Language selection for applications

by M. H. HALSTEAD
Purdue University
Lafayette, Indiana

It may strike you as a truism to note that if the solution to a problem depends upon too many variables, we are apt to reduce the multiplicity of variables to one, base an important decision upon that one, and thereafter proceed as though there had never been, and would never be, any reasonable alternative to our choice.

This approach, when applied to the problem of choosing a programming language with which to implement a given class of problems, results in advice of the following sort: For scientific and engineering problems, use FORTRAN; for problems with large data bases, use COBOL; for command-and-control, use JOVIAL; and for systems implementation, use PL/I. Except for academic work avoid ALGOL; and with respect to those opposite ends of the spectrum, APL and PL/I, merely report that you are waiting to see how they work out. Now, obviously, advice of this type might be correct, but just as clearly, it will sometimes be erroneous, primarily because it ignores most of the variables involved.

It would seem only prudent, therefore, to examine some of those dimensions along which languages might be compared, with a view toward increasing our understanding of their relative importance in the decision process. However, there are four items which we should discuss in some detail first. These are:

1. How a programmer spends his time.
2. The difference between local and global inefficiency.
3. The role of the expansion ratio.
4. Variance in programmer productivity.

For the first item, I will use the best data of which I am aware, from an unpublished study of Fletcher Donaldson, done some years ago. Without presenting the details of his study, we note that his measurements of programmers in two different installations, when reduced to hours of programming activity per 40 hour week, yielded the following figures.

<table>
<thead>
<tr>
<th>Group 1</th>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Understanding Objective</td>
<td>3</td>
</tr>
<tr>
<td>B. Devising Methods</td>
<td></td>
</tr>
<tr>
<td>B1. Finding Approach</td>
<td>4</td>
</tr>
<tr>
<td>B2. Flow Charting</td>
<td>3</td>
</tr>
<tr>
<td>C. Implementation</td>
<td></td>
</tr>
<tr>
<td>C1. Writing Program (Coding)</td>
<td>8</td>
</tr>
<tr>
<td>C2. Preparing Test Data</td>
<td>2</td>
</tr>
<tr>
<td>D. Testing</td>
<td></td>
</tr>
<tr>
<td>D1a. Finding Minor (Syntactic) Errors</td>
<td>4</td>
</tr>
<tr>
<td>D1b. Correcting Minor Errors</td>
<td>1</td>
</tr>
<tr>
<td>D2. Eliminating Test</td>
<td>1</td>
</tr>
<tr>
<td>D3a. Finding Major (Logical) Errors</td>
<td>6</td>
</tr>
<tr>
<td>D3b. Correcting Major Errors</td>
<td>3</td>
</tr>
<tr>
<td>E. Documenting</td>
<td>3</td>
</tr>
<tr>
<td>Totals</td>
<td>40</td>
</tr>
</tbody>
</table>

From these data, let us combine those activities which are almost certainly independent of any language which might be chosen: Understanding the Objective, and Finding an Approach. For installation 1 this gives 7 hours, and for installation 2 it gives 7.14 hours, or roughly one day per week. From this it is apparent that, no matter how much improvement might be expected from a given language, it can only operate upon the remaining four days per week.

With respect to the problem of global versus local inefficiencies in programs, there are even fewer data, but the broad outlines are clear, and of great importance. Let us look first at local inefficiency. This is the inefficiency in object code produced by a compiler which results from the inevitable lack of perfection of its optimizing pass. According to a beautiful study reported by Knuth, the difference to be expected from a "finely tuned FORTRAN-H compiler" and the "best conceivable" code for the same algorithm and data structure averaged 40 percent in execution time. This is not to say that the difference between well written machine code and code compiled from FORTRAN will show an average difference of the full 40 percent, for even short programs will seldom be "the best conceivable" code for the machine.

With the present state of measurements in this area it is too soon to expect a high degree of accuracy, but let us
accept for the moment a figure of 20 percent for the average inefficiency introduced by even the best optimizing compilers. Now this inefficiency is the only one we are in the habit of examining, since it applies linearly to small as well as to large programs, and we are usually forced to compare only small ones.

The other form of inefficiency is global, and much more difficult to measure. This is the inefficiency introduced by the programmer himself, or by a programming team, due to the size and complexity of the problem being solved. It must be related to the amount of material which one person can deal with, both strategically and tactically, at a given time. Consequently, this form of inefficiency does not even appear in the small sample programs most frequently studied. But because programs in higher level languages are more succinct than their assembly language counterparts, it is responsible for the apparent paradox which has been showing up in larger tests for more than a decade. This paradox probably first appeared to a non-trivial group during the AFADA Tests in 1962, in which a Command-and-Control type program was done in six or seven languages and the results of each compared for efficiency against a “Standard” done in assembly language. When the initial results demonstrated that many of the higher level language versions, despite their obvious local inefficiencies, had produced object code programs which were measurably more efficient than the “Standard,” the paradox was attributed to programmer differences, and the entire test was redone. In the second version, the assembly language version was improved to reflect the best algorithm used in a higher level language, and the paradox dissipated. As recently as 1972, the paradox was still showing up, this time in the Henrikson and Merwin study. In that study, several operating programs which had been written in assembly language were redone in FORTRAN.

