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 i Figure 2.1. The head of the tombstone names the program *P*. The base of the tombston names the implementation language *L*, i.e., the language in which the program i expressed.

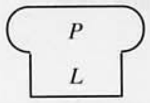


Figure 2.1 Tombstone representing a program *P* expressed in language *L*.

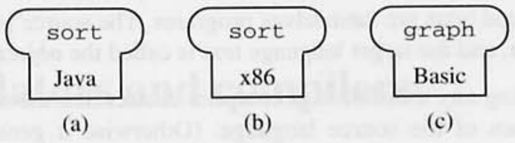
Example 2.2 Tombstone diagrams representing programs

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 ab breviate '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 represente by a pentagon inside which *M* is named, as shown in Figure 2.2.

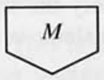


Figure 2.2 Tombstone representing a machine *M*.

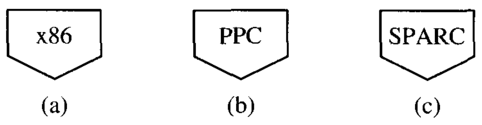
Example 2.3 Tombstone diagrams representing machines

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.

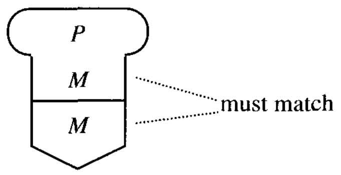


Figure 2.3 Running program *P* on machine *M*.

Example 2.4 Tombstone diagrams representing program execution

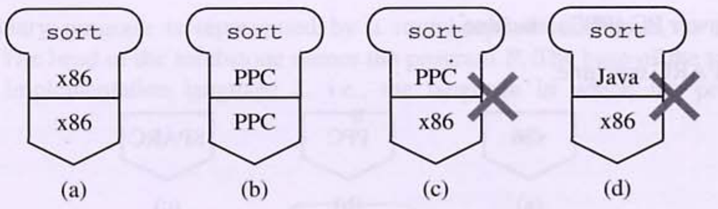
The following diagrams show how we represent:

(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]

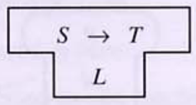


Figure 2.4 Tombstone representing an S-into-T translator expressed in language L.

Example 2.5 Tombstones representing translators

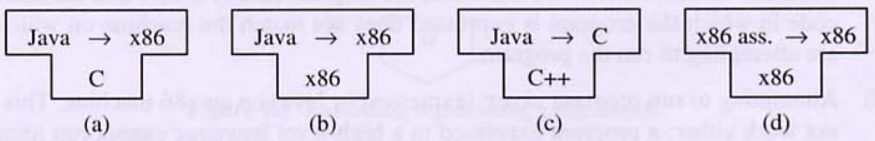
The following diagrams show how we represent:

(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.)

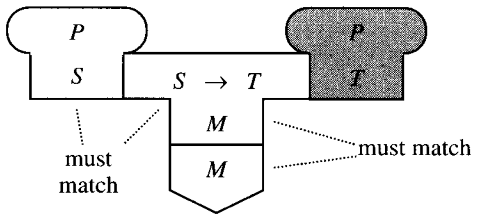
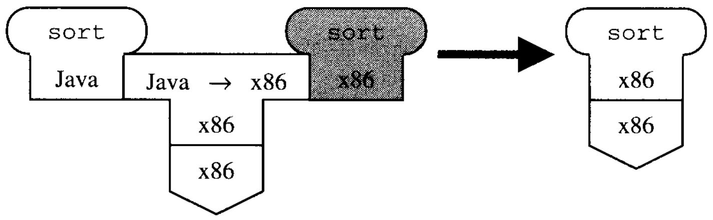


Figure 2.5 Translating a source program P expressed in language S to an object program expressed in language T, using an S-into-T translator running on machine M.

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.



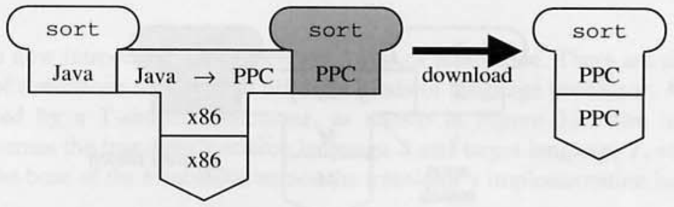
The second stage of the diagram shows the object program being run, also on an x86 machine.

□

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:

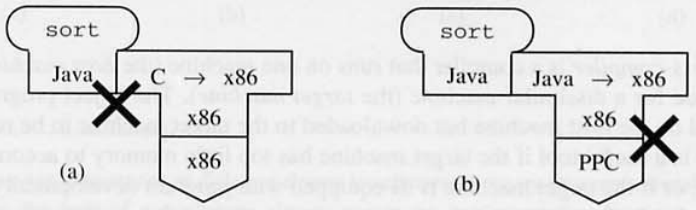
- A translator (like any other program) can run on a machine *M* only if it is expressed in machine code *M*.
- The source program must be expressed in the translator's source language *S*.
- The object program is expressed in the translator's target language *T*.
- The object program is semantically 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.)

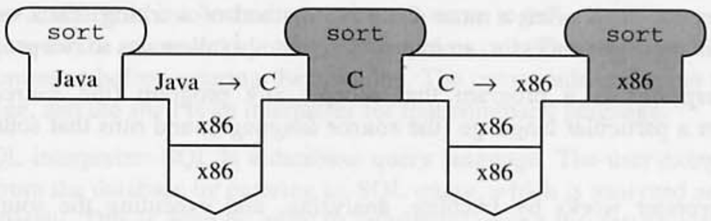
□

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.



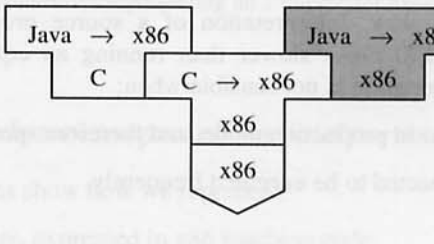
The two-stage compiler is functionally equivalent to a Java-into-x86 compiler.

□

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.

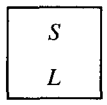


Figure 2.6 Tombstone representing an *S* interpreter expressed in language *L*.

Example 2.12 Tombstones representing interpreters

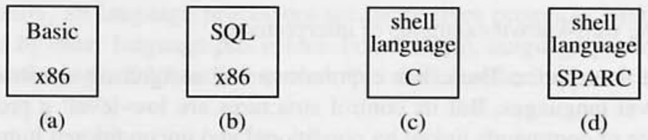
The following diagrams show how we represent:

(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.

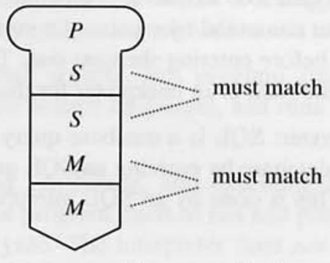


Figure 2.7 Interpreting a program *P* expressed in language *S*, using an *S* interpreter running on machine *M*.

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.