1984

Techniques for grading programming labs

Kathleen Muller

Rochester Institute of Technology
School of Computer Science and Technology

TECHNIQUES FOR GRADING PROGRAMMING LABS

by
Kathleen Muller

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Computer Systems Management

Approved by: Lawrence A. Coon 12/13/84
Dr. Lawrence Coon

John A. Biles 12/13/84
Professor John A. Biles

James Hammerton
Professor James Hammerton

December, 1984

DEDICATED TO

ALL OF MY FAMILY

TECHNIQUES FOR GRADING PROGRAMMING LABS

I Kathleen Muller Kathleen A. Muller

hereby grant permission to the Wallace Memorial Library, of Rochester Institute of Technology, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

ACKNOWLEDGEMENTS

I would like to thank all the professors at Rochester Institute of Technology who helped me obtain a fine education in the field of computer science. I would especially like to thank Larry Coon, Al Biles, and Jim Hammerton for all their time and effort on this thesis.

I would like to thank my husband Tom for being so supportive, and my children, Anna and Eric for being so understanding while their mother did her work.

ABSTRACT

Techniques for manual and automated grading of programming labs are discussed. Topics investigated include: general grading of programming labs, plagiarism detection, program documentation, program output, and program efficiency.

This investigation led to the development of automated grading tools that report on style and point to possible instances of plagiarism. The techniques utilized will be discussed and their use demonstrated.

TABLE OF CONTENTS

UNIX is a Trademark of Bell Laboratories

1. RESEARCH GENERAL INFORMATION

1.1 RESEARCH BACKGROUND

The intent of this thesis is to determine the automated tools that are available for grading programming labs and detecting plagiarism. In researching the topic five categories emerged on which instructors might concentrate on when grading programs. These categories are:

- General Grading of Programming Labs,
- Plagiarism Detection,
- Program Documentation,
- Program Output,
- Program Efficiency.

Figure 1.1 contains a chart of the tools, both manual and automated, available for use in each of the different categories. Figure 1.2 contains a chart of the information collected for automated tools.

In addition to automated tools, basic information on how to weight these categories was found and is reported on in the sections to follow.

At the end of this thesis is an annotated bibliography as well as a bibliography. The annotated bibliography provides a brief review of the articles or books which are referenced in this thesis and is organized by category, whereas the bibliography is organized alphabetically by author.

Purpose of Tool	Language used for	Article found in	Implementation notes
GENERAL GRADING	any language	[HHRT83]	grading sheet
	Pascal	[Meek83]	program style assessor
	any language	[Morg82]	use of a rubber stamp
	any language	[MiPe80]	grading sheet
	Pascal	[Rees82]	program style assessor
	Pascal	[Rose83]	program style assessor
	Fortran	[RoSo80]	program style assessor
		[RoTo77]	program style assessor
PLAGIARISM			
	Cobol		
	Basic	[DLSo81]	program plag detection
	Pascal	[Grie81]	program plag detection
	Fortran	[Otte77]	program plag detection
	Pascal	[Rees82]	program plag detection
	Fortran	[RoSo80]	program plag detection
PRETTY-PRINTING	Pascal	[Bate81]	prettyprinter
	Pascal	[Bond79]	indentation algorithm
	PL/1	[Clif78]	connector lines
	PL/1	[CoSm79]	statement reformatter
	Lisp/Rlisp	[HeNo79]	prettyprinter
	Pascal	[LeHu77]	prettyprinter
OUTPUT			
	Basic	[Chan78]	match output
	Algol	[FoWi65]	match output
		[Holl60]	match output
	Algol	[Naur64]	match output
EFFICIENCY			
	Pascal	[MaMi76]	execution time
	Snobol	[RiGr75]	execution time
	Pascal	[Site78]	execution time
	C		'prof'

Figure 1.1 CHART OF AVAILABLE TOOLS

VALUES COUNTED	A	B	C	D	E #
1. n1 - # of unique operators	p	s	p	p	s,p
2. n2 - # of unique operands	p	s	p		
3. N1 - total operators	p	s	p		
4. N2 - total operands	p	s	p		
5. N - size of program N1+N2	p				
6. code lines			p		
7. variables declared (and used)			p		
8. total control statements			p		
9. total lines		s	c		p
10. average line length					s
11. code comment lines			c		
12. use of comments					s
13. use of indentation					s
14. total of non-comment characters					p
15. use of blank lines as separators					s
16. multiple statement lines			c		
17. constants and types			c		
18. number of reserved words					s,p
19. variables declared (not used)			c		
20. length of identifier					s
21. number of procedure/functions		s	c	p	s,p
22 total calls to subroutine				p	
23. total input statements				p	
24. var parameters			c		
25. value parameters			c		
26. # and kind of data structure		s			
27. procedure var (includes 21,24)			c		
28. total conditional statements				p	
29. # and kind of control structure		s			
30. for statements			c		
31. repeat statements			c		
32. while statements			c		
33. goto statements			c		s
34. assignment statements		s		p	
35. loop statements				p	
36. indenting function			c		
37. % of embedded spaces					s,p
38. vocabulary of the program		s			
39. volume of the program		s			
40. level of the program		s			
41. intelligence content		s			
42. effort of the program		s			

s - information used in style program
p - information used in plagiarism program
c - information counted but not used in plagiarism program
A-[Otte77] ; B-[RoSo80] ; C-[Grie81] ; D-[DLSo81] ; E-[Rees82]

Figure 1.2 TABLE OF INFORMATION COLLECTED BY TOOLS

2. GENERAL GRADING OF PROGRAMMING LABS

2.1 INTRODUCTION AND BACKGROUND

Instructors confronted with large numbers of programs to grade tend to defend themselves in several ways: they may employ a cadre of graders or teaching assistants, they may decrease the number of programming assignments, or they may be forced to grade so hastily that they seize one or two simplistic criteria often unrelated to their course objectives. Unfortunately, this results in evaluation inconsistencies, a loss of student confidence in grading fairness, and a diminished level of student competence in programming [HHRT83].

It becomes important for the sake of both students and instructors that efficient, objective criteria for grading programs be developed. These criteria should accurately measure a student's achievements and avoid errors in evaluation [Morg82]. By developing some kind of standard grading technique, the student knew precisely what was expected [MiPe80].

2.2 PREVIOUS WORK

In researching the previous work on grading of programming labs two approaches were found - manual grading systems and automatic grading systems.

Manual grading systems all basically took the same approach. In each case evaluation criteria for a programming assignment was defined, and then a rating scheme to be used to grade the program was developed.

Knowing the uncertainty of marking by manual methods, it was thought that automatic assessment of style using simple algorithms could produce results just as valid and with improved consistency. At the same time automatic assessment would completely eliminating time-consuming manual inspection of program listings [Rees82].

A discussion of both manual and automatic approaches follows. (See Sections 2.2.1 and 2.2.2).

2.2.1 MANUAL APPROACHES

This section includes five different manual approaches to the grading of programming labs.

2.2.1.1 G. WEINBERG AND E. SCHULMAN

Weinberg and Schulman [MiPe80] graded programs by ranking the students according to the following criteria:

- number of program statements,
- number of hours in completing the assignment,
- output clarity,
- program clarity.

2.2.1.2 D. CLUTTERHAM

Clutterham [MiPe80] used the following criteria for grading a program, assigning points for each criterion:

- correct answers,
- program efficiency in terms of length
  (# of statements in instructor's program divided by
  # of statements in student's program multiplied by
  total points for the criterion),
- correct termination of program.

2.2.1.3 N. MILLER AND C. PETERSON

Miller and Peterson [MiPe80] used forms attached to each program with the evaluation criteria listed, along with the weight given for each criterion. They felt that the weighting factors helped make the grading more objective. They also graded so that students received only 80% of the grade if they met the minimum requirements. The other 20%

was for students who did more than what was required.

Four sample forms were presented by the authors. One was the original form the authors used, the other three were other instructor's adaptations of the original form. The original and one of the adaptations follows:

ORIGINAL APPROACH

Algorithm (10%)
- Structure chart showing calling hierarchy (5%)
- Detailed algorithm expression for each module (5%)

Program style and clarity (25%)
- Internal documentation (10%)
- Meaningful identifiers (5%)
- Formatted listing (10%)

Output (45%)
- Correct for specific input (35%)
- Easy to read (5%)
- Graceful termination (5%)

Refinements above minimum (20%)
- Algorithm clarity, efficiency, and/or elegance (5%)
- "Elegant" implementations (10%)
- Output embellishments (2%)
- Exemplary program design and implementation (3%)

AN ADAPTATED APPROACH

Top down design (40%)
- Detailed problem definition (20%)
- Refinement of the problem using a level by level approach (20%)

Program style and clarity (20%)
- Description of all data structures (5%)
- Meaningful identifiers (5%)
- Proper indentation (5%)
- Modular design (5%)

Output (20%)
- Correctness (15%)
- Well organized and readable (5%)

Refinements - Superior work (20%)
- Program length (5%)
- Output embellishments (5%)
- Exemplary program style and clarity (10%)

2.2.1.4 G. MORGAN

Morgan [Morg82] used the same approach of listing the criteria to evaluate, and then rating each criterion. Morgan [Morg82] used a rubber stamp applied to the front of each program to grade each program, rather than an attached form. A sample format for the rubber stamp as it might be filled out follows:

Timely	2	3	4	(5)
Problem definition	2	3	4	(5)
I/O design	2	3	(4)	5
Logic design	2	(3)	4	5
Source program	2	3	(4)	5
Test validity	2	3	(4)	5

This student would receive an 83% for the lab, since there were 25 points awarded out of a possible 30.

2.2.1.5 R. HAMM, K. HENDERSON, M. REPSHERT and K. TIMMER

Hamm, Henderson, Repshert and Timmer [HHRT83] borrowed an approach used in grading English Compositions called the "Diederich Scale". They felt that there was a similarity between writing a computer program and writing an English paper. Thus using a similar approach in the grading of each was appropriate.

The following concepts tied English compositions to computer programs:

- both are the solution to a communication problem
  - the composition - communicates with other persons
  - the program - communicates with a computer
- both start with an outline or flowchart
- both implement the outline or flowchart
- both have qualities of style and individuality
- both create a heavy paper-load on the instructor
- both students expect a consistent grading between instructors.

The proposed system had a weighting scheme similar to that of Miller and Peterson [MiPe80]. A list of criteria, with a sample weight scale for an English composition follows:

	p-poor; a-adequate; g-good				
	p		a		g
ideas	2	4	6	8	10
organization	2	4	6	8	10
flavor	1	2	3	4	5
wording	1	2	3	4	5
usage	1	2	3	4	5
spelling	1	2	3	4	5
punctuation	1	2	3	4	5

A list of criteria, with a sample weight scale for a computer program follows:

	p-poor; a-adequate; g-good					
	p			a		g
execution of the program,	0	7	13	20		
correctness of the output,	0	7	13	20		
design of the output,	0	4	8	12	16	20
design of the logic,	0	4	8	12	16	20
design of test data,	0	4	8	12	16	20
internal documentation,	0	4	8	12	16	20
external documentation,	0	4	8	12	16	20

A program was written to generate a specific form for each assignment, so that changes to the list of criteria graded and the weight assigned to each could be easily made. The form would contain some identifying information and the criterion and weight assigned to each criterion for each assignment (similar to the previous example). The forms for the appropriate assignment would then be filled out by the instructor and attached to each student's program.

2.2.2 AUTOMATED APPROACHES

This section includes three different approaches to the automatic grading of programming labs.

2.2.2.1 S. ROBINSON and S. TORSUN

Robinson and Torsun [RoTo77] used an approach whereby the set of submitted solutions were automatically assessed relative to a solution produced by the instructor. They used a program which classified each source statement by its relative importance to the execution of the whole program, then a report was produced that listed the following for each program statement:

- an estimate of execution time (a),
- a count of the frequency of execution (b),
- an estimate of total execution time (a*b).

From these three values the importance factor for each statement was calculated. The importance factor was the relative contribution of a statement to the overall execution time of the program, expressed as ten times the percentage of total execution run time. Thus an importance factor could range from 0 to 1000, with a larger value indicating a higher cost statement. The importance factor was then used to produce a graph. The x coordinate being the importance factor and the y coordinate being the statements rank in order of importance. The student's graph was then compared to the instructor's graph.

Robinson and Torsun showed that as programming style was improved the graph would mold to the instructor's solution.

A major problem with this method, as with the other automated methods, was that if output was not also looked at, the program could fit into the correct measurements, yet not solve the problem it was designed for. Also this system will not mark or take into consideration original solutions or readability [RoTo77].

