AN ANALYSIS OF CONCEPTS OF PLAGIARISM AS THEY ARE APPLIED IN COMPUTER PROGRAMMING

by

John Bradley Connely

A Research Report
In Partial Fulfillment of the
Requirements for the Degree
Master of Science
(Computer Science)

November 1988

TABLE OF CONTENTS

MEMBERS OF COMMITTEE

Curtis R. Cook, Ph.D., Chairman

Tim Budd, Ph.D.

Ted Lewis, Ph.D.

CHAPTER I

INTRODUCTION

Statement of the Problem

There has been concern voiced for a number of years about the problem of program plagiarism. A number of SIGCSE Bulletin articles have discussed the issue. However, it is not at all clear as to what behavior we are actually referring, nor to what extent it is occurring. How do faculty define 'plagiarism?' How do students define the term? What do students typically do when they plagiarize? Are there discernible patterns in plagiaristic behavior? Are there ways to accurately measure such behavior? Do some types of students plagiarize more than others? Does plagiarism occur more in some kinds of classes than in others? Can we use some kind of metrics to determine when program plagiarism has taken place? What solutions are used or have been suggested to control or to eliminate the occurrence of plagiarism? Are they effective? Lastly, is there any consensus in the computer science profession on these issues and, if not, should there be and can there be?

Summary of the Findings

There is no clear-cut, easily applied, generally accepted, adequate definition of computer plagiarism. On a superficial

level it is easy to get agreement that copying other's work and submitting it as your own is wrong. However, when an effort is made to apply this definition to actual cases it readily becomes apparent that there are too many gray areas. To what extent can a student paraphrase an already existing publicly available algorithm in order to adapt it to a specific problem without being accused of plagiarism? What is the body of "common knowledge" in computer science that does not require citation? Up to what point is collaboration acceptable and when not? As of yet there are no generally agreed upon answers.

It is difficult to tell if there is very much program plagiarism and how serious a problem it is. This is significantly due to the lack of an adequate definition of the term. The comments in the literature are largely anecdotal in nature and cannot be used as a basis for a valid generalization. One can be of the opinion that any plagiarism is too much and that even the suspicion of successful plagiarism is demoralizing to the honest students. That is an acceptable position, but it does not answer the question of how much program plagiarism is occurring. The general opinion is that program plagiarism occurs mostly in lower division introductory classes and that it is committed mostly by non computer science majors.

There have been a number of attempts to formulate some form of metrics to determine the degree of similarity of homework in an effort to use such information as a means of detecting

possible instances of plagiarism. The schemes have been useful and productive of a greater understanding of the issues involved. They have foundered for a variety of reasons. One of them again being the lack of an adequate development of the concept of program plagiarism. The schemes have had a rather limited applicability to the comparison of whole programs which have undergone cosmetic alterations. For them to work at all, the original had to be submitted in the same homework set. So program plagiarism which involves using any other source of information is outside the realm of the use of the metrics that have been discussed.

There apparently are definite patterns involved in student plagiarism, at least as can be determined by asking students what they would do if they set out to plagiarize. There are patterns of what type of changes they would make and in what order. The protocol studies indicated that non-cosmetic changes are common. Thus a metric which would only catch cosmetic transformations would chance missing a significant number of cases of program plagiarism.

None of the solutions proposed to curb program plagiarism are anything more than piecemeal efforts. They all have some merit. They all may help. There is always a need for a clear-cut policy and procedures if one works in an institutional setting. The general solution must involve all of the aspects suggested. There must be negative reinforcement for those hopefully few who

stray, and positive procedures and teaching styles to help guide the many. Here, too, a better definition of program plagiarism is absolutely necessary.

Finally, as is now evident, except at an abstract level there is little consensus as to the nature, extent or seriousness of the problem. There is a consensus that there is a problem, that it has negative consequences, and that it should be addressed.

CHAPTER II

PLAGIARISM

Literary Concepts

Is a specific behavior an act of plagiarism? Is program plagiarism a frequent occurrence? Is plagiarism a serious problem in computer science programming?

These questions point to a basic difficulty that must be dealt with before any reasonable answers can be found to the questions themselves. The problem concerns the conceptual difficulty of the meaning of the word 'plagiarism'. What then is 'plagiarism' and, specifically, what is it in terms of computer programming? Certainly we cannot categorize behavior, nor determine the extent of the problem before the problem itself is clearly defined, nor should we make moral judgments before we have determined what behavior constitutes plagiarism. In fact, it would seem morally questionable on our part to condemn a student's behavior if we have not previously explained what behavior is unacceptable and why this is so. Thus, the analysis of the concept is a primary concern both for the authorities who attempt to apply it and for the students who are supposed to understand and abide by it.

Plagiarism is commonly defined as presenting somebody else's words or ideas as one's own, i.e., fraudulently copying something

(5:1031). Size is largely irrelevant. "...the copied matter may range from a few sentences to a whole paper copied from another student or from a book or a magazine" (8:635).

William W. Watt states that,

> ... It is as immoral to steal from another person's writing as from his ... wallet.
> ...
> There are, to be sure, degrees of plagiarism. For every student who commits grand literary larceny--lifting an entire theme word for word--there are a hundred fundamentally honest classmates who indulge in various kinds of petty larceny through ignorance of the laws of literary ethics. (See also Fowler, 482-484 on this point)
> ...
> ... The general principles for all honest writing can be summarized briefly. Acknowledge indebtedness:
> 1. Whenever you quote another's person's actual words.
> 2. Whenever you use another person's idea, opinion, or theory, **even if it is completely paraphrased in your own words.** (10:5-6)

On the other hand,

> When you write a research paper, you use information from three kinds of sources: (1) your independent thoughts and experiences; (2) common knowledge, the basic knowledge people share; and (3) other people's independent thoughts ... Of the three, you must acknowledge only the third, the work of others.
>
> ... even when [someone else's ideas] are expressed entirely in your words and format, they require acknowledgment.
> ...
> Common knowledge consists of the standard information of a field of study as well as folk literature and common sense observations. (4: 482-484)

The MLA Handbook would include proverbs and familiar quotations along with common knowledge (6:28). Oregon State University's "Handbook for Writing Teachers" further states that

> ... there are several situations where the rule

> [concerning plagiarism] is relaxed. Information generally known and accepted in your field is not documented. ... Sources are not cited for information that your readers are unlikely to question or can easily verify.(37:7)

To summarize at this point, plagiarism, in general, is copying or paraphrasing of someone else's work and, by not giving credit for the words or ideas or theories used, allowing others to assume that they are yours. This is so regardless of the amount of material plagiarized. There are degrees of opprobrium depending on your intent, e.g., if you plagiarize through ignorance of proper footnoting procedure and without deliberate intent, it is not nearly so serious a breach of ethical behavior as if you knowingly plagiarize and deliberately try to obscure that fact. As a complication, however, if you copy information which is common knowledge in general or common knowledge in your field, "information that your readers are unlikely to question or can easily verify" (37:7), material from folk literature, proverbs or familiar quotations, then you need not cite the source and you are not committing plagiarism. The above considerations would seem to indicate that plagiarism is to some extent a matter of the context of the situation, i.e., it will depend on your intent and yours and others understanding of 'common knowledge'. We are faced with a dilemma at this point. It is not at all obvious how we should acknowledge <u>everything</u> that is borrowed (words and ideas) which really, ultimately, means the vast plurality of what we know, but not acknowledge

that which is of common knowledge. Lastly, whatever plagiarism is defined to be, it is felt to be immoral, dishonest, unethical, larcenous and a form of stealing.

Computer Science Concepts

In an effort to clarify the relationship of the concept of plagiarism as it relates to computer programming, a number of SIGCSE journal articles have modified and expanded on the literary definitions of plagiarism.

One of the earlier papers asserts that plagiarism has most likely occurred when two program listings can be determined to be equivalent or "(nearly) identical" (30:30).

Ottenstein enters several caveats, however:

> ... it is possible for identical work to be performed independently, the semantic equivalence of two items cannot always be shown deterministically, and there is a subjective area between plagiarism and paraphrasing.
> ...
> ... Unfortunately, a student who cheated on only part of a program will not be detected.(30:30-31)

Thus, he clearly assumes that plagiarism is a kind of copying. He considers plagiarism to have occurred when the copying involves cheating, although he makes no effort to define what is meant by 'cheating' nor does he address the issues of intent, the possibility of standard information, or the question of legitimate collaboration. Ottenstein's search for similarity and invariants is analogous to seeking either the exact text or the ideas and structure which lie behind a prose work. His cautions are interesting also. Moreover, where they are not

relevant to traditional plagiarism is instructive. It makes sense when dealing with any kind of plagiarism to be aware that " ... the semantic equivalence of two items cannot always be shown deterministically, and there is a subjective area between plagiarism and paraphrasing." It is not made clear how to distinguish between legitimate paraphrasing (or even what it is) and the assumption that a similarity of invariants between two papers should be considered as a possible instance of cheating. It is not very likely in prose that "... it is possible for identical work to be performed independently". Furthermore, it would seem odd to assert in grading an English composition that "... a student who cheated on only part of a program will not be detected". The reason for the first deviation from a prose work is simply that computer languages are extremely limited in terms of vocabulary and syntax in comparison with natural languages, so, with a moderately simple assignment given to an entire class, there might well be solutions submitted that were very similar. The second deviation relates more to the mechanical way in which Ottenstein and others propose to ascertain whether program plagiarism has taken place. Although variations might deal with different computer languages and seek to implement varying ideas of invariance, the methods generally result in a comparison of the totality of one aspect of a program with another, e.g., the number of unique variables in program A versus the number in program B. Thus to the extent that any part of a program is <u>not</u>

plagiarized, it may start to significantly affect the metrics being used to compare the two programs. Ottenstein does not assert that plagiarism is simply a matter of copying a complete program, but that his method will only be appropriate to measuring such a case. Furthermore, if a large segment of several programs consisted of what might be considered common knowledge, it would also becloud any assertion that similarity should be viewed as possible plagiarism.

Mary Shaw chaired a computer science departmental committee at Carnegie-Mellon University in 1979 to, among other things, "... draft a policy statement defining cheating .... (33:72). In Appendix I of her paper she quotes extensively from the Carnegie-Melon University Student Handbook.

> Cheating includes but is not necessarily limited to:
>
> 1. The use of unauthorized materials including computer programs ....
> ...
> 3. The submission of work that is not the student's own.
> 4. Plagiarism.
> ...
> 7. Collaboration in the preparation of an assignment, unless specifically required by the department .... (33:74)

Shaw notes that "... the nature of the computer introduces unique problems" (33:72) in applying standard university policies on cheating and plagiarism. To clarify to students the nature of plagiarism, her department prepared and distributed a handout that specifies that the question of cheating depends on "the intent of an assignment or exam, the ground rules specified by the instructor, and the behavior of the student" (33:75).

Cheating is to be suspected if two programs are so similar that one can be transformed into the other by mechanical means, e.g., "renaming variables, rearranging statements and expressions and making systematic changes of data structures such as the substitution of integers ranging over [0,1] for booleans" (33:73).

