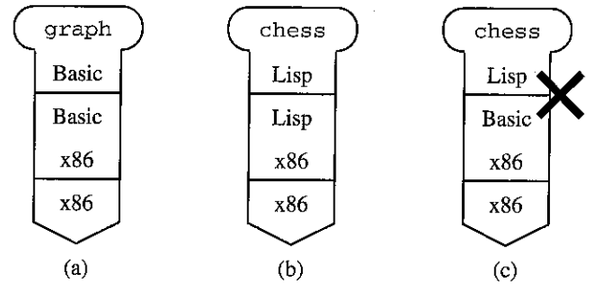
Example 2.13

The following diagrams show how we represent:

(a) Running program `graph` (expressed in Basic) on top of a Basic interpreter, which itself runs on an x86 machine.

(b) Running program `chess` (expressed in Lisp) on top of a Lisp interpreter, which itself runs on an x86 machine.

(c) Attempting to run program `chess` (expressed in Lisp) on top of a Basic interpreter. Of course, this will not work; the diagram clearly shows that the language in which the program is expressed does not match the interpreter's source language.



□

2.3 Real and abstract machines

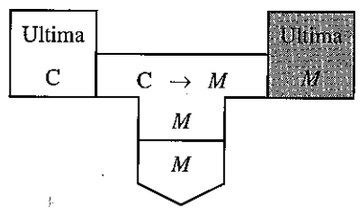
The interpreters mentioned in Example 2.12 were all for (relatively) high-level languages. But interpreters for low-level languages are also useful.

Example 2.14 Hardware emulation

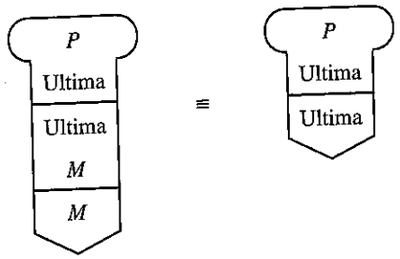
Suppose that a computer engineer has designed the architecture and instruction set of a radical new machine, Ultima. Now, actually constructing Ultima as a piece of hardware will be an expensive and time-consuming job. Modifying the hardware to implement design changes will likewise be costly. It would be wise to defer hardware construction until the engineer has somehow tested the design. But how can a paper design be tested?

There is a remarkably simple method that is both cheap and fast: we write an interpreter for Ultima machine code. E.g., we could write the interpreter in C:

We can now translate the interpreter into some machine code, say *M*, using the C compiler:



This gives us an Ultima interpreter expressed in M machine code (shaded gray above). Now we can run Ultima machine-code programs on top of the interpreter, which itself runs on M, as shown below left:



In all respects except speed, the effect is the same as running the programs on Ultima itself, as shown above right.

This kind of interpreter is often called an *emulator*. It cannot be used to measure the emulated machine's absolute speed, because interpretation slows everything down. But emulation can still be used to obtain useful quantitative information: counting memory cycles, estimating the degree of parallelism, and so on. It can also be used to obtain qualitative information about how well the architecture and instruction set meet the needs of programmers. □

Running a program on top of an interpreter is functionally equivalent to running the same program directly on a machine, as illustrated in Example 2.14. The user sees the same behavior in terms of the program's inputs and outputs. The two processes are even similar in detail: an interpreter works in a fetch–analyze–execute cycle, and a machine works in a fetch–decode–execute cycle. The only difference is that an interpreter is a software artifact, whereas a machine is a hardware artifact (and therefore much faster).

Thus a machine may be viewed as an interpreter implemented in hardware. Conversely, an interpreter may be viewed as a machine implemented by software. We sometimes call an interpreter an ***abstract machine***, as opposed to its hardware counterpart, which is a ***real machine***. An abstract machine is functionally equivalent to a real machine if they both implement the same language L. This is summarized in Figure 2.8.

A related observation is that there is no fundamental difference between machine codes and other low-level languages. By a machine code we just mean a language for which a hardware interpreter exists.

Figure 2.8 An abstract machine is functionally equivalent to a real machine.