2.2.2.2 S. ROBINSON (ITPAD)

The approach of Robinson [RoSo80] was to use modified code optimization techniques and software science measures to analyze FORTRAN source programs. Each student's program went through three phases of analyze. With the information that the system collected, the following functions were performed:

- each student's program was examined visually for certain design requirements,
- the progress of a student through a quarter was evaluated,
- the programming assignments were evaluated to see if the student was using the desired concepts,
- possible plagiarism was looked for,
- suggestions were given to the students for program improvements.

The first phase was the lexical analysis phase. In this phase fourteen program characteristics (listed in Figure 1.2) were tracked. The information gathered in this phase was used to create two profiles, the student's profile and the assignment's profile.

The student's profile contained information about the control structures, retreating edges, and data structures that a student employed throughout the quarter. A retreating edge is the edge of a program graph which represents a return to the beginning of a loop. The student's profile aided in determining whether or not a student had mastered a particular topic. An instructor's model program was used for comparing programs. Following is a sample of the information contained in a student's profile for four programming assignments.

STUDENT PROFILE - OUTPUT FROM S. ROBINSON

ASSIGNMENT NUMBER	1	2	3	4
Control Structures:				
if-then		12	3	
if-then-else			1	
else-if				
Logical if				
with goto	4			
without goto				
while		2		
for				
indexed do			1	3
goto	5			
Data Structures				
real	3			5
integer	5	6	6	5
Basic Blocks	8	23	12	6
Retreating				
edges	7-2	22-2	11-2	3-3 4-2 5-1

The assignment profile contained the software science measures: the control structures used, the retreating edges, the number of basic blocks and the data structures used by the students for each assignment [RoSo80]. The assignment profile gave insight into how effective a programming assignment was at displaying a student's understanding of a particular concept. It did so by revealing the general concepts used by the students to solve the problem. Did the student use the new material in the assignment or did they use older material that they felt more comfortable with?

Three sample assignment profiles from Robinson's approach follows. This information contains the different ranges of values (low, model, high) for each criterion counted per assignment.

ASSIGNMENT PROFILE - OUTPUT FROM S. ROBINSON

****

Assignment Profile of Program 1

****

	low	model	high
- unique variables	3	4	13
- total variables	14	18	46
- unique operators	4	5	8
- total operators	10	13	34
- assignments	1	5	11
- length	24	31	76
- vocabulary	9	9	19
- volume	79	98	323
- level	.30	1.0	1.3
- intelligence content	6.1	8.7	30.5
- effort	59	98	1061
- leaders	2	3	16

****

Assignment Profile of Program 2

****

	low	model	high
- unique variables	6	7	17
- total variables	32	46	95
- unique operators	4	5	7
- total operators	13	25	53
- assignments	4	6	26
- length	45	71	132
- vocabulary	10	12	23
- volume	150	254	565
- level	.48	1.0	1.12
- intelligence content	10.9	15.5	99.0
- effort	87	254	1045
- leaders	4	17	36

****

Assignment Profile of Program 3

****

	low	model	high
- unique variables	7	9	19
- total variables	19	21	34
- unique operators	4	6	10
- total operators	10	13	80
- assignments	6	7	39
- length	31	47	189
- vocabulary	12	15	22
- volume	111	184	756
- level	.24	1.0	1.62
- intelligence content	12.4	16.2	44.0
- effort	85	184	1685
- leaders	5	7	26

The second phase was the analysis of program structure. This phase obtained characteristics by:

- dividing the program into basic blocks,
- constructing a flow graph from basic blocks, the flow graph was constructed by examining all statements that could cause transfer to other basic blocks,
- constructing a directed acyclic graph (DAG) for each basic block, the DAG presents a picture of how the the value computed by each statement in a basic block were used by subsequent statements in the block,
- performing data flow analysis on the flow graph,
- detecting loops in the flow graph.

The third phase was the analysis of program characteristics. This phase analyzed the characteristics detected in the second phase to determine if the student should receive a message containing advice on how they might improve their program. The messages were selected from common programming errors and could be specialized by the instructor for an individual assignment [RoSo80].

This implementation concentrated on evaluating a programming assignment, but unlike the other approaches, it assigned no grade to a student's program.

2.2.2.3 M. REES (STYLE)

Michael Rees' [Rees82] approach to grading an assignment's style was called STYLE. STYLE was designed to accept as input the source of a syntactically correct program, make measures on individual criterion in one pass, on a line by line basis, and yield a style mark out of 100%.

The final mark was influenced by a weighting table supplied by the instructor.

The data collected on each assignment along with, in parenthesis, whether the value should be a high or low number and notes on changes made by Rees to his original implementation follows:

Layout

- line length - the average number of "significant" characters per line (LOW),
- comments - percentage of all program lines comprised wholly or partially of comments (HIGH),
- indentation - percentage of lines indented in any way (changed to calculate changes of indentation on a line by line basis) (HIGH),
- blank lines - percentage of blank lines in a program, (changed to blank lines were subtracted from total line count before other measures were calculated) (HIGH),
- embedded spaces - additional spaces embedded within a line (HIGH).

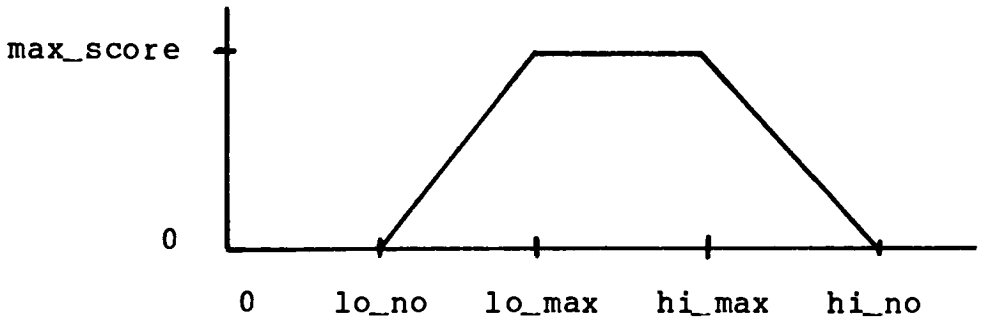
Identifiers

- program decomposition - number of procedures and functions (HIGH) By dividing this figure into the total number of lines, a measure of module length was obtained (LOW),
- variety of reserved words - count of the number of different reserved words used (HIGH),
- length of identifiers - average length of all the programmer-defined identifiers (HIGH),
- variety of identifiers - number of different programmer-defined identifiers, (changed to number of different identifiers as a function of program length) (MID),
- labels and gotos - count the number of occurrences of the reserved words "label" and "goto" (zero).

The grade for each value was obtained using the following parameters:

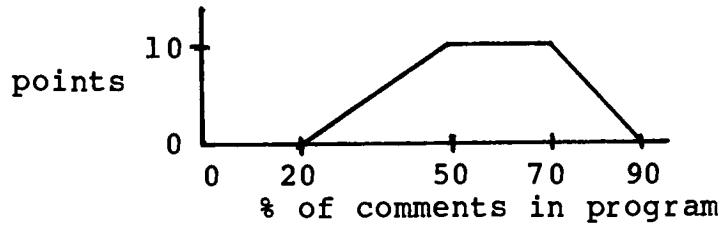
- max_score - the maximum percentage mark allowed for the criterion,
- lo_max, hi_max - the low and high value range of the criterion which will yield the maximum grade for that criterion,
- lo_no to lo_max - the interval of the criterion which will yield a grade from zero to the max_score on a linear basis,
- hi_max to hi_no - the interval of the criterion which will yield a grade from the max_score to zero on a linear basis,
- lo_no, hi_no - any criterion below lo_no and above hi_no yields a zero mark.

A visual representation of how these values work follows:



An illustration of the grading of a criterion follows. If an instructor wished to grade commenting as 10% of the grade and was looking for between 50% to 70% commenting for a perfect grade, and for anything less than 20% or greater than 90% as being a zero grade, then the system parameters might be set-up as follows:

- max_score = 10
- low_no = 20
- low_max = 50
- hi_max = 70
- hi_no = 90



This would result in assigning 0 points for less than 20% or greater than 90% comments; 10 points for between 50% to 70% comments; and a linear grade between 0 and 10 points for between 20% to 50% or 70% to 90% comments.

The sum of each criterion's weighted grade yielded the style grade. A sample setting for the parameters (max_score, low_no, low_max, hi_max, hi_no) and output from two sample programs follows on the next page.

Another observation made by Rees was that programs which used some form of a prettyprinting before being graded for style, changed the style mark by less than 5%.

OUTPUT FROM REES

****
SAMPLE OF PARAMETER SETTINGS
****

Measure	max_score	low_no	low_max	high_max	high_no
chars/line	15	12	15	25	30
% comments	10	15	20	25	35
% indentation	12	60	70	80	90
% blank lines	5	8	10	15	20
% spaces	8	8	12	18	20
proc/fnc length	20	10	20	35	50
# reserved words	10	22	26	40	41
id. length	20	7	9	15	16
# identifiers	0	0	0	0	0
label and gotos	-20	1	3	199	200

****
OUTPUT OF THE PARAMETERS FOR TWO COURSES
****

	Program 1 Ave. 350 lines Pascal			Program 2 Ave. 750 lines Pascal		
Measure	low	mean	max	low	mean	max
chars/line	14	20	34	9	13	18
% comments	3	21	31	0	16	35
% indentation	0	74	98	39	72	94
% blank lines	0	5	27	0	17	33
% spaces	2	7	55	3	11	20
proc/fnc length	15	32	174	17	37	77
# reserved words	10	23	26	17	23	29
id. length	5	8	10	7	10	15
# identifiers	24	46	87	13	41	97
labels and gotos	0	0	3	0	0	3
Marks	35	60	84	44	64	95

[Rees82]

2.3 SUMMARY

Both the automated and manual tools offered the same benefit to the student and instructor - they provided a consistent grading method. The automated approaches also offered the benefits of being efficient for the instructor and objective for the student. Style is not the only aspect of a program that should be looked at by the instructor, but the tools reported on could aid at making the evaluation of this category efficient and objective.

In partial fulfillment of this thesis an automatic style grader was developed with a similar grading approach as Rees [Rees82]. See Section 7 for further details.

3. PLAGIARISM

3.1 INTRODUCTION AND BACKGROUND

The problem of the possibility of plagiarized programs compounds the already difficult responsibility of evaluating students' programs.

> The acquisition of skills in computer programming can be, and often was, a challenging and rewarding experience. Unfortunately, the need to teach larger classes consisting of a wider variety of students had introduced many problems. Outstanding among these was the tendency of students to resort to unorthodox means in fulfilling course requirements. In other words, students cheat [Mill81].

There are a variety of reasons and pressures which cause students to cheat on programming assignments: some students plagiarize because they can not do the work themselves, some students plagiarize to prove they can pull a fast one on the instructor and get away with it, some students desire to get something for nothing, other students only cheat on assignments that they feel were busy work [Mill81]. But the biggest reason of all was that the monetary and social rewards were very attractive, or at least perceived as such, in this field [HwGi82].

Students should be given a sense of values regarding their chosen field. Employers who hire Computer Science graduates should be able to trust a student's knowledge and ability in the subject [Mill81]. Thus a responsibility to prevent or detect plagiarism falls on the instructor.

When cheating occurred in courses it:

- failed to establish a standard of professional integrity,
- reduced the ability to make accurate assessments of student's skills,
- demoralized honest students who feel (often with reason) that they were in competition with the cheaters,
- wasted the energy of both faculty and students,
- encouraged the cheaters to believe that cheating pays and that good grades were a substitute for understanding [Shaw80].

Students can plagiarize programming assignments in a variety of ways:

- copying a program and changing only the author's name,
- copying a program and changing the documentation,
- copying a program and changing the variable names,
- transposing statements when the ordering of the statements does not effect the results,
- breaking up single statements such as declarations and output statements,
- stealing programs written by other students,
- copying a program and changing the logic a little,
- copying a program and changing the logic a lot,
- copying a program given in an earlier class,
- having someone else write all or part of the program,
- copying a program by changing only the line numbers (Basic and Fortran),

[HwGi82],[DLSp81],[Mill81].

A discussion of ways in which students plagiarize and some methods used in dealing with plagiarism follows. Preventive approaches are discussed in Section 3.2.1 and detection approaches are discussed in Section 3.2.2.

3.2 PREVIOUS WORK

Detection of plagiarized programs is a complicated issue. Both Ottenstein [Otte77] and Donaldson, Lancaster and Sposato [DLSp81] realized that using a grader alone was inadequate for detecting plagiarized programs. In the area of plagiarism prevention there were a variety of approaches, the next section will discuss some of them.

