The Effects of Using a Nonprocedural Computer Language on Programmer Productivity

By: Elie C. Harel
Ephraim R. McLean
Graduate School of Management
University of California
Los Angeles, California

Introduction

Programmer productivity is a growing concern to all organizations involved with information systems and data processing activity. The steady rise of programmer salaries, the scarcity of qualified programmers, and the growing demand for the services of existing programming staffs are all contributing to an interest in improving programmer productivity. Previous research has shown that the computer language used, the type of application being programmed, and programmer characteristics (e.g., experience with a given programming language, experience on the job) are three major variables affecting programmer productivity [2]. In the last few years, a fourth generation of programming languages has been developed. These nonprocedural languages are now being used by a growing number of organizations, both by the professional programming staff and, of late, by end users themselves. These fourth generation languages are assumed to provide a substantial benefit to both groups, but no systematic research has yet been conducted to test their superiority over the more traditional third generation, procedural languages.

Abstract

An empirical experiment was conducted in order to compare a third generation, procedural language (COBOL) with a fourth generation, nonprocedural language (Focus) in terms of programmer productivity and program efficiency. Six applications were developed in two languages by different programmers who had been matched according to their level of expertise. The results of the experiment showed that the applications were programmed significantly faster, particularly by beginning programmers, using the nonprocedural language. On the other hand, it was found that the procedural language was significantly faster than the nonprocedural language in terms of CPU execution time.

Keywords: Application development, programmer productivity, program efficiency, procedural and nonprocedural languages, query languages.

ACM Categories: C.4, D.3.2, H.2.3, J.1, K.6.3

To address this shortcoming, this article focuses on the first of these two groups and studies the effects of language selection on programmer productivity as a function of programmer expertise and application complexity. It describes the results of a field experiment conducted in an actual business environment, comparing third and fourth generation languages for a variety of applications. These applications consisted of generating various types of reports typical of the work performed in many DP organizations. The primary goal of the experiment was to identify those situations in which the use of a nonprocedural language resulted in higher programmer productivity, particularly as a function of application complexity and programmer expertise. A secondary goal was to measure the hardware resource utilization of the procedural and nonprocedural languages to see what penalties, if any, were incurred in using the latter.
The Nature of Programming Languages

The evolution of programming languages can be divided into four distinct stages or generations. They are:

- First generation — machine languages
- Second generation — assembly languages
- Third generation — high level languages
- Fourth generation — very high level languages.

The first three (e.g., BAL, COBOL, FORTRAN) are procedural in nature, that is, "individual statements are prescriptive and programs specify sequences of computer processing steps" [12]. The fourth, and newest generation is nonprocedural in character, sometimes called "English-like." These languages (e.g., Focus, RAMIS) "state merely what the result is to be, not how to obtain it" [3]. With nonprocedural languages, users need only describe the data and the relations that are appropriate to the application, not the detailed program steps.

Different types of languages are used for different purposes. However, there is no clear consensus to date regarding when and where the use of a nonprocedural, fourth generation language is preferable to the use of a third generation language. Some comparisons between procedural and nonprocedural languages have been published recently [9, 20], but very little empirical research has been performed.

Reisner et al. [13] compared the human factors aspects of two fourth generation, nonprocedural query languages, SQUARE AND SEQUEL. The goals of the study were to determine the degree to which nonprogrammers could use these languages without extensive training, and whether or not there was a difference in general usability between them. A laboratory experiment was conducted using students, with programming experience and the language used as the independent variables and the students' test results as the dependent variable. The results were inconclusive regarding the differences between the two languages, although they found that nonprogrammers performed better with the language that more closely resembled English (i.e., SEQUEL).

Welty and Stemple [21] also studied two fourth generation query languages (SQL and TABLET) that differed primarily in their degree of procedurality. They found that students without programming experience, who studied the procedural language, wrote difficult queries better than those who studied the nonprocedural language. However, the time to develop or write the queries was not measured in their study.

Both of these studies had students as subjects and did not use real world applications. Also, the subjects did not have access to interactive terminals (the exams were on paper), and so the environments in which the studies were conducted were not typical of the current practice in most commercial environments.