Cheating should also be suspected whenever a student cannot explain his/her solution or how it was arrived at. Shaw also seeks to define cheating/plagiarism by listing several examples:

> Here are some examples of cases which are clearly cheating and clearly not cheating.
>
> Cheating
>
> - Turning in someone else's work as your own (with or without his knowledge). Turning in a completely duplicated assignment is a flagrant offense.
> - Allowing someone else to turn in your work as his or her own.
> - Several people writing one program and turning in multiple copies, all represented (implicitly or explicitly) as individual work.
> - Using any part of someone else's work without the proper acknowledgment.
>
> Not Cheating
>
> ...
> - Getting or giving help on how to solve minor syntax errors.
> - High-level discussion of course material for better understanding.
> - Discussion of assignments to understand what is being asked for. (33:75)

Shaw's paper generally supports the traditional idea of plagiarism, i.e., it is the falsely claimed use of someone else's work; it is cheating. She ignores the question of common knowledge, unless she possibly intended to mean this by the term

"unauthorized material." She points out that there are aspects of the problem of plagiarism that are unique to computer science, at least in their emphasis. For instance, collaboration is forbidden in the preparation of an assignment unless specifically required. However, by introducing the term 'collaboration' the conceptual difficulty is not eased, but rather made worse. Shaw then feels compelled to mention types of collaboration which are acceptable, even though not specifically required by the department [a contradiction?], e.g., "getting or giving aid on solving minor syntax errors", "high-level discussions" or "discussion of assignments". Note that a number of additional difficulties have been introduced. The person who helps, aids or collaborates may now also be culpable. It is not at all clear, certainly to a novice, if indeed to anyone, where the boundaries for "minor", "high-level" and "discussion" are to be found.

Shaw does address the earlier mentioned contextual nature of plagiarism by stipulating that the question of cheating does involve "the intent of an assignment or exam, the ground rules specified by the instructor, and the behavior [intent?] of the student". It can be a matter of partial or wholesale copying. She continues that plagiarism includes the transformation of copied materials through mechanical means and that cheating should be suspected "whenever a student cannot explain his/her solution or how it was arrived at."

These are interesting and worthwhile observations, but they

also obscure any specific definition by, in reality, allowing completely contradictory requirements from course to course or from instructor to instructor. Such statements are useful in an heuristic sense, but serve poorly as guidelines for students. They do serve to raise our consciousness of the new, or at least more numerous, difficulties of clarifying the concept of plagiarism in relation to computer programming. They are not, however, self-defining. It is not clear what is meant by unauthorized materials, nor the degree to which a student should be able to explain what he/she has done. The solution may have been honestly stumbled upon without any great insight, but great persistence.

A panel discussion moderated by Philip L. Miller elicited several comments on the nature of plagiarism:

> ... the tendency of students to resort to unorthodox means in fulfilling course requirements.
> ... such tactics as copying programs, stealing programs written by other students, and paying to have programming assignments written for them .... [Dodrill] (29:26)

In addition, another member of the panel noted that:

> In computer science it is particularly valuable for students to work cooperatively. Throughout their professional careers they will be working in teams and it is a poor educational system which does not prepare them for this. We should foster teamwork, rather than isolated individual activity; we should train students to work together, rather than looking upon it with suspicion; and we should encourage the sharing of ideas, rather than a jealous secrecy. There is nothing inherently unethical about such collaborative work. [Lidtke](29:27)

A member of a different panel supported this last point of view:

1. A certain amount of student collaboration is a powerful and useful teaching method. It invariably leads to look-alike programs.
2. Use of standard subprograms for often used algorithms should be tolerated. [Criss] (26:263)

Here we are seeing both the extension of ideas as the concept of plagiarism is applied to computer programming and the conflict which arises when the opprobrium generally felt toward plagiarism is not felt to be appropriate in regards to certain aspects of the proposed extended definition. If "a certain amount of collaboration" is good and the "use of standard subprograms should be tolerated" then mutual aid under some conditions and verbatim copying or close paraphrasing of common algorithms are both possibly acceptable. In other words, collaboration and cheating are not synonymous nor are copying, paraphrasing and plagiarism. We are back to a question of context.

Hwang and Gibson presented a long list of activities that would count as efforts to plagiarize a program.

> Students have devised an assortment of ways for cheating on programming assignments. Below, are listed several of them:
>
> 1. Copying a program by changing only the author's name.
> 2. Having someone else write all or part of the program.
> 3. Copying a program given in an earlier class.
> 4. Copying a program by changing only the line numbers.
> 5. Copying a program by changing the documentation.
> 6. Copying a program by changing the logic a little.
> 7. Copying a program by changing the variable names.
> 8. Copying a program by changing the logic a lot.(25:51)

The above list does little in general to further our insight

into plagiarism, although it does support the central core of meaning. Of interest, however, are points number 3 and number 8. Does number 3 really mean that it is plagiarism to use one's own work in a different class? That would seem an odd extension to the basic meaning of 'plagiarism', but might be of interest as an extension of the meaning of 'cheating'. The problem with number 8 is that once again we are faced with a concept with extremely vague boundaries. Is there no point at which a change of logic becomes so extensive that the result is no longer plagiarism?

Janet Cook, as part of a much larger concern with overall student ethics in a computer science environment, suggests meanings for 'collaboration', 'consultation' and 'plagiarism'.

> Collaborative problem solving can be valuable in computing. ... Unless group work is explicitly authorized for a project, however, assume that it is to be done by each student individually.
>
> When working on an "individual" project, the line between legitimate and illegitimate consultation is drawn at the point where a solution is put into writing on paper, in a machine, or elsewhere. The detailed development of the written solution should be one's own independent work.
>
> 1. Plagiarism, any act of accessing or copying another person's work and submitting it as one's own.
>
> Individually,
> - copying another person's program, perhaps modifying parts, and turning it in as one's own work. (16:464)

Here, as in the other papers, we find puzzling gray areas. It seems clear to say that collaboration is forbidden with individual projects, but legitimate consultation is acceptable.

Is the difference one between joint work and asking advice? But is not collaboration between students often a form of one asking advice of another, i.e., consultation? Apparently consultation is all right up until the students begin writing their program, but reality is not so precise. Many students do not ponder and workout the entire program in their minds and then type it in. Is any consultation allowed, e.g., a further discussion of the use of sentinels, after coding has started? What if the consultation before coding started involved detailed analysis of every aspect of the assignment to the point where the students involved had extremely similar code, even though each one actually wrote his/her program individually?

For the meaning of 'plagiarism', no new issues have been raised. Once again it is left vague as to whether one can plagiarize a partial program and what amount and kind of modifications can occur before one no longer is submitting someone else's work.

Faidhi and Robinson (21) have come closest to an operational definition of plagiarism by developing a taxonomy of levels of plagiarism from novice to expert:

> Level 1--represents the changes in comments and indentation.
> Level 2--represents the changes in level 1 and changes in identifiers.
> Level 3--represents the changes of level 2 and changes in declarations (i.e., declaring extra constants, changing the positions of declared variables and shuffling the procedures/functions, etc.).
> Level 4--represents the changes of level 3 and changes in program modules (i.e., changing a function to a procedure, merging two procedures into one, creating new

procedures).
Level 5--represents the changes of level 4 and changes in the program statements (i.e., FOR instead of WHILE, etc.).
Level 6--represents the changes of level 5 and changes in the decision logic (i.e., changes in expression).(21:18)

They also note that:

> Novice student plagiarism mainly utilizes certain stylistic and syntactic changes, whilst expert programmers may introduce semantic changes (e.g., changing the data structures used, changing an iterative process to a recursive process, etc.) as well. (21:11)

Such a table of transformations as shown above is certainly of interest and potentially very useful, but there is no justification given for either the entries of a given level or the order of the levels. While many would intuitively agree with some aspects of the table, it is not at all obvious that Level 5 is intrinsically more difficult than Levels 4 or 6. Lacking a theoretical or even empirical foundation, one cannot simply assert that a hierarchy is valid as stated. In addition, such a table would seem to imply that no matter how extensive such changes are, it is still plagiarism if you start from a copied program or, perhaps, even a copied fragment. Yet the authors later state:

> ... [that] when the original program has been so significantly changed that it can no longer be considered to be copied (and if it were, would require a plagiarizing skill exceeding that required to produce an independent program in the first place). (21:18)

Are we to assume that if the changes made are extensive enough and skillful enough, then it ceases to be plagiarism?

Student Concepts

As part of the protocol analyses--which will be discussed in

detail in the next chapter--nineteen of the twenty-two subjects were asked for a brief definition of plagiarism (three were no longer available at the time). All responded in some form that plagiarism was using or copying someone else's work or ideas without giving credit. A number of them added comments which indicate that they have given some further thought to the matter. Bibliographic numbers refer to Appendix E.

- ... [plagiarism] ... would not include adapting someones [sic] ideas or thought into something of your own design. (4)
- ... It [plagiarism] doesn't apply to general ideas and procedures that might be considered in the "public domain". (6)
- ... [plagiarism] is the direct, unrefined usage of someone else's information, without the proper permissions/ acknowledgments. (7)
- Plagiarism occurs when a document or parts of a document are copied, or copied and modified, and then used ... [by] those who do not have permission (legal or implied) ... (9)
- ... If the originator is unknown and the piece of work is deemed accessible to the public, then I would not consider it plagiarism. (16)
- ... If someone else liked the idea and spent his energies developing the idea again, that is not plagerism [sic]. (17)
- Plagiarism--copy of somebody elses [sic] work word for word without giving proper credit to them [sic]. (21)

Summary

Glaring in its absence is a generally accepted, clear-cut and easily understood definition of the concept 'plagiarism'. Through all of the above mentioned articles does run the common thread that plagiarism is a form of copying where proper credit

is not given to the original author. Everyone would seem to agree that to put your name on someone else's complete program, with perhaps some minor cosmetic modifications, is plagiarism. This would be so whether you stole the program, bought it or it was given to you. There is no disagreement that copying an entire program and claiming it as your own is wrong. There seems to be agreement, where the issue is noted, that copying part of someone else's program is also plagiarism. Thus copying from someone else with an intent to deceive is an essential feature of plagiarism.

A number of related concepts have also been mentioned: cheating, stealing, collaboration, paraphrasing, and the idea of common knowledge and its use.

Cheating is also an essential feature of plagiarism. Whatever else plagiarism is, it is clearly an attempt to circumvent the explicit and/or implicit guidelines of a course. It is an attempt to delude the instructor as to the fulfillment of the program requirements. It is not a priori self-evident what those requirements are or even that they would be considered reasonable by other professionals in the computer science field. A great deal will probably depend on the intentions of the instructor and the context of the class, but whatever the requirements are, for the student to deliberately mislead the instructor as to whether the rules of the course vis-a-vis the program assignment were honestly followed, is cheating.

An important distinction can be made between cheating or

stealing. Stealing would assume lack of permission and probably some degree of injury or loss being sustained by the original owner. Yet in program plagiarism the owner may have gifted the program to the plagiarizer and/or for all intents and purposes suffered no loss. Again, if the program or part of it had been written by the current user, but for a previous course, there could be no question of theft. Thus 'stealing' or 'theft' is not an essential aspect of plagiarism, but, rather, a frequently concomitant characteristic. Plagiarism may often involve stealing, but the concepts are not synonymous.