3.2.1 PREVENTIVE APPROACHES

Hwang and Gibson [HwGi82] summarized five different approaches to dealing with plagiarism and their success with these approaches. Included in the following discussion are references to the other author's researched that strengthened their position.

1. Set up a punishment policy to discourage students from cheating. Hwang and Gibson [HwGi82] felt that this method was ineffective, since it was essentially negative. A totally negative attitude was not the complete solution to the problem, but it was part of the solution. Miller [Mill81] stated that the consequences of plagiarizing should be reasonable yet severe enough to point out that it will not be tolerated. Whatever penalties were declared, offenders must be dealt with fairly and firmly. The student should be aware of what the consequences of plagiarism will be.

A list of possible disciplinary actions is given below:

- actions within the course,
  - sharing the grade among guilty students [Mill81],
  - negative credit for the assignment,
  - no credit for the assignment and loss of a letter grade for the course,
  - makeup assignment over the same material, no credit,
  - forced drop in the course,
  - failure in the course,
- actions within the Computer Science Department,
  - suspension from Departmental courses for a designated period,
  - expulsion from Departmental courses,
- actions by the University,
  - warning,
  - probation,
  - suspension from the University for a designated period,
  - expulsion from the University [Shaw80].

2. Set up a software plagiarism detection system. Hwang and Gibson [HwGi82] questioned if this approach would catch every type of cheating, they felt it might be rather expensive. This approach is covered in Section 3.2.2.

3. Raise the consciousness of the students to understand and appreciate what they must know in order to obtain a degree. This was a positive approach, but Hwang and Gibson [HwGi82] realized that students were too interested in passing the course for it to be an effective one.

4. Inform the students that they may be called into the office at any time to verify what they "claim" to have learned on a programming assignment. Hwang and Gibson [HwGi82] felt this method was a cynical approach which bred mistrust and was not too effective. It was also apt to invite confrontations between students and instructors.

5. Assign grades according to the ratio of programming assignments and exams (including routine quizzes). This was the method supported by Hwang and Gibson. Six different ratio methods were discussed with the advantages and disadvantages of each in the article [HwGi82]. The methods along with Hwang's and Gibson's labels are listed below:

A - exams weighted proportionately heavier than programming assignments,
B - programming assignments weighted proportionately heavier than exams,
C - exams and programming assignments weighted approximately equally,
D - final exam used as evidence of what the student had learned - Fail the final - Fail the course
E - programming assignment related quiz associated with each programming assignment,
X - percentage on programming assignment-related quiz applied to the score on the programming assignment,
Example: 100 points total
80 points on program assignment
90% points for programming assignment related quiz
80 * 90 = 72 total points on the project
Y - score obtained on the programming assignment-related quiz added to the score obtained on the programming assignment,
Example: 100 points total (50/50)
40 points for quiz
45 points for programming assignment
40 + 45 = 85 total points on the project

Hwang and Gibson [HwGi82] discarded methods B, C and E as being too lenient on those who cheat; methods D and X were better but could penalize the honest student if they happen to have a bad day. Thus methods A and Y were the better choices with Y being the best because of its fewer listed disadvantages.

The advantages of method Y were as follows:

- encourages students to do the programming assignments in order to do well on the programming assignment quiz,
- the total grade actually represented the student's understanding of the programming assignment,
- the grade was proportional to the time and effort expended,
- the method represented the students' grade very well for all unexpected situations.

The disadvantages of Method Y were as follows:

- if the student had a bad day, the grade on the quiz would not represent their true ability,
- if the programming assignment quiz did not represent the programming assignment well the grade would not represent the student's ability.

Shaw [Shaw80] outlined a series of actions which an instructor could use when accusing a student of cheating:

- make copies of the evidence (Ex. program) for the student and the Department, retaining the original,
- in the presence of a witness confront the student with the allegation,
- if after the confrontation the instructor decided to impose a penalty, the instructor should so inform the student by letter, the letter should state the basis for the action, the assigned penalty and student's right to appeal to the University Committee on Discipline within one calendar week.

Throughout the entire process, it was essential that all meetings, decisions, and actions be documented in writing.

Shaw also outlined actions the computer science department, the computer center and the faculty could follow for the prevention and detention of plagiarism. These are listed below:

- possible department actions,
  - develop an on-line system to detect programs that were similar, (see Section 3.2.2)
  - provide an adequate number of available, knowledgeable consultants to advise students in the lower level courses,
  - establish facilities for in-class examination of the student's programming,
  - maintain records on cheating incidents in department courses,
  - spread the word that the department does not condone cheating,
- possible computer center actions,
  - upgrade on-line assistance including help facilities, debuggers, and on-line explanations of routinely encountered errors,
  - routinely provide information on computer usage including the amount of time each student is connected to various systems,
  - provide closed trash cans for the disposal of program listings,
- possible instructor actions,
  - provide students with a hand-out stating the cheating policy and disciplinary action,
  - base judgement on the student's mastery of course material on work done in monitored situations to the extent educational objectives permit,
  - provide guidelines for user consultants, indicating what kinds of help they should and should not give to students in each course,
  - use any automatic detection procedures that become available. (see Section 3.2.2) [Shaw80]

3.2.2 DETECTION APPROACHES

In the plagiarism detection process instructors have designed systems to detect similarities in student programs. The next sections will present five different views on the detection of plagiarism.

3.2.2.1 K. OTTENSTEIN

The earliest article available on this subject was Ottenstein [Otte77]. His approach was conservative but effective. His method was to count Halstead's [Hals77] software science criteria:

- n1 - the number of unique operators,
- n2 - the number of unique operands,
- N1 - the total number of occurrences of operators,
- N2 - the total number of occurrences of operands.

for each student's program. Operators consisted of control structures as well as the normal program operators. Reserved words other than control structures were not counted. Each occurrence of an operator or operand was called a token. He also calculated the size of the program (in tokens), N, which was N1 + N2. These five values were assigned to each program and were the basis for comparison between programs. Ottenstein's reporting of this information was very simple. The values n1, n2, N1, N2, and N for each students' information was reported on a line. These lines were sorted by program size (N). Thus an instructor would look for those programs with equal N values and then

look back at the other values to determine if there was a need to manually review the similar programs.

This approach was successful in detecting programs changed by:

- reordering time independent statements,
- recommenting,
- reformatting of the text,
- renaming the variables and labels.

It would not detect a student who cheated on only part of a program. Donaldson, Lancester and Sposato [DLSp81] questioned how effective this method was for introductory courses where there may be only slight variation in the final results.

3.2.2.2 S. ROBINSON (ITPAD)

Robinson's and [RoSo80] approach for the collection and reporting of student program information was expanded to detect plagiarism. (see Section 2.2.2.2 for a discussion of the basic implementation.) The method for detecting possible collaborators followed this procedure:

- group the program by the number of leaders, (leaders were a type of statement),
- compare the number of statements in each basic block, then eliminate the programs which match less than 50% of the time,
- compare the control structures and retreating edges, then eliminate the programs that have different values,
- compare the data structures, and eliminate programs with a difference of more than one for each data type.

Since this approach has more detail and was less restrictive in selecting similar programs than Ottenstein [Otte77], it matched more students. The question is whether the extra information was worth the extra time and resources required. The Robinson results did not show much justification for the extra detail. In fact after visual inspections most of the extra programs which they selected appeared not to have been plagiarized.

3.2.2.3 S. GRIER (ACCUSE)

Grier's [Grie81] approach to plagiarism detection is an extension of Ottenstein's [Otte77]. Grier's program, ACCUSE, calculated the four Halstead [Hals77] software science criteria plus 16 others (see Figure 1.2). Through testing different combinations of the 20 elements, seven were retained to determine a correlation number.

An interesting calculation was used by Grier [Grie81] to determine the correlation number between two programs. The correlation scheme involved computing an increment for each pair of affected programs based on the equation:

increment = "importance factor" -
(pcounta - pcountb)

where pcounta and pcountb represent criterion counts for the two programs compared. If the (pcounta - pcountb) was less than or equal to some "window size", depending on the particular criterion then the increment was calculated.

The importance factor was the weight for each criterion which affected the increment value. Each of the seven increments was totaled to form a correlation number.

The following is a list of the seven increments discussed, listed also are the increments window size and importance factor and notes on how they were calculated:

- Unique operators - (Begin and End ignored)
  window size 5
  importance factor 6

- Unique operands - (for each assignment operator two operands were subtracted)
  window size 5
  importance factor 6

- Total operators - (does not include assignment operators, Begin and End ignored)
  window size 3
  importance factor 5

- Total operands
  window size 3
  importance factor 5

- Code lines - (decremented for each assignment operator, ignore blank lines, comments, and declarations, count only executable lines of code)
  window size 3
  importance factor 5

- Variables declared (and used)
  window size 2
  importance factor 3

- Total control statements
  window size 1
  importance factor 2 [Grie81]

Grier also produced the following five reports:

- report of each student's program's 20 criteria as listed in Figure 1.2 measured by ACCUSE,
- report of each student's program's 7 criteria as listed in Figure 1.2 used to compute the correlation number,
- a triangular matrix whose entry in the matrix is the correlation number between each program pair,
- frequency distribution graph that indicates the number of pairs of programs with the same correlation numbers,
- a list of all pairs of programs which have a correlation number greater than or equal to 28 (32 was maximum correlation number).

These were then used to determine which programs might have been plagiarized. A manual inspection of each suspected program was still necessary. This approach was successful in detecting programs changed by the following means:

- reordering of time independent statements,
- recommenting,
- reformatting of text,
- renaming variables and labels,
- adding unnecessary initialization and assignment statements,
- adding excess declarations.

ACCUSE was designed to be as inexpensive to use as possible. Thus the idea of utilizing a front end of a compiler was replaced with Ottenstein's [Otte77] approach of a fast counter. The result was a compromise between speed and comprehensive analysis [Grie81].

3.2.2.4 J. DONALDSON, A. LANCASTER, and P. SPASATO

Donaldson, Lancaster and Sposato [DLSp81] approach to plagiarism included two data collection phases: one to gather information on the structure of the program, and the other to gather information on the content of the assignment. There were also two data analysis phases, one to evaluate each type of information collected.

The first data collection phase (for FORTRAN assignments) kept track of the following criteria:

- total number of variables,
- total number of subprograms,
- total number of input statements,
- total number of conditional statements,
- total number of loop statements,
- total number of assignment statements,
- total number of calls to subprograms,
- total number of statements of type 2-7.

The second data collection phase characterized the assignment by the order in which statements occurred. Each type of statement was given a character code (Ex. X for logical if). As the assignment was processed a string of character codes was produced.

The first data analysis phase performed the following three types of calculations on the information gathered in the first data collection phase:

1. Sum of the difference - corresponding criterion values were subtracted and the absolute values of the difference were summed. This gave some indication of how two assignments differed in content.

2. Count of similarity - each similarity factor starts at zero and was incremented by one for each corresponding criterion value which was equal. This showed how many criterion values were equal but not which ones.

3. Weighted count of similarity - this method was an extension of number 2 above. Instead of incrementing by one, the increment was by the weight given the criterion values. This allowed the instructor to weight the criterion value according to what was expected of the particular assignment.

The second data analysis phase worked with the string of character codes from the second data collection phase. It compressed identical characters in succession. The resulting strings were compared. If all the characters of the string matched that of another student, it meant the two assignments had the same order of statements [DLSp81].

This approach was successful in detecting programs changed by the following means:

- transposing statements when the ordering of the statement does not effect the results,
- altering format statments,
- breaking up single statements such as declarations and output statments,
- renaming variables and labels,
- recommenting.

3.2.2.5 M. REES (CHEAT)

The method for detecting plagiarism started with Rees' STYLE program (see Section 2.2.2.3). Robson [Rees82] added to STYLE and created a post processor called CHEAT which looked for similar programs. After some experimenting the following criteria were selected for comparison:

- total of non-comment characters,
- % of embedded spaces,
- number of reserved words,
- number of identifiers,
- total number of lines,
- number of procedure/functions.

This approach was similar to the others in that the criteria for each student was compared. Student programs with similar values were then verified for possible plagiarism.

3.3 SUMMARY

The tools and techniques for the detection of plagiarism can only point to possible plagiarized programs. It is still necessary to manually inspect the suspected programs to confirm plagiarism. The tools should report broad enough information on possible plagiarism so that changes in plagiarism approaches will be flagged. If a plagiarism tool is designed too restrictively it may create a false sense of security for the instructor.

The benefit of having both a plagiarism policy and detection mechanism was to create a cheating deterrent.

