1 INTRODUCTION

1.1 Objectives

The use of computer languages is an essential link in the chain between human and computer. In this text we hope to make the reader more aware of some aspects of

- Imperative programming languages - their syntactic and semantic features; the ways of specifying syntax and semantics; problem areas and ambiguities; the power and usefulness of various features of a language.

- Translators for programming languages - the various classes of translator (assemblers, compilers, interpreters); implementation of translators.

- Compiler generators - tools that are available to help automate the construction of translators for programming languages.

This book is a complete revision of an earlier one published by Addison-Wesley (Terry, 1986). It has been written so as not to be too theoretical, but to relate easily to languages which the reader already knows or can readily understand, like Pascal, Modula-2, C or C++. The reader is expected to have a good background in one of those languages, access to a good implementation of it, and, preferably, some background in assembly language programming and simple machine architecture. We shall rely quite heavily on this background, especially on the understanding the reader should have of the meaning of various programming constructs.

Significant parts of the text concern themselves with case studies of actual translators for simple languages. Other important parts of the text are to be found in the many exercises and suggestions for further study and experimentation on the part of the reader. In short, the emphasis is on "doing" rather than just "reading", and the reader who does not attempt the exercises will miss many, if not most, of the finer points.

The primary language used in the implementation of our case studies is C++ (Stroustrup, 1990). Machine readable source code for all these case studies is to be found on the IBM-PC compatible diskette that is included with the book. As well as C++ versions of this code, we have provided equivalent source in Modula-2 and Turbo Pascal, two other languages that are eminently suitable for use in a course of this nature. Indeed, for clarity, some of the discussion is presented in a pseudo-code that often resembles Modula-2 rather more than it does C++. It is only fair to warn the reader that the code extracts in the book are often just that - extracts - and that there are many instances where identifiers are used whose meaning may not be immediately apparent from their local context. The conscientious reader will have to expend some effort in browsing the code. Complete source for an assembler and interpreter appears in the appendices, but the discussion often revolves around simplified versions of these programs that are found in their entirety only on the diskette.

1.2 Systems programs and translators

Users of modern computing systems can be divided into two broad categories. There are those who never develop their own programs, but simply use ones developed by others. Then there are those who are concerned as much with the development of programs as with their subsequent use. This latter group - of whom we as computer scientists form a part - is fortunate in that program development is usually aided by the use of high-level languages for expressing algorithms, the use of interactive editors for program entry and modification, and the use of sophisticated job control languages or graphical user interfaces for control of execution. Programmers armed with such tools have a very different picture of computer systems from those who are presented with the hardware alone, since the use of compilers, editors and operating systems - a class of tools known generally as **systems programs** - removes from humans the burden of developing their systems at the machine level. That is not to claim that the use of such tools removes all burdens, or all possibilities for error, as the reader will be well aware.

Well within living memory, much program development was done in machine language - indeed, some of it, of necessity, still is - and perhaps some readers have even tried this for themselves when experimenting with microprocessors. Just a brief exposure to programs written as almost meaningless collections of binary or hexadecimal digits is usually enough to make one grateful for the presence of high-level languages, clumsy and irritating though some of their features may be.

However, in order for high-level languages to be usable, one must be able to convert programs written in them into the binary or hexadecimal digits and bitstrings that a machine will understand. At an early stage it was realized that if constraints were put on the syntax of a high-level language the translation process became one that could be automated. This led to the development of **translators** or **compilers** - programs which accept (as data) a textual representation of an algorithm expressed in a **source language**, and which produce (as primary output) a representation of the same algorithm expressed in another language, the **object** or **target language**.

Beginners often fail to distinguish between the compilation (*compile-time*) and execution (*run-time*) phases in developing and using programs written in high-level languages. This is an easy trap to fall into, since the translation (compilation) is often hidden from sight, or invoked with a special function key from within an integrated development environment that may possess many other magic function keys. Furthermore, beginners are often taught programming with this distinction deliberately blurred, their teachers offering explanations such as "when a computer executes a *read* statement it reads a number from the input data into a variable". This hides several low-level operations from the beginner. The underlying implications of file handling, character conversion, and storage allocation are glibly ignored - as indeed is the necessity for the computer to be programmed to understand the word *read* in the first place. Anyone who has attempted to program input/output (I/O) operations directly in assembler languages will know that many of them are non-trivial to implement.

A translator, being a program in its own right, must itself be written in a computer language, known as its **host** or **implementation language**. Today it is rare to find translators that have been developed from scratch in machine language. Clearly the first translators had to be written in this way, and at the outset of translator development for any new system one has to come to terms with the machine language and machine architecture for that system. Even so, translators for new machines are now invariably developed in high-level languages, often using the techniques of **cross-compilation** and **bootstrapping** that will be discussed in more detail later.