2.4 Interpretive compilers

A compiler may take quite a long time to translate a source program into machine code, but then the object program will run at full machine speed. An interpreter allows the program to start running immediately, but it will run very slowly (up to 100 times more slowly than the machine-code program).

An ***interpretive compiler*** is a combination of compiler and interpreter, giving some of the advantages of each. The key idea is to translate the source program into an *intermediate language*, designed to the following requirements:

- it is intermediate in level between the source language and ordinary machine code;
- its instructions have simple formats, and therefore can be analyzed easily and quickly;
- translation from the source language into the intermediate language is easy and fast.

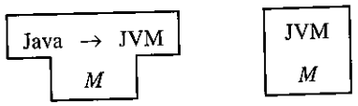
Thus an interpretive compiler combines fast compilation with tolerable running speed.

Example 2.15 Interpretive compilation

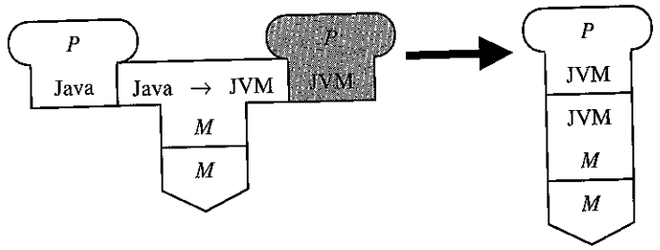
Sun Microsystems' Java Development Kit (JDK) is an implementation of an interpretive compiler for Java. At its heart is the Java Virtual Machine (JVM), a powerful abstract machine.

JVM-code is an intermediate language oriented to Java. It provides powerful instructions that correspond directly to Java operations such as object creation, method call, and array indexing. Thus translation from Java into JVM-code is easy and fast. Although powerful, JVM-code instructions have simple formats like machine-code instructions, with operation fields and operand fields, and so are easy to analyze. Thus JVM-code interpretation is relatively fast: 'only' about ten times slower than machine code.

JDK consists of a Java-into-JVM-code translator and a JVM-code interpreter, both of which run on some machine M:



A Java program P is first translated into JVM-code, and then the JVM-code object program is interpreted:



□

Interpretive compilers are very useful language processors. In the early stages of program development, the programmer might well spend more time compiling than running the program, since he or she is repeatedly discovering and correcting simple syntactic, contextual, and logical errors. At that stage fast compilation is more important than fast running, so an interpretive compiler is ideal. (Later, and especially when the program is put into production use, the program will be run many times but rarely recompiled. At that stage fast running will assume paramount importance, so a compiler that generates efficient machine code will be required. In Java, this problem is typically addressed by a so-called *just-in-time compiler*. See Section 2.8 and Exercise 2.7.)

2.5 Portable compilers

A program is *portable* to the extent that it can be (compiled and) run on any machine, without change. We can measure portability roughly by the proportion of code that remains unchanged when the program is moved to a dissimilar machine. Portability is an economic issue: a portable program is more valuable than an unportable one, because its development cost can be spread over more copies.

The language in which the program is expressed has a major impact on its portability. At one extreme, a program expressed in assembly language cannot be moved to a dissimilar machine unless it is completely rewritten, so its portability is 0%. A program expressed in a high-level language is much more portable. Ideally, it only needs to be recompiled when moved to a dissimilar machine, in which case its portability is 100%. However, this ideal is often quite elusive. For example, a program's behavior might be altered (perhaps subtly) by moving it to a machine with a different

character set or different arithmetic. Written with care, however, application programs expressed in high-level languages should achieve 95–99% portability.

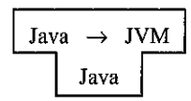
Similar points apply to language processors, which are themselves programs. Indeed, it is particularly important for language processors to be portable because they are especially valuable and widely-used programs. For this reason language processors are commonly written in high-level languages such as Pascal, C, and Java.

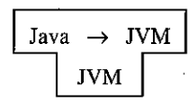
Unfortunately, it is particularly hard to make compilers portable. A compiler's function is to generate machine code for a particular machine, a function that is machine-dependent by its very nature. If we have a C-into-x86 compiler expressed in a high-level language, we should be able to move this compiler quite easily to run on a dissimilar machine, but it will still generate x86 machine code! To change the compiler to generate different machine code would require about half the compiler to be rewritten, implying that the compiler is only about 50% portable.