In partial fulfillment of this thesis a tool was developed that uses a counting approach similar to Donaldson, Lancaster and Sposato [DLSp81]. See Section 7 for further details.

4. PROGRAM DOCUMENTATION

4.1 INTRODUCTION AND BACKGROUND

A program that is easy to read and understand is easier to test, maintain and modify [Clif78]. Failure to adequately document software leads to: higher production and maintenance costs, customer dissatisfaction, and the development of useless programs [Fole83]. These statements sum up the importance of a well documented program. Readability is improved through the proper use of identifier variable names, comments, modularity and formatting. The understanding of a program is improved by having good supporting documentation as well as a well documented program.

Section 4.2 supplies more information on formatting standards and automated tools to enforce the standards. Sections 4.3 and 4.4 contain supporting documentation standards.

4.2 PREVIOUS WORK

When program formating standards are developed it should be kept in mind that they should be unambiguous, flexible enough to provide for a programmer's individuality and dynamic [LHSi77]. Once the standards are set the computer can do the formatting of the program, using the standards as a guideline. This automated formatting of a program is called prettyprinting.

There were many formatting standards, each of which could have their own prettyprinter program to enforce their standards. There have been many articles written on different indenting techniques and approaches to handling the different formatting standards. The articles reviewed presented each author's approach to a formatting standard or a criticism of another author's approach.

Bates [Bate78], Bond [Bond79], Conrow and Smith [CoSm79], Grogono [Grog79], Ledgard, Hueras and Singer [LHSi77], and Peterson [Pete77] all seem to present a traditional formatting approach. (see Section 4.3 - General Prettyprinting Rules, Alignment Rules and Indentation Rules). To go along with these traditional standards some authors presented prettyprinter programs to enforce the standard. Some authors were Bates [Bate81], Bond [Bond79], Conrow and Smith [CoSm79], Hearn and Norman [HeNo79], Ledgard and Hueras [LeHu77].

There were also the less orthodox formatting standards. Bates [Bate81] and Ramsdall [Rams79] placed semicolons at the beginning of the line to represent the logic structure of the program. Clifton [Clif78] placed incremented numbers at the beginning of the line to show the logic level of a program. Crider [Crid78] placed the BEGIN and END words in Pascal at the end of the line.

The following authors point out other benefits of a standard formatter program beside the improved readability of a program. Bates [Bate81] and Ledgard and Hueras [LeHu77] developed a prettyprinter to save time at the editing function. Conrow and Smith [CoSm79] developed a statement reformatter that was able to run prior to and faster than a complier. This reformatter was then used to show if the program coding logic matched the intended logic of the student. Errors were reported by showing unexpected indentation patterns.

Hearn and Norman [HeNo79] saw several benefits in the use of a prettyprinting program: it could create standard program formatting for a group project, it could also set a good example for a new programmer and it allowed for storing programs in compacted form.

When developing a prettyprinter program several points should be kept in mind.

A good formatter should always do the following:

- be consistent in generating output,
- allow easy specification of input,
- never lose text because it cannot properly format,
- work at a "reasonable" speed [Marc81],
- handle the entire language not just the execution control structures [Bate81].

The following approaches could be used when designing a prettyprinter:

- rebuild the source file from compacted files,
- use keywords such as begin and end to trigger indentation or line splitting,
- use the prior method plus some measure of program complexity to guide in its indentation,
- make a prepass of the program to aid in determining line breaks,
- build a data structure to represent the entire printed form of the program and then pass over doing the actual printing,
- use two routines - one to set up the printing process and the second to do the formatting [HeNo79].

These approaches could possibly be implemented in the following ways:

- linked list of tokens which have been considered but not yet printed [Bate81],
- a stack to determine indentation and de-indentation,
- a table used to see if any special prettyprinter action was associated with a symbol [LeHu77].

Beside program format one must also be concerned with the total documentation of the programming project. Ledgard, Hueras and Singer [LHSi77] and Marca [Marc81] outlined other documentation topics which needed to have formal standards developed. Foley [Fole83] outlined the contents for supporting documentation. Each of these articles is a good source for developing or updating a total documentation standard.

Section 4.3 contains a sample of a program standard. The program standard was developed by combining the information from Ledgard, Hueras, and Sidger [LHSi77], and Marca [Marc81]. Section 4.4 contains a sample of a program documentation standard. The documentation standard was done by Foley [Fole83].

4.3 SAMPLE PASCAL PROGRAM STANDARDS

- GENERAL RULES
    - each program must include the following:
        - title, author, program purpose, input files, output files,
        - an overview of the program and any procedures and functions,
        - a summary of large sections of code,
        - notes on complicated lines of code,
        - visual aids to assist in the general reading of the program,
        - meaningful identifier names.
- DECLARATIONS
    - all messages printed by a program must be specified in a constant declaration,
    - all scalars that remain constant throughout a program must be specified in a constant declaration,
    - no function may alter any of its actual parameters or change the value of any global variable,
    - separate and indent key work from data declarations,
    - list each identifier on its own line,
    - align all attributes,
    - comment declarations,
    - separate different lists with white space,
    - alphabetize each data list.
- CONTROL STRUCTURES
    - goto's are not allowed,
    - nesting of any combination of if, for, while, case and repeat statements must be no more than four levels,
    - whenever there is a chance of misrepresentation parentheses should be used,
    - avoid the use of negatives in a conditional expression.

- GENERAL PRETTYPRINTING RULES

    - each statement must begin on a separate line,
    - each line must be less than or equal to 72 characters,
    - comments that are appended at the end of a line of code and that are continued on successive lines must be written so that they are under the initial comment fragment,
    - keywords begin, end, if, then, else, do, repeat, until must be on a line by themselves, followed by supporting comments,
    - at least one blank line must appear before label, const, type and var declarations,
    - at least three blank lines must appear before procedure and function declarations,
    - at least one space must appear before and after ' := ' and ' = ',
    - at least one space must appear after ': ',
    - at least one space must appear after '(* ' and '} ' and before ' *)' and ' }' in a comment.

- ALIGNMENT RULES

    - program, procedure and function headings begin at the left margin,
    - the main begin and end block for a program, procedure and function must line up with the heading,
    - each statement within a begin-end, if-then-else, case, while-do, repeat-until must be aligned.

- INDENTATION RULES

    - indent consistently - three places is a minimum,
    - the bodies of label, const, type, and var declarations must be indented from the beginning of the corresponding header keywords,
    - each statement within a begin-end, if-then-else, case, while-do, repeat-until must be indented,
    - separate code from comments with white space.

[LHSi77]
[Marc81]

4.4 SAMPLE PROGRAM DOCUMENTATION STANDARDS

- HIGH LEVEL DOCUMENTATION
    - description of the problem,
        - name, user application, purpose of program,
    - input information,
        - name, type, format, data elements, organization,
    - processing information,
        - name, conditions, steps, special detail,
    - output information,
        - name, type, format, data elements, organization,
    - sample operations
        - input, output,

- INTERMEDIATE LEVEL DOCUMENTATION
    - structure chart,
    - internal data structures,
    - pseudocode,
    - module interface tables,

- LOW LEVEL DOCUMENTATION
    - program code,
    - comments,

- RUN LEVEL DOCUMENTATION
    - test data,
    - test results.

[Fole83]

4.5 SUMMARY

Documentation is a very important part of a successful life of a program. To aid in getting the students used to proper documentation, adherence to formatting, program commenting, identifier naming and modularity conventions all could be a part of the program grade. Just as important as proper program documentation is complete supporting documentation. Standards and a grading criterion for these should also be developed.

Currently the level of program documentation of the program (program format, amount of comments and identifier length) can be measured automatically. What still needs visual inspection is the supporting documentation. Therefore the level of documentation required will determine how much of the documentation grade can be automated.

The level of documentation required from the student will be dependent on the level of the programming course. But as courses progress so should the level of documentation required. Hollingsworth [Holl83] had a grading policy on documentation. "Until the documentation meets strict and high standards the student gets no grade whatsoever for his homework effort." This may seem harsh, but it works to improve the student's quality of documentation.

5. PROGRAM OUTPUT

5.1 INTRODUCTION AND BACKGROUND

Up to this point a student's program has been shown to have a variety of attributes. What has not been discussed is whether or not it works the way it is suppose to and how to measure whether it does. There were a couple of ways to aid in the determination of whether the program was correct or not.

> As tools for realizing correct programs, program testing and program proving were at the ends of a spectrum whose range is the number of times the program must be executed. To establish its correctness through testing, one must execute the program at least once for all possible unique inputs; thus an infinite number of times. To establish its correctness through a rigorous correctness proof, one need not execute the program at all; but one may be faced with a tedious, if not difficult, formal analysis. These two extreme points of the spectrum offer other contrasts as well. Correctness proofs usually ignore certain realities encountered in actual test runs. On the other hand one may finish a proof of correctness, but seldom does one ever finish testing a program [King75].

It is not intended in this thesis to present a detailed discussion of the theory of testing. Instead testing methods in general will be touched on.

5.2 GENERAL TESTING APPROACHES

One of the first automated "grader" of student's labs was introduced in 1960 by a current instructor at Rochester

Institute of Technology, Hollingsworth [Holl60]. The original approach was to use the automated "grader" to monitor the success or failure of a student's attempt at running a program. The article pointed out many limitations of this implementation.

Hollingsworth has recently added a more vigorous approach to the testing of a student's program [Holl83]. He felt the major considerations when developing this type of testing tool were:

1. Will the answers be computed ahead of time or will an instructor's program generate the answers at the same time as the student?

2. Will all the test data be available to the student or only that data which caused a program to operate incorrectly?

3. How will the student's information be stored?

His most recent method was to format each student program as a subroutine, and then store all the subroutines in individual files. The automated grader would then be given a list of the files for processing. The processing of each student file consists of computing a teacher's result and a student's result for each data input, as read. The comparison of the results were reported in a unique file for each student.

Naur's [Naur64] approach was very similar to Hollingsworth [Holl83].

Forsythe's and Wirth's [FoWi65] program "Grader2" appeared to be a direct growth of Hollingsworth's [Holl60] original ideas. The major difference was that this second implementation resolved all of the system problems reported in Hollingsworth's first article [Holl60]. The grading system also included a program called "Test" for more advanced courses to weight the quality, ie. reliability and effectiveness, of a program. In the articles example of evaluating an integration function, the accuracy of the integral answer and the number of evaluations of the function to find the integral were measured.

Another method was to automatically compare the student's output with the instructor's output. On a UNIX system the "diff" command would be used. This command compared each line of two output files and reports the differences found between the two files. This approach is only good for fixed or specifically formatted output, and allows for the option of ignoring spaces.

The objectives for developing an automatic grader for testing a student's program could include the following:

- it should relieve the instructor of a routine and repetitive aspect of grading,
- it should better evaluate the correctness of programs,
- it should make grading of computer programs more consistent and objective,
- it should be simple and convenient to use.

An interactive testing system was discussed by Laski [Lask80]. This was an attempt to provide the user with access to all areas of the code in order to detect incorrect situations. This method of testing was accomplished by sectioning a program in terms of a program model with control and data flow components. The testing approach followed the usual progression of programmer testing process from top down.

Berry [Berr83] discussed the development of test cases to be used in the testing of a compiler. This method collected information on how the compiler would be used, and then used that information to generate test cases. This method then limited the number of test cases and combinations tested, based on usage, from an infinite number of possible test cases and combinations to those test cases which might be more useful.

Chapman [Chap82] developed a method of generating test cases that paid special attention to the environment that will use the test cases.

Two tools called SELECT and EFFIGY had been developed by Boyer, Elspas, and Levitt [BELe75] and King [King75]

respectively. These are interactive debugging/testing tools useful in program development. The systems perform a symbolic execution of the paths of the program.

A system utility to aid students in the creation, maintenance and testing of test files to be used with their labs was presented by Naur [Naur64].

By far the most literature was found on the controversial area of the theoretical proof of program correctness. Tanenbaum's [Tane76] insight into why extensive testing will not be replaced by the theoretical proofs, was very realistic. His reasoning came from the following observations:

- correctness proofs can not demonstrate that the formal specifications have not missed a subtle, but critical point,
- proving a program is far more difficult than writing a program,
- programmers may not always be aware of things which they should be aware of:
  - correct semantics of the programming language,
  - correct semantics of the operating system calls,
  - correct understanding of how the hardware works,
  - computers have a finite amount of memory (important in proving a recursive application),
  - a real time system program may fail due to timing considerations.