Holz [4] compared a nonprocedural, fourth generation language (ADF) with a procedural, third generation language (COBOL). However, his study was not conducted as a controlled experiment; therefore, the results cannot be generalized. He found that the use of the nonprocedural language resulted in high programmer productivity in terms of person-days, while the machine performance (efficiency) was only slightly lower.

Perrott et al. [11] developed simulation models using two third generation languages (FORTRAN and Pascal). The models were then compared from the point of view of efficiency; i.e. compilation time, execution time, memory requirements, and lines of code. Pascal showed a significant improvement over the use of FORTRAN. However, of interest here, they found that the higher level of abstraction of Pascal also contributed to shorter development time and improved maintenance experience.

Munnecke [9] compared a third generation language (COBOL) with a fourth generation language (MUMPS) on a strictly linguistic basis. In his paper he stated his belief that the role of a computer language should be to support users linguistically in their own terms, to adapt to their needs, and to be as forgiving and friendly as possible. Not surprisingly, he concluded that the nonprocedural, fourth generation language that he studied was much better suited to meeting these needs than was the third generation language. In fact, he felt that COBOL is an
almost archaic language, somehow surviving within the modern programming environment.

McCracken [7] also discussed the evolution of programming languages from third to fourth generation and gave examples of the differences between them. He contended that non-procedural languages are easier to use in terms of both learnability and programming time and result in more satisfied programmers. He predicted that the growth in the number of COBOL programmers will slow down quite soon, and that fourth generation languages will eventually replace the use of almost all procedural languages.

Reisner [14] recently summarized much of the human factors research that has been done on query languages over the past six years. In addition to discussing the research itself, she identified three aspects of a language that she felt would affect its ease-of-use: syntactic form, procedurality, and data models. Furthermore, she posited that improved ease-of-use is directly related to increased productivity.

In earlier articles, both Shneiderman [16] and Wasserman [19] discussed the design techniques needed for evaluating programming languages. Shneiderman argued that as computer use becomes more widespread, programming amateurs and novices will demand facilities that are simpler to use. He identified five tasks which underlie programming language design: learning, program understanding, program writing, debugging, and modification.

Shortly after this, Chrysler [2] introduced a framework within which the variables affecting programmer productivity could be studied. His six variables were:
1. hardware,
2. source language,
3. programming mode,
4. programmer characteristics,
5. programming problem (application), and
6. organizational characteristics.

A seventh variable was later introduced by Benbasat, et al, [1]:
7. software engineering technology.

In this study, Chrysler found a significant correlation among most of these independent variables and his dependent variable, programming time. He concluded that one could predict the amount of time necessary to develop a program by measuring certain variables. However, he only used one language, COBOL, in his study; therefore, the effect of different languages on programmer productivity was not measured. In a later study, Jeffery [5] found that program development time depended primarily on the number of lines of code written. He found no relation between development time and the programmer's expertise.

Given the importance of the topic, it is surprising that so few research studies have been published. One suspects that such companies as IBM and TRW are actively investigating the matter internally, but it is difficult to obtain the results of their research. On the other hand, university-based researchers are frequently forced to use students as surrogates for professional programmers, a compromise that calls into question the generality of the findings. Moreover, when research can be conducted in the real world it is usually impossible to have controlled conditions. Few companies are willing to do the same application twice or to group programmers according to some research design rather than the needs of the business. This study is, therefore, one of the few in which real programmers are writing real applications in a real setting.

An Empirical Study

The research setting

This experiment was conducted at the Administrative Information Services (AIS) Department at the University of California, Los Angeles. Unlike many universities that combine their business and academic computing activities, at UCLA these two areas are intentionally kept separate and distinct. Each has its own computer (Administrative Computing has an Amdahl V-7; Academic Computing, an IBM 3033), is located on different parts of the campus, reports to different vice chancellors, and has completely separate staffs. The programmers at AIS are full time professionals, working on typical business
applications like payroll, accounts payable, and general ledger, and are thus fully comparable to their counterparts in private industry. All of the applications undertaken as part of this study were a regular part of their assigned duties. Thus, in spite of being located on a university campus, the department is very much like the typical business data processing organization. This is important to note for, as was mentioned earlier, university-based studies are often discounted as not being applicable to the real world. The fact that this research took place at a university is only coincidental; the results should have general applicability.

