proceedings of the eleventh annual computer personnel research conference june 21-22, 1973

edited by
theodore c. willoughby
the state university of new york at binghamton

the special interest group on computer personnel research [SIGCPR] of the association for computing machinery

proceedings of the eleventh annual computer personnel research conference june 21-22, 1973

edited by
theodore c. willoughby
the state university of new york at binghamton

the special interest group on computer personnel research [SIGCPR] of the association for computing machinery

A copy of the most recent SIGCPR Annual Conference Proceedings is sent to each member of SIGCPR as a benefit of membership. Additional copies of these and previous annual proceedings may be purchased from: Order Dept., P. O. Box 12105, Church Street Station, New York, New York 10249. Price is $5.00; all orders must be prepaid.

Foreword

The Eleventh Annual Computer Personnel Research Conference, sponsored by the Special Interest Group on Computer Personnel Research (SIGCPR) of the Association for Computing Machinery (ACM), was held at the Center of Adult Education of The University of Maryland, College Park, Md., June 21 and 22, 1973.

The purpose of the annual computer personnel research conference is to identify and discuss the common problems and needs of those individuals concerned with the selection, training, evaluation and other aspects of computer personnel management. Research and problems reported are of interest to personnel managers, computer installation managers, and researchers involved with personnel administration of computer centers.

Representative papers are presented throughout the conference, followed by prepared and open discussion. The papers are published in these proceedings and constitute what is considered by the SIGCPR Executive Committee to be the most significant work brought forth this past year.

Omitted in this publication of the proceedings is "Profiling Pl/I Source Programs," by James L. Elshoff and Robert L. Beckermerger of General Motors Research Laboratories, as well as the discussion by James H. Burroughs, U.S. Air Force. Also omitted are workshop presentations, chaired by Elizabeth Campbell of the Massachusetts Institute of Technology and Dr. Harlan Mills, IBM Federal Systems Division.

SIGCPR seeks to explore computer personnel problems and provide access to a constantly growing body of computer personnel data gathered nationally at all levels in the computer field. Its membership includes representatives of large corporations, university groups, government organizations, small companies and service bureaus whose interests include scientific, business and research programming and computing. Further information on the group's activities may be obtained from Malcolm Gotterer, Professor, Department of Mathematical Sciences, Florida International University, Miami, Florida.

Special acknowledgement is made of the Conference Planning Committee: Dr. Gotterer (Chairman), Florida International University; Robert A. Dickmann (Arrangements

Chairman), U.S. Department of Labor; and Ashford Stalnaker (Program Chairman), Georgia Institute of Technology.

Theodore C. Willoughby
Editor

Contents

KEYNOTE ADDRESS: COMPUTER MAGIC 1
David Freedman, State University of New York at Binghamton

RESEARCH NEEDS IN THE NEXT DECADE 11
Robert Reinstedt, RAND Corporation

ORGANIZING AND MANAGING COMPUTER PERSONNEL: CONCEPTUAL APPROACHES FOR THE MIS MANAGER 19
Cyrus F. Gibson and Richard L. Nolan, Harvard Business School

A NEW APPROACH TO PROGRAMMER APTITUDE TESTING 49
Charles J. Testa, University of Maryland

ANALYSIS OF CAREERS FOR GEORGIA YOUTH IN COMPUTER OCCUPATIONS, 1971-76 62
John L. Fulmer, Georgia Institute of Technology

LARGE SYSTEM OPERATORS AS PROFESSIONALS AND OTHER OPERATOR CONSIDERATIONS 80
Richard H. Branton, Southern Services, Inc.

COGNITIVE PREDICTORS OF SUCCESS IN COMPUTER PROGRAMMER TRAINING 98
Stuart J. Jacobs, Southern Connecticut State College

PANEL: CERTIFICATION OF COMPUTER PERSONNEL 115
Fred H. Harris, The University of Chicago;
Robert Reinstedt, RAND Corporation;
J. L. Hughes, IBM

KEYNOTE ADDRESS:

Computer Magic

David Freedman
State University of New York at Binghamton

THE TRANSMISSION OF MAGIC

Computer programming is a profession which requires training. This training is offered in universities, computing companies, and through professional organizations. It is also offered informally at computer installations, in coffee shops, in car pools, and at most places where there is more than one programmer or at least one programmer and a machine.