The advantages and disadvantages of the theoretical (using proofs) and the empirical (using test cases) approachs to proving program correct was covered in Howden [Howd78]. A summary of these follows:

- theoretical approach
    - advantages
        - if a particular testing strategy had a sound theoretical basis, then the user of the strategy knew what had been proven and what had not been proven.
    - disadvantages
        - the proofs of the theorems depend on the availability of a test oracle which can recognize the correctness of a specific kind of test output, these may not always be available,
        - the theorems may require certain assumptions about the correct version of the program which are difficult to justify.

- empirical approach
    - advantages
        - it can be applied to any testing methodology.
    - disadvantages
        - lack of a sound mathematical basis.

Correctness proofs have their place, but they can easily lull one into a false sense of security, and therein lies the potential danger. They should be regraded as an important technique for insuring internal consistency, but they can not eliminate problems having to do with the "programming environment" rather than the internal program logic. Testing can not guarantee this either, but well thought out testing can possibly detect some errors that proofs inherently miss. The two methods do not compete; rather they complement each other. If possible, all programs should be proven and tested [Tane76].

5.3 SUMMARY

Test cases are a good tool to use both in the academic and development testing environment. The theoretical approach to testing verification is being researched, yet it has a long way to go to become realistic. Both approaches, theoretical and empirical complement each other and will continue to have their place in future testing strategies.

6. PROGRAM EFFICIENCY

6.1 INTRODUCTION AND BACKGROUND

Program efficiency was not usually included as a part of a student's programming lab grade. Thus writing efficient programs was not a concern of students, they just wanted to get the assignment completed. Efficiency is included here because at some point a programmer has to be concerned with efficiency, if not at school, then possibly some day at work. People may argue that because of unlimited computer resources this does not need to be a concern. But as more users become aware of the potential of the computer they will require more applications, especially real time applications. Unless care is taken in designing efficient programs these application requirements may not be met.

There are several areas of efficiency which can be looked at:

- Efficiency of development,
- Run time efficiency,
- Efficiency of space utilization.

Section 6.2 discusses tools available to evaluate different types of program efficiencies.

6.2 PREVIOUS WORK

Efficient use of the computer can be measured in a couple of ways. How often a student compiles a program can tell something of how much desk checking was done and how effective it was. Matwin and Missala [MaMi76] explained a tool that tracked the execution time of procedures. This information can then be used to improve the most used procedures to make the total program more efficient. Ripley and Griswold [RiGr75] explained the same type of measurement tool with the added capability of measuring the entire program, or part of a program. This measuring can also be done in an interactive mode. This last mode allowed one to make changes and immediately see the results.

> An article by Sites [Site78] explained the benefits of having both statement counts and procedure timings. Statement counts alone can not find real time performance problems, and procedure times alone cannot find detailed algorithm problems. Together, they form a cheap, simple-to-implement pair of tools which should be designed into every compiler system.

Efficiency tools which were implemented on UNIX for C programs are called "prof" and "time". More information on these tools is available from the UNIX Programmers Manuals.

6.3 SUMMARY

Program efficiency is not a major concern when developing a program. As Van Tassel [VanT78] said "If it doesn't work, it doesn't matter how efficient it is" and "Readability is usually more important than efficiency." Thus efficiency must be placed in the proper perspective. It is not harmful for an instructor to introduce the topic of program efficiency and the tools available to aid in its attainment. What better time for the student to be aware of writing efficient programs then when they are beginning to develop their programming skills?

7. TOOLS DEVELOPED

7.1 INTRODUCTION

This thesis includes five programs in a system which looks at the style of a student's Pascal program and also attempts to flag possible plagiarized programs. The five programs in this system are discussed in Section 7.2

7.2 PROGRAM EXPLANATION

Sections 7.2.1, 7.2.2, 7.2.3, 7.2.4, and 7.2.5 explains the purpose of each program written for this thesis, along with the input and output file information and any procedures used in the programs. Where appropriate those steps taken for efficiency and the algorithms used are also explained.

7.2.1 STR.COM.C - COMMENT STRIPPER

purpose:	This C program will remove all comments from a Pascal program. This means anything between {} and (* *) is not output. Anything between a single or double quote is also removed from the program. A count of the number of comment lines in a program is also kept.	
input file:	student's program	
	created by	: the student
	input	: the program
output file:	standard output	
	used by	: style.c
	output	: uncommented program
	commfile	
	used by	: style.c
	output	: count of comment lines in the program
procedures:	none	

In the interest of creating an efficient system comments were stripped using a C program instead of 'lex'. The program used a case process to eliminate the appropriate information. For Pascal programs characters between ', ", {}, and (* *) are removed.

7.2.2 TOKEN.L - LEXICAL ANALYZER

purpose: This program is written in lex. Its function is to take a student's Pascal program and break it into operator, operand, or reserved word tokens. When token.l is called from style.c it looks for the appropriate token to return to style.c. When a token is found the type of token is returned using the defines in token.h, and the actual token, if needed, is returned in lex_text.

input files: none.

output files: none.

procedures: none.

In the interest of creating an efficient system this lex program is organized to look for the most frequently used tokens first, where possible.

7.2.3 TOKEN.H - HEADER

purpose: This file contains the defines used in the programs token.l and style.c. These defines represent the different types of Pascal tokens. These tokens are separated from a program by token.l and processed by the program style.c.

input files: none.

output files: none.

procedures: none.

7.2.4 STYLE.C - STYLE GRADER

purpose:

This C program accepts a student's Pascal program immediately after the comments have been stripped by str.com.c. This program performs the following functions:
- calls token.l and token.h to return token values to this program,
- the token values returned are evaluated and handled as operators, operands, reserved words or comments,
- a table of operands, reserved words, and constants is created using a hash function,
- the information collected about the program is used to produce a style report, style grade and information for a plagiarism report.

input files:

standard input
- created by : str.com.c
- input : uncommented program

commfile
- created by : str.com.c
- input : comment count

output files:

plagfile
- used by : plag.c
- output : plagiarism information

standard output
- used by : student, professor
- output : style report

procedures:

build_tbl
- called by : main
- calls : yylex, lookup, ins_tbl, strip_p_f

lookup
- called by : build_tbl
- calls : hash, strcmp, cnt_declaration, cnt_structure, cnt_i_o

hash
- called by : lookup

```
ins_tbl
        called by : build_tbl
        calls     : strsave
strsave
        called by : ins_tbl
        calls     : strcpy
strip_p_f
        called by : build_tbl
        calls     : strsave
cnt_operators
        called by : main
cnt_res_wd
        called by : main
        calls     : cnt_declaration
                  : cnt_structure
                  : cnt_i_o
cnt_declaration
        called by : cnt_res_wd
        calls     : lookup
cnt_structure
        called by : cnt_res_wd
        calls     : lookup
cnt_i_o
        called by : cnt_res_wd
        calls     : lookup
cnt_id
        called by : main
style
        called by : main
print_out
        called by : main
plagiarism
        called by : main
```

In the interest of creating an efficient system two methods are used.

First a case process is used to handle each token returned from token.l. The case statement is organized to handle the most frequently used tokens first.

Second the tokens are stored using a hash function to provide the index into the entry table. This gives fast access for storing and retrieving tokens.

The following defines the different algorithms used in this program.

The hash algorithm takes the sum of each character in the string passed to it and then divides this sum by a prime number, which is also the size of the table. Collisions when making entries in the table are handled by looking for the next available table entry after the position hashed to.

The style grade is produced based on the following processes. First a count of the total number and number of unique for each of the following criterion is produced:

1. operators equals the sum of the occurrences of :=, *, +, -, /, div, mod, **, <>, >=, <=, =, <, >, or, and, not, in,
2. declarations equals the sum of the reserved words array, boolean, char, const, file, integer, label, packed, program, real, record, set, type, var, varying,
3. control structures equals the sum of the reserved words begin, case, do, downto, else, end, for, goto, if, of, otherwise, repeat, then, to, until, while, with,
4. input/output equals the sum of the reserved words get, put, read, readln, reset, rewrite, write, writeln,
5. procedures is the sum of the number of procedures,
6. functions is the sum of the number of functions,
7. numeric constants is the sum of the numerics in the program.
8. identifier is the sum of all identifiers not including any reserved words,
9. comment is the number of comments counted in the program str.com.c.

Second a grade is calculated for each of the nine criterion above. The total count for each value is used, but the unique count, or a percentage, or a combination of any of these, for each criterion, could be used. The grade is calculated similar to the approach used by Rees [Rees82]. See Section 2.2.2.3. for a more complete explanation.

The grade for each criterion is calculated using the variables max_score, lo_no, lo_max, hi_max, hi_no. The max_score is given for values between lo_max and hi_max. A zero grade is given for values less than lo_no or greater than hi_no. A linear grade between zero and max_score is given for values between low_no and low_max or hi_max and hi_no.

The style grade for each student is the sum of each criterion's grade.

The output for the style report consists of an individual student report consisting of the following information for each criterion except comments: number unique, number unique per lines of code, total number, total per lines of code, and a style mark. The comment line consists of total comment lines, comments per lines of code, and a style mark. The next line consists of the number of lines of code and the last line is the total style grade for the students. A sample output of a student style report follows.

STUDENT PROGRAM NUMBER: programl.p
Wed Oct 10 19:05:28 EDT 1984
INSTRUCTOR: instructor's name
CLASS: class identification

STUDENT'S STYLE INFORMATION

	number unique	unique/ lines	total	total/ lines	style mark
operators	16	0.03	245	0.52	11.00
declarations	9	0.02	37	0.08	11.00
control structures	13	0.03	348	0.75	11.00
input/output	4	0.01	45	0.10	11.00
procedures	25	0.05	80	0.17	11.00
functions	4	0.01	30	0.06	11.00
numeric constants	23	0.05	188	0.40	11.00
identifiers	51	0.11	394	0.84	0.00
no. of comment lines			278	0.60	12.00
number of code lines			467		
total style score					89.00

7.2.5 PLAG.C - PLAGIARISM DETECTOR

purpose: This C program reads in a file which contains an entry for each student. The entry includes a string of values output by style.c. These values are compared on a student by student basis calculating 2 values for each pair. The two values are:
DIFFERENCE - the sum total of the weighted absolute value of the difference between two student's criteria.
SIMILARITY - a total of the number of criteria with a zero difference

input files: plagfile
created by : style.c
input : plagiarism information for each student

output files: standard output
output : difference and similarity for a pair of students

procedures: read_input
called by : main
calculate
called by : main
print_out
called by : main

The plagiarism information for each student consists of the information passed from style.c. Style.c passes the total for the eight criteria; comment count is not passed. These eight values for each student are read into a table. The calculations used on these values are similar to the ones used by Donaldson, Lancaster, and Sposato [DLSp81]. See Section 3.2.2.4 for further details.

The first value produced is the difference value. The difference is obtained by calculating, for each criterion, the absolute difference of two student's criteria. These differences can be weighted for importance through a predefined multiplication factor. The importance factors are uniquely defined for each criterion. The sum of the weighted absolute difference of all the criteria for a pair of students is the difference value for that pair of students.

The second value produced is the similarity value. The similarity value is obtained by keeping a running total of all the criteria which are identical between a pair of students. This total for a pair of students is the similarity value.

Example:

criterion	1	2	3	4	5	6	7	8
student 1 values	10	10	10	10	10	10	10	10
student 2 values	20	20	10	10	10	10	10	10
importance values	1	2	1	1	1	1	1	1

Difference value for students 1 and 2 is 10 + 10(2) = 30

Similarity value for students 1 and 2 is 6

The output for the plagiarism report consists of lines which state the students' numbers of the students being compared, the difference value for the two students, an asterisk if the difference value is below a predefined value, the similarity value for the two students, and a double asterisk if the similarity value is above a predefined value.

The report is organized such that it reports how the first student compares with the rest of the students, then how the second student compares with the other students starting with the third student, etcetera, until all students have been compared. A sample output of a plagiarism report follows.

INSTRUCTOR: instructor's name
CLASS: class identification
Mon Oct 29 14:49:04 EST 1984

PLAGIARISM REPORT

STUDENTS		DIFFERENCE		SIMILARITY	
1	2	182		0	
1	3	211		0	
1	4	186		0	
1	5	193		0	
1	6	0	*	8	**
2	3	55	*	0	
2	4	6	*	5	**
2	5	63		1	
2	6	182		0	
3	4	49	*	0	
3	5	84		0	
3	6	211		0	
4	5	63		1	
4	6	186		0	
5	6	193		0	

* - represents values below the minimum allowed
** - represents values above the maximun allowed

7.2.6 SUMMARY

On the next page is a diagram which shows how the different programs are tied together.