The study variables

Chrysler's study identified six research variables. This study investigated three of them, holding the other three constant. The variables held constant were:

1. Hardware—All programmers used the same hardware.
2. Programming mode—All program development was done on-line.
3. Organizational characteristics—All programs were developed within the same organization during the same period of time.

The variables to be manipulated were:

1. Source language used—Third generation COBOL or fourth generation Focus.
2. Type of application—Whether simple or complex.
3. Programmer expertise—Beginning or experienced programmers.

The dichotomy of these independent variables was a simplification of the actual situation in order to provide meaningful results within the limitations of the study. Ideally, application complexity and programmer expertise should be subdivided into additional steps or levels. For example, the variable "programmer expertise" could be divided into several levels depending on the years of experience with the language being used or with the application being undertaken, rather than a simple "beginner" and "expert." Such expanded scales, however, must wait until substantially larger sample sizes can be obtained.

Source Language Selection—Due to practical resource limitations, only two languages were selected for use in this study. COBOL was selected to represent procedural, third generation languages and Focus was selected from among the several new nonprocedural, fourth generation languages. Ryge [15] has reported that COBOL is the dominant business computer language, with over four times the use of either Assembler or PL/1, which rank next in popularity. Thus it seemed an obvious choice. Focus, developed by Information Builders Inc., is a database management system as well as a generalized report writer. It is somewhat English-like, with nonprocedural query features similar to such other languages as RAMIS II and SOL and, therefore, appeared to be representative of the mainstream of fourth generation languages.

Application Selection—In order to accurately reflect a real world situation, actual scheduled applications were chosen for programming in connection with this experiment. These applications involved mostly report generation from preexisting files. They were then divided into two groups—"simple" and "complex" programs. The term complex may be somewhat misleading, for none of the applications were truly complex. Simple and not-so-simple would probably be more accurate, although more awkward descriptors. The measure of program complexity was determined by expert opinions through a questionnaire administered to project managers and experienced programmers after they had read the program specifications. Due to time and resource limitations, only six applications were selected—three simple and three complex. None of the common measures of application complexity (e.g., lines of code, number of variables) [18, 22] were used because program complexity had to be determined from the specifications only, before the beginning of the programming phase. However, measures of the completed programs, e.g., lines of code, number of GOTO statements, number of I/O statements, etc., showed the classification of program complexity to be quite consistent with the previous ratings of the experts. The simple programs had fewer lines of code, involved
fewer GOTO statements, and required less time to develop than did the more complex ones.

Programmer Selection — Twelve programmers were chosen to be involved in the experiment, all of whom had at least six months exposure to the applications environment. Their level of experience was determined by a questionnaire which elicited information on their familiarity with either COBOL or Focus in terms of formal training, years of experience, and number of programs written. They were also asked about their familiarity with the AIS organization and the various applications under development within AIS. In addition, project managers were asked for their opinions regarding the programmers' expertise.

Each programmer was graded with regard to the language that he or she would be using. For example, a person could be an expert in one language and a beginner in another. A programmer was classified as a beginner in a particular language if he or she had programmed less than twenty programs in that language. All those who had written more than twenty programs were classified as experts. This selection process also matched the managers' opinions regarding the programmers' expertise. Those who had written less than a total of five programs did not participate in the experiment at all. It was felt that they were still on the steep part of their learning curve and any information about their productivity would be meaningless. Using these criteria, six programmers were classified as beginners and six as experts.

The Dependent Variables

There were two main dependent variables that were evaluated in this research: programmer productivity and program efficiency. The first was measured by recording the following actual times (in person-hours):

1. time to understand the specifications,
2. program design time,
3. programming time,
4. testing and debugging time,
5. consulting time, and
6. documentation time.