For many years anthropologists have studied the differences between science and magic through the efforts of Bronislaw Malinowski who observed that primitive tribes practice both.(1) Science is used in those situations where the people have a great deal of knowledge and control, but magical elements enter when there are factors which no amount of manipulation can control. Malinowski's people -- the Trobriand Islanders -- are extremely scientific in planting gardens, building boats and caring for livestock, but follow strict magical rites to make their gardens grow, provide calm weather for sailing and encourage the reproduction of their pigs. Science may cure them, but magic tells them why they are sick.

For a number of years we have studied the formal and informal training of computer programmers in both industrial and university environments. We have noticed that programmers are rigorous and careful in dealing with situations which they can control. When dealing with a new and unknown situation, however, programmers often use magical manipulations. If these manipulations are successful, they become ritualized and are passed on from one programming generation to the next.

At the School of Advanced Technology, an entering student is required to show proficiency in six areas: APL, PL/1, Logic and Boolean Algebra, Probability and Statistics, Numerical Analysis, and Set Theory. Proficiency is demonstrated by passing an on-line multiple-choice examination. Students lacking the necessary background in any area are offered short courses -- video tape presentations supplemented by discussions and a computer laboratory in which the student can receive help with programming problems. After completing all proficiency

work, the student enters core courses in architecture, applications, and programming.

Of the core courses, only the architecture course is designed to be theoretical. In the two workshop courses, students are taught to consider such things as numerical validity, time and space efficiency, and that any reliable program is more efficient than any unreliable program. Theory is not entirely neglected, for students are also introduced to the formal aspects of a language -- the formal language definition of PL/1,[2] and syntactic and semantic rules for approaching other languages. These required core courses represent the only necessary intersection of all the students' training -- an intersection with only moderate theoretical emphasis.

The programming class is divided into working groups, each group having the same specifications and deadlines. Our school's grading policy (pass-no record) encourages cooperative rather than competitive learning, and there is a great deal of intergroup cooperation. At the computer center, non-student jobs are given high priority, and students find the best turn-around times late at night. Since the center is practically empty at these hours, the students have a quiet atmosphere to discuss their programs and thus to learn from each other.

This focal point of interaction provides an ideal locus for our studies of the psychology of computer programming[3] using the anthropologist's technique of "participant observation"[4]. From hundreds of hours of such observation, we have discovered that at least as much education occurs during these all night sessions as in scheduled classes. It is in such informal settings, as in almost every computer center in the country, that the oral tradition of computer magic is passed from person to person.

MANA

During one semester, the final programming project required the use of an OS/360 sort -- one purpose of which was to introduce the student to the use of technical manuals. One week before the deadline, no group had a working program. A collective panic seized not only the groups but the instructor also. Each night, the entire class arrived at the computer center with box lunches and pillows, planning to sleep over. Finally, on the last remaining night, the mistake was found: two parameters in the manual were given incorrectly. Since there was no time to explain the error, the "magical" cards were passed from group to group until all programs worked. We refer to the cards as magical because the students

treated them as if they had "mana". Mana is a supernatural force found in an object or individual. It is most often discussed in the Melansian context although similar beliefs exist throughout the world(5)(6). One cannot acquire mana nor cause it to enter an object. Once acquired, it cannot be prevented from leaving, but while it remains, the object has supernatural power. A stone ax which has mana will cut better than any other ax, a tree with mana will grow better than any other tree, and a deck of computer cards with mana will work better than any other deck. On this penultimate night, working better simply meant working. Indeed, these cards were so powerful that they were passed on to the students in the next programming class, but by then the specifications had been altered and the sorting sequence was slightly different. In spite of the change, many of the students had so much faith in the magic cards that they turned in incorrect programs only to find that the mana had "fled." We have on more than one occasion observed a student waiting for a particular terminal to be vacant, or a particular key punch to be open before beginning to work. When we inquired as to his motivation, we found that he felt the terminal or key punch worked better for him, although there was nothing physically wrong with the other available machines.