The first major translators written may well have been the Fortran compilers developed by Backus

and his colleagues at IBM in the 1950's, although machine code development aids were in existence by then. The first Fortran compiler is estimated to have taken about 18 person-years of effort. It is interesting to note that one of the primary concerns of the team was to develop a system that could produce object code whose efficiency of execution would compare favourably with that which expert human machine coders could achieve. An automatic translation process can rarely produce code as optimal as can be written by a really skilled user of machine language, and to this day important components of systems are often developed at (or very near to) machine level, in the interests of saving time or space.

Translator programs themselves are never completely portable (although parts of them may be), and they usually depend to some extent on other systems programs that the user has at his or her disposal. In particular, input/output and file management on modern computer systems are usually controlled by the **operating system**. This is a program or suite of programs and routines whose job it is to control the execution of other programs so as best to share resources such as printers, plotters, disk files and tapes, often making use of sophisticated techniques such as parallel processing, multiprogramming and so on. For many years the development of operating systems required the use of programming languages that remained closer to the machine code level than did languages suitable for scientific or commercial programming. More recently a number of successful higher level languages have been developed with the express purpose of catering for the design of operating systems and real-time control. The most obvious example of such a language is C, developed originally for the implementation of the UNIX operating system, and now widely used in all areas of computing.

1.3 The relationship between high-level languages and translators

The reader will rapidly become aware that the design and implementation of translators is a subject that may be developed from many possible angles and approaches. The same is true for the design of programming languages.

Computer languages are generally classed as being "high-level" (like Pascal, Fortran, Ada, Modula-2, Oberon, C or C++) or "low-level" (like ASSEMBLER). High-level languages may further be classified as "imperative" (like all of those just mentioned), or "functional" (like Lisp, Scheme, ML, or Haskell), or "logic" (like Prolog).

High-level languages are claimed to possess several advantages over low-level ones:

- *Readability:* A good high-level language will allow programs to be written that in some ways resemble a quasi-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting, a highly desirable property when one considers that many programs are written once, but possibly studied by humans many times thereafter.

- *Portability:* High-level languages, being essentially machine independent, hold out the promise of being used to develop portable software. This is software that can, in principle (and even occasionally in practice), run unchanged on a variety of different machines - provided only that the source code is recompiled as it moves from machine to machine.

  To achieve machine independence, high-level languages may deny access to low-level features, and are sometimes spurned by programmers who have to develop low-level machine dependent systems. However, some languages, like C and Modula-2, were specifically designed to allow access to these features from within the context of high-level constructs.

- *Structure and object orientation:* There is general agreement that the structured programming movement of the 1960's and the object-oriented movement of the 1990's have resulted in a great improvement in the quality and reliability of code. High-level languages can be designed so as to encourage or even subtly enforce these programming paradigms.

- *Generality:* Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.

- *Brevity:* Programs expressed in high-level languages are often considerably shorter (in terms of their number of source lines) than their low-level equivalents.

- *Error checking:* Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages - or at least their implementations - can, and often do, enforce a great deal of error checking both at compile-time and at run-time. For this they are, of course, often criticized by programmers who have to develop time-critical code, or who want their programs to abort as quickly as possible.

These advantages sometimes appear to be over-rated, or at any rate, hard to reconcile with reality. For example, readability is usually within the confines of a rather stilted style, and some beginners are disillusioned when they find just how unnatural a high-level language is. Similarly, the generality of many languages is confined to relatively narrow areas, and programmers are often dismayed when they find areas (like string handling in standard Pascal) which seem to be very poorly handled. The explanation is often to be found in the close coupling between the development of high-level languages and of their translators. When one examines successful languages, one finds numerous examples of compromise, dictated largely by the need to accommodate language ideas to rather uncompromising, if not unsuitable, machine architectures. To a lesser extent, compromise is also dictated by the quirks of the interface to established operating systems on machines. Finally, some appealing language features turn out to be either impossibly difficult to implement, or too expensive to justify in terms of the machine resources needed. It may not immediately be apparent that the design of Pascal (and of several of its successors such as Modula-2 and Oberon) was governed partly by a desire to make it easy to compile. It is a tribute to its designer that, in spite of the limitations which this desire naturally introduced, Pascal became so popular, the model for so many other languages and extensions, and encouraged the development of superfast compilers such as are found in Borland's Turbo Pascal and Delphi systems.

The design of a programming language requires a high degree of skill and judgement. There is evidence to show that one's language is not only useful for expressing one's ideas. Because language is also used to formulate and develop ideas, one's knowledge of language largely determines *how* and, indeed, *what* one can think. In the case of programming languages, there has been much controversy over this. For example, in languages like Fortran - for long the *lingua franca* of the scientific computing community - recursive algorithms were "difficult" to use (not impossible, just difficult!), with the result that many programmers brought up on Fortran found recursion strange and difficult, even something to be avoided at all costs. It is true that recursive algorithms are sometimes "inefficient", and that compilers for languages which allow recursion may exacerbate this; on the other hand it is also true that some algorithms are more simply explained in a recursive way than in one which depends on explicit repetition (the best examples probably being those associated with tree manipulation).