The best overall measure for programmer productivity was felt to be the sum of program design time, programming time, and testing and debugging time. The other three variables, collected for the sake of completeness, proved to be relatively independent of the language used and thus were not included in the analysis.

After the programs were debugged and approved, the following efficiency measurements were taken:

1. total CPU time for compilation,
2. total CPU time for execution,
3. total clock time for execution,
4. total number of I/O operations,
5. number and size of input and output files, and
6. total lines of source code.

Among these, the best measures of efficiency were felt to be the total CPU time for execution, the total number of I/O operations, and the total lines of source code. However, the absolute values of the first two are not the appropriate measures because of program variations; i.e., simple programs may take a long time to execute because of large files, while complex programs with smaller files might be quite fast. Thus the ratios of execution times and of I/O operations between the two languages were used as the experimental measures of efficiency.

The other measures appeared to depend on a number of factors. For example, total clock time for execution varied significantly with the machine load, i.e., the number and type of jobs that were on the computer at the same time. Similarly, the CPU time for compilation was not considered because it was quite small for the COBOL programs and did not exist for the Focus programs (nonprocedural languages are generally executed interpretively).

The Research Hypotheses

The study was designed to test the following two hypotheses.

H.1 The use of nonprocedural languages, rather than procedural languages, results in higher programmer productivity when one or more of the following is true:
A. The programmers do not have substantial experience with the languages.
B. The applications are simple in nature.
Previous research \[4, 14\] has shown that non-procedural languages are generally more productive than procedural languages in terms of programming time. On the other hand, it has also been shown that complex programs are better written in procedural languages, although not necessarily faster \[21\]. This first hypothesis was therefore based on the above research and on accumulated industry experience. It was assumed that as the level of expertise increased, the differences between the languages would decrease in terms of programming time. In cases where the programmer has substantial experience or where the application is complex, there should be no significant difference between the use of a procedural or a nonprocedural language on programmer productivity; i.e., the results should be inconclusive.

H.2 Procedural languages are more efficient than nonprocedural languages in terms of:

A. CPU execution time,
B. input/output operations, and
C. lines of source code.

The nature of most nonprocedural languages forces them to be less efficient than procedural languages in terms of CPU time. These languages are often based on a lower generation procedural language, e.g., FORTRAN or Assembler, and are executed interpretively. Moreover, since they provide more services to the programmer (or nonprogramming end user), the tradeoff lies within the effectiveness domain. Previous research has thus shown that procedural languages tend to be more efficient than nonprocedural languages in terms of CPU time \[4\].

As for the other two dimensions of efficiency, I/O operations and lines of code, they are only modestly important. Because of the nature of the languages being tested and the applications being coded, there was little reason to expect significant differences in I/O operations; these should be approximately equal for both versions. In sharp contrast, the lines of code should be substantially different between the procedural and nonprocedural versions. In fact, this difference should be true almost by definition. By their very nature, nonprocedural languages are designed to be more concise.

The Conduct of the Study

As discussed previously, the experiment was conducted at the Administrative Information Services Department of the University of California, Los Angeles, as part of an ongoing research project on the use of nonprocedural languages. All the subjects who participated in the study were full time programmers who worked on applications as part of their regular jobs. The six applications that were selected for the experiment were all drawn from the regular schedule of programs which were to be written during this time period. Each application was classified as either simple or complex, and each was written in both languages by different programmers.

All of the programming assignments chosen for the study were typical business applications, mainly the programming of new or special reports from existing files. None of the applications had to create their own data files. This point is important in interpreting the results of the efficiency comparisons between the two languages. Both had to access pre-existing VSAM and sequential files, resulting in substantial overhead penalties. This was particularly true for the fourth generation language, Focus, which might have performed more efficiently if the files had been created with Focus in mind.

Each programmer was classified as either a beginner or an expert in each language and was then pair-matched so that each pair was composed of programmers with the same level of experience in the two languages being compared. For example, one pair consisted of an expert in COBOL coupled with an expert in Focus. Each then programmed the same application in the language he or she knew.

Identical program specifications were given to each of the programmers personally by one of the authors, and the general research objectives were explained at that time. To ensure that the output of the two languages would be comparable, the final reports produced by each version were required to be identical in terms of both form and content.