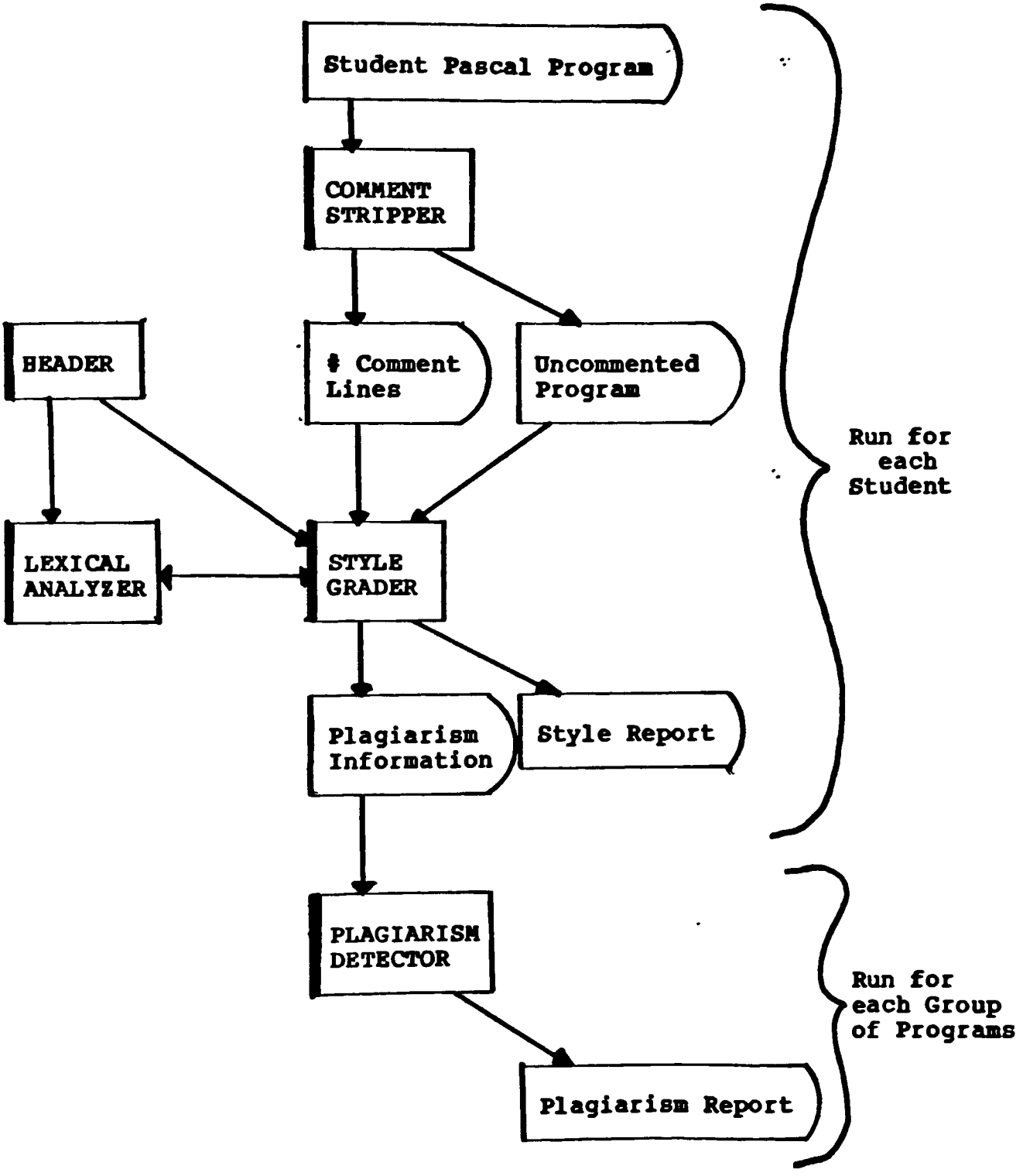
A sample of timings for 40 programs, each 717 lines and 15264 character long on a VAX 11/780, follows:

	MIN	MAX
user seconds	192	203
system seconds	112	114
real time	10:34	15:31

Below is a sample shell script which could be used to evaluate a programming class's programs.

```
#!/bin/sh
for i in *.p
do
#set file to name of file without the .p ending
        file=`expr $i : ".p"`
#place heading on the student style report
        echo "    " > $file.s
        echo "    " >> $file.s
        echo "STUDENT PROGRAM NUMBER: $i " >> $file.s
        date >> $file.s
        echo "INSTRUCTOR:              instructor's name " >>
                                                $file.s
        echo "CLASS:                   class identification " >>
                                                $file.s
#perform comment stripping and
#style grading for each pascal program
        str.com.out < $i | style.out >> $file.s
done
#place heading on the plagiarism report
echo "     " > plagreport
echo "     " >> plagreport
echo "CLASS:      class identification" >> plagreport
echo "INSTRUCTOR: instructor's name " >> plagreport
date >> plagreport
#perform the plagiarism check
plag.out >> plagreport
```

DIAGRAMS OF THE PROGRAMS FLOW



7.3 RESULTS OF THE TOOLS

The next Sections 7.3.1 and 7.3.2 state the objective for each tool developed and the results of the tools which were developed.

7.3.1 STYLE GRADER PROGRAM

The objective when creating the automatic Style Grader was to create an efficient and objective method for grading programming labs.

The results showed that it did both. It was efficient in terms of being able to free an instructors time from having to look at each program for style information. It was also efficient in terms of the computer resources required to run the grader. See Section 7.2.6 for timing results.

The grader was objective. Each student was graded based on the same criteria and rating scheme using a computer. The resulting grade was more of a relative grade. It provided the instructor with general information about the programming ability of the student evaluated.

When the grades obtained from this style grade were matched with the actual grades of the sample students, the style grader correctly represented the student's ability. What was shown was the 'A' students received the highest style grade, while those students close to failing received the lowest style grade.

The style grader did meet its objective, yet it must be remembered that style is only one portion of a programming lab's grade. Other areas an instructor might consider are the actual content and style of the program, and the correctness and format of the program output.

7.3.2 PLAGIARISM DETECTION PROGRAM

The objective when creating an automatic plagiarism detection system was to create a system which would efficiently report on possible cases where plagiarism might have taken place. The system would still require a visual inspection of those programs reported as possibly plagiarized before any action would be able to be taken.

The results showed that the system was efficient. See Section 7.2.6 for timing results.

Summarized below is how effective this system would be at reporting programs plagiarized by the methods discussed in Section 3.1. Types of plagiarism which will be reported on are those programs changed by the following means:

- change the author's name,
- change the documentation,
- change the variable names,
- transpose statements when ordering of statements does not affect results,
- break up single statements such as declarations and output,
- programs stolen from a member of the class which is being matched,
- change the logic a little.

Types of programs which might not be reported are programs which:

- change the logic a lot,
- copy a program from a student not in the group matched,
- a program written completely by someone else,
- partial collaboration.

The information obtained from the student sample reported all students plagiarized on a punch and run programming assignment. This would be expected because they are basically the same program. Two other groups of student's programs matched showed no possible plagiarism. The last group matched detected two groups of plagiarized programs. One group of two students and one group of three students. These results matched what actually occurred in the class. The plagiarism report in Section 7.2.5 is the actual report for this last group.

This system can quickly report possible plagiarized programs. Yet it is still important to visually inspect those programs suspected to determine if they are actually plagiarized. This system will not catch all plagiarism situations, there are some sophisticated techniques used. But it will give some indication. This system may have to be adjusted from time to time, as students catch on to ways to "beat the system".

7.4 USER INFORMATION

The following items will have to be set by the user:

- str.com.c - nothing should be changed,
- token.l - nothing should be changed,
- token.h - nothing should be changed,
- style.c
    - TBL_SIZE - should be a prime number for the hashing function large enough to handle the number of unique identifiers, operators and reserved words in a program, as well as room for collisions.
    - NO_ANALYSIS - should be the number of criteria used in the style analysis portion of the program,
    - max_score - is the maximum score by criterion used for the style grade,
    - low_no - is the low score where by criterion values less than this value will receive a 0 style grade for that style criterion,
    - low_max - is the low score where by criterion values greater than this and less than the next value (high_max) will receive the max_score for that style criterion,

- high_max - is the high score where by
  criterion values less than
  this and greater than the prior
  value (low_max) will receive the
  max_score for that style criterion,
- high_no - is the high score where by
  criterion values greater than
  this value will receive a 0 style
  grade for that style criterion,
- values between low_no and low_max or
  high_max and high_no will
  receive a linear grade between 0 and
  the max_score,
- *style_par - is the definition used in the
  output line for each style criterion,
- param_id - is a table of the criterion
  variables used for the style grade,
- style() and printout() - procedures which
  need to be updated for any style
  output changes,
- plagiarism() - a procedure which needs to be
  updated for any plagiarism output
  changes.

Style.c needs to run immediately after str.com.c in order to record the correct number of comments for any given program. Within style.c if multiple items are hashed to the last table entry location, an error is reported and the item is not entered in the table. The program style.c will look for a string of any form of end. (example End. END .) to represent the end of a student's program.

- plag.c -

- NO_ANALYSIS - the number of criteria used in the plagiarism check,
- MIN_DIFF - the minimum difference allowed in the plagiarism check. Anything below the value is flagged by an *,
- MIN_SIM - the minimum similarity allowed in the plagiarism check. Anything above the value is flagged by an **,
- imp[] - is the weight given to each criterion. This is used in the plagiarism output.

7.5 SUGGESTIONS FOR FUTURE EXTENSIONS

The following items could be looked at if consideration were given to use this system for another language:

- str.com.c - update to handle the given language's comment format,
- token.l and token.h and style.c - update to handle the languages identifiers, operators and reserved words,
- style.c - update to output the appropriate style information,
- plag.c - update to output the appropriate plagiarism information.

8. SUMMARY

The results of the research and the programs developed for this thesis consistently pointed out the benefits of automated tools.

In all the categories which programs could be graded on, the benefits were the same. Automated tools proved to be objective, consistent, efficient, in terms of both computer and instructor resources, and timely.

But among the benefits were certain limitations which must be kept in perspective. These limitations follow:

- The style grade provided a good relative grade, but there still remained a need for an actual grade.
- The plagiarism detection system matched possible plagiarized programs, yet it was still important to visually inspect those programs matched to verify plagiarism.
- The information gathered automatically for documentation was limited to that which itself was automated. Written reports or supporting documentation could not be included into a fully automated grade.
- It is important to review the algorithms used to produce the automated information. Without keeping up with the changes that may be initiated by the students the tools could leave the instructor with a false sense of security in the grade or information reported.

This research proved to me that the benefits far outweight the limitations of automated tools. My overall impression is that automated tools for grading students' programming labs offer many benefits to both the student and instructor.

ANNOTATED BIBLIOGRAPHY

GENERAL GRADING OF PROGRAMMING LABS

[Hals77] Elements of Software Science ;Halstead, Maurice; Elsevier North-Holland, Inc; New York; 1977.

This book presented a theoretical study of computer programs. Half the book was devoted to the development of algorithms which can be applied to programs. The second half of the book was devoted to the application of these algorithms. [RoSo80], [Grie81], [Otte77] were articles which reference this book.

[HHRT83] "A Tool for Program Grading: The Jacksonville University Scale" ; Hamm, R.; Henderson, Kenneth; Repsher, Marilyn; Timmer, Kathleen; ACM SIGCSE ; Vol.15; No.1; Feb 83; pp 248-252.

This article discussed the problem of consistency in the grading of programming assignments. The solution was a variation of a grading technique for English compositions called the Diederich scale. The article suggests seven factors to be weighted by the instructor for each program. To simplify this process a grading sheet with the instructor's: factors, weighting scheme, total value of the project and title of the project was generated automatically.

[HeUl75] "Global Data Flow Analysis Problems"; Hecht, Matthew; Ullman, Jeffrey; SIAM Journal on Computing ; Vol.4; No.4; Dec 75; pp 519-532.

This article presents a variety of definitions, theorems lemmas, and the appropriate proofs for graph theory. The intent of the article was to develop a "bit propagation algorithm" for solving global data flow analysis problems, such as "available expressions" and "live variables"." This article was used as a reference to [RoSo80] in the DAG implementation.

[Knut71] "An Empirical Study of FORTRAN Programs"; Knuth, Donald; Software-Practice and Experience ; Vol.1; No.2; Feb 71; pp 105-133.

This article presented the findings and interpretation of a study of a sample of FORTRAN programs from a variety of sources. The study was conducted to find out actually what programmers do and how FORTRAN was used.