RITUAL MAGIC

Successful magic, just like successful science, has been known to spread through time and space. A Job Control deck for a compile and go operation was hastily constructed in New York City in 1966. A duplicate version of the deck was shown to a programmer in Stockholm some three years later. They had no idea where the deck originated, for it had been in the installation when they arrived. Although there had been numerous changes in the system since the construction of the deck, no one in the Swedish installation wished to take the responsibility of changing the supernatural. After all, the programs were working -- or seemed to be.

To the anthropologist, such duplicated cards represent not "mana" but a different type of magic called "ritual" magic.(6) There is nothing supernatural -- no mana -- about the cards themselves, since this was a duplicated version. The power is in the correct carrying out of the ritual. As long as the formula is correct, the magic will work. The efficiency of such magic is not easy to disprove. Even if the desired result is not achieved, one can never be certain that the ritual was performed correctly. Perhaps the reproducer failed.

Ritual magic beliefs affect computer programs in efficiency as well as in reliability. A programmer who has become comfortable with a prescribed procedure is reluctant to change his method. This fear of the unknown, and its consequence, is not unique in computing. In a small Swiss valley, the people produce a particularly good and inexpensive cheese. The cheese is always in demand, and demand far exceeds the supply. Unfortunately, the people can't increase the supply for they are supporting nearly the optimum number of cows given the ecological constraints. When asked why their cheese has its unique flavor, most of the inhabitants agree it is because of the grazing. They are not sure of the exact type of grass or combination of flowers, but they are afraid to use chemical fertilizers to increase the growth for fear of changing the taste of the cheese. Since no one seems to know the reason for the taste of the cheese, this conservative attitude may be extremely functional.

In computer programming, a conservative attitude often leads to inefficient operations. If there exists a routine, large scale application which must be run with frequency, little or no modification will be attempted for fear of dire consequences. Programs designed for 705's and 1401's with limited capacity are still being emulated on spanking new 360's and 370's because no one in the installation knows how they were constructed, what their limitations are, or even how they work. Ritual magic retards change.

NAME MAGIC

Man in his attempt to control his environment has named his universe. The book of Genesis tells how Adam walked around his garden naming each plant and animal in its turn. In many American Indian tribes, a warrior receives from his guiding spirit a secret name, or secret song, which he jealously guards for it brings him power. Were his enemy to know his secret name, his enemy could control him.

The computer programmer knows and respects the magic of names. The entire process of finding mnemonics and acronyms is full of magical qualities. For example, we were shown an assembler language program which was in an endless loop. This program contained a counter, stored in a location named ONE. Unfortunately, the value at location ONE was zero. When reading such programs, programmers rarely check to see the stored value, for certainly nothing other than 1 could be found in location ONE, especially since the programmer knows he put it there.

Ritual magic occurs often among APL programmers. We have observed students sitting at the terminal repeating a function which continues to return error messages. Belief is evidently strong enough to keep them trying the same unsuccessful program, particularly if it comes from a public library. Program libraries are the high point of ritual magic.

To the casual user, a public library is part of the computer. He does not know the specifications for which the program was written, nor the "fuzz" factor of APL, nor what the output is supposed to look like. As an example, in one of our university's physics courses, students check their experimental results with an APL program. If the results of the program do not match the experimental results, the student then has an opportunity to repeat his experiment. Even assuming that the program is "reliable," it is clear that this practice encourages magical beliefs. The student is being told that his observations are incorrect, but the pronouncement of the deity (in this case, IBM 370/155) is gospel. He may not, in all probability, have the necessary skill or confidence to check or question the program.

In another case, a professor was introduced to a particular program by her graduate student. Neither the graduate student nor the professor had ever seen the program code. Rather they had heard about the documentation from someone who had used the program, and they played with the input format until they got the desired results. The professor then assigned the project to the class. Certainly this is an excellent use of the university's computing facilities. This method of teaching introduces new techniques and makes education dynamic and individually oriented. However, it also increases the magical aspects of the computer.