Each subject received three tracking forms: a programmer characteristics form, a time tracking form, and a program characteristics form. The subjects were instructed to regard the pro-
Programmer Productivity

<table>
<thead>
<tr>
<th>Programmer Expertise</th>
<th>Type of Application</th>
<th>Language Used</th>
<th>COBOL</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginner</td>
<td>Simple</td>
<td>2 programs</td>
<td>2 programs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Complex</td>
<td>1 program</td>
<td>1 program</td>
<td></td>
</tr>
<tr>
<td>Expert</td>
<td>Simple</td>
<td>1 program</td>
<td>1 program</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Complex</td>
<td>2 programs</td>
<td>2 programs</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1. Experimental Design**

Programming assignment as a routine task, but were asked to keep track of the specific time spent on each application. Also, the programmers were told not to communicate with each other about the programs they were writing during the development process. There was no further interaction between the programmers and researchers until the programs were debugged and approved. Once the applications were complete, the forms were collected and analyzed.

The experimental design is shown in Figure 1. It is an unbalanced split-plot, factorial design, composed of two between-block treatments — programmer expertise and application complexity — and one within-block treatment — language. Each of these treatments has two possible levels. This design, according to Kirk [6], is the most appropriate one for experiments having two or more treatments, with repeated measures, and with subjects who are heterogeneous.

The results of the programming of the six applications, each done in COBOL and Focus, are shown in Figure 2.

**Programmer productivity**

To test the hypothesis on programmer productivity, five comparisons were made. Development time (i.e., the sum of design, programming, and test times) were tested as a function

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Productivity</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Progr’er Expertise</td>
<td>Appl. Type</td>
</tr>
<tr>
<td></td>
<td>Beginner</td>
<td>Simple</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Complex</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Complex</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Expert</td>
<td>Simple</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Complex</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2. Experimental Results**
of language type for each of the following combinations:
1. All applications written by beginner programmers.
2. All applications written by expert programmers.
3. All simple programs regardless of programmer experience level.
4. All complex programs regardless of programmer experience level.
5. All applications regardless of programmer experience level (i.e., the overall comparison between the use of procedural and non-procedural languages).

Since these comparisons were conducted on subsets of the same data, the error term of the entire design was used. The error was $E = 132.2031$ (with 2 degrees of freedom) using a two-way analysis of variance (ANOVA). All analysis was done using SAS programs. In order to calculate the F-ratios for the planned comparisons, separate SAS programs were executed, with the sums-of-squares for each of the runs divided by the common error term. Although the sample sizes are small, it can be argued that the results are still valid because the F distribution used in the ANOVA is quite robust even if the assumption of homogeneity of the error variance is not completely satisfied. Also, because both the programs and the programmers were randomly selected, there is no reason to believe that the underlying populations are anything but normally distributed. The results are shown in Figure 3.

These results show that there is a significant overall difference in development time between third and fourth generation languages. Fourth generation, nonprocedural languages were found to be faster to program in every case; and overall there is a three-to-one improvement in development times $(297.75/98.82 = 3.01)$.

There is also a significant difference between third and fourth generation languages regarding the development time of beginner programmers for applications of any degree of complexity. Programmers who do not have substantial experience with the language being used will be more productive using a fourth generation language than with a third generation one. With regards to experienced programmers, no significant differences were found between the languages, although the raw data shows that the nonprocedural language was consistently better than the procedural language.

If the level of significance is relaxed somewhat (i.e., alpha = 0.10), a significant difference is found between third and fourth generation languages regarding development time for both simple and complex applications. Thus it appears that, for both beginner and expert programmers, the nonprocedural language is more productive for both simple and complex applications. This contradicts the widely held notion that there will not be a significant productivity gap between the two languages for complex applications. This finding can perhaps be explained by the complexity classification used in this study. As indicated earlier, the programs that were classified as complex were not, in fact, very complex. Consequently, the comparative advantages of each of the languages, which