[Meek83] "Style Analysis of Pascal Programs"; Meekings, B. A.; ACM SIGPLAN ; Vol.18; No.9; Sept 83; pp 45-54.

This article presented a revised implementation of the style checker program discussed in [Rose83]. The design of the style checker was in [Rees82]. The article noted seven problems with the original programs. The following programs were included in the article:

- style - a shell program which controls the style-checking for each of its supplied parameters in turn,
- style.cnt.awk - an awk program to count the total number of lines and the number of commented lines,
- style.com.sed - a sed program which deletes comments from the program text,
- style.detab.c - a C program which replaces tabs by the appropriate number of spaces,
- style.str.sed - a sed program which removes strings delineated either by ' or by " from the text, assuming that strings cannot be multi-line,
- style.dict - a dictionary of Pascal's reserved words and standard identifiers,
- style.met.awk - an awk program which corresponds to [Rees82] original proposal with a minor modification,
- style.stan.c - a C program which performs the final analysis of the statistics produced by earlier programs.

[Morg82] "Evaluating Students' Computer Programs"; Morgan, George; Balance Sheet ; Vol.63; No.4; Feb 82; pp 207-209.

This article discussed a method of grading a student's lab by using a predefined rubber stamp. The stamp had six criteria (timely, problem definition, I/O design, logic design, source program, and test validity) and four weighting factors. The instructors then marked the appropriate factor for each criterion to arrive at a grade.

[MiPe80] "A Method for Evaluating Students Written Computer Programs in an Undergraduate Computer Science Programming Language Course"; Miller, Nancy; Peterson, Charles; ACM SIGCSE ; Vol.12; No.4; pp 9-17.

This article referenced three approaches to assigning a grade to a computer program:

- ranking the programs on four characteristics,
- assigning points to three different categories,
- allowing 80% of the points for minimum requirements and 20% for extra work in the areas of program style and output clarity.

The last approach was discussed in detail in this article. The reliability of the approach was discussed and sample grading forms and output from this implementation were included.

[Rees82] "Automatic Assessment Aids for Pascal Programs"; Rees, Michael; ACM SIGPLAN ; Vol.17; No.10; Oct 82; pp 33-42.

This article discussed an automatic assessment aid for grading programs according to their style. Ten measures were described to be used in the program called STYLE. A description of the calculation of the marks and an assessment of the results was included. A couple other modifications to STYLE, including an addition of a program called CHEAT to match students' programs looking for plagiarism was also discussed. [Rose83] and [Meek83] were articles about this style checker and include actual code to implement this system.

[Rose83] Letter to the Editor; Rosenthal, David; ACM SIGPLAN ; Vol.18; No.3; March 83; pp 4-5.

This brief letter included four programs to implement the style checker of [Rees82]. [Meek83] was an article providing modifications and improvements on this article's implementation.

[RoSo80] "An Instructional Aid for Students Programs"; Robinson, Sally; Soffa, M. L.; ACM SIGCSE; Vol.12; No.1; Feb 80; pp 118-129.

This article briefly defined the reason for a system that would provide students with programming suggestions, instructors with information about a student's progress, and flag the possibility of plagiarized programs. The article then continued by explaining a program called ITPAD which will do the above for FORTRAN programs. There were three phases to the ITPAD system:

- Lexical phase - computes fourteen program characteristics,
- Analysis of program structure obtains characteristics using five code optimization steps,
- Analysis of program structure to determine if the student should receive a message about program improvements.

The article then contained a complete examination of the results of this system including a complete set of sample output examples for a simple program. References were made to [Otte77] approach to plagiarism detection. An approach for plagiarism detection using ITPAD was also discussed.

[RoTo77] "The Automatic Measurement of the Relative Merits of Student Programs"; Robinson, S. K.; Torsun, I. S.; ACM SIGPLAN; Vol.12; No.4; April 77; pp 80-93.

This article gave a brief introduction to several methods used to determine how a program behaved along with some references on these methods. It continued with a method of generating an Importance Factor: a rank profile graph for the students and instructors programs. The students' graphs were matched with the instructors graph and based on the varition of the two a grade was determined. An explanation of the grading was given. This system only allows a student's program to slightly vary from the style of the instructor in order to receive a good grade.

[VanT78] Program Style, Design, Efficiency, Debugging, and Testing ; Van Tassel, Dennis; Prentice Hall, Inc; New Jersey; 1978.

This book was a very complete and easy to understand book which touched on five major sections of programming. The topics included were: program style, program design program efficiency, program debugging and program testing. This book appeared to summerize all the concepts which needed to be kept in mind when writing a good computer program.

[DaHi82] "Finger Printing a Program"; Dakir, Karl; Higgins, David; Datamation ; Vol 28; April 82; pp 133-144.

This article discussed plagiarism of software in the business world. The detection of plagiarism could be looked at from five levels as listed below:

- the changing of procedure and data names,
- the rearrangement of the code,
- the change of the operating environment and/or adding additional code,
- the input and output were all that matched,
- there was no proof of copying.

[DLSp81] "A Plagiarism Detection System"; Donaldson, John; Lancaster, Ann-Marie; Sposato, Paula; ACM SIGCSE ; Vol.13; No.1; Feb 81; pp 21-25.

This article discussed briefly the problems of plagiarism and four approaches to its detection as listed below:

- relay on the grader of the assignments,
- [Otte77] - create a four tuple for each program and compare each four tuple,
- [RoSo80] (ITPAD) - build a graph to represent the structure of each student's program, Then compare programs by counting attributes of this representation,
- [DLSp81] - collect and analyze data about each program.

The details of the last approach were covered in this article. The system counts eight parameters, and creates a coded statement string. There were also four algorithms used to analyze the data, these were also explained. The system was designed for FORTRAN, COBOL and BASIC.

[Grie81] "A Tool that Detects Plagiarism in Pascal Program"; Grier, Sam; ACM SIGCSE ; Vol.13; No.1; Feb 81; pp 15-20.

This article discussed in detail a Pascal plagiarism detection system. The system started with the basic ideas of [Otte77]. This system counted twenty parameters, correlated seven of these parameters, and produced five different reports. The method for determining a correlation between two programs was also explained.

[Hals77] see General Grading of Programming Labs

[HwGi82] "Using an Effective Grading Method for Preventing Plagiarism of Programming Assignments"; Hwang, C. Jinshong; Gibson, Darryl; ACM SIGCSE ; Vol.14; No.1; Feb 82; pp 50-59.

This article presented a very complete discussion on plagiarism of programming assignments. It discussed ways students cheat, current methods for the detection or prevention of cheating, general grading methods currently used along with some advantages and disadvantages of these methods. Two experimental grading methods and their implementation and the advantages and disadvantages were also discussed. The article recommended the final grading method discussed as a means for preventing plagiarism.

[Hwan82] "Preventing the Plagiarism of Programming Assignments"; Hwang, C. J.; ACM SIGCSE ; Vol.14; No.1; Feb 82; pp 262-264.

This article highlighted the comments of six panel members on the topic of program plagiarism. The comments included material from [HwGi82], plus the proper disposal of programs to hinder cheating, using structured walk throughs and peer pressure.

[Mill81] "Plagiarism in Computer Sciences Courses"; Miller, Philip; ACM SIGCSE ; Vol.13; No.1; Feb 81; pp 26-27.

This article highlighted the comments of four panel members on program plagiarism. The comments included drawing the line between working together and copying, the use of good automatic detection methods and imposition of severe penalties for offenders.

[Otte77] "An Algorithmic Approach to the Detection and Prevention of Plagiarism; Ottenstein, Karl; ACM SIGCSE ; Vol.8; No.4; 1977; pp 30-41.

This article discussed an automated approach to plagiarism detection which used four software science parameters from [Hals77]. The parameters were the number of unique operators and operands and the total number of occurrences of operators and operands. Sample programs and the results were also presented.

[Rees82] see General Grading of Programming Labs

[RoSo80] see General Grading of Programming Labs

[Shaw80] "Cheating Policy in a Computer Science Department"; Shaw, Mary; ACM SIGCSE ; Vol.12; No.2; July 80; pp 72-76.

This article discussed the research of a committee formed to define cheating and its penalties, and to recommend methods of preventing and detecting cheating. The article discussed the background of plagiarism, the prevention and detection of cheating, dealing with cheating, and policy recommendations. This article also included appendices which contain the following:

- University Rules on Cheating and Plagiarism,
- Computer Science Cheating at Other Schools,
- Student Information: Cheating Policy; Computer Science Department,
- Implementation of the Prevention and Detection Policy.

[Bate78] Letter to the Editor; Bates, David; ACM SIGPLAN ; Vol.13; No.3; March 78; pp 12-15.

This letter criticized the content of the article [Pete77] on program formatting and presented an example of the Bates' recommendation. The difference between Peterson and Bates occurred in the following areas:

- Number of statements per line (recommended average of three statements per line),
- Indentation norm (recommended two spaces).

[Bate81] "A Pascal Prettyprinter with a Different Purpose"; Bates, Rodney; ACM SIGPLAN ; Vol.16; No.3; March 81; pp 10-17.

Bates purpose in designing a prettyprinter was to save time in editing. The article described the Bates' unorthodox and personally confusing method of developing his pretty-printer (Ex. semicolons placed to the left of a statement). The technique of using tokens to implement his style of pretty printer, and how comments were handled was also discussed.

[Bond79] "Another Note on Pascal Indentation"; Bond, Reford; ACM SIGPLAN ; Vol.14; No.12; Dec 79; pp 47-49.

This article discussed programming indentation as a question of style, and as such it was difficult to develop an objective criteria. Bond proposed an indentation algorithm to be used in a prettyprinting program with the advantages it was simple, flexible, and general. A perceived difference with the algorithm was that else and labels do not stand out.

[Clif78] "A Technique for Making Structured Programs More Readable"; Clifton, Mitchell; ACM SIGPLAN ; Vol.13; No.4; April 78; pp 58-63.

This article mentioned a couple of current techniques used to make structured programs easier to read. It also introduced a technique which automatically generated numbered lines to connect separate parts of control structures. An example in PL/1 was included. Some of the current techniques for easier reading of programs mentioned included: use of indentation, comments, nesting level numbers, darker keywords (ALGOL 68), and flowcharts.

[Crid78] "Structured Formatting of Pascal Programs"; Crider, John; ACM SIGPLAN ; Vol.13; No.11; Nov 78; pp 15-22.

This article discussed a technique for indenting Pascal programs called Structured Format. Examples for the different statements were explained. A perceived difference was that the begin and end were placed at the end of the line. Because of this variation Crider introduced a concept called indented end relationship, if a line contained end or until symbols then the number of indentation increments that it had relative to the following line, were equal to the total number of end or until symbols that it contained. This concept did not apply to the last end symbol.

[CoSm79] "NEATER2: A PL/1 Source Statement Reformatter"; Conrow, Kenneth; Smith, Ronald; Comm of the ACM ; Vol.13; No.11; Nov 79; pp 669-675.

This article discussed NEATER2 a PL/1 source statement reformatter. Its two major uses were as follows:

- reformat a program by indentation to indicate its logic level and to make it easier to read,
- count the number of times each statement was executed during execution.

Both of these aided in the complete and efficient testing of a program. Diagnostic messages and the fifteen options of the program were also discussed.

[Fole83] "Program Documentation at Wichita State University" Foley, David; ACM SIGCSE ; Vol.15; no.1; Feb 83; pp 133-136.

This article explained the reasons for formal program documentation, the universities documentation standard, and examples of the documentation standard. An outline of the contents for standard documentation follows:

- High level documentation,
  - description of the problem, input information, processing information, output information, sample operations,
- Intermediate level documentation,
  - structure chart, internal data structures, pseudocode module interface tables,
- Low level documentation,
  - program code, comments,
- Run level documentation,
  - test data, test results,

[Grog79] "On Layout, Identifiers and Semicolons in Pascal Programs"; Grogono, Peter; ACM SIGPLAN ; Vol.14; No.4; April 79; pp 35-40.

Grogono discussed the readability of programs. His program layout aspect was quit similar to [LSHi77]. The other areas discussed were identifiers, semicolons and closing keywords. The article also critiqued many of the other articles included in this bibliography on this topic.

[Gust79] "Some Practical Experiences Formatting Pascal Programs" ; Gustafson, G. G.; ACM SIGPLAN ; Vol 14; No.9; Sept 79; pp 42-49.

This article discussed three methods of formatting Pascal programs. Two methods were described by [Pete7] and [Crif78] and the third was a combination of the two. The article also gave a table comparing the advantages and disadvantages of each method.