If we direct our attention to their comparison of FORTRAN V and SLEUTH, the assembly language for the 1108, we find that two of their three programs gave the same execution times, while for the third they report:

“In actual performance, the FORTRAN version ran significantly better than the SLEUTH version. If the SLEUTH version were to be recoded using the FORTRAN version as a guide, it is probable that it could be made to perform better than the FORTRAN version.”

While the dimensions of global inefficiency have not yet been well established, it tends to become more important as the size of a job increases. Since it is a non-linear factor, it overtakes, and eventually overwhelms, the local inefficiency in precisely those jobs which concern all of us most, the large ones.

This brings us, then, to a consideration of the Expansion Ratio, the “one-to-many” amplification from statements to instructions, provided by computer languages and their compilers. Since the global inefficiency varies more than linearly with program size, it follows that anything which will make an equivalent program “smaller” in its total impact upon the minds of the programmers will reduce its global inefficiency even more rapidly than the reduction in size itself. With an expansion ratio of about four, it appears that the effects of local and global inefficiencies balance for programs of about 50 statements or 2000 instructions, but these figures are extremely rough.

The fourth item listed for discussion, the variance in programmer productivity, enters the language picture in several ways. First, of course, it is because of the large size of this variance that many of the measurements needed in language comparisons have been so inconclusive. But this need not blind us to another facet. Since the time, perhaps 15 years ago, when it was first noted that chess players made good programmers, it has been recognized that programming involved a nice balance between the ability to devise good over-all strategies, and the ability to devote painstaking attention to detail. While this is probably still true in a general way, the increasing power of computer languages tends to give greater emphasis to the first, and lesser to the second. Consequently, one should not expect that the introduction of a more powerful language would have a uniform effect upon the productivity of all of the programmers in a given installation.

On the basis of the preceding discussion of some of the general considerations which must be borne in mind, and leaning heavily upon a recent study by Sammet, let us now consider some of the bulk properties which may vary from language to language. While it is true that for many purposes the difference between a language and a particular compiler which implements it upon a given machine is an important distinction, it is the combined effect of both components of the system which must be considered in the selection process. For that purpose, the costs of some nine items directly related to the system must somehow be estimated. These will be discussed in order, not of importance, but of occurrence.

1. Cost of Learning. If one were hiring a truck driver to drive a truck powered by a Cummins engine, it would be irrelevant as to whether or not an applicant’s previous experience included driving trucks with Cummins engines. It would be nice if prior familiarity with a given language were equally unimportant to a programmer, but such is not quite the case. The average programmer will still be learning useful things about a language and its compiler for six months after its introduction, and his production will be near zero for the first two weeks. According to a paper by Garrett, measurements indicate that the total cost of introducing a new language, considering both lost programmer time and lost computer runs, was such that the new language must show a cost advantage of 20 percent over the old in order to overcome it. Since those data were taken quite a long while ago, it is probably safe to estimate...
that increasing sophistication in this field has reduced the figure to 10 percent by the present time.

2. Cost of Programming. Obviously, this is one of the most important elements of cost, but difficult to estimate in advance. If we restrict this item strictly to the activity classified as C1 in Donaldson’s study, then it would appear to be directly related to the average expansion ratio obtained by that language for the average application making up the programming load. Here the average job size also becomes important, and for a given installation this appears to increase. In fact, some years ago Amaya noted through several generations of computers, it had been true that “The average job execution time is independent of the speed of the computer.”

3. Cost of Compiling. While it has been customary to calculate this item from measurements of either source statements of object instructions generated per unit machine time, some painstaking work by Mary Shaw has recently shown that this approach yields erroneous results. She took a single, powerful language, and reduced it one step at a time by removing important capabilities. She then rewrote a number of benchmark programs as appropriate for each of the different compilers. She found that when compilation time was measured for each of the separate, equivalent programs, then the more powerful compiler was slower only if it contained features not utilizable in a benchmark. For those cases in which a more powerful statement was applicable, then the lessened speed per statement of the more powerful compiler was dramatically more than compensated for by the smaller number of statements in the equivalent benchmark program.

4. Cost of Debugging. Here again, since debugging varies directly with the size and complexity of a program, the more succinctly a language can handle a given application, the greater will be the reduction in debugging costs. While the debugging aids in a given compiler are important, their help is almost always limited to the minor errors rather than the major ones.

5. Cost of Optimizing. Since a compiler may offer the option of optimizing or not optimizing during a given compilation, it is possible to calculate this cost seperately, and compare it to the improvements in object code which result. For example, FORTRAN-H has been reported to spend 30 percent more time when in optimization mode, but to produce code which is more than twice as efficient. FORTRAN-hand, Frailey has demonstrated that, under some conditions optimization can be done at negative cost. This condition prevails whenever the avoidance of nonessential instructions can be accomplished with sufficient efficiency to overcompensate for the cost of generating them.