<table>
<thead>
<tr>
<th>Comparison of Development Times</th>
<th>COBOL (hr.)</th>
<th>Focus (hr.)</th>
<th>Sums-of-Squares</th>
<th>F-ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginner programmers only</td>
<td>181.75</td>
<td>36.75</td>
<td>3504.51</td>
<td>26.51*</td>
</tr>
<tr>
<td>Expert programmers only</td>
<td>116.00</td>
<td>62.07</td>
<td>484.74</td>
<td>3.67</td>
</tr>
<tr>
<td>Simple programs only</td>
<td>112.75</td>
<td>29.32</td>
<td>1160.09</td>
<td>8.78**</td>
</tr>
<tr>
<td>Complex programs only</td>
<td>185.00</td>
<td>69.50</td>
<td>2223.37</td>
<td>16.82**</td>
</tr>
<tr>
<td>All cases (overall)</td>
<td>297.75</td>
<td>98.82</td>
<td>3297.76</td>
<td>24.98*</td>
</tr>
</tbody>
</table>

*Significant F-ratio at the 0.05 level
**Significant F-ratio at the 0.10 level

Figure 3. Productivity Results
Programmer Productivity

<table>
<thead>
<tr>
<th>Test</th>
<th>N</th>
<th>Mean Ratio</th>
<th>Standard Deviation</th>
<th>t-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU time</td>
<td>6</td>
<td>32.9935</td>
<td>29.7337</td>
<td>2.6362*</td>
</tr>
<tr>
<td>Number of I/O Operations</td>
<td>6</td>
<td>6.0000</td>
<td>6.1448</td>
<td>1.9936**</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>6</td>
<td>9.1216</td>
<td>9.5572</td>
<td>2.0825*</td>
</tr>
</tbody>
</table>

*Significant t-value at the 0.05 level
**Significant t-value at the 0.10 level

Figure 4. Efficiency Results

might have come to light in a truly complex programming application, were not found.

Program efficiency

As discussed earlier, the measures of program efficiency were defined as the ratios of the CPU times (for execution), the ratios of the number of I/O operations, and the lines of code. To test the hypothesis regarding these measures, one-tail, one-sample independent t-tests were conducted with the ratio data, again using the SAS software package. The null hypotheses for both the CPU and I/O tests were that the ratios should be less or equal to 1.0. The results are shown in Figure 3.

Figure 4 shows that the ratios of CPU times between nonprocedural and procedural languages is significantly larger than 1.0; i.e., procedural languages execute considerably faster than nonprocedural languages in terms of CPU time. This conclusion holds true for all levels of program complexity and programmer expertise.

The mean value for the CPU ratio for the six applications was 32.99. In other words, the nonprocedural programs were over thirty times slower than the procedural versions. This figure must be used with extreme caution, however. As discussed earlier, the nonprocedural language programs were developed with external input files; files that did not conform to Focus’s internal database structure. This substantially reduced the efficiency of execution.

In addition to the time penalty for having to access external files, the CPU compilation time was not included in the efficiency calculations. This compilation figure may become quite significant if the procedural language program requires extensive testing and debugging or if the program is to be changed frequently. For instance, the testing time measured for the procedural language was significantly longer than for that of the nonprocedural language; therefore, an overall efficiency ratio might be quite different depending upon the number of compilations and the number of production executions.

The ratio I/O operations between the nonprocedural and the procedural languages was not significantly different at the 0.05 level but was significant at an alpha level of 0.10. This result may have been biased by one program which, because of its nature, required significantly more I/O operations in the nonprocedural language version. Without this program, the ratio of I/O operations between the two languages for the other five applications would have been very close to one.

As a final comparison, the ratio of lines of code written in the procedural and nonprocedural languages was significantly larger than 1.0; i.e., the programs written in COBOL are, on average, over nine times longer than their Focus equivalents. This is probably the least surprising of all the research findings. The hallmark of fourth generation languages is their powerful and concise notation.