There are two divergent schools of thought as to how programming languages should be designed. The one, typified by the Wirth school, stresses that languages should be small and understandable,

and that much time should be spent in consideration of what tempting features might be omitted without crippling the language as a vehicle for system development. The other, beloved of languages designed by committees with the desire to please everyone, packs a language full of every conceivable potentially useful feature. Both schools claim success. The Wirth school has given us Pascal, Modula-2 and Oberon, all of which have had an enormous effect on the thinking of computer scientists. The other approach has given us Ada, C and C++, which are far more difficult to master well and extremely complicated to implement correctly, but which claim spectacular successes in the marketplace.

Other aspects of language design that contribute to success include the following:

- *Orthogonality:* Good languages tend to have a small number of well thought out features that can be combined in a logical way to supply more powerful building blocks. Ideally these features should not interfere with one another, and should not be hedged about by a host of inconsistencies, exceptional cases and arbitrary restrictions. Most languages have blemishes - for example, in Wirth's original Pascal a function could only return a scalar value, not one of any structured type. Many potentially attractive extensions to well-established languages prove to be extremely vulnerable to unfortunate oversights in this regard.

- *Familiar notation:* Most computers are "binary" in nature. Blessed with ten toes on which to check out their number-crunching programs, humans may be somewhat relieved that high-level languages usually make decimal arithmetic the rule, rather than the exception, and provide for mathematical operations in a notation consistent with standard mathematics. When new languages are proposed, these often take the form of derivatives or dialects of well-established ones, so that programmers can be tempted to migrate to the new language and still feel largely at home - this was the route taken in developing C++ from C, Java from C++, and Oberon from Modula-2, for example.

Besides meeting the ones mentioned above, a successful modern high-level language will have been designed to meet the following additional criteria:

- *Clearly defined:* It must be clearly described, for the benefit of both the user and the compiler writer.

- *Quickly translated:* It should admit quick translation, so that program development time when using the language is not excessive.

- *Modularity:* It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.

- *Efficient:* It should permit the generation of efficient object code.

- *Widely available:* It should be possible to provide translators for all the major machines and for all the major operating systems.

The importance of a clear language description or specification cannot be over-emphasized. This must apply, firstly, to the so-called **syntax** of the language - that is, it must specify accurately what form a source program may assume. It must apply, secondly, to the so-called **static semantics** of the language - for example, it must be clear what constraints must be placed on the use of entities of differing types, or the scope that various identifiers have across the program text. Finally, the

specification must also apply to the **dynamic semantics** of programs that satisfy the syntactic and static semantic rules - that is, it must be capable of predicting the effect any program expressed in that language will have when it is executed.

Programming language description is extremely difficult to do accurately, especially if it is attempted through the medium of potentially confusing languages like English. There is an increasing trend towards the use of formalism for this purpose, some of which will be illustrated in later chapters. Formal methods have the advantage of precision, since they make use of the clearly defined notations of mathematics. To offset this, they may be somewhat daunting to programmers weak in mathematics, and do not necessarily have the advantage of being very concise - for example, the informal description of Modula-2 (albeit slightly ambiguous in places) took only some 35 pages (Wirth, 1985), while a formal description prepared by an ISO committee runs to over 700 pages.

Formal specifications have the added advantage that, in principle, and to a growing degree in practice, they may be used to help automate the implementation of translators for the language. Indeed, it is increasingly rare to find modern compilers that have been implemented without the help of so-called **compiler generators**. These are programs that take a formal description of the syntax and semantics of a programming language as input, and produce major parts of a compiler for that language as output. We shall illustrate the use of compiler generators at appropriate points in our discussion, although we shall also show how compilers may be crafted by hand.

Exercises

1.1 Make a list of as many translators as you can think of that can be found on your computer system.

1.2 Make a list of as many other systems programs (and their functions) as you can think of that can be found on your computer system.

1.3 Make a list of existing features in your favourite (or least favourite) programming language that you find irksome. Make a similar list of features that you would like to have seen added. Then examine your lists and consider which of the features are probably related to the difficulty of implementation.

Further reading

As we proceed, we hope to make the reader more aware of some of the points raised in this section. Language design is a difficult area, and much has been, and continues to be, written on the topic. The reader might like to refer to the books by Tremblay and Sorenson (1985), Watson (1989), and Watt (1991) for readable summaries of the subject, and to the papers by Wirth (1974, 1976a, 1988a), Kernighan (1981), Welsh, Sneeringer and Hoare (1977), and Cailliau (1982). Interesting background on several well-known languages can be found in *ACM SIGPLAN Notices* for August 1978 and March 1993 (Lee and Sammet, 1978, 1993), two special issues of that journal devoted to the history of programming language development. Stroustrup (1993) gives a fascinating exposition of the development of C++, arguably the most widely used language at the present time. The terms "static semantics" and "dynamic semantics" are not used by all authors; for a discussion on this point see the paper by Meek (1990).