6. Cost of Execution. If, but only if, a good optimizing compiler exists for a given language, then it can be expected that the local inefficiencies of different high level languages will be roughly comparable. The inevitable global inefficiencies will still exist, however. It is here, as well as in programming cost, that a language well suited to the application can yield the greatest savings. Again, the data are inadequate both in quantity and quality, but cost savings of a factor of two are well within my own experience. This does not apply to those languages which are primarily executed in interpretive mode, such as SNOBOL, where large costs of execution must be recovered from even larger savings in programming costs.

7. Cost of Documentation. In general, the more powerful, or terse, or succinct a language is for a given application, the smaller the amount of additional documentation that will be required. While this is true enough as a general statement, it can be pushed too far. It seems to break down for languages which use more symbolic operators than some rough upper limit, perhaps the number of letters in natural language alphabets. Allowing for this exception in the case of languages of the APL class, it follows that the documentation cost of a language will vary inversely with the expansion ratio obtainable in the given applications area.

8. Cost of Modification. Since it is well known that any useful program will be modified, this item is quite important. Here any features of a language which contribute to modularity will be of advantage. Block structures, memory allocation, and compile time features should be evaluated in this area as well as for their effect on initial programming improvement.

9. Cost of Conversion. Since hardware costs per operation have shown continual improvements as new computers have been introduced, it is only reasonable to expect that most applications with a half-life of even a few years may be carried to a new computer, hence this element of potential cost should not be overlooked. If one is using an archaic language, or one that is proprietary, then the cost of implementing a compiler on a new machine may well be involved. While this cost is much less than it was a decade ago, it can be substantial. Even with the most machine-independent languages, the problems are seldom trivial. Languages which allow for the use of assembly language inserts combine the advantage of permitting more efficient code with the disadvantage of increasing the cost of conversion. As noted by Herb Bright, a proper management solution to this problem has existed since the invention of the subroutine, and consists of properly identifying all such usage, and requiring that it conform to the subroutine linkage employed by the language.

In examining the preceding nine elements of cost, it is apparent that many of them depend upon the expansion ratio of a language in a given application area. In deciding
upon a new language, or between two candidates, it might be useful to attempt to plot them upon a globe, with longitude representing possible application areas, and latitude representing any convenient function of the expansion ratio. The plot should look something like Figure 1, where it can be seen that languages range from machine language, in which any application may be handled, girdling the equator, through general purpose, procedure oriented languages in the tropics, to highly specialized, problem oriented languages in the arctic. The pole, which implies an infinite expansion ratio for all applications areas, must be reserved for programming via mental telepathy.

While there is some current research under way,12 which may yield more basic insight into the problems in this area, it is only a year or so old, and the most that can be said is that it is not yet sufficiently developed to be of present assistance.

A very interesting technique which has moved part of the way from research to practice, however, should be mentioned in conclusion. This involves a practical approach to the problem of language extension. Unlike the extensible-language approach, which seemed to open the door to a dangerous, undisciplined proliferation of overlapping and even incompatible dialects within a single installation, this alternate approach to language extension is based upon the older concept of precompilers. As suggested by Garrett13 and demonstrated by Ghan,14 dramatic savings in programming costs can be achieved in shops having sufficient work in any narrow application area. This has been done by designing a higher level, more specialized language, and implementing a translator to convert it to a standard procedure oriented language. In this process, the higher-level language may readily permit inclusion of statements in the standard procedure oriented language, and merely pass them along without translation to the second translator or compiler. This process, by itself, has the obvious inefficiency that much of the work done by the first translator must be repeated by the second. While the process has proved economical even with this inefficiency, Nelin15 has recently demonstrated the ability to reorganize, and thereby remove the redundant elements, of such a preprocessor-compiler system automatically, provided that both are written in the same language.

In summary, let us first note that we have not offered a simple table with line items of preassigned weights, nor a convenient algorithm for producing a yes-no answer to the question “Should I introduce a language specifically to handle a given class of programming jobs.” Instead, we realize that, with the current state of the art, it has only been feasible to enumerate and discuss those areas which must be considered in any sound management decision.

From these discussions, however, we may distill at least four guidelines. First, it is abundantly clear that great economies may be realized in those cases in which the following two conditions prevail simultaneously:

1. There exists a language which is of considerably higher level with respect to a given class of applications programs than the language currently in use, and
2. The given class of applications programs represents a non-trivial programming work load.

Secondly, there is important evidence which suggests that a higher level language which is a true superset of a high level language already in use in an installation may merit immediate consideration.

Thirdly, it must be remembered that data based upon comparisons between small programs will tend to underestimate the advantage of the higher level language for large programs.

Finally, the potential costs of converting programs written in any language to newer computers should neither be ignored, nor allowed to dominate the decision process.

REFERENCES

3. The AFADA Tests, conducted by Jordan, were unpublished, but see Datamation, Oct. 1962, pp. 17-19.