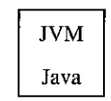
It might seem that highly portable compilers are unattainable. However, the situation is not quite so gloomy: a compiler that generates intermediate language is potentially much more portable than a compiler that generates machine code.

Example 2.16 A portable compiler kit

Consider the possibility of producing a portable Java compiler kit. Such a kit would consist of a Java-into-JVM-code translator, expressed both in Java and in JVM-code, and a JVM-code interpreter, expressed in Java:



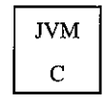




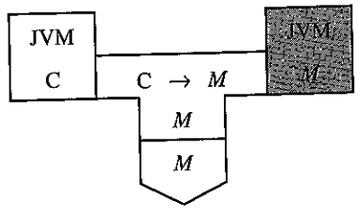
How can we make this work? It seems that we cannot compile Java programs until we have an implementation of JVM-code, and we cannot use the JVM-code interpreter until we can compile Java programs! Fortunately, a small amount of work can get us out of this chicken-and-egg situation.

Suppose that we want to get the system running on machine M, and suppose that we already have a compiler for a suitable high-level language, such as C, on this machine.

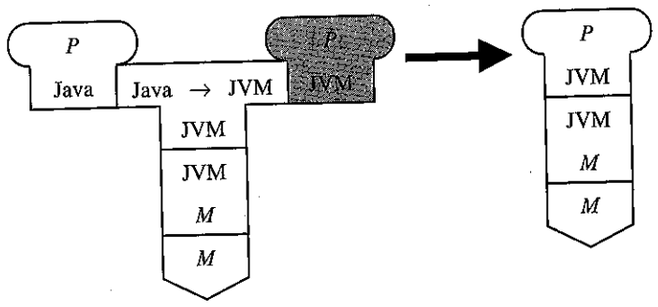
Then we rewrite the interpreter in C:



and then compile it:



Now we have an interpretive compiler, similar to the one described in Example 2.15. There is one difference: the compiler itself, being expressed in JVM-code, has to run on top of the JVM-code interpreter:



The JVM-code interpreter is much smaller and simpler than the compiler, so rewriting the interpreter is an easy job (a few days' work for an experienced programmer). Consequently, our example compiler kit as a whole would be about 95% portable. If no suitable high-level language is available, it is even feasible to rewrite the interpreter in assembly language.

Notice that the compiler expressed in Java is not actually needed to bootstrap the portable compiler. It would, however, be used to generate the compiler expressed in JVM-code. It would also prove to be useful in later development of the compiler after the initial move to machine *M*.

□

The Java compiler in Example 2.16 must be interpreted, so compilation of a Java source program will be slow. However, the compiler can be improved by bootstrapping, as we shall see in Section 2.6.1.

2.6 Bootstrapping

A language processor, such as a translator or interpreter, is a program that processes programs expressed in a particular language (the source language). The language processor is expressed in some implementation language.

Now suppose that the implementation language *is* the source language: the language processor can be used to process itself! This process is called ***bootstrapping***. The idea seems at first to be paradoxical, but it can be made to work. Indeed, it turns out to be extremely useful. In this section we study several kinds of bootstrapping.

2.6.1 Bootstrapping a portable compiler

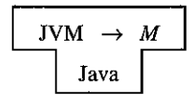
In Sections 2.4 and 2.5 we looked at interpretive and portable compilers. These work by translating from the high-level source language into an intermediate language, and then interpreting the latter.

A portable compiler can be bootstrapped to make a true compiler – one that generates machine code – by writing an intermediate-language-into-machine-code translator.