[HeNo79] "A One-Pass Prettyprinter"; Hearn, Anthony; Norman, Arthur; ACM SIGPLAN ; Vol.14; No.12; Dec 79; pp 50-58.

This article began with a detailed explanation of the need for a source text formatter. It then continued to explain five methods used to implement this process. These were listed below:

- Simply rebuild the source file from compacted files,
- Use of keywords such as BEGIN and END to trigger indentation or line splitting,
- Use the second method plus some measure of program complexity to guide in its indentation,
- Make a prepass of the program to aid in determining line breaks.
- Picture compile technique - build a data structure to represent the entire printed form of the program and then pass over doing the actual printing.

The new method proposed two coroutines: one to set up the printing process and the second to do the formatting. This new method was explained in this article and was implemented in LISP and RLISP. A sample output was shown.

[Holl83] "The Grading of Students Computer Homework"; Hollingsworth, Jack; 1983 ASEE Annual Conf Proceedings ;1983; pp 755-756.

This article discussed the details of how Hollingsworth graded three categories of a student's program. These categories include the following:

- correctness of the program,
- documentation of the program,
- comments - including program purpose and variable usage; descriptive variable names; program comments and well labeled results,
- quality of the student's testing of the program.

[LeHu77] "An Automatic Formatting Program for Pascal"; Ledgard, Henry; Hueras, Jon; ACM SIGPLAN ; Vol.12; No.7; July 77; pp 82-84.

This article discussed an automated implementation of the formatting standards in [LHSi77]. It also discussed the general approach used in developing a prettyprinter. This implementation used a stack to keep track of indentation. A short sample was also given.

[LHSi77] "A Basis for Executing Pascal Programmers"; Ledgard, Henry; Hueras, Jon; Singer, Andrew; ACM SIGPLAN ; Vol.12; No.7; July 77; pp 101-105.

This article was a well put together set of coding standards for Pascal programmers. Included were rules on general topics, declarations, control structures, prettyprinting, alignment and indentation. A sample program using the standards was also provided. [LeHu77] was an article about an automated implementation of these standards.

[Marc81] "Some Pascal Style Guidelines"; Marca David; ACM SIGPLAN ; Vol.16; No.4; April 81; pp 70-80.

This article presented a complete discussion on a set of guidelines for Pascal programming style. Guidelines were given for the areas of comments, identifiers names, and conditionals. This article was the best one I read on this material.

[Mohi78] "Prettyprinting Pascal Programs"; Mohilner, Patricia; ACM SIGPLAN ; Vol.13; No.7; July 78; pp 34-39.

This article discussed prettyprinting in general. It also compared and contrasted and suggested improvements to the prettyprinting approach presented by the Ledgard, Hueras [LeHe77] and Peterson [Pete77].

[Pete77] "On the Formatting of Pascal Programs"; Peterson, James; ACM SIGPLAN ; Vol.12; No.12; Dec 77; pp 83-86.

This article discussed Peterson's ideas on how to format a Pascal program in order to improve its readability. The basic guidelines of Peterson were as follows:

- each statement will appear on a line by itself,
- simple statements included within compound statements were equally indented,
- compound parts of the structured statement were indented under the controlling expression,
- other miscellaneous line spacing guidelines such as double spacing before and after comments.

[Rams79] "Prettyprinting Structured Programs with Connector Lines"; Ramsdall, John; ACM SIGPLAN ; Vol.14; No.9; Sept 79; pp 74-75.

Ramsdall used the Pascal semicolon at the beginning of a line to represent connections within the program. Ramsdell had also combined the ideas of Clifton [Clif78] and Crider[Crid78] in his approach.

[Berr83] "A New Methodology for Generating Test Cases for a Programming Language Compiler"; Berry, Daniel; ACM SIGPLAN ; Vol.18; No.2; Feb 83; pp 46-52.

This article discussed two approaches to testing compilers. The first approach tried to test all the features of the compiler which quickly became a large number of cases. The second approach collected information on how the compiler would be used in order to limit the cases and combinations tested.

[BELe75] "Select--A Formal System for Testing and Debugging Programs by Symbolic Execution"; Boyer, Robert; Elspas, Bernard; Levitt, Karl; ACM SIGPLAN ; Vol.10; No.6; June 75; pp 234-245.

This article was written about an automated testing tool called SELECT. This tool was intended to be a compromise between the formal proofs of program correctness and the current debugging facilitied. SELECT returns basic conditions on input variables that cause the different program paths to be executed. Basic symbolic values for program variables at program output were also produced. The article included: information on SELECT's features, a couple examples, and a comparison of SELECT with other types of testing methods. An evaluation of the system and some possible extensions were also included.

[Chan78] "An Automated System for Grading Basic Programs"; Chand, Donald; Association for Educational Data Systems 16th Annual Convention ; May 1978; pp 253-257.

This article discussed as automated tool written in AlgolW to evaluate the test results from a Basic program. It was written from the users point of view outlining the necessary steps to make the system functional. Each time a student used this grader it: recorded the program size, updated the number of times this exercise had been graded, calculated a ratio of correctness to total cases, recordeded the date, and the execution time. A sample case was also provided.

[Chap82] "A Program Testing Assistant"; Chapman, David; Comm of the ACM ; Vol 25; No. 9; Sept 82; pp 625-634.

This article discussed the testing assistant program. This program would generate program test cases in a variety of formats and would keep a file of errors as a result of the tests. Sample dialogue for the testing of an air traffic controller simulator was given. This system worked with LISP. References to similar types of programs, TINKER and others unnamed were made.

[FoWi65] "Automatic Grading Programs"; Forsythe, George; Wirth, Niklaus; Comm of the ACM ; Vol. 8; No. 5; May 65; pp 275-278.

This article discussed two programs to automatically generate test data to be used to test and grade a student's ALGOL program. Grading of programs was dependent on the correctness of the output. This approach also referenced and discussed Hollingsworth [Holl60]. The problems discussed in [Holl60] were answered in this article. The article explained in detail how to implement the two graders along with the actual code for each.

[Holl60] "Automatic Grader for Programming Classes"; Hollingsworth, Jack; Comm of the ACM ; Vol. 3; No. 10; Sept 1960; pp 528-529.

This article discussed the use of a program to automatically generate test data for a student's program and determine if the student's program generated the correct results. The article discussed the benefits of such a system along with a flow chart of how to implement one. Some difficulties and limitations of the technology of the time were also discussed.

[Holl83] see Program Documentation

[Howd78] "Theoretical and Empirical Studies of Program Testing"; Howden, William; IEEE Tran on Software Eng ; Vol. SE-4; No. 4; July 78; pp 293-298.

This article presented the advantages and disadvantages of theoretical and empirical testing. It also gave information on the different techniques of testing. Theoretical testing included graph theory methods and algebraic methods. Empirical testing includes path, branch structure, special value and symbolic testing, and program analysis by interface consistency and anomaly analysis. This article also had a good bibliography on testing.

[King75] "A New Approach to Program Testing"; King, James; ACM SIGPLAN ; Vol.10; No.6; June 75; pp 228-233.

This article discussed an interactive debugging and testing system called EFFIGY. This tool allowed one to choose a place between individual test runs and general correctness proofs. A lengthy example of an actual EFFIGY session was provided.

[Lask80] "A Hierarchical Approach to Program Testing"; Laski, Janusz; ACM SIGPLAN ; Vol.15; No.1; Jan 80; pp 77-85.

This was an informational and technical article on an approach to program testing. This approach was good for both large and small projects. The approach was to debug a program using a real environment in a hierarchical manner. This approach started with a test of basic functions and then added options and decision testing. The results of the different levels of testing would be used to aid in determining where the error might occur.

[MiHo78] Tutorial: Software Testing and Validation Techniques ; Miller, Edward; Howden, William; IEEE Computer Society; New York; 1978.

This book contained a complete series of articles on testing covering a variety of topics.

[Naur64] "Automatic Grading of Students' Algol Programming"; Naur, Peter; Bit 4 ; 1964; pp 177-188.

This article discussed an automated tool written in Algol to evaluate the efficiency and logical completeness of a student's algorithm. The method assumed a correct program. The article included the techniques used, the actual program to do the checking, sample results and some suggestions for improvements. The ideas and techniques were good but because of when it was written the actual implementation was obsolete.

[Newc80] "Use of Program Generator to Improve Student Productivity in a Small-Computer Lab"; Newcomer, Larry; ACM SIGCSE ; Vol.12; No.2; July 80; pp 40-42.

This article discussed the benefits of using a program generator to set up test files for students automatically.

[Tane76] "In Defense of Program Testing or Correctness Proofs Considered Harmful"; Tanenbaum, Andrew; ACM SIGPLAN ; Vol.11; No.5; May 76; pp 64-68.

This was a very interesting article emphasizing the importance of actual testing of a program. The article was Tanenbaum's reaction to the current state of the theoretical proof of program correctness.

[MaMi76] "A Simple, Machine Independent Tool for Obtaining Rough Measures of Pascal Programs"; Matwin, S.; Missala, M.; ACM SIGPLAN ; Vol.11; No.8; Aug 76; pp 42-45.

This article proposed a simple technique to measure program performance. The technique calculated the execution time and relative to total execution time for each procedure in a program.

[MoRo82] Tools and Techniques Computer Performance Evaluation for Effective Analysis ; Morris, Michael; Roth, Paul; Von Nostrand Reinhold Company; New York; 1982.

This book covered all aspects of evaluating computer performance. The sections that monitored software performance, which included discussions of software monitors and program optimizers, were used for this thesis.

[RiGr75] "Tools for the Measurement of SNOBOL4 Programs"; Ripley, G. David; Griswold, Ralph; ACM SIGPLAN ;Vol.10; No.5; May 75; pp 36-52.

This article discussed three modes in which SNOBOL programs can be measured as well as how the measurement was implemented. The modes were as follows:

- measure the entire program - using a switch in the SNOBOL interpreter,
- measurement under program control - using a built in MEASURE function,
- measure interactively - using the SPIDER program.

The article explained what was measured and gives examples of output using the different modes.

[Site78] "Programming Tools: Statement Counts and Procedure Timings"; Sites, Richard; ACM SIGPLAN ; Vol.13; No.12; Dec 78; pp 98-101.

This article nicely defined the uses of execution time statements counts and procedure times. Examples of a statement count for a hash routine and procedure timing of the Cray-1 Pascal complier(1977) were provided.

[VanT78] see General Grading of Programming Labs

[Yuva75] "Gathering Run-Time Statistics Without Black Magic"; Yuval, G.; Software-Practice and Experience ; Vol.5; No.1; Jan 75; pp 105-108.

This was a technical article explaining how to collect run time statistics, without: slowing the system, changing hardware or changing the Operating System. The implementation was explained for the CDC6000, Cyber, and Pascal/6000.

BIBLIOGRAPHY

[Bate78] Letter to the Editor; Bates, David; ACM SIGPLAN ; Vol.13; No.3; March 78; pp 12-15.

[Bate81] "A Pascal Prettyprinter with a Different Purpose"; Bates, Rodney; ACM SIGPLAN ; Vol.16; No.3; March 81; pp 10-17.

[BELe75] "Select--A Formal System for Testing and Debugging Programs by Symbolic Execution"; Boyer, Robert; Elspas, Bernard; Levitt, Karl; ACM SIGPLAN ; Vol.10; No.6; June 75; pp 234-245.

[Berr83] "A New Methodology for Generating Test Cases for a Programming Language Compiler"; Berry, Daniel; ACM SIGPLAN ; Vol.18; No.2; Feb 83; pp 46-52.

[Bond79] "Another Note on Pascal Indentation"; Bond, Reford; ACM SIGPLAN ; Vol.14; No.12; Dec 79; pp 47-49.

[Chan78] "An Automated System for Grading Basic Programs"; Chand, Donald; Association for Educational Data Systems 16th Annual Convention ; May 1978; pp 253-257.

[Chap82] "A Program Testing Assistant"; Chapman, David; Comm of the ACM ; Vol 25; No.9; Sept 82; pp 625-634.

[Clif78] "A Technique for Making Structured Programs More Readable"; Clifton, Mitchell; ACM SIGPLAN ; Vol.13; No.4; April 78; pp 58-63.

[CoSm79] "NEATER2: A PL/1 Source Statement Reformatter"; Conrow, Kenneth; Smith, Ronald; Comm of the ACM ; Vol.13; No.11; Nov 79; pp 669-675.

[Crid78] "Structured Formatting of Pascal Programs"; Crider, John; ACM SIGPLAN ; Vol.13; No.11; Nov 78; pp 15-22.

[DaHi82] "Finger Printing a Program"; Dakir, Karl; Higgins, David; Datamation ; Vol 28; April 82