Chapter 1

OVERVIEW

1.1 OBJECTIVES OF TRANSLATION

A translator permits the programmer to express an algorithm in a language other than that of the machine which is to execute the algorithm. Why should the programmer select a language other than machine language? After all, in so doing he commits himself to the costs incurred in performing translation, and these costs would be obviated by programming in machine language. Nevertheless, the machine-language programmer is rare. The reason is that machine language is very ill-designed for human communication. This is not the fault of the machine designers. The purpose of machine language is to express algorithms in a form in which the machine can interpret them efficiently. Because automatic computers utilize two-state storage and switching devices almost exclusively, sequences of binary digits provide the most natural form for expressing instructions. Sequences of binary digits prove very unnatural, however, for humans to construct or to understand. Consequently, programmers prefer to express algorithms in a different form.

But it is more than personal preference which mandates the use of more expressive languages than those whose sentences are merely strings of zeros and ones. The greater ease of expression in other languages confers important benefits. One is increased accuracy of programming. Errors are easier to avoid and to detect both when a larger symbol set than $\{0,1\}$ is used, and when the program text is shorter, as is typically the case. Another benefit is increased programmer productivity: what can be written more easily can be written more rapidly. A particularly important benefit is the relaxation of the restriction that the programmer be intimately familiar with the computer to be used. If a language is available in which concepts are expressed in terms related not to the computer, but rather to the problem which the user is trying to solve, then the power of the computer can be placed at the disposal of workers who are not computer experts.

Moreover, if the language is independent of the computer, programs can be transported between computers which have different machine languages. This leads to further increase in programmer productivity and to the advantages of exchanging programs. The cost, for any number of programs, is limited to the construction and use of one translator for each desired machine language.

The input to a translation program is expressed in a *source* language. It may represent either a nonalgorithmic specification of what the computer is to accomplish or an algorithmic specification of the steps in that accomplishment. The result of performing the translation may be the execution of the required algorithm (whether specified explicitly or not) on a machine whose language, the *host* language, is other than the source. Alternatively, the translator may produce as output a representation of the algorithm in a *target* language. That target language is often the machine language of some computer, which is then able to execute the algorithm. Sometimes a non-machine language is chosen as target, typically a language which is itself the source language for another translator, which then yields the desired result. If the target language is machine language, the translator output is often referred to as *machine code.*

1.2 SOURCE LANGUAGES

Algorithmic source languages differ in the degree to which they reflect the structure of the target machine. Those in one group are designed to permit the programmer to control machine operation in detail. They necessarily reflect the machine structure explicitly, and are known as *assembler* languages. A given assembler language is intended for use with a single machine design. Assembler language differs from the binary machine language chiefly in that operations and their operands can be referred to symbolically, without concern for actual encodings or numeric addresses.

Representation of an algorithm in machine-independent terms is considered to be at a higher level of abstraction than is offered by assembler languages. The most common designation for languages which permit algorithms to be so expressed is *programming languages*. Sometimes they are called "user-oriented" to distinguish them from the *machine-oriented* assembler languages. Among the best known languages of this group are Fortran, COBOL, Algol, PL/I, Pascal, and APL. These programming languages are characterized by more powerful primitive operations and more powerful control structures than are available in machine language or in assembler language. An example of their operations is reading a record

from a file specified only by name; another is exponentiation. Typical control structures include repetition of a group of instructions until a specified condition is satisfied. Often, but not necessarily, these languages are designed to resemble a natural language, such as English, more closely than could an assembler language.

Problem-oriented is sometimes used as a synonym for "user-oriented", but often it connotes instead a degree of specialization toward a particular problem area. The languages LISP for list processing and SNOBOL for character string manipulation could be placed here, along with COGO for civil engineering or APT for machine-tool control.

Nonalgorithmic source languages differ widely in their areas of application. Examples include specifications of reporting programs and of sorting programs. The former specifications include descriptions of data fields and report formats; the latter include descriptions of data records, keys, and desired sequences. In many operating systems, the command language provides another example of a nonalgorithmic source language.

1.3 TRANSLATION MECHANISMS

In translating an algorithm, whether into direct execution on the host machine or into a target-language program for later execution on the target machine, it is necessary for the translator both to analyze and to synthesize. It must analyze the source-language representation of the algorithm to determine what actions are ultimately to be performed. It must also synthesize those actions, either into direct performance or into a target-language representation.

Source-language analysis incorporates three stages: lexical, syntactic, and semantic. *Lexical* analysis is the determination of what symbols of the language are represented by the characters in the source-language text. For many programming languages, lexical analysis would classify "PRESSURE" as an identifier, "13" as an integer, "(" and "<=" as special symbols, "'LANGUAGE'" as a character string, and ".314159E+01" as a so-called "real" number. Examination of the text to identify and classify symbols is known as *scanning*, and lexical analyzers are therefore often called *scanners*.