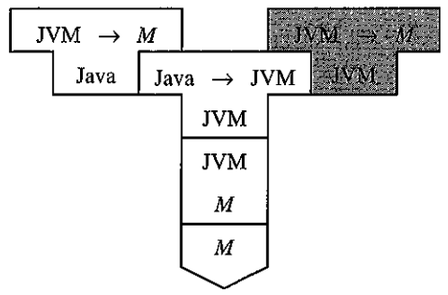
Example 2.17 Bootstrapping an interpretive compiler to generate machine code

Suppose that we have made a portable Java compiler kit into an interpretive compiler running on machine *M*, as described in Example 2.16. We can use this to build an efficient Java-into-*M* compiler, as follows.

First, we write a JVM-code-into-*M* translator, in Java:

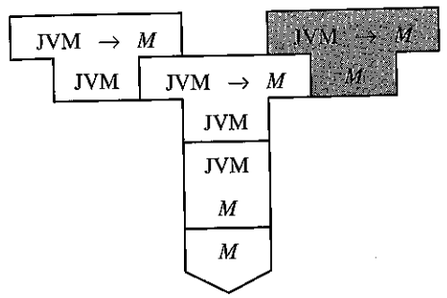


(This is a substantial job, but only about half as much work as writing a complete Java-into-*M* compiler.) Next, we compile this translator using the existing interpretive compiler:



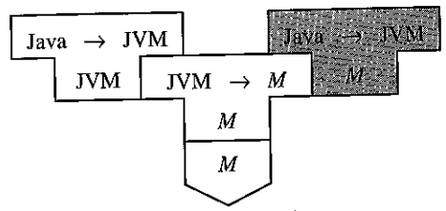
This gives a JVM-code-into-*M* translator expressed in JVM-code itself.

Next, we use this translator to translate *itself*:

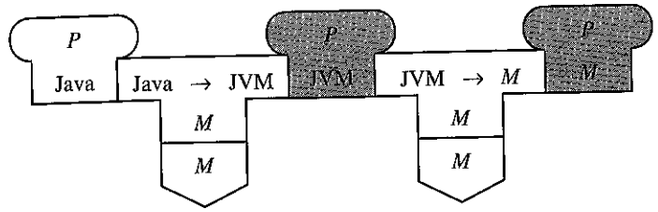


This, the actual bootstrap, gives a JVM-code-into-*M* translator expressed in machine code *M*.

Finally, we translate the Java-into-JVM-code translator into machine code:



Now we have implemented a two-stage Java-into-*M* compiler:



Moreover, the compiler is expressed in machine code, so compilation of a Java source program is much faster than in Example 2.16.

□

2.6.2 Full bootstrap

We have seen that a program, if it is to be portable, should be written in a suitable high-level language, *L*. That implies a commitment to the language *L* throughout the program's lifetime. If we wish to make a new version of the program (e.g., to remove known bugs, or to make it more efficient), we must edit the *L* source program and recompile it. In other words, the program is maintainable only as long as an *L* compiler is available.

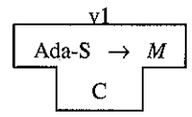
Exactly the same point applies to a language processor expressed in *L*. In Example 2.10, we saw how a Java compiler, expressed in C, could be translated into machine code by a C compiler (and thus enabled to run). However, this Java compiler can be maintained only as long as a C compiler is available. If we wish to make a new version of the Java compiler (e.g., to remove known bugs, or to generate better-quality machine code), we will need a C compiler to recompile the Java compiler.

In general, a compiler whose source language is *S*, expressed in a different high-level language *L*, can be maintained only as long as a compiler for *L* is available. This problem can be avoided by writing the *S* compiler in *S* itself! Whenever we make a new version of the *S* compiler, we use the old version to compile the new version. The only difficulty is how to get started: how can we compile the *first* version of the *S* compiler? The key idea is to start with a subset of *S* – a subset just large enough to be suitable for writing the compiler. The method is called ***full bootstrap*** – since a whole compiler is to be written from scratch.

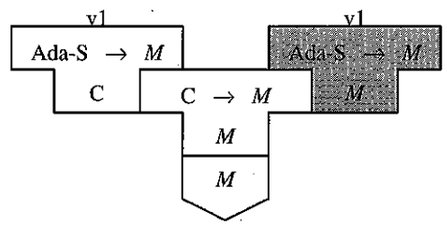
Example 2.18 Full bootstrap

Suppose that we wish to build an Ada compiler for machine *M*. Now Ada is a very large language, so it makes sense to build the compiler incrementally. We start by selecting a small subset of Ada that will be adequate for compiler writing. (The Pascal-like subset of Ada would be suitable.) Call this subset Ada-S.