There is a great deal of confusion concerning the term "collaboration". Two positions have been taken. The first is that since all work must be done independently, any collaboration is cheating unless specifically authorized. This is not as clear-cut as it seems for surely no one intends that no questions may be asked at all of anyone concerning anything. The second position is that collaboration is acceptable up to a certain point. This is more realistic and forces one to consider more carefully what is meant by "collaboration" and what values accrue from it. Several of the articles noted in this chapter have spoken to the positive pedagogical values of student collaboration, but the difficulties of setting clear limits and how to cope with the effect that collaboration may well have on similarity of programs are still to be solved. In any case, collaboration per se cannot be considered a type of plagiarism.

The problem of paraphrasing needs to be viewed in two aspects. One would be where minor changes were made to obscure the original form of a program, i.e., cosmetic or novice changes. There is a consensus that this is plagiarism. The second aspect, however, is somewhat more complex. Is there a point at which the original idea or algorithm has been changed so much that its use no longer constitutes plagiarism? There is disagreement on this point, both as to whether such a point exists and, if so, where it is. What is really needed is a defensible rational for the degree of similarity that can be taken to indicate plagiarism.

The problem which has received the least attention is that of common knowledge. In the traditional definition of the term "plagiarism", one could use "common knowledge" without the necessity of citing the source. Aside from the traditional difficulty of how a novice is supposed to know what is general knowledge, computer science has the additional difficulty that very little effort seems to have gone either into spelling out what is included in the concept or in teaching students to work with such a concept and learn what it does and does not include. This is a critical lack because in a larger sense, part of becoming a professional in any field is both learning what is "common knowledge" in one's field and how to give credit where credit is due. But, what seems to have been overlooked is that a sense of both is necessary since it really is impossible to give credit for everything.

CHAPTER III

EXTENT OF THE PROBLEM

There has been a remarkable consistency in the data available relating to both the concern about program plagiarism and the perception of its occurrence. Ottenstein's SIGCSE article in 1976 (30) marks an early attempt at coping with plagiarism by devising a computerized means to apply a metric to the problem of program plagiarism. Shaw's comments in 1980 about cheating (33) are still relevant and up-to-date. In it she reports on a survey in which she received replies from the computer science departments at the Universities of British Columbia, California (Berkeley), Colorado, Pennsylvania State, Princeton, Stanford, Tennessee, Waterloo, Wisconsin, and Yale. The following is part of her summary of the responses:

> The factors most conducive to cheating are generally believed to be large class size, beginning students, and out-of-class programming assignments. Unfortunately, all three factors are usually present in the most commonly offered computer science course--introductory programming. Our colleagues uniformly believe this is where the cheating monster lurks. As with Nessie of Loch Ness there does not seem to be a clear picture of this beast, only a variety of eyewitness reports. ... Most respondents reported one or two blatant cases per course and a feeling that many more less obvious cases went undetected.
>
> ...
>
> We asked about explicit definitions of and policies toward cheating. All departments reported the same basic system: no single, explicit definition or policy at the departmental level and a brief definition and elaborate policy at the university level. (33:75)

A sample of relevant SIGCSE articles largely support Shaw's summary. Grier's article in 1981 asserts that "Plagiarism has become a problem in introductory Computer Science courses", and that "Sophisticated plagiarism is not the problem ..." He justifies his latter statement by assuming that the student intelligent enough to plagiarize with sophistication has no need to plagiarize" (23:15). He asserts that "... [students] cheat more now, ... [and that this is] a rising phenomenon ... (23:18).

Donaldson et al. take a somewhat different tack. They mention that there is a "feeling of many faculty members that incidents of plagiarism are quite prevalent". They do not say that there is more plagiarism in lower level courses, however, just that it has been more difficult to detect at that level due to the use of graders, less complex assignments and several sections of a course having a similar assignment (19:21).

In the panel discussion moderated by Philip Miller, the panel members all assume that cheating is a problem. William Dodrill is more explicit when he notes that

> ... the need to teach larger [introductory] classes consisting of a wider variety of students has introduced many problems. Outstanding among these is the tendency of students to resort to unorthodox means in fulfilling course requirements. In other words, students cheat. (29:6)

He then goes on to characterize such courses as containing "Many students [who] must take computer programming whether they have an interest in the subject or not" (29:26). A member of the same panel, Mary Dee Harris Fosberg, notes that "Plagiarism on

programming assignments has been a persistent problem for Computer Science educators, ..." (29:27).

Hwang and Gibson, on the other hand, do not limit plagiarism to introductory courses, freshmen or to non-computer science majors.

> Cheating on programming assignments, it would seem has become a way of life for many, and fraudulent misrepresentation of one's credentials for entrance into the marketplace is increasingly commonplace....
> ...(We simply assume that cheating on programming assignments is a highly undesirable practice which has adverse effects upon the students' preparation for their later professional performance in the computing industry.) (25:50)

However, interesting enough, when Hwang moderated a panel at the same meeting where the previously mentioned paper was delivered, his opening statement was that

> Plagiarism on programming assignments, particularly in lower-level computer classes, has been a problem of considerable concern. We suspect that the strongest contributing factor to the increase in this kind of cheating is the computing profession's reputation for being able to offer such high-salaried positions. (26:262)

Here he continues to seem to stress that those going into the computer profession, i.e., computer science majors, are the major group, but apparently mostly in lower division classes. A member of that same panel, Clinton Fuelling, noted that "Plagiarism seems to be a continual problem in the teaching of computer science courses in which programming assignments are requirements" (26:263).

A year later, in 1983, R. Wayne Hamm did not select freshmen, introductory courses or non-majors when asserting that

"Too often students encounter classes in which plagiarism is rampant, ..." (24:248). All that we can assume from this is that plagiarism was thought to be a serious problem.

Repeating the observations of previous writers, Bensen states that "... [in those classes where plagiarism occurred] the background of the class in question was varied" (13:25). Finally, the more recently published works by Cook (16), Faidhi and Robinson (21) and Jankowitz (27) indicate the continuing concern with the problem of program plagiarism, e.g., Faidhi talks about "... novice plagiarism [being] ... quite a common occurrence" (21:11).

Three recent unpublished surveys (See Appendix E) conducted by this author support the continuing authenticity of the findings that were published at the beginning of the 1980's by Shaw.

1. In a survey (See Appendix C) taken in August of 1987 of the computer science departments at the 19 campuses of the California State University system, 14 of the 17 reporting do not have departmental policies on plagiarism, but rely on the general university rule which is taken from the California Administrative Code, Title V, Article 1.1, Section 41301. Part (a) of this section states that students may be expelled, suspended or put on probation for "Cheating or plagiarism in connection with an academic program at a campus" (2:66). It is left up to individual campuses to expand on the definitions of

'plagiarism' and 'cheating'.

2. In a survey (see appendix D) of May 1987 taken by the Computer Science Department faculty at Oregon State University, 100% of the responses--19 returned out of 29--thought that plagiarism was occurring. On a scale of 1 (not very serious) to 5 (very serious), the mean was 2.7 overall, but 3.23 for lower division service courses and 3.53 for lower division major courses. This clearly is somewhat at odds with many other reports in that plagiarism is perceived as happening throughout the curriculum, although more so in lower division classes, and it is not at all limited to non-majors.
3. The same survey as in #2 above was undertaken at Cal Poly San Luis Obispo in December 1987. Again of those responding (21 out of 28) 100% reported that plagiarism is occurring. On the scale of 1 to 5 (not very serious to serious), the overall average was 3.01, somewhat higher than at OSU, although not significantly so. Lower division service courses had a mean of 3.31, whereas lower division major courses had a mean of 3.0.

Thus the problem continues as the decade is coming to an end, much in the same terms as at the beginning. Plagiarism is thought to frequently occur, more so in large lower division courses than upper division and, it is generally perceived, more so in service courses than in major courses. It is considered a

moderately serious problem. Novice plagiarism is felt to be much more of a problem than sophisticated plagiarism.

Of particular interest is the lack of any quantification of the problem of plagiarism. All we have are the personal feelings and experiences of a variety of computer science faculty. There is really very little evidence as to the extent or the demographic characteristics of the problem.

CHAPTER IV

PLAGIARISTIC TECHNIQUES SUGGESTED BY STUDENTS

The literature available concerning program plagiarism is faculty focused. It represents their perceptions and value judgments. Plagiarism is abhorred, but it is usually discussed in the abstract. Moreover, even in those instances where specific types of plagiaristic behavior are listed, there is no way to determine whether the lists are in some sort of order of precedence or to what extent they are representative of the totality of actions that students might actually take. It would be of interest to discover what students would suggest as to the nature of plagiarism and what they might do if they were to set out to obscure the original characteristics of a copied program.

This chapter describes the results of two studies designed to solicit this information from students. Two techniques were used in an effort to collect data on students' perceptions on the question of plagiaristic practices. One was to conduct classroom surveys. Each student was asked to make a list of everything he/she could think of to do if the goal were to obscure the authorship of a copied program.

The second was to conduct a series of observations of

students as they worked at obscuring the original nature of a copy of a program. The student was requested to verbalize his/her thinking as decisions were being taken and as changes were being made at the terminal. This type of experimental design is called a protocol analysis (34)(36). Each student was given a listing (See appendices A and B) of the original program and a copy of the expected input and output. The student was allowed approximately five minutes to become familiar with the material. Then, after logging onto an account containing a copy of the program to be modified, the subject had thirty minutes in which to make changes. The vi editor was used by twenty out of the twenty-one subjects. One student had been interviewed at Oregon State University and had used the MacPascal editor. A protocol analysis is felt to be potentially more valid than either an observer simply taking notes on what was observed or the subject predicting what he/she would do before the process takes place or trying to recollect what took place after the process was concluded.

A general problem became apparent when the analysis of the data from both the classroom surveys and the protocol analyses began. The responses were both somewhat idiosyncratic and very diverse so that a scheme of categorization had to be developed. Two such schemes were eventually used. The first was to divide responses into two broad rubrics--cosmetic/trivial and non-cosmetic/logic--and list specific responses under the appropriate

group.

The second way was to use five functional categories:

1. Typographic: spacing/blank lines, indentation, variable names, comments, order of declarations, etc. All those ways that do not affect the semantics of the program in any way.
2. Logic: changing loop types, changing if-then-else to switch and vice versa, changing relational operator in condition, DeMorgan's laws, etc.
3. I/O and Files: strings and output form changed, I/O error checking, creating separate files, etc.
4. Modularization: combining or splitting functions or procedures, replacing parameters with globals and vice versa, etc.
5. Data Structures: changing from linked lists to arrays or vice versa, creating new temporary variables, creating or erasing of constants, etc.

FIRST ANALYSIS

CLASSROOM POLL RESULTS

Seven classes were surveyed (See Appendix E). Two of them were taken by sophomores (CSc 218), two by juniors (CSc 345) and three by a mixture of sophomores, juniors and seniors (CSc 204--C & Unix). So on average the students in the classes surveyed were juniors.

The individual responses have been grouped to get central tendencies. The individual responses themselves, since they do not refer to a particular program, nor even necessarily to a particular language, make the answers often more abstract and/or vague and thus less susceptible of being specifically listed in subcategories than the behaviors observed in the protocol analysis.