Syntactic analysis is the determination of the structure of the source-language representation. If the source-language text is that of an assembler language with a fixed format for each instruction, this analysis could be trivial. In a particular assembler language the symbol in card columns 10–14 might, for example, always specify the operation code of the machine-language instruction to be performed. If the format is not so constrained,

or if the source-language is more complex, syntactic analysis poses more of a challenge. Lexical analysis of the programming-language program fragment "A + B * C" would establish that it is composed of three identifiers separated by two special symbols. It would probably also determine that the special symbols represent binary operations. The major task in syntactic analysis of the fragment would be to determine whether the operation represented by "+" was to be performed before or after that represented by "*". In making the determination, the syntactic analyzer would apply a specification of the rules for constructing symbol strings of the programming language. This set of rules is known as the *grammar* for the language. Application of a grammar to determining syntactic structure is known as *parsing*, and syntactic analyzers are therefore often called *parsers.*

Semantic analysis determines the *meaning* of the source-language program in the sense that it identifies the actions specified by the program. For the string "A + B * C" semantic analysis would determine what particular actions are specified by "+" and by "*". Although semantic analysis is, for reasons we shall discuss later, often performed in conjunction with syntactic analysis, it is conceptually a different process.

The synthesis of action by the translation program involves one of two mechanisms. *Interpretation* is the direct performance of the actions identified in the process of analyzing the source-language program. For each possible action there exists a host-language subroutine to perform it. Interpretation requires the proper subroutine to be called at the right time with the appropriate parameters. *Generation* is the creation of target-language code to perform at a later time each action identified by analysis of the source-language program. Appropriate parameters assist in shaping the code sequence to be produced. Many persons restrict the term "generation", using it only for source languages nearer the machine-oriented end of the spectrum, and apply "compilation" to those more user-oriented.

1.4 TYPES OF TRANSLATORS

Translators which synthesize actions by interpretation are called *interpreters*. Although we have been discussing translation *programs*, it is important to observe that interpreters may be made of hardware as well as software. In fact, every computer is an interpreter of its own machine language, because it translates machine-language instructions into actions. As a result, every sequence of program translations includes a hardware interpreter at the end, even if no software interpretation precedes final execution. For reasons which will become clear when translation mechanisms are later

examined carefully, interpretation is easier than generation. Consequently, interpreters are usually the easiest translators to write, but they tend also to result in much slower execution than do generative translators.

It would seem natural to designate as "generators" those programs which translate by generation. The term is indeed used, but it is customarily restricted to programs, such as report program generators (RPG) and sort generators, whose input is nonalgorithmic. Of the generative translators from algorithmic source language, the most important are assemblers, compilers, linkers, and loaders. For the simplest *assembler*, the target language is machine language and its source language has instructions in one-to-one correspondence with those of machine language, but with symbolic names for both operations and operands. The translator just converts each assembler-language instruction into the corresponding machine-language instruction, collecting those instructions into a program. Less elementary assemblers translate the program into a target-language form which permits the program to be combined with other programs before execution.

Many programs contain sequences of instructions which are repeated in either identical or nearly identical form. The repetitious writing of such sequences is obviated by the *macro processor*, which allows a sequence of source-language code to be defined once and then referred to by name each time it is to be assembled. Each reference, which may use parameters to introduce controlled variation, results in the generation of source-language text for a subsequent translation. *Conditional* macro processors provide for the conditional performance of part of the translation.

The *compiler* translates from a programming language as source to a machine-oriented language as target. Each source-language instruction is usually translated into several target-language instructions.

The *linker* (also "binder", "consolidator", or "linkage editor") takes as input independently translated programs whose original source-language representations include symbolic references to each other. Its task is to resolve these symbolic references and produce a single program. There is typically little difference between the linker's source and target languages.

The *loader* takes a program produced by assembler, compiler, or linker, and prepares that program to be executed when ultimately resident in a particular set of physical main storage locations. The loader's target language is machine language; its source language is nearly machine language.

Loading is intimately bound with the storage management function of operating systems, and is usually performed later than assembly and compilation. Some actions in linking must be deferred until load time and others may be deferred until execution. It is therefore convenient to classify

linkers and loaders as control programs. This they are, but they are translators as well and interact closely with assemblers and compilers. It is instructive to study linkers and loaders in both the operating system context and the translator context.