We write version 1 of our Ada-S compiler in C (or any suitable language for which a compiler is currently available):

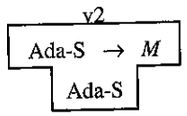


We compile version 1 using the C compiler:



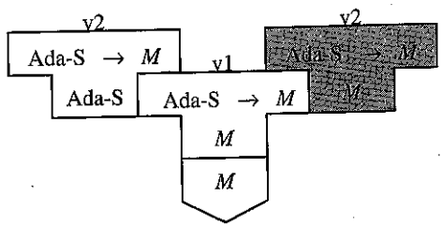
This gives an Ada-S compiler for machine *M*. We can test it by using it to compile and run Ada-S test programs.

But we prefer not to rely permanently on version 1 of the Ada-S compiler, because it is expressed in C, and therefore is maintainable only as long as a C compiler is available. Instead, we make version 2 of the Ada-S compiler, expressed in Ada-S itself:



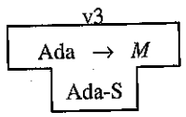
This rewriting of the compiler is not a hard job, because all the algorithms and data structures have already been developed and tested in version 1. (In fact, we could have wisely anticipated the rewriting, by refraining from using C features with no direct counterparts in Ada-S.)

Now we use version 1 to compile version 2:

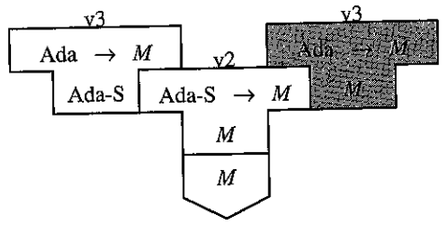


As usual, we can test version 2 of the Ada-S compiler by using it to compile and run Ada-S test programs. We have now broken our dependency on C, because the version-2 Ada-S compiler is expressed in Ada-S itself.

Finally, we extend the Ada-S compiler to a (full) Ada compiler, giving version 3:



and compile it using version 2:



This gives us an Ada compiler expressed in Ada itself. (Actually it is expressed in a subset of Ada, but that does not matter.) This compiler can be used to maintain itself by using version 3 to compile version 4, and so on.

□

2.6.3 Half bootstrap

Suppose that we have a compiler that runs on a machine *HM*, and generates *HM*'s machine code; now we wish to move the compiler to run on a dissimilar machine *TM*. In this transaction *HM* is called the *host machine*, and *TM* is called the *target machine*.

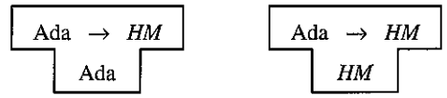
If the compiler is expressed in a high-level language for which we have a compiler on *TM*, just getting the compiler to run on *TM* is straightforward, but we would still have a compiler that generates *HM*'s machine code. It would, in fact, be a cross-compiler.

To make our compiler generate *TM*'s machine code, we have no choice but to rewrite part of the compiler. As we shall see in Chapter 3, one of the major parts of a compiler is the *code generator*, which does the actual translation into the target language. Typically the code generator is about half of the compiler. If our compiler has been constructed in a modular fashion, it is not too difficult to strip out the old code generator, which generated *HM*'s machine code; then we can substitute the new code generator, which will generate *TM*'s machine code.

If the compiler is expressed in its own source language, this process is called a ***half bootstrap*** – since roughly half the compiler must be modified. It does not depend on any compiler or assembler being already available on the target machine – indeed, it depends *only* on the host machine compiler!