	Mentioned by % of the group
Cosmetic/trivial changes	100
1. Change variables	92
2-3. Other types of cosmetic changes	87
4. Change comments	77
Non-cosmetic/logic changes (Accepting the list as given below for the Protocol Analysis as a definition of this category.)	54
Mean of non-cosmetic/logic changes suggested per person for the seven classes.	72

PROTOCOL ANALYSIS RESULTS

There were twenty-one such studies conducted (See Appendix E). C was used in eighteen. Pascal in three. The average student was a senior: one sophomore, four juniors, twelve seniors and four graduate students. The mean number of courses in which they had used vi as an editor was 6.9. The mean number of courses in which they had used C--for the eighteen who did-- was 4.6.

	Mentioned by % of the group
Cosmetic/trivial changes	100
1. Change variables:	95
a. Identifiers, e.g., global and local	95
b. Procedure/function names	43
c. Array names	10
d. Constants	19
e. Tag/type names	14
2. Change strings	
a. Input	67
b. Output	81
3. Change general listing format	
a. Add blank lines	29
b. Use tabs, different indentation	62
c. Change order of declarations	19
d. Change order of functions, e.g., alphabetize them	34
e. Change alignment of braces, begin/ends, colons, etc.	67
f. Use multiple declarations, e.g., float a,b,c	24
4. Change comments	95

The following are the types of changes suggested:

a. Erase all comments, put in your own
b. Rewrite the comments by changing some words and phrases here and there
c. Outline the comment blocks with stars
d. Place the comments in different places
e. Comment each line, not just one block of statements
f. Overcomment or undercomment in terms of the original program
g. Change comment characteristics to one's own personal style

Non-cosmetic/logic changes

These are listed in decreasing order of frequency. NINETY-FIVE percent of the students (20/21) indicated at least one change in this category.

	Mentioned by
1. Add constants	9
2. Change to different type loop	7

3.	Take some in-line code and develop another function	7
4.	Change if/then/else to switch	5
5.	Combine 2 functions into 1	5
6.	Shorten main by grouping function calls with some commonalty	4
7.	Replace parameters with globals	4
8.	Replace globals with parameters	3
9.	Change switch to if/then/else	
10.	Create temp variables to lengthen formulas, etc.	2
11.	Change data structures, e.g., linked lists for arrays	2
12.	Modify formulas	2
13.	Reverse the logic on if/then/else e.g., change the else clause so that it is tested first where possible	2
14.	Use error checking on input statements	2
15.	Use functions/returns instead of parameters or globals	1
16.	Get rid of any temp variables	1
17.	Use loops for multiple spaces or stars for borders	1
18.	Create separate files and link	1
19.	Add date function to output	1
20.	Reverse comparisons, e.g., > -- <=	1
21.	Break up large procedures	1
22.	Apply DeMorgan's rules where possible with multiple comparisons	1

The following is a listing of how many non-cosmetic suggestions were made per person during the protocol study.

1. 2
2. 1
3. 1
4. 0
5. 4
6. 3
7. 1
8. 6
9. 4

10. 3 Mean = 2.7 suggestions
11. 1
12. 1
13. 1
14. 2
15. 2
16. 8
17. 3
18. 1
19. 4
20. 6
21. 3

The following is a sublist of language specific suggestions made during the protocol studies. Where relevant they have been included in the above lists. Thus this is not a disjoint set.

Peculiar to C (all but three students used C)

1. Add define statements to create constants	9
2. Eliminate newline statements, include newlines with other statements	8
3. Pretty print the file (use cb)	3
4. Use tab instead of spaces	2
5. Initialize the globals when declared	2
6. Add error check for scanf	1
7. Add 'void' before function name	1
8. Initialize the globals as a group, e.g., float a=b=c=0	1
9. Create and use a 'typedef'	1
10. Change arrays to pointers	1
11. Compile to different name than a.out	1
12. Rename source code file	1
13. Exchange 'A +=B' for 'A = A + B'	1
14. Break source file into several files	1

Peculiar to Pascal (three students)

1. Add complete list of 'forward' statements	1
2. Get rid of unnecessary 'forwards'	1
3. Use 'with do' wherever possible	1
4. Remove any semicolons before and 'end'	1

Summary of First Analysis

	Survey classes	Protocol studies

Students	155	21
Made cosmetic changes	100%	100%
Made non-cosmetic changes	54%	95%
Number of non-cosmetic changes per student	0.72	2.7

N.B., the students in the protocol studies almost all made non-cosmetic type changes and made approximately four times as many per student as those in the surveyed classes.

SECOND ANALYSIS

A Spearman Rank Correlation of the protocol studies and the class surveys was calculated using the five classes of changes. The order represents the volume of changes suggested or made.

	Survey	Protocol	
Typo	1	1	
Logic	2	5	
I/O	3	2	rho = .3
Module	4	4	
Data	5	3	

This was interesting, but would seem to indicate that something rather different was going on in the minds of the different groups. Upon looking at the material again, however, it was noticed that the majority of 'Data' entries for the protocol studies concerned the use of 'define' statements. When their weight was subtracted, on the basis of those being specific to the program language being used and its specific character, the results were

quite different.

	Survey	Protocol	
Typo	1	1	
Logic	2	4	
I/O	3	2	rho = .7
Module	4	3	
Data	5	5	

With the weighting scheme used, the differences among categories for the class survey results were larger than those for the protocol study.

Survey results

1. Typo	747	57%
2. Logic	250	19%
3. I/O	196	15%
4. Module	86	07%
5. Data	37	03%

Protocol results

1. Typo	107	38%
2. I/O	61	22%
3. Data	41	15%
4. Module	38	14%
5. Logic	32	12%

It was also of interest to determine whether the two groups would suggest or make changes in a similar order. The following represents the order in which a specific class of change was first demonstrated:

	Survey	Protocol	
Typo	1	1	
Logic	2	4	
I/O	3	2	rho = .7
Module	4	3	
Data	5	5	

Thus, the order which represented the number of changes in

these particular two studies was also the order in which the type of class was first suggested or made. The rho value indicates a substantial or marked correlation in both cases. Whether students are looking at a specific program or imagining what they would do with a hypothetical program apparently made little difference as to the order in which they would try a particular class of change.

Summary

All of the students participating in either the classroom surveys or the protocol studies mentioned cosmetic changes. There were noticeable differences, however, between the two groups as regards non-cosmetic/logic changes. Only fifty-four percent of the students in the classroom survey mentioned a non-cosmetic type change whereas ninety-five percent of those in the protocol study did so. The difference in numbers of such suggestions is even more striking. The average student in the classroom survey suggested .72 non-cosmetic changes, i.e., fewer than one change per student. The average student in the protocol study suggested 2.7 changes.

It was of interest to determine whether there was a correlation between the number of courses in which a student had used the vi editor and the number of non-cosmetic/logic changes which were suggested. One might hypothesize that the greater the facility with an editor, the greater the ability to plagiarize. There was, however, little or no significant relationship: rho = -.304.

There was a marked correlation between the order of changes recommended by the surveyed classes and those recommended by the students in the protocol study when a language specific item was removed (rho = .7). This is of particular interest because the students in the study represent various experiences with C, Pascal, Modula2 and Fortran. The results may then represent a general attitude toward program changes by students who have had training in high level languages.

Finally, although the students in the survey and those in the protocol study tended to approach the task in a similar manner, the distribution of effort was quite different. Fifty-seven percent of the effort in the surveyed classes would have gone for cosmetic changes, whereas only thirty-eight percent of the effort in the protocol studies was applied in this category. Overall there was a more even distribution of effort among the different categories of changes by the protocol students.

Reviewing the class surveys and the protocol studies in another light, it was found that the class surveys, on the whole, contained more abstract statements, whereas the protocol studies were more specific. Clearly the people in the protocol studies were responding to having a concrete object on which to focus. This would argue that their responses toward programmatic changes are more realistic.

CHAPTER V

SUGGESTED METHODS TO CURB PLAGIARISM

A Similarity Checker

A number of the articles discussed various metrics which could be used to measure the degree of similarity of the programs handed in for a particular assignment (14,15,19,21,23,27,30). Most of the articles specifically mentioned that student awareness of an automated means to detect seemingly unreasonable similarity should discourage plagiarism. The more direct goal of the use of various metrics would be to detect suspect programs.

Practical Advice

Several authors listed one or more tactics that a teacher might use to prevent or diminish the amount of plagiarism. Darrell Criss offered a list of do's and don'ts.

1. Do not use same problem assignments over and over, thus preventing copy returned solutions from fraternity or sorority files.
2. Require different style header comment each year that a course is given.
3. Encourage student to use a unique style or language for program annotation--embedded comments. This does not counter the use of standard program statement format, which is important in their training for future commercial or industrial work.
4. Make it well known that you do check for submissions being exact or near copies of a fellow students work--and levy a penalty or reject the work

if it is clear that one or the other has not done his own work.

5. Periodic quizzes on specific details of required programs will force students to do their own work in order to be prepared for the quizzes. (26:262-263)

Darryl Gibson pointed out that cheaters often wait for bright students to discard intermediate copies of their assignments. Procedures for the proper disposal of such material would avoid this problem (26:263).

Jerry P. Harshany takes a less serious view of the problem.

> I have never considered the detection of plagiarism to be worth the time and effort that is required, where this aspect of the grading process is one of the major goals. Blatant cases may, of course, not be ignored or passed without a comment.
>
> I rely on several methods for "discouraging" (is this preventing?) plagiarism and encouraging self-expression in a program. (26:264)

Positive Techniques

Betty Hwang found that requiring a structured walkthrough, for instance on a day the logic design was due, was a highly effective way to prevent cheating. Peer pressure at that time creates a strong motivation to come prepared rather than publicly demonstrate one's lack of understanding (26:264).

Ernest Ferguson reports on the use of a conference grading method. Each student in the class is to sign up for a one-on-one fifteen minute conference with the professor. At that time the student is to submit his/her final listing and output. During the conference each student is asked to explain the algorithm(s) used in the program. The professor grades the work and explains how the

grade reflects the style, syntax, etc. of the program (22:361-365).

Hwang and Gibson argue that

> Much of this problem [cheating on programs], certainly not all of it, could be alleviated if we who are charged with the preparation of future computing professionals could guarantee, to the extent possible, that our graduates have in fact learned the material and are in fact competent ... We can do this only if we can design and adopt practices which will systematically require the students to master both the theoretical and practical aspects of the discipline. (25:52)

To achieve their goal they examine five different types of combinations of examinations and program assignments in regard to how each is weighed when generating the final course grade. There is a great deal of detail with pro's and con's pointed out, but the following gives a brief idea of their ideas:

1. Exams given greater weight than programs. More negative characteristics than positive ones.
2. Programs given greater weight than examinations. More negative characteristics than positive ones.
3. Examinations and programs given equal weight. More negative characteristics than positive ones.
4. Final exam given all weight. More negative characteristics than positive ones.
5. This entry is split into X and Y subtypes. Both are representative of a function where the final grade on the programming project is derived from relating the score on the relevant quiz to the provisional score on the project. X--multiply the percentage made on the quiz with the

provisional score on the project to obtain the final project grade. Slightly negative evaluation.

Y--add the score made on the quiz to the provisional score for the project to obtain the final project score.

Type 5.Y was felt to be the best method. It was not perfect, but it was significantly superior to types 1, 2, 3, and 4. It was somewhat better than type 5.X.