1.5 HISTORICAL NOTES

The earliest computers, even those considered large at the time, executed single programs written in raw machine language. To keep track of storage use, the programmer customarily prepared by hand a "memory map" on which he wrote symbolic names for the variables whose values occupied the corresponding locations. Computers became larger and faster. When main storage reached a thousand words or so, the memory map became too unwieldy. Moreover, it was much easier to think of symbolic operation codes, such as "LOAD" and "ADD", rather than of the decimal or even binary numbers used to represent them in machine language. These needs led to the elementary symbolic assembler. Another early development was the *absolute* loader, a program which would take a machine-language program, whether prepared by the programmer or produced by an assembler, read it into main storage, and transfer control to it.

Within a few years, two major new features appeared. One was designed for computers whose main storage was on magnetic drum, for which the maximum random access time greatly exceeded the minimum, by a factor of perhaps 50. The *optimizing* assembler assigned to data and instructions those storage locations which would minimize execution time. Because the main storage media in current use present a uniform random access time, this feature is no longer needed. The second feature, provision of macro generation and conditional assembly, soon extended the power of assemblers. Unlike the first, this feature was not limited to a particular hardware design, and is still in widespread use.

A nearly parallel development was due to the rather limited instruction repertoires of early machines. Instructions which were desired, but not provided in the hardware, were incorporated into a language which could be used instead of machine language or the similarly circumscribed assembler language. Since the hardware was unable to interpret programs in the resulting language, interpretation was performed by programs, called *interpreters*. Among the most popular features provided by the early interpreters were three-address instruction formats and floating-point arithmetic operations.

Not only the floating-point arithmetic subroutines, but other subroutines,

both arithmetic and nonarithmetic, constituted growing libraries of subroutines available to all users of a system. If the subroutines were written to be executed while occupying fixed locations in storage, then their use was highly constrained, because not more than one program can occupy a given storage location. Added to this was the problem that the user program which called the subroutines needed storage of its own. The conflicting storage requirements were resolved by producing both the system subroutines and the assembler output in a form in which addresses were specified not absolutely, but only relative to the start of the program. A *relocating loader* would read the program into whatever storage locations were available, and then insert the correct absolute addresses which corresponded to the starting location chosen.

Prior to loading, another step became convenient. This was the linking of user program to subroutines, or of separately written or translated user programs to each other, resulting in a single program for loading. Symbolic names defined in one of the modules to be linked could be referenced in another module which was prepared separately without access to the definition. The required resolution of intermodule symbolic references was provided by linkers.

The poor execution efficiency of interpreters led to a desire to perform generative translation, yet without losing the extended source-language capabilities of existing interpreters. This most difficult of the language translation requirements was met by the compiler. In fact, the earliest successful compiler particularly stressed target-language efficiency to enhance its chances of adoption. For quite a few years compilers were limited to problem-oriented languages designed for restricted fields of application, usually either business or numeric calculation. More widely applicable languages subsequently became available, and so have the corresponding compilers.

1.6 FUNCTION AND IMPLEMENTATION

In describing software, it is important to distinguish, as it is for hardware, among *architecture, implementation,* and *realization**. The architecture, or *function*, of a program is *what* the program does, as specified in its external description. The implementation is *how* the program performs that function, how it is organized internally. The realization is the *embodi-*

*Brooks [1975, p. 49] discusses the application to software engineering of this important insight due to Blaauw [1966, 1970].

ment of that organization in the particular language in which it is written and on the particular machine on which it runs.

The realization of translators is intimately bound with techniques of programming, and is dependent upon the details of specific languages and machines. To avoid excessive dependence on those matters, we shall concentrate on function and implementation. The choice of function does not dictate the choice of implementation. As for most programs, there is more than one way to implement the desired function. Indeed, different philosophies of translator design emphasize characteristics which result in widely varying implementations. Among the characteristics which influence implementation, or are consequences of the implementation, are the following.

Translation speed
Translator size
Simplicity
Generality
Ease of debugging in source language
Target-language code speed
Target-language code size

The various characteristics cannot in general be established independently; for example, efficiency of compiled code is usually obtained at the expense of compiler time and space.

In the following chapters we shall examine both the functions of the principal types of translators, and the details of how generation and interpretation are performed. In discussing implementations, we shall not attempt to cover all possible approaches, but shall concentrate instead on typical forms of implementation. This will be particularly true for compilers. Because their function is the most complex of all the translators, compiler implementation shows the widest variety. Entire large books have been written on compiler design, and the material on compilers in this book must necessarily be illustrative rather than exhaustive.

FOR FURTHER STUDY

Two good overviews of system software, both considerably more detailed than this opening chapter, are Chapter 3 of Gear [1974] and Chapter 12 of Freeman [1975]. For the early history of translators, good sources are Knuth [1962] and Rosen [1964, 1967a, 1969].