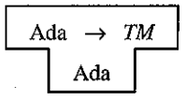
Example 2.19 Half bootstrap

Suppose that we have a Ada compiler that generates machine code for machine *HM*. The compiler is expressed in Ada itself, and in *HM*'s machine code:

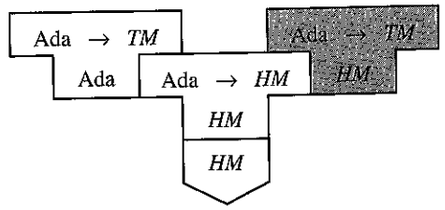


We wish to bootstrap this compiler to machine *TM*. To be precise, we want a compiler that runs on *TM* and generates *TM*'s machine code.

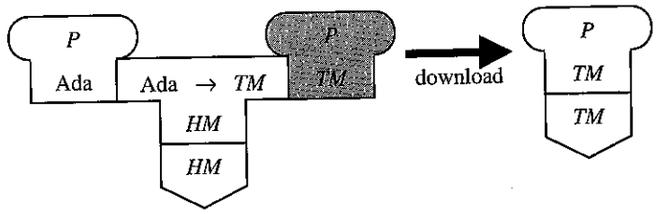
First, we modify the compiler's code generator to generate *TM*'s machine code:



We compile the modified compiler, using the original compiler, to obtain a cross-compiler:

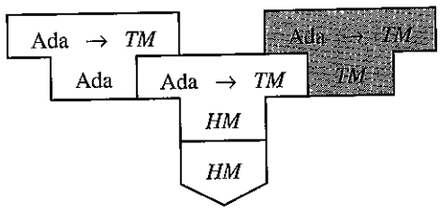


How do we test the cross-compiler? We can run it on *HM* to compile Ada test programs into *TM*'s machine code, and then download the object programs to *TM* to be run:



(Visual inspection of the object code is also a good idea, but practicable only for small test programs.)

Once we are satisfied that the cross-compiler is correct, we can use it to compile *itself* into *TM*'s machine code (the actual bootstrap):



Finally, we download the Ada-into-*TM* compiler (expressed in both Ada and *TM*'s machine code) to the target machine *TM*, and subsequently maintain it there.

□

2.6.4 Bootstrapping to improve efficiency

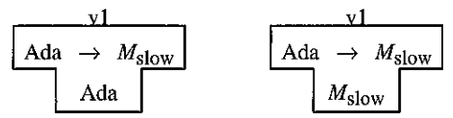
The efficiency of an ordinary program can be measured with respect to either time or space: how fast does it run, and how much storage space does it require?

When we discuss the efficiency of a compiler, the situation is more complicated. We can measure the efficiency of the compiler itself, and we can measure the efficiency of the object programs it generates.

In this chapter, we are not concerned with techniques for generating efficient object programs. But we can show that bootstrapping is a useful *strategy* for taking a simple compiler and upgrading it to generate more efficient object programs. The basic idea is to use the existing version of the compiler to compile the new version, and to do this repeatedly to make better and better versions.

Example 2.20 Bootstrapping to improve object code

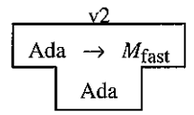
Suppose that we have an Ada compiler, version 1, that generates slow machine code. Version 1 is expressed in slow machine code, as well as in Ada:



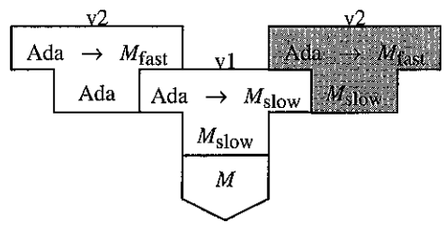
In the diagrams we will use notation like M_{fast} and M_{slow} to indicate fast and slow machine code, respectively. (Note that M_{fast} and M_{slow} are the same language, the machine code M; the subscripts are merely indications of code *quality*.)

When we compile Ada programs, both the version-1 compiler and its object programs will be slow. (Why?) Our objective is to make a fast compiler that generates fast object programs.