More Elaborate Procedures

Mary Shaw's committee report for the Computer Science Department at Carnegie-Mellon University (33) is the description of an effort to develop a formal document concerning cheating that would implement the general university policy by addressing the unique problems of program plagiarism. Her report is an elaboration of her overall recommendation to her department.

... Specifically, the Department should

> •establish an interpretation of cheating in computer science that supplements and extends the University definition of cheating and plagiarism
> •develop technological and policy mechanisms for preventing and detecting cheating
> •set forth procedures and sanctions for dealing with cheating incidents, and uniformly enforce them (33:72)

Basically the suggested procedure is an attempt to legislate against behavior which is believed to be undesirable. Professors are to be responsible for ferreting out wrongdoers. Possible punishments are delineated for various degrees of unacceptable activities. The report goes into some detail to explain the need

for such procedures and the negative impact on all concerned if cheating is allowed to occur without any adequate attempt at prevention.

Janet Cook has also been involved with an effort to develop policies and procedures concerned with student misbehavior in a computer science environment. Her paper is, however, more student oriented. She emphasizes that "Students are unsure of what is expected of them" (16:462). The body of her article contains two sample policies. One deals with a policy toward microcomputer software. The other, and by far the more involved, deals with the ethics expected of computer science students in relation to individual and group projects, behavior in any of the labs, work with software and files and the accessing of computer budget accounts, i.e., the ethical and unethical usage of one's own and others' computer accounts.

In an effort to ensure that the students who will be affected by these policies will be cognizant of them, each student is given a copy of the document and required to sign and date a statement that he/she has read it. Overall, Cook's procedures and policies are as elaborate as Shaw's, although they cover somewhat different concerns. Both documents represent a high degree of positive effort and address significant problems. They are both examples of an effort to come to grips with unacceptable student behavior by explanations, rules, regulations and proposed punishments.

Some Pedagogical Concerns

While voicing their opprobrium towards cheating and plagiarism, several individuals also tried to call upon a larger context to give some sense of direction to their concerns.

Dodrill muses that

> The primary difficulty in teaching computer programming is not necessarily centered around detecting and punishing cheating cases, but rather on how to teach a discipline with the unique characteristics of computer programming in a way that will encourage individual effort and reward individual achievement. Examples of questions which might be posed in order to improve teaching methods include: How can student interest in computer programming be stimulated? What can be done to reduce the frustrations inherent in writing and debugging code? What should be expected (and what should not be expected) of students taking introductory programming courses? How can individual performance and achievement be measured effectively for grading purposes. (29:26)

Doris K. Lidtke puts her finger on an important aspect of computer science as a subject:

> In computer science it is particularly valuable for students to work cooperatively. Throughout their professional careers they will be working in teams and it is a poor educational system which does not prepare them for this. We should foster teamwork, rather than isolated individual activity; we should train students to work together, rather than looking upon it with suspicion; and we should encourage the sharing of ideas, rather than a jealous secrecy. There is nothing inherently unethical about such collaborative work. At the same time it is incumbent upon instructors and the profession in general to encourage mutual honesty, open frankness about how results have been achieved and enthusiasm for a subject which can be approached cooperatively. (29:27)

Summary

There is a wide enough variety of suggestions. Similarity checkers are basically a negative action. They are used to catch

cheaters. They would also seem to have some discouraging effect. A number of ad hoc practical pieces of advice could be helpful. They amount to specific means to make it harder to cheat. Harshany offers the clearest and easiest advice to follow. He does not think that plagiarism is normally a big enough problem to worry about.

There were several positive techniques offered: a structured walkthrough, conference grading and a specific mixture of testing and grading. They are of particular interest because they offer the possibility of improved learning alongside of a reduction in cheating. The article by Hwang and Gibson on using testing procedures to ascertain the understanding of the programming assignment seems particularly pertinent since it will involve the instructor in a greater effort to write relevant examinations and motivate students to better understand the programming assignments.

Two articles specified in great detail the establishment of policies aimed at defining, preventing and, if necessary, punishing cheating. Such policies and procedures may be necessary, but they too, are basically negative and will do little to improve learning.

Dodrill and Lidtke asked the more far-seeing questions. Rather than concentrate on the discordant aspects of computer science in an effort to suppress them, we should ask what the nature of the discipline is and how best to prepare students for

it. In the process we may reduce our problems or find ways to handle them differently.

All the suggestions were felt by their authors to have been of benefit. Perhaps anyone getting involved and doing something has a positive effect. It is not that they are not useful. It is very difficult to accumulate any but circumstantial evidence as to the effect of these differing techniques. How does one measure the degree that a policy has curbed some behavior, especially if there is very little data as to the extent of the behavior?

CHAPTER VI

AN EXPOSITION AND COMPARISON OF SOME SUGGESTED METRICS

A Review of Relevant Articles

The early paper by Karl Ottenstein (30) represents an effort to use Halstead's basic software science parameters, i.e., a four tuple of size measurements of the number of unique operators, the number of unique operands, the total number of occurrences of operators and the total number of occurrences of operands, to detect similarities among student homework papers written in FORTRAN. Ottenstein's work is based on some observations made by Bulut in his Ph.D. dissertation. Ottenstein quotes Bulut as stating that the chances of two programs having equal four tuples was "slim" (30:31).

Ottenstein's paper is both an expansion of Bulut's work and a critique of it. He quantifies "slim" by noting that:

> Assuming [a] ... normal distribution [of programs], there is clearly a greater likelihood of finding a pair of independently written programs with equal parameter values near the means as there is of finding such a pair with values on the tails. Thus, we can be more confident of a partition's accuracy as its individual parameter values approach the tails of their distribution curves. (30:31)

He cautions that the method as it stands is only valid for cosmetic alterations. Furthermore it can only be usefully applied with the assumption that entire programs have been copied. However, Ottenstein asserts that " ... Most non-cosmetic

alterations fall into one of six well-defined impurity classes, all of which are detectable by a slightly more sophisticated counter" (30:31). He defines impurity classes by citing the following as listed in Bulut:

> (1) self-canceling operations
> (2) ambiguous usage of an operand
> (3) synonymous usages of operands
> (4) common subexpressions
> (5) unnecessary replacements
> (6) unfactored expressions (30:32)

Finally, Ottenstein concludes that this method can be applied to other programming languages. However, earlier he had noted that the ideal function which would <u>prove</u> plagiarism is unobtainable because

> ... it is possible for identical work to be performed independently, the semantic equivalence of two items cannot always be shown deterministically, and there is a subjective area between plagiarism and paraphrasing. (30:30)

Thus this key paper offers a technique immediately applicable to classes taught in FORTRAN where the instructor's goal is to detect similarity between whole programs where one of these may be a copy with only cosmetic changes. Whether two programs with equal four tuples represent an example of plagiarism is less likely if their tuples lie near the mean of the tuples of the other programs handed in for that assignment. In any case, the degree of similarity cannot prove plagiarism because of subjective and non-deterministic aspects of programming. Ultimately the instructor must exercise his/her judgment.

Robinson and Soffa used Ottenstein's techniques as a control method in their project whose purpose was to develop a tool to aid in program advising (32). The language used was FORTRAN. Their own method for detecting possible collaborators used code optimization techniques and the following steps:

1. Group the programs by the number of leaders.
2. Compare the number of statements in each basic block. Eliminate the programs which match less than 50% of the time.
3. Compare the control structures and retreating edges. Eliminate the programs that have different values.
4. Compare the data structures. Eliminate programs with a difference of more than one for each data type.(32:125)

Robinson and Soffa did not discuss the effectiveness of their software tool in terms of cosmetic versus non-cosmetic changes. Hence, there was no testing done along those lines. Their tool was successful " ... in calling attention to a greater number of possible copies. The final evaluation must be made by the instructor" (32:125). In comparing the results of their scheme and the results of running the same data using the Ottenstein technique they concluded that the "Ottenstein approach is an effective but conservative approach." (32:125), i.e., it used too fine a mesh.

Donaldson et al. alludes to the Ottenstein technique (19). Their paper was concerned with FORTRAN, but it was noted that the method had been used with Cobol and BASIC. In reference to Ottenstein, it was asserted that for program assignments of the size typical of introductory classes there was a greater than "slim"

chance of tuples being quite similar. This is not as pertinent as it might appear, since Ottenstein himself had stated that Bulut's original notion of the significance of unique tuples had to be modified in terms of the mean of the group of papers submitted.

After a brief discussion of Ottenstein's methods, the size measurements used by Donaldson were elaborated upon. The Computer Science Department at Bowling Green University had recently implemented an automatic detection system. It was felt that those students in introductory classes who did plagiarize used quite simplistic techniques, e.g., renaming variables, changing the ordering of statements, and changing format statements. Therefore, a detection scheme was developed to measure the following:

1. Total number of variables
2. Total number of subprograms
3. Total number of input statements
4. Total number of conditional statements
5. Total number of loop statements
6. Total number of assignment statements
7. Total number of calls to subprograms
8. Total number of statements of type 2-7 (19:22)

In addition, the sequence of statements in the program were represented by a coding scheme, e.g., VVV===RHR=DI=EE ... would represent three declaration statements, three assignment statements, Read, While loop, Read, assignment statement, Do loop, If Then, assignment statement, End IF or While, End IF or While, etc. (19:22-23).

After the data analysis was accomplished, the degree of similarity or difference between the counters of any two assignments was determined. Later an algorithm was used to

match the code sequences of different programs to try to find a match. Then, an instructor, by comparing both the counts and the structures of two programs, could make a judgment as to the possibility of plagiarism.

The paper gave no mention of any attempt to use any other technique as a control nor was there any mention of using the technique itself in an experimental situation. It concluded by stating that:

> ... Already, the system has proved to be a useful tool. it has enabled instructors who have used it to detect cases of plagiarism that had gone unnoticed by the graders. (19:25)

Sam Grier at USAF Academy applied the Halstead measures to an introductory computer science course in Pascal. It was asserted that "Sophisticated plagiarism is not the problem ..." (23:15). Ottenstein's basic ideas were applied, but three more measures were added in an effort both to adjust the technique to Pascal and to include some means to circumvent several tricks which might skew the measurements. The three additional measures were code lines, variables declared (and used) and total control statements. Grier labeled his program 'Accuse'. As an example of the effort to fine tune the measurements:

> ... for every assignment operator found, two operands are subtracted from "total operands," and "code lines" is decremented. This should prevent Accuse from being misled by unnecessary initializations and unnecessary assignment statements.

And,

> Accuse is also selective about what it calls operators. Software Science considers a BEGIN END combination as an

> operator .... Because BEGINs and ENDs can be added to Pascal code where not required, Accuse chose to ignore them. Parentheses and several other operators are ignored by Accuse for essentially the same reason. (23:16)

Grier noted that Accuse will not uncover changes made by the sophisticated plagiarist (23:15). It will not prove plagiarism, only indicate its possibility. There was no attempt made to compare the results of Accuse with other plagiarism detection tools. The testing of Accuse reported in the article was modest. "The correlation scheme is admittedly ad hoc. The only thing that can be said in its defense is that it seems to work" (23:18).

Berghel and Sallach's 1984 study (14) presented a comparison between the Halstead metric as presented by Ottenstein and a list of features they finally settled upon after using various statistical measures. The programmers were considered to be "novice" and the programs were in FORTRAN.