If there is a modification in the program, or if natural selection occurs(7), the program may fail to give reliable results. By the time this happens the graduate student may have left the school. Even if he is still in attendance, he may not have the expertise to understand the change. By the time any explanation filters down to the undergraduate students in the class, it will surely seem like magic.

This situation may be generalized to a non-academic environment.

Because of the high rate of turnover in many computer installations, and the reluctance to alter existing software, many routine programs have never been examined by anyone in the installation. They may work, but no one really knows "how they work" -- ritual magic.

Another example of name magic occurs in naming variables which are used as loop counters. In our video tapes, as well as in the APL text, the letter "I" is invariably used for a counter. As might be expected, the letter "I" is used most in student programs, followed closely in frequency by the word "COUNT".

There is nothing necessarily wrong with ritually imitaing programs unless imitation replaces learning. Another classic example of ritualistic imitation was found in our computer laboratory. The programmer wished to increment his counter by 2, and wrote in two separate statements:

```
I←I+1

I←I+1
```

When asked why he had not written

```
I←I+2
```

he responded that he was not certain it would work, as he had never seen that type of increment. This astounding reply led us to do a small informal experiment. When students were asked if the statement

```
I←I+1
```

was syntacticly correct, they invariably said "yes" but faltered when shown

```
I←I+2
```

Again, as in the case of the Swiss farmer, if you don't know what you are doing, but it works, don't change it.

Loop structures seem to be filled with examples of imitation. A vast majority of the APL programs we have read have the increment and test statement at the end, as in the following loop:

```
loop: (expression)
      (expression)
            .
            .
            .
      →(N>I←I+1)/LOOP
```

Even if the initial value of "I" were greater than that of "N", the program would go through the loop once. When this defect is explained to the programmers, they

usually admit that they have never thought of that situation. Some programmers, however, have hotly responded "Well, 'I' could never BE greater than 'N'." Perhaps, but in computing, never is a long time, and time is expensive.

THE DESIRE TO BELIEVE

The right-to-left rule of APL is a natural place for magical ideas to arise. When a student asks why does APL read from right to left, he may receive one of a number of possible answers:

1. "It's more natural."

2. "It has to be done that way because of the hardware."

3. "Don't worry about it, just remember: APL ALWAYS READS FROM RIGHT TO LEFT."

Unfortunately, to the inquiring mind, always does not appear to be always. We overheard a dialogue between an instructor and a student that went something like this:

Student: "Does APL always read from right to left?"

Instructor: "Yes, always."

Student: "Then how come the lamp (comment indicator) appears on the left?"

Instructor: "Well, I guess in that case it doesn't read from right to left."

Similar questions have been raised by students about indexing from left to right, and the "take" and "drop" functions operations from left to right. Unfortunately, the answers to these questions are often "because," which may be true, or is the best possible answer, but it leads to such rules as:

APL ALWAYS READS FROM RIGHT TO LEFT, EXCEPT WHEN IT READS FROM LEFT TO RIGHT

which is remarkably similar to "THE PROGRAM WORKS IN ALL CASES...EXCEPT THAT ONE" or "THE RITUAL ALWAYS WORKS, EXCEPT WHEN IT DOESN'T."

For example, we were debugging an indexing program, and the latest version had a "little bug." After three people had looked at the program, one finally found that an operation was carried out in a subroutine which was called only when the input had two page numbers. If there was only one page number, this routine was by-passed, and so was a necessary correction. The programmer who had done most of the work looked up smilingly and said, "Well, most of the entries will have two page numbers anyhow" and trotted off to do the correction. Indeed, the bug had only appeared in one case, and, after all, the program did work in all cases except that one.