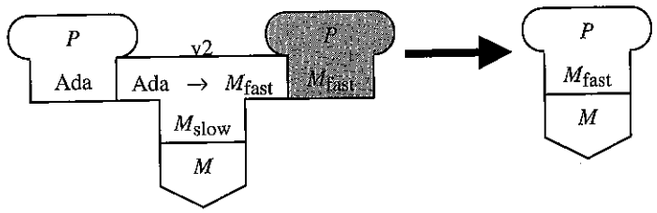
First, we modify version 1 to make a version-2 compiler that generates faster machine code:



We can use version 1 to compile version 2:

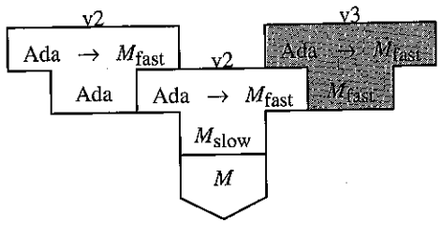


This gives us a better compiler, which we can use to compile Ada programs:

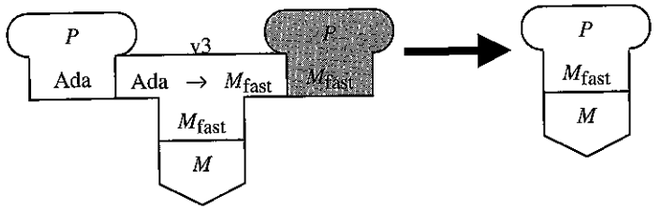


Compilation will still be slow (since the compiler is expressed in slow machine code), but the object program will be fast (since the generated machine code is fast).

The final stage of bootstrapping is to use version 2 to compile itself:



This gives us version 3, a fast compiler that generates fast object programs:



□

In practice, the bootstrapping steps illustrated in Example 2.20 would be used many times, as the compiler is gradually improved to generate better and better object code.

2.7 Case study: the Triangle language processor

The Triangle language processor will be used as a case study throughout this book. It consists of a compiler, an interpreter, and a disassembler. We will study how they work in the following chapters. Here we examine the Triangle language processor's overall structure. (See Figure 2.9.)

The compiler translates Triangle source programs into TAM code. *TAM* (Triangle Abstract Machine) is an abstract machine, implemented by an interpreter. TAM has been designed to facilitate the implementation of Triangle – although it would be equally suitable for implementing Algol, Pascal, and similar languages. Like JVM-code (Example 2.15), TAM's primitive operations are more similar to the operations of a high-level language than to the very primitive operations of a typical real machine. As a consequence, the translation from Triangle into TAM code is straightforward and fast.

The Triangle-into-TAM compiler and the TAM interpreter together constitute an interpretive compiler, much like the one described in Example 2.15. (See Exercise 2.2.) The TAM disassembler translates a TAM machine code program into *TAL* (Triangle Assembly Language). It is used to inspect the object programs produced by the Triangle-into-TAM compiler.

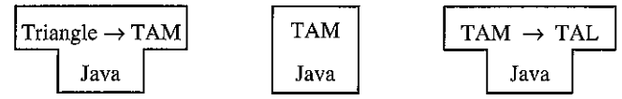


Figure 2.9 The compiler, interpreter, and disassembler components of the Triangle language processor.

2.8 Further reading

A number of authors have used tombstone diagrams to represent language processors and their interactions. The formalism was fully developed, complete with mathematical underpinnings, by Earley and Sturgis (1970). Their paper also presents an algorithm that systematically determines all the tombstones that can be generated from a given initial set of tombstones.

A case study of compiler development by full bootstrap may be found in Wirth (1971). A case study of compiler development by half bootstrap may be found in Welsh and Quinn (1972). Finally, a case study of compiler improvement by bootstrapping may be found in Ammann (1981). Interestingly, all these three case studies are interlinked: Wirth's Pascal compiler was the starting point for the other two developments.

Bootstrapping has a longer history, the basic idea being described by several authors in the 1950s. (At that time compiler development itself was still in its infancy!) The first well-known application of the idea seems to have been a program called `eval`, which was a Lisp interpreter expressed in Lisp itself (McCarthy *et al.* 1965).

Sun Microsystems' Java Development Kit (JDK) consists of a compiler that translates Java code to JVM code, a JVM interpreter, and a number of other tools. The compiler (`javac`) is written in Java itself, having been bootstrapped from an initial