Their general findings were:

> Once the two metrics had been calculated, the validity of their profiles was estimated by comparison with the independent judgment of the graders. The general result of these comparisons is that the Halstead metric consistently detected similarities which did not exist. The alternative metric, in contrast, showed itself to be consistently more reliable. (14:68)
>
> ... Only if most programs fell into the narrow range where the Halstead features are focused could that metric provide superior identification of program similarity. (14:69)

Thus, the Halstead metrics used provided too broad a mesh.

Berghel and Sallach go on to make several interesting comments which are relevant to the entire question of developing a theory from which eventually a valid plagiarism detection tool

might be drawn.

> ... Our factor analysis demonstrates that a number of features, of which Halstead's compose only a part, appear to lie along the same underlying dimension.
> ...
> We are forced to conclude that there is nothing unique about the features isolated by the Halstead metric. While the application of quantitative methods to program structure has shown itself to be productive, the Halstead features seem to have no unique theoretical or practical properties which make them singularly effective indicators of program structure. More specific features (such as a count of the frequency of a specific operator like assignment statements) and more general features (such as total code lines, or perhaps even the size of an object module) may be equally or more effective than the components of the Halstead metric. ..., the isolation of the most powerful indicators ... is a task which is as yet incomplete. More fundamentally, perhaps contextual factors will prevent <u>any</u> set of features from achieving this type of conceptual primacy. (14:70)

Faidhi and Robinson's paper (21) dealt with the interesting question of the degree of sensitivity of plagiarism detection schemes. Their main concern was the plagiarism being done by novice programmers in introductory courses.

The study was in two parts. One was concerned with testing a variety of metrics to ascertain whether they inappropriately suggested plagiarism. This was found to be so for the Halstead primitive software science measures used by Bulut and Ottenstein (21:12). It was also found to be the case as regards " 'derived' software science" measures (21:15) and as regards a set of metrics previously used by the authors themselves to reflect an evaluator's opinion of a program's style (21:15). All three metrics found extensive plagiarism where there was none. There was no effort made to discover if the three would not find

plagiarism where it was present.

The second aspect of the study was to select two sets of measures which would hopefully be less broad in detecting plagiarism. One set was a list made up of general features that it was believed a novice programmer would most likely alter, e.g., number of comment lines, number of blank lines, average procedure/function length and number of reserved words. (21:15) The other set was an attempt to form a list which represented "hidden" features or invariants (21:15-16). In combination, the two sets contained twenty-four items. When these items were applied to the test sample used for part one of the study, the new measures reported no plagiarism. Thus, the combined set was less broad than the metrics tested in part one. In addition, the authors were able to conclude that the empirical metrics set, i.e., the combined one they had just constructed, was a minimal set.

As a last test, they took a student program and transformed it successively through six levels of ostensible complexity (See pp. 16-17). At each level all of the metrics used in the study were utilized. Their conclusion was:

> ... We notice that all the metrics detect that these programs are variants of the original. However, the correlation difference between level 1 and level 6, and between metric sets, shows the increased sensitivity of the empirical metrics to the changes made in the original program. ... The other metrics [from the first part of the study] ... will continue to suspect plagiarism even when the original program has been so significantly changed that it can no longer be considered to be copied (and if it were, would require a plagiarizing skill exceeding that required

to produce an independent program in the first place). (21: 18)

The evidence given for the broadness of several of the metrics used in the study is of significance. The second part of the study seemed to support the argument that those same metrics do, however, at least point to plagiarism when it is present. The merit of the combined set of measurements in not finding evidence of plagiarism where it is not present is extremely valuable. It is intriguing, perhaps shedding some needed light on our general concept of plagiarism, that the more a copied program is modified, the less we might consider it a plagiarism. If this would become a standard perception, it would certainly be useful to have a tool such as developed by Faidhi and Robinson that would indicate by means of a correlation coefficient the relationship of the copy to the original. (21:18)

Jankowitz (27) has taken a different direction in his effort to develop a tool to detect program plagiarism. He alludes to earlier approaches "mostly based on Halstead's" metrics. His work, however, is concerned with " ... the order in which procedures are referenced during static execution, ..." (27:1). If the program is too small, e.g., consisting of only three or four procedures, the order of procedure calls is trivial and he would then compare "each procedure in the first program with every other procedure in the second program" (27:1).

The patterns generated by the technique can be obscured by splitting or merging procedures, but unless this were done for

every procedure in the program, parts of the original pattern would still be identifiable. Jankowitz uses the rule of thumb that more than 50% of the procedures must match before plagiarism is seriously considered (27:5).

In initial testing several cases of plagiarism were detected. Perhaps the wider application of such a scheme is of even more interest, i.e., as a means to 'fingerprint' a program.

Summary

A number of the above studies utilize Halstead's software science metrics in one way or another. Probably the first effort to apply the Halstead ideas to program plagiarism was Ottenstein's. In many ways the work was prototypical. It was focused on the assumption that similarity of program structure, i.e., invariants, is evidence of plagiarism, but could never be considered proof. It sought to ascertain the degree of similarity between entire programs. It could not be used to compare a 'mixed' copy with an original. It attempted to ferret out novice plagiarism, i.e., plagiarism based on cosmetic changes.

Various additional measures have been either added to the Halstead metrics or used in comparison with them in an effort to test for similarity, e.g., Robinson and Soffa utilized code optimization techniques in order to develop tallies of different characteristics. These could then be compared among a set of programs. Overall, however, there has not been very much

replication of experimentation. Only a few of the schemes have involved any control groups. The validity of the techniques is often open to question, e.g., different researchers have made contradictory claims as to whether the Halstead measurements are too fine or too broad. The size of the samples involved in the experiments is often quite small.

Berghel and Sallach had one of the better experimental designs. Part of their conclusions are especially pertinent as they argue that the Halstead metrics have no unique properties that make them any more effective than other metrics. Moreover, perhaps "contextual factors will prevent <u>any</u> set of features from achieving ... conceptual primacy" (14:70).

Faidhi and Robinson have taken the interesting tack of developing a complex metric and then testing it against an original program, which they then successively altered in ways suggested by a taxonomy of program complexity which they had developed. They reported great accuracy in determining that the repeated modifications were, in fact, a form of the original. This seems a promising direction to take, but apparently no test was conducted on actual student papers, nor has there been a follow up study reported.

Jankowitz mentions the Halstead metrics and points out that most of the previous approaches to the problem of program plagiarism are based on them and that they attempt to "analyze a procedure without regard to the context in which it is

referenced" (27:1). By utilizing the order in which procedures/ functions are called during static execution, he is not limited to novice plagiarism or an inspection focused primarily on whole copies. The system seems promising, but further testing is necessary.

The use of metrics to test for similarity of structure has been productive of a number of interesting and useful ideas, but so far there has been a number of difficulties, both theoretical and practical in nature, which has frustrated the attempt to develop a practical, reliable and valid means of determining program plagiarism. Practically, the tools available as of yet are too limited and relatively untested.

Moreover, if the results of the protocol studies can be shown to be generally true, any metric which focused primarily on cosmetic changes would yield invalid results in perhaps a majority of the cases, i.e., either too many instances of plagiarism would be missed or too many innocent cases would be flagged. Cheat detectors based on the Halstead metrics would be fooled by many of the changes suggested and carried out by the students involved in this study.

Of particular interest at this point are the protocol results in which 95% of the subjects suggested an average of approximately three types of non-cosmetic changes per person. Given this finding, the two most promising metrics would be those of Faidhi and Robinson and those of Jankowitz. The metrics of

Faidhi and Robinson have not been tested on actual programs, but appear to be able to detect similarity where significant non-cosmetic modifications have been made. Their taxonomy is not grounded in any theory of program complexity, nor does it appear to include changes in data structures, but it does represent an interesting effort to provide a structure by which to categorize the types of changes which can be made.

The work of Jankowitz takes a different approach entirely. It is the only one that fits Berghel and Sallach's concern about a means of involving context in the utilization of the metrics. If the program has few procedure/function calls it is clearly ineffective, however. In larger programs it would be effective as long as a student did not extensively modify the nature of the procedures in the sense of the order in which they were called. Since this type of change was observed several times in the protocol studies, the method would appear to be again a partial means and would need to be combined with something like the metrics suggested by Faidhi and Robinson.

CHAPTER VII

SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

Background

There has been a continuing concern over the extent of program plagiarism. It is widely felt to exert a negative impact on all involved: the students innocent or guilty, the instructors, the university, and the general public who hire and/or use the services of students who have taken computer science classes.

In spite of the efforts which have been made to analyze the problem, to find means to determine to what extent program plagiarism has taken place, to develop policies to deal with it and, in general, to curb it, the issue is still largely unresolved. A primary reason for this is the lack of a clear and widely accepted definition of program plagiarism and the concepts related to it. There is also a lack of information as to what students actually and typically do when they plagiarize a program.

The Problem

The purpose of this study was to analyze the concept of computer plagiarism and those concepts related to it and to seek to discover patterns in student attempts at program plagiarism. This was done by seeking answers to the following questions:

1. Is there a coherent, generally accepted agreement as to the concept of program plagiarism?

   No, there is not. There is an agreed upon core, e.g., someone copying an entire program. There are too many gray areas, however, where it is neither clear to the instructor nor the students as to what is acceptable behavior.

2. Can it be shown that program plagiarism is a serious problem in computer science classes?

   No it cannot. There is a widely held feeling among computer science faculty that there is a problem, but there is no objective data to substantiate the rate of incidence, so it is difficult to judge the seriousness of the matter.

3. Are there patterns involved in program plagiarism? Can it be demonstrated what changes students are more or less likely to make?

   Both the classroom surveys and the protocol analysis support the concept that there are patterns as to the type of changes which are likely to be made, the order in which they are made and the relative proportion of one type of change to another.

4. Can it be determined which of the suggested means to control or reduce program plagiarism are effective?

   No it cannot. All of the suggested mechanisms would

probably be of some use, if only to reduce the temptation to cheat. It may be of greater importance to seek out means which will enhance the learning process, e.g., a better means of testing understanding, than to put into place a means of catching students who have given in to the pressure to plagiarize. There is no direct evidence, however, as to either the relative or the absolute merit of any of the suggested schemes.

5. Do any of the metrics proposed as a means to uncover suspected program plagiarism effectively discover actual instances of such plagiarism and, at the same time, not falsely report programs that were not plagiarized?

   None of the proposed metrics are adequate to discover the range of plagiaristic techniques suggested by the protocol studies. None of them have been used across a variety of languages with adequate statistical controls and adequate samples of students to ensure a high degree of validity. Thus none of them can be relied upon.

Conclusions

There is no right answer in an absolute sense to a question of the definition of a term. A term such as 'plagiarism' serves a social need in a given context. We need to develop our meaning of the term to meet the practical needs of computer science.

We need to ask "What are the practical results of this definition over that one?" "How can we make the term meaningful

to our students?" Any effort to be successful will have to take into consideration and incorporate the related concepts of collaboration, consultation, paraphrasing and common knowledge.

In a larger sense we need to develop a coherent vision of professional behavior as it relates to computer science and develop ways to communicate this to students. There should be a clear articulation between being a student of the discipline and being a professional person.