COMBATING MAGICAL INFLUENCES

Magical beliefs start early so they must be fought early. For example, it is difficult for the beginning student to separate the relevant variations of a language from the irrelevant variations. Anthropological linguists are faced with a similar problem when they attempt to study a new language. They attempt to collect as many "utterances" as possible and then attempt to understand when variations are meaningful. For instance, In Standard American English, the "p" sound as in pit is phonetically different from the "p" sound as in spit, the difference being aspiration. However, to a speaker of the language this phonetic difference is not phonemically significant. That is, it conveys no difference in meaning. The two different "p" sounds are allophones of the same phoneme, i.e. there is no difference in meaning between an aspirated "p" and an unaspirated "p" in English.

Understanding computer languages is a similar situation. There are times when the choice of names for a variable is not influenced by meaning, but rather free variation. In APL, for instance, the choice of a name for a variable does not influence the attributes of that variable whereas in FORTRAN it does. If the rule could be established that in APL the choice of names never makes a difference, whereas in FORTRAN it always makes a difference, then the problem would be solved. But what of the case in APL where there is a global variable in a public library? If the student chooses the same name for a variable in his routine the program might not work. Depending on the name, the program's behavior is altered. Clearly, examples of when differences are differences must be presented.

This can be done in a number of ways. Again, working from the model of linguistics, the instructor can present examples in which the variation presents no significant difference, and then contrast these with

examples where the variation is significant, showing that it is not the choice of name which makes the difference, but rather the context or environment in which the name is being used. We have found that these differences can be taught by a rather simple exercise. After a group of students has finished a working program, the program is given to a second group. Their assignment is to rewrite the program, using different names, and different syntactic structures. In this manner, students learn that:

```
PUT SKIP LIST(X,Y);
```

and

```
PUT LIST(X,Y) SKIP;
```

are forms in free variation, but

```
PUT SKIP LIST(X,Y);
```

and

```
PUT SKIP LIST(Y,X);
```

are not.

In the first example, the variation has not affected the output, in the second example it does. Instances of significant vs. insignificant variation must be pointed out explicity. Programmers must be able to justify their actions. Only then will there be uniformity and efficiency in software. Even if a section of code works -- find out why and demonstrate it. For he may indeed come to realize during the grand demonstration that it only works most of the time. Otherwise, "magic" remains a reasonable explanation for the differences.

Magic cannot prevent magical beliefs. Magic should be combated by science. Students of data processing should be trained in the scientific method. They should assume that their programs do not work, and only when they can demonstrate by a series of hypotheses and proofs that the program performs as expected, are they finished. Simply, science is supposed to be the model for data processing, and a scientific model must be used.

Research Needs in the Next Decade

Robert Reinstedt
RAND Corporation

When faced with the topic of what the thrust of research might be in the next decade it becomes very quickly evident that it is easier to pick one research effort, design it, gather and process the data and report on it than it is to look into the future through a cloudy crystal ball and see much besides fog. What motivates one to keep looking (aside from the fact that a speech has been promised on the subject) is the optimist's syndrome or, for those of you who are familiar with the story, "I know there's a pony around here somewhere." That "there is research around here somewhere" is really not in doubt, but what shape it will take and the specific direction it will take in the next decade is a matter which does indeed require more attention.

Some of the concomitant questions must also be addressed. My talk today might be viewed as analogous to planning a curriculum; this is an essential part of the educational system, but without the other ingredients such as faculty, students, facilities, etc., is worth very little in and of itself. For me to simply outline a shopping list of interesting research needs and ignore the problems of accomplishing such research would be equally worthless.

So, before we start on some major effort to watch apples falling from trees we had better recruit some Newtons. Fortunately, for the programming field, there is a larger population of researchers available from which to draw and contributions can be made which do not nearly require the genius of a Newton. But good, solid researchers <u>are</u> required and in order to accomplish the needed research the environment must be such that it attracts competent individuals to perform studied analyses. And although personnel researchers <u>don't</u> need to compare with Newton, they have at least one thing in common, and that is to be gainfully employed for purposes of eating. To date there has been pitifully little monies available for any kind of personnel research in the programming field. What research has been done has been largely under the auspicies of individual corporations, in some cases bootlegged or on an individual's own time, or by someone whose primary goal is a thesis; a limited number of studies have been subsidized by grants, and, predictably enough, some of the most intensive research has been carried out by test developers.