WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

AN INTRODUCTION TO COMPUTER SOFTWARE

by

DAVID N. NESS*

419-69

September, 1969

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139
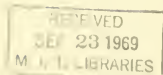
AN INTRODUCTION TO COMPUTER SOFTWARE

by

DAVID N. NESS*

419-69

September, 1969

*Assistant Professor, Sloan School of Management, M.I.T.

Dewey

AN INTRODUCTION TO COMPUTER SOFTWARE

A computer program is a <u>procedure</u> (or algorithm) for performing some computation. This procedure must be written in, or translatable into the <u>machine language</u> of a given machine for us to be able to perform (or "run") the program on that machine.

In our discussion here it will sometimes prove useful to be able to draw an analogy between human (or natural) languages and machine languages. In doing so we must remember not to stretch the analogy too far, if for no reason other than the fact that computer languages are usually a great deal more precise than natural languages and thus such analogies are bound to break down.

The languages of different computers differ from one another in ways parallel to natural language. Some of them look very different from one another while others share some resemblance. As a general rule, however, the languages differ enough so that a procedure expressed in the machine language of a given machine cannot be performed or understood* by a different machine. This is analogous to asking someone who only understands English to execute a procedure written in French.

----

*We often say "executed" in this context.

We have come to call computer programs which solve real world (as opposed to computer generated) problems <u>applications programs</u>. Such programs are obviously the ultimate goal of any problem solver who seeks the aid of the computer. As we will see, however, there is much more to the subject of computer software than simply applications programs.

After several years of experience writing applications programs in machine language, some programmers began to focus on the question of making the man/machine communication process more efficient. They observed that computer-aided problem solving normally consisted of two distinct phases:

1) Expressing the solution to a problem in some convenient
    (often mathematical but not necessarily so) form,
2) Translating such an expression into the machine language
    of a particular computer.

The natural question was whether these two phases could be separated or not.

One of the earliest, and surely the most important, answer to this question was proposed by the designers of the FORTRAN (FORmula TRANslator) language. Their idea was to design a language for expressing numerical calculations in a form which was more convenient than machine language. This language was not, for most purposes, quite as convenient as a natural language, but it possessed one extremely important property that no natural language has. FORTRAN can be translated, with virtually no ambiguity, into a machine language. In fact, it is possible to write down an explicit procedure for making this translation.

Thus if I can express my solution to some problem in FORTRAN, then someone who only understands the translation procedure (not the original problem) can translate it into machine language.

In point of fact, of course, this translation procedure is written in machine language, and it is the machine itself that performs the translation. Such a translation procedure is commonly called a compiler. The input to a compiler is called source language and the output is called object (or target) language. The object language need not necessarily be a machine language, although it often is.

In this situation, then, I use the computer twice in the process of solving some problems. I write my procedure in FORTRAN source language. I have the FORTRAN compiler translate this into machine language, and then I take this machine language procedure and actually have the machine execute it. The computer is used twice, once to translate and again to execute.

Let us look at this process in the analogy as it is vitally important in understanding the role of software. Let us assume we have a machine which can execute procedures written in English. Let us further assume that we have a number of people who would rather write their procedures in French. I sit down and write an unambiguous procedure for translating from French into English. I write this procedure down in English. Now when I want to execute some procedure written in French, I first have my

machine translate it into English (using my translation program), and then I have the machine execute the result of that translation process (see Figures 1-2).

Given a compiler like the FORTRAN compiler, I can think of my computer as though its language were FORTRAN, rather than its own machine language. Indeed it would be possible, although probably not very sound economically, to build a machine whose language was FORTRAN (i.e., it would be wired to perform FORTRAN procedures directly). Thus a machine with a FORTRAN compiler looks like a piece of "FORTRAN hardware". A real FORTRAN machine differs from a machine with a FORTRAN compiler, however, in that a change in our FORTRAN language would require physically rewiring the real machine, while it would only be necessary to change some instructions in the translation procedure in the second case. Thus it is easier to change the program than it is to change the hardware. Therefore we come to the name software.

In our analogy, a machine which executes procedures expressed in French is the analogue of the FORTRAN machine. Note that if we wanted to change the meaning of a French word it would be necessary to rewire this machine. We would not have to touch our English machine, however, as it would suffice simply to alter the translation procedure, and the machine would not need to be modified.