Recommendations

1. The development of a broader concept of what is and what is not computer plagiarism. It should be generally acceptable and applicable to all computer science courses. It should involve an examination of the related concepts of common knowledge and collaboration.
2. A continuing examination of the application of metrics to the problems of similarity and invariance. The practical need of devising a simple yet valid plagiarism checker is important. The theoretical need to more clearly determine what is "similarity" and what are "invariants" in the field of computer programming has important implications for the concepts of plagiarism, common knowledge and collaboration.
3. Independent of the extent of program plagiarism, the development of ways to promote more authentic learning of the subject matter and professional attitudes of computer

science are important and relevant. Work should continue on the pedagogical, tactical and administrative means to achieve reasonable and valid academic goals for all students.

A P P E N D I C E S

```
program Prufung(input,output);
(* John B. Connely
   July 20, 1988

   This program accepts as input the id number, the number of
   hours worked, the rate of pay and the number of dependents
   claimed. After the data for one person has been input, the
   program simulates printing out a check for the EXOTIC SofA
   COMPANY of Corvallis.
   Each check will list the ID number, the gross pay, federal
   and state taxes, the social security contribution and the
   netpay.
*)


type
    Employee_model = record       (* standard employee record *)
      SS_num : integer;
      Hours,
      Rate : real;
      Num_of_dep : 1..10;
      Gross_pay,
      Fed_tx,
      State_tx,
      Social_security,
      Net_pay : real;
    end;
var
   Answer : char;          (* Answer and Continue are both used to control *)
   Continue : boolean;     (* the loop in MAIN.                            *)
   Indiv_Person : Employee_model;

procedure StateTax(var Persons: Employee_model); forward;

procedure FedDependents(var Persons: Employee_model); forward;

procedure StateDependents(var Persons: Employee_model); forward;

procedure InputRecord(var Persons : Employee_model);
 (* input module of personnel data *)
begin
        writeln;
        write('Please input the social security number. ');
        read(Persons.SS_num);
        writeln;
        write('Please input the hours worked. ');
        read(Persons.Hours);
        writeln;
        write('Please input the rate of pay. ');
        read(Persons.Rate);
        writeln;
        write('Please input the number of dependents. ');
        read(Persons.Num_of_dep);
        writeln; writeln;
end;
```

```
procedure ComputeGrossPay( var Persons: Employee_model);
(* To determine the gross pay based on regular rate for the first 40
   hours, time and a half for the next ten hours, and double time for the
   number of hours worked over 50 hours per week. *)
const
  First_overtime = 10;       (* the first 10 hours after 40 *)
  Standard_week  = 40;       (* standard work week *)
begin
  with Persons do
    begin
      if Hours <= 40 then Gross_pay:= Hours * Rate

        else if (Hours > 40) and (Hours <= 50) then
           Gross_pay:= Standard_week * Rate + (Hours - 40) * (Rate * 1.5)

           else
              Gross_pay:= Standard_week * Rate + First_overtime * (Rate * 1.5)
                          + (Hours - 50) * (Rate * 2);
     end
end;

procedure FedTax(var Persons : Employee_model);
(* To figure the federal tax based on the gross pay of each individual.
   A deduction is made based on the number of dependents claimed *)
const
    Higher_fed_rate = 0.28;    (* the federal tax rate above $500 per week *)
    Lower_fed_rate  = 0.20;    (* the federal tax rate if below $501 per week *
begin
    with Persons do
      begin
        if Gross_pay <= 500 then
            Fed_tx := Lower_fed_rate * Gross_pay
          else
              Fed_tx := Higher_fed_rate * Gross_pay
      end;
      FedDependents(Persons);
        (*  to calculate the deduction from the federal
            tax obligation due to number of dependents *)
end;

procedure FedDependents; (* var Persons: Employee_model *)
(* calculates a deduction based on the number of dependents claimed *)
var
  People : integer;
begin
  People := Persons.Num_of_dep;
  case People of
    0: ;

    1: Persons.Fed_tx := Persons.Fed_tx -  5;
       (* 1 deduction claimed equals $5 deducted from the tax obligation *)

    2: Persons.Fed_tx := Persons.Fed_tx - 10;

    3: Persons.Fed_tx := Persons.Fed_tx - 15;
```

```
    4: Persons.Fed_tx := Persons.Fed_tx - 20;

    5: Persons.Fed_tx := Persons.Fed_tx - 25;

    otherwise
      writeln('There is an error in the dependent input. ');
  end;
    if Persons.Fed_tx < 0 then Persons.Fed_tx := 0;
     (* No one should have a deduction greater than the tax owed *)
end;

procedure StateDependents; (* var Persons: Employee_model *)
(* A deduction from the state tax based on number of deductions *)
var
  People : integer;
begin
  People := Persons.Num_of_dep;
  case People of
    0: ;

    1: Persons.State_tx := Persons.State_tx -  2;
       (* A $2 deduction for each claimed dependent *)

    2: Persons.State_tx := Persons.State_tx - 4;

    3: Persons.State_tx := Persons.State_tx - 6;

    4: Persons.State_tx := Persons.State_tx - 8;

    5: Persons.State_tx := Persons.State_tx - 10;

    otherwise
      writeln;
  end;

  if Persons.State_tx < 0 then Persons.State_tx := 0;
     (* No one should have a deduction greater than the tax owed. *)
end;


procedure StateTax; (* var Persons : Employee_model *)
(* To calculate the correct state tax due based on the level of gross pay
   earned per week *)
const
    Higher_state_rate = 0.09; (* the state tax rate above $500 per week *)
    Lower_state_rate  = 0.04; (* the state tax rate if below $501 per week *)
begin
  with Persons do
    begin
      if Gross_pay <= 500 then
         State_tx := Lower_state_rate * Gross_pay
      else
         State_tx := Higher_state_rate * Gross_pay;
    end;
    StateDependents(Persons);
  (*   To deduction an amount from the state tax based on the number
```

```
      of dependents claimed. *)
end;

procedure ComputeSS_Deductions(var Persons : Employee_model);
(* To figure the amount of social security tax owed based on a flat rate of
   8 percent of the gross pay. Called by ComputeNetPay*)
const
  SS_rate = 0.08;    (* standard percentage of salary for social security *)
begin
  Persons.Social_security := SS_rate * Persons.Gross_pay;
end;

procedure ComputeTaxWithholding(var Persons: Employee_model);
begin
   FedTax(Persons);
   StateTax(Persons);
end;

procedure ComputeNetPay(var Persons: Employee_model);
(* Figures net pay from amounts figured previously. *)
begin
   ComputeTaxWithholding(Persons);
   ComputeSS_Deductions(Persons);
   with Persons do
     begin
       Net_pay := Gross_pay - ( Fed_tx + State_tx + Social_security);
       if Net_pay < 0 then writeln('Error in netpay figure. ');
     end;
 end;

procedure PrintCheck(Persons : Employee_model);
(* Procedure to print out the simulated check for each employee. *)
begin
  writeln;
  writeln;
  writeln;
  writeln;
  writeln('*************************************************************');
  writeln('*                    THE EXOTIC SofA COMPANY                  *');
  writeln('*                       Corvallis, Oregon                     *');
  writeln('*                                                             *');
  writeln('                         *');
    write('* SS# = ',Persons.SS_num);
  writeln('                                        *');
    write('* Gross Pay = ',Persons.Gross_pay:12:2);
  writeln('                                        *');
    write('* Federal Tax = ',Persons.Fed_tx:10:2);
  writeln('                                        *');
    write('* State Tax = ',Persons.State_tx:12:2);
  writeln('                                        *');
    write('* Social Security = ',Persons.Social_security:6:2);
  writeln('                                        *');
  writeln('*                                                             *');
    write('*                                      NETPAY = ',Persons.Net_pay:10:2);
  writeln('       *');
  writeln('*                                                             *');
```

```
  writeln('************************************************************');
  writeln; writeln; writeln;
end;


begin                                   (* M A I N *)
   Continue:= true;
   while Continue do
      begin
        InputRecord(Indiv_Person);
        ComputeGrossPay(Indiv_Person);
        ComputeNetPay(Indiv_Person);
        PrintCheck(Indiv_Person);
        writeln('To process another employee input a "Y", else input an "N" '
        readln(Answer);
        if (Answer = 'Y') or (Answer = 'y') then Continue := true
           else Continue := false;
      end;

end. (* End of program *)
```

```
/* This program is to accept interactive input from a
keyboard and produce paycheck information. It consists of
functions which will:
            1. Request the name, id number, hours worked
               per week and the rate of pay.
            2. Use the hours worked per week and the
               rate of pay, to calculate the grosspay.
            3. Use the grosspay to figure the tax due.
            4. Use the grosspay and the tax due, to figure
               the net pay.
            5. When all the input has been processed, the
               individual paycheck information and the
               cumulatative totals will be printed out.
*/


#include <stdio.h>


struct person{
/* This is an array of records which will be used to load
all the input data and the results of the functions. Then
the information will be printed out for each person.
*/
  char name[20];
  int id;
  float hours;
  float rate;
  int depend;
  float grpay;
  int deduction;
  float tx;
  float net;
  } payroll[20];


  float totalgrosspay;
  float totalnetpay;
  float totaltax;
  /* These global variables are to be used to accumulate the
  total amounts of gross pay, net pay and tax for any given week.
  */

  dependents(ii)
  int ii;
  /* This function will calculate the tax deduction in dollars
  generated by the number of dependents claimed.
  */
  {
     int deduct;

     int t;
     for (t = 1; t <= ii; t++){
       deduct = payroll[t].depend;
       switch (deduct)
       {
```

```
        case 0: payroll[t].deduction = 0; break;
        case 1: payroll[t].deduction = 5; break;
        case 2: payroll[t].deduction = 10; break;
        case 3: payroll[t]. deduction = 15; break;
        case 4: payroll[t].deduction = 20; break;
        default : payroll[t].deduction = 25; break;
       }
   }
}


netpay(ii)
int ii;
/* This function will calculate the netpay by subtracting the
deduction from the tax and the tax from the grosspay. At the
end of the function it will add the current net pay to the
accumulative total net pay and add the modified tax to
the cumulative total tax.
*/
{
   float grospay;
   float tax;
   int t;
   int takeoff;

   for (t = 1; t <= ii; t++){
     grospay = payroll[t].grpay;
     tax = payroll[t].tx;
     takeoff = tax - payroll[t].deduction;
     if (takeoff < 1 )
       {
        payroll[t].tx = 0;
        payroll[t].net = grospay;
        }
       else
         {
           payroll[t].net = grospay - takeoff;
           payroll[t].tx = takeoff;
         }

      totaltax = totaltax + payroll[t].tx;
      totalnetpay = totalnetpay + payroll[t].net;
     }
}


grosspay(ii)
int ii;
/* This function will calculate the grosspay by
figuring rate times hours for the first 40 hours, 11/2
times rate for the hours between 40 and up to 50, and
double time for the hours over 50 worked in a single
week. At the end of the function, it will add the
current grosspay to the cumulative total grosspay.
*/
{
```

```
  int t;
  float hrs;
  float rte;
  for (t = 1; t <= ii; t++){
    hrs = payroll[t].hours;
    rte = payroll[t].rate;
    if (hrs <= 40) payroll[t].grpay = hrs * rte;
    else
      if (hrs <= 50)
           payroll[t].grpay = (40 * rte) + (1.5 * rte
           * (hrs - 40));
      else
         payroll[t].grpay = (40 * rte) + (1.5 * rte *
         10) + ( 2 * rte * (hrs - 50));
    totalgrosspay = totalgrosspay + payroll[t].grpay;
    }
}


input(ii)
int ii;
/* This function will request that the last name, id,
hours worked per week and rate of pay be typed in at
the keyboard.
*/
{
  int t;
  for (t = 1; t <= ii; t++){
    printf("Type in the last name: ");
    scanf("%s",payroll[t].name);
    printf("\n");
    printf("Type in the ID: ");
    scanf("%d", &payroll[t].id);
    printf("\n");
    printf("Type in the hours worked: ");
    scanf("%f",&payroll[t].hours);
    printf("\n");
    printf("Type in the rate of pay: ");
    scanf("%f",&payroll[t].rate);
    printf("\n");
    printf("Type in the number of dependents:");
    scanf("%d",&payroll[t].depend);
    printf("\n");
    }
}

tax(ii)
int ii;
/* This function will calculate the tax due on the
basis of a 5% tax on the first hundred dollars of pay,
a 7% tax on the second 100 dollars of pay and a 10% tax on
any pay over 200 dollars.
*/
{
  int gropay;
  int t;
```

```
    for (t = 1; t <= ii; t++){
      gropay = payroll[t].grpay;
      if (gropay <= 100) payroll[t].tx = .05 * gropay;
        else if (gropay <= 200)
           payroll[t].tx = (.05 * 100) +
             (.05 * (gropay - 100));
             else
               payroll[t].tx = (.05 * 100) +
                 (.07 * 100) + (.10 * (gropay - 200));
    }
  }

  printchecks(ii)
  int ii;
  /* This function will print out a header for the
  company and the pertinent information for each employee
  which is needed to complete a weekly paycheck */
  {
      int t;

      printf("\n\n\n\n\n\n\n\n\n\n");
      printf("                        ACME TOOL and DIE COMPANY");
      printf("\n");
      printf("                              1223 Broadway");
      printf("\n");
      printf("                             San Luis Obispo");
      printf("\n");
      printf("                               California");
      printf("\n");
      printf("                             (805) 543-7771");
      printf("\n");
      printf("\n");
    for (t = 1; t <= ii; t++){
      printf("Name:       %s \n",payroll[t].name);
      printf("ID:         %d \n",payroll[t].id);
      printf("Dependents: %d \n",payroll[t].depend);
      printf("Grosspay:   %6.2f \n",payroll[t].grpay);
      printf("Tax:        %6.2f \n",payroll[t].tx);
      printf("Netpay:     %6.2f \n",payroll[t].net);
      printf("\n\n");
    }
  }

printtotals()
  {
    printf("           The Payroll Totals \n\n");
    printf("Total Grosspay:     %.2f\n",totalgrosspay);
    printf("Total Tax:          %.2f\n",totaltax);
    printf("Total Netpay:       %.2f\n",totalnetpay);
  }

main()
  {
    int i, t;
    float gross;
    totalgrosspay = 0;
```

```
    totalnetpay = 0;
    totaltax = 0;
    printf("How many records to input? ");
    scanf("%d",&i);
    printf("\n");
    input(i);
    grosspay(i);
    dependents(i);
    tax(i);
    netpay(i);
    printchecks(i);
    printtotals();
}
```