Summary and Implications

The results of this experiment can be summarized as follows. The nonprocedural language (Focus) was found to be significantly superior to
the procedural language (COBOL) in terms of programmer productivity, as measured by the total of design, programming, and test times for six common business applications. This difference was most pronounced for the more simple and straightforward programs. The study did not include extremely large or complex programs where it is possible that this difference might disappear or even reverse. Additionally, the benefit of improved programmer productivity was found to be larger for beginning programmers than for their more experienced colleagues.

This finding has implications for those organizations that are considering the decentralization of data processing into the various user departments. As stated in the beginning of this article, both professional programming staffs and end users are now starting to use these new fourth generation tools. These “end-user developers” [10] are likely to be similar to beginning programmers in terms of their programming skills — or lack thereof — and thus may derive a greater benefit from using nonprocedural languages than their more skilled counterparts in the data processing department.

The price paid for these improvements is computer time. The execution times for the Focus programs were significantly longer than for those in COBOL. However, compilation times were not included in the machine usage calculations. Extensive testing or frequent program modification would both result in substantial compilation times, thus reducing the CPU efficiency advantage of COBOL. Also, the tradeoff between computer time and “people time” is increasingly tilting in favor of people. In other words, the decreasing cost of hardware, coupled with the increasing cost of personnel, make the penalty of using computers inefficiently smaller and smaller as time passes. Finally, the applications chosen for this experiment made use of preexisting, external files which were not organized by means of any database management system. The execution penalties of fourth generation languages may be markedly reduced by specifying the files and databases with these languages in mind.

For an organization considering the use of fourth generation languages, Figure 5 gives some general guidelines for applications similar to the ones used in this study. These guidelines are naturally quite subjective and therefore should be used with caution.

The first two guides are derived directly from the results of this study and suggest that the greatest benefits to be obtained from the use of fourth generation tools are for fairly simple report-type applications, done by relatively inexperienced personnel, be they programmers or end users. The last two guides are not a direct outgrowth of the study but are logically consis-

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Procedural Language (3rd Generation)</th>
<th>Nonprocedural Language (4th Generation)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Skill</strong></td>
<td>Bad</td>
<td>Very Good</td>
</tr>
<tr>
<td>Beginning</td>
<td>Good</td>
<td>Very Good</td>
</tr>
<tr>
<td>Expert</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Application Type</strong></td>
<td>Bad</td>
<td>Very Good</td>
</tr>
<tr>
<td>Simple</td>
<td>Fair</td>
<td>Good</td>
</tr>
<tr>
<td>Average</td>
<td>Not Tested</td>
<td>Not Tested</td>
</tr>
<tr>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Program Execution</strong></td>
<td>Very Good</td>
<td>Fair</td>
</tr>
<tr>
<td>Infrequent</td>
<td>Very Good</td>
<td>Bad</td>
</tr>
<tr>
<td>Frequent</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Program Modification</strong></td>
<td>Fair</td>
<td>Good</td>
</tr>
<tr>
<td>Infrequent</td>
<td>Bad</td>
<td>Very Good</td>
</tr>
<tr>
<td>Frequent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. Guidelines for Language Usage
tent with it. They suggest that for programs that are run very frequently, the execution time penalties of fourth generation languages may be quite heavy and recoding in a more efficient procedural language may be worthwhile. Conversely, if the programs are frequently modified, requiring extensive recompilations, the benefits of execution efficiency found in third generation languages may be more than offset by the cost of these compilations.

Finally, it should be stressed that this study was conducted exclusively among professional programmers and that, as great as the benefits may be for this group, the benefits among end users may be greater still. The enhancement in human effectiveness will more than compensate for any loss in computer efficiency.

References

About the Authors

Elie C. Harel is currently a Ph.D. candidate in Computers and Information Systems at the UCLA Graduate School of Management. He received a B.Sc. in Electrical Engineering from the Technion-Israel Institute of Technology, and a M.Sc. in Management Information Systems from Tel-Aviv University. He is also the Manager of Data Administration, End-User Computing, and EDP Control for UCLA’s Administrative Information Services.

Ephraim R. McLean is an Associate Professor of Information Systems and Director of the Computers and Information Systems Research Program, both at the UCLA Graduate School of Management. His research and teaching are in the areas of strategic planning for MIS and decision support systems, and end user computing.