In talking about computer languages we often have occasion to mention the level of a language. This is a very imprecise notion, but generally we consider one language to be of a higher level than another if it is
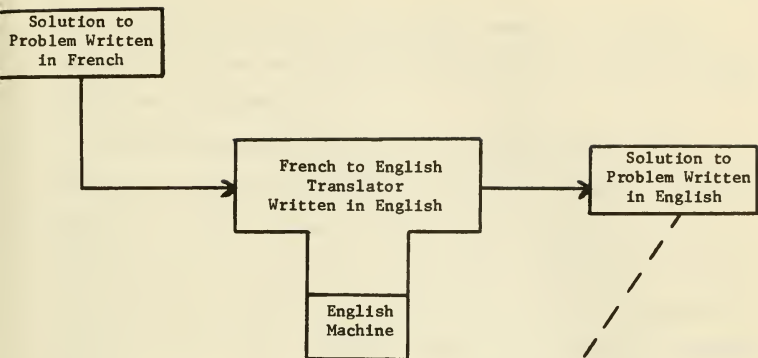
```
┌─────────────┐
│ Solution to │
│Problem Written│
│  in French  │
└─────────────┘
```

```
┌──────────────────┐      ┌─────────────┐
│ French to English│      │ Solution to │
│    Translator    │ ───> │Problem Written│
│ Written in English│      │  in English  │
└──────────────────┘      └─────────────┘
        │  English  │
        │  Machine  │
        └───────────┘
```

Figure 1
Translation from French to English

```
┌─────────┐
│  Data   │
│  for    │
│ Problem │
└─────────┘
```

```
┌─────────────┐      ┌──────────┐
│ Solution to │      │ Solution │
│Problem Written│ ──> │   to     │
│ in English  │      │ Problem  │
└─────────────┘      └──────────┘
   │ English │
   │ Machine │
   └─────────┘
```
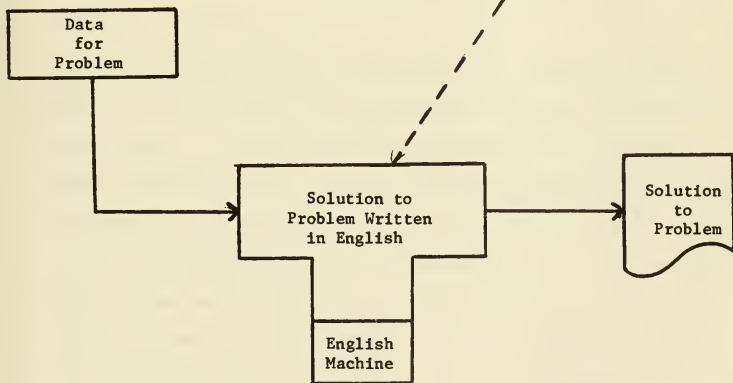
Figure 2
Execution of Translated Procedure

closer to normal human terms than to machine language. Thus machine language, for a given machine, is very low level while a natural language is of an extremely high level.

As might be suspected from some of the discussion above, the restriction that a computer language (like FORTRAN) be directly translatable into machine language imposes some rather significant constraints on its structure and character. First, such languages usually are very demanding in their grammar. They often require a strict attention to such things as punctuation which are not directly indicated by the nature of the problems that the languages are intended to help in solving. As we will discuss in a moment, this aspect significantly affects the ease with which it is possible to develop the passive and active vocabularies appropriate to using such a language.

A second point of importance with respect to computer languages is that for the most part they are constrained to dealing with some specific problem domain. The greater part of the history of the developments in computer languages center around the creation of languages appropriate to some broad, but not completely general, area of problem solving. Thus we see the development of many computer languages. COBOL (the COmmon Business Oriented Language) is directed towards making it easy to express the solution to many (hopefully common) business data processing problems. GPSS and SIMSCRIPT, on the other hand, are directed towards problems associated with the development and construction of discrete simulation models.

This indicates that it is always appropriate to ask what broad problem area a computer language is directed towards helping us attack. Since FORTRAN is generally directed towards numerical computation, for example, it may be quite difficult to use it to help us solve a problem in business data processing (maintaining a list of customer addresses might be an example). Similarly it may be inappropriate to use a simulation language to solve a numerical computational problem.

There are, however, many other dimensions to the software problem. So far we have only really considered compilers which translate one computer related language into another. We must also deal with <u>interpreters</u>. As their name indicates, these procedures are qualitatively analogous to those used by human simultaneous translators. Our compilers translated a procedure written in French into one written in English. This procedure could be performed, if desired, at some later point in time. Any interpreter, on the other hand, would perform each statement as it was being translated. This procedure, while not intrinsically different, makes certain things easier and other things more difficult.

For example, an interpreter is slower (generally speaking) than operating a compiler. Consider a statement which is executed several times. An interpreter would translate the statement each time just before performing it. A compiler, on the other hand, would translate it only once, and then execute or perform it each time without any translation being necessary.

On the other side of the coin, however, the interpreter allows some flexibility not easily attained by the compiler. If the performance of some step in the procedure modifies some other step in the procedure (clearly nothing we have said so far prevents this from being the case), then it may be easier to retranslate the statement every time. In such cases the translation time is a necessary expense, and such things as error detection may be considerably simplified.

Another important dimension for the classification of computer languages is efficiency. Here we must be very careful as efficiency can be measured on several dimensions. We will consider only three rather narrow interpretations without going into the potential complexity of some of the more global measures.

First, we might measure the efficiency of the compiler or translation procedure itself. Since the process of translating a source language procedure into object language requires time, this is of concern to us. We would like the translation process to require as little time as possible. A compiler which is efficient in this sense translates a given program in less time. We call this measure compile-time efficiency.

A second measure of efficiency concerns the procedure which results from the translation process. This is somewhat more difficult to see. Let us go to the analogy for a moment. If I write a procedure directly in English it will presumably be more efficient (i.e., shorter and require less time to perform) than an otherwise equivalent procedure which was originally written in French and then translated into English

conceive of would be incredibly difficult, we cannot use it today. This consideration is very important when designing a computer language. Some languages are much easier to implement than others.

Another important dimension of classification is ease of use. Here, too, things are perhaps more complex than they might appear to be on the surface. First let us consider active vs. passive use. As a speaker of English I have an active vocabulary which is much smaller than my passive vocabulary. I recognize and understand many words which I would never think of using. There is a considerable difference between reading or listening (which exercise the passive vocabulary) and writing or speaking (which exercises the active vocabulary).

In computer languages a similar phenomenon arises. Some languages which are quite easy to read may be very difficult to write. For example, the statement "ADD PAY TO TOTAL" may seem clear and easy to write, but if the language does not allow the statement "ADD PAY TO SUM" (because "SUM" is a word reserved for use only in certain contexts), then it may prove to be difficult to write.

A second important consideration with respect to ease of use is the sophistication of the user. Naive users and sophisticated users need and want different things. A naive user may well want a language where statements are self-explanatory. This helps him remember what statements to use, and helps quickly recall the purpose of each statement. Self-explanatory statements are usually somewhat longer than would otherwise be necessary. Since the naive user usually is not (or should not) be attacking huge problems this verbosity is not very painful.

by our translation procedure. In a similar fashion, a good machine language programmer is usually able to write a program which requires less space and runs more quickly than an equivalent program generated by the FORTRAN compiler. We will call this measure run-time efficiency.

The compiler designer can often choose a balance between these two. Let us take a conventional FORTRAN compiler and the WATFOR compiler as an example. In WATFOR all attention was directed at compile-time efficiency with no particular concern for run-time. This produced a compiler which compiles programs exceedingly quickly, but the programs which are produced may not run very efficiently. This is quite appropriate, for example, in an academic environment where a student's program is compiled again and again until it works, and then after a successful test run it is thrown away.

The typical FORTRAN compiler, on the other hand, spends a substantial amount of time during compilation in an attempt to produce a procedure which makes efficient use of the machine at run time. This is obviously more appropriate to an environment where the testing phase is only a small part of the time during which the procedure will be used.

The third measure of efficiency is of little direct concern to us as users of software. This is the efficiency (and ease) with which the language itself can be implemented. English might be a nice computer language to use, but as any implementation that we presently could

To the sophisticated user, however, excessive verbosity is a nuisance. Such a user is often writing large programs where extra wordiness may detract from overall readability. To some extent use of a concise language is analogous to the use of jargon (in the best sense) to communicate amongst workers in a field. Such communication is often clear to the initiated, but obscure to those less familiar with it.

There are many other dimensions of classification which could be discussed here. Our purpose, however, was only to present some of the basic concepts and terminology. Let us close this paper by considering a particularly important example.

Let us assume that we have written all of our procedures in French, and used a French-to-English translator to get them to run on our English machine. Now someone comes along and sells us a German machine, which in this case has some technological advantage. Must we rewrite all of our procedures? Clearly not, as all we need is a French-to-German translator, and all of our old procedures will still be useful.

This is analogous, of course, to the data processing facility which has its programs written in higher level languages. With 100 programs in FORTRAN and 50 in COBOL and two (the FORTRAN compiler and the COBOL compiler) in Machine Language A, it is only necessary to rewrite* the two compilers, rather than the 150 programs, to change to Machine Language B.

---

*This is an overstatement. In most practical circumstances some further effort is required, but it is usually relatively small when compared to the task of rewriting a program from the beginning.

Let us close by considering a question that an astute observer might have asked as we considered our last analogy, "Why not write an English-to-German translator (i.e., a Machine Language A to Machine Language B translator)?" This is where our analogy breaks down. Translation between two machine languages is typically _much_ more complicated than translation from a higher-level language. It is simply a case of the flexibility of the source language getting in the way. Thus we must be careful not to overextend the analogy which has been presented.