APPENDIX C

August 21, 1987

Department Head/Chair
Computer Science

Dear Sir/Madam:

I'm conducting a survey to try to roughly determine whether plagiarism involving student programs is considered enough of a problem to have generated a formal departmental policy on the matter.

If there is such a written policy in your department I'd very much appreciate receiving a copy of it.

Thank you,

John B. Connely, Professor
Computer Science Department
Cal Poly State University
San Luis Obispo, CAL 93407

We have such a policy. YES NO

APPENDIX D

June 3, 1987

*** In the following, plagiarism refers to computer program plagiarism.

1. Do you believe that plagiarism does occur? Yes No

2. If so, how serious is the problem?
(please circle your response)

	Not very				very
a. Lower division major classes?	1	2	3	4	5
b. Lower division service classes?	1	2	3	4	5
c. Upper division classes?	1	2	3	4	5
d. Graduate classes?	1	2	3	4	5

3. Do you feel that the departmental policy on computer program plagiarism is helpful and well-defined?

Yes No No opinion

4. Do you feel that the students clearly understand the concept?

Yes No No opinion

5. Do you look for or do you instruct your grader to look for plagiarism when you are grading a program assignment?

Yes No

6. Rate the following common forms of plagiarism.
(Please add any others that seem important to you.)

	Very common	common	not common
a. Change only program name			
b. Change only comments			
c. Change only variable names			
d. Shuffle order of procedures/ functions			

e. Change program logic

f. Insert part of another student's program

g. Other?

7. Which do you feel would do most to reduce the problem of plagiarism?
(Please circle the <u>one</u> you favor)

a. A more detailed definition.

b. Better means of detection.

c. Better communication of the idea of plagiarism to the students.

d. Harsher penalties.

e. Lower percentage of the class grade based on the programming.

8. Would you use a plagiarism detection tool if there were one available?

Yes No

9. Please circle the appropriate level.

I generally teach:

a. Lower division service courses

b. Lower division major courses

c. Upper division courses

d. Graduate courses

APPENDIX E

Classroom Surveys

1. CSc 204, Introduction to C and Unix.
   March 1988.

2. CSc 204, Introduction to C and Unix.
   March 1988.

3. CSc 204, Introduction to C and Unix.
   June 1988.

4. CSc 218, Advanced Pascal/ Introduction to Modula 2.
   December 1987.

5. CSc 218, Advanced Pascal/ Introduction to Modula 2.
   December 1987.

6. CSc 345, Data Structures. Language: Modula 2.
   December 1987.

7. CSc 345, Data Structures. Language: Modula 2.
   July 1988.

Protocol Interviews

1. Ayson, Laurie. In C. July 28, 1988.

2. Beebe, Andy. In Pascal. June 10, 1987.

3. Cron, Chris. In C. July 22, 1988.

4. Crook, Ernie. In Pascal. November 9, 1988.

5. Dalke, Darrell. In C. July 25, 1988.

6. David, Paul. In C. July 26, 1988.

7. Dimmick, John. In C. July 16, 1988.

8. Grandjean, Paul. In C. July 22, 1988.

9. Jones, Bob. In C. July 25, 1988.

10. Kamimoto, Norman. In C. July 22, 1988.

11. Kaut, Debbi. In C. July 27, 1988.

12. Mach, Roger. In C. July 20, 1988

13. Maughmer, Mike. In Pascal. July 26, 1988.

14. Nakamura, Lee. In C. July 20, 1988.

15. Neuman, Phil. In C. July 22, 1988.

16. Otteson, Ingrid. In C. July 27, 1988.

17. Salter, Jim. In C. July 19, 1988.

18. Sartor, Vince. In C. July 19, 1988.

19. Sobel, Andy. In C. July 20, 1988.

20. Stark, Heather. In C. July 28, 1988.

21. Toftler, Elizabeth. In C. July 28, 1988.

Surveys

1. Connely, John B. "Plagiarism". Survey of Oregon State University Computer Science Faculty. May 1987.

2. Connely, John B. "A Plagiarism Policy". Survey of California State University and College Departments of Computer Science. August 1987.

3. Connely, John B. "Plagiarism". Survey of California Polytechnic State University Computer Science Faculty. December 1987.

BIBLIOGRAPHY

Books

1. Best, John W. Research in Education. Englewood Cliffs: Prentice-Hall, 1970.

2. California Polytechnic State University at San Luis Obispo Catalog, 1984-1986.

3. Conte, S.D., H.E. Dunsmore and V.Y. Shen. Software Engineering Metrics and Models. Menlo Park: Benjamin/Cummings, 1986.

4. Fowler, H. Ramsey. The Little, Brown Handbook. Boston: Little, Brown and Company, 1980.

5. Funk & Wagnalls Standard College Dictionary. New York: Funk & Wagnalls, 1977.

6. Gibaldi, Joseph and Walter S. Achtert. MLA HANDBOOK for Writers of Research Papers, Theses, and Dissertations. New York: Modern Language Association, 1980.

7. Halstead, M.H. Elements of Software Science. New York: Elsevier North-Holland, 1977.

8. Perrin, Porter G. Writer's Guide and Index to English. Chicago: Scott, Foresman and Company, 1959.

9. Turabian, Kate L. A Manual for Writers of Term Papers,Theses, and Dissertations. 4th ed. Chicago: University of Chicago Press, 1973.

10. Watt, William W. An American Rhetoric. New York: Rinehart and Company, 1955.

11. Wilson, John. Language and the Pursuit of Truth. Cambridge: Cambridge University Press, 1960.

12. _____. Thinking With Concepts. Cambridge: Cambridge University Press, 1963.

Articles

13. Benson, M. "Machine Assisted Marking of Programming assignments", ACM SIGCSE Bulletin, Vol. 17, No. 3 (September 1985), 24-25.

14. Berghel, H.L. and D.L. Sallach. "Measurements of Program Similarity in Identical Task Environments". SIGPLAN Notices, Vol. 19, No. 8 (August 1984), 65-72.

15. _____. "Computer program plagiarism detection: the limits of the Halstead metric". Journal of Educational Computer Research. Vol. 1, No. 3 (1985), 295-315.

16. Cook, Janet M. "Defining Ethical and Unethical Student Behaviors Using Departmental Regulations and Sanctions." ACM SIGCSE Bulletin, Vol 19, No. 1 (February 1987), 462-468.

17. Cross John A. and James L. Wolfe. "Paperless Submission and Grading of Student Assignments", ACM SIGCSE Bulletin, Vol. 17, No. 1 (March 1985), .

18. Denenberg, Stewart A. "Test Construction and administration Strategies for Large Introductory Courses", ACM SIGCSE Bulletin, Vol. 13, No. 1 (February 1981), 235-243.

19. Donaldson, John L. et al, "A Plagiarism Detection System". ACM SIGCSE Bulletin. Vol. 13, No. 1 (February 1981), 21-25.

20. Ericsson, K.A. and H.A. Simon, "Verbal reports as data". Psychological Review. Vol. 87, No. 3 (1980), 215-251.

21. Faidhi, J.A.W. and S.K. Robinson. "An empirical approach for detecting program similarity and plagiarism within a university programming environment". Computers and Education. Vol. 11, No. 1 (1987), 11-19.

22. Ferguson, Ernest. "Conference Grading of Computer Programs". ACM SIGCSE Bulletin. Vol. 19, No. 1 (February 1987), 361-365.

23. Grier Sam. "A Tool that Detects Plagiarism in Pascal Programs". ACM SIGCSE Bulletin, Vol. 13, No. 1 (February 1981), 15-20.

24. Hamm, R. Wayne, et al. "A Tool for Program Grading: The Jacksonville University Scale." ACM SIGCSE Bulletin, Vol. 15, No. 1 (February 1983), 248-252.

25. Hwang, C. Jinshong and Darryl E. Gibson. "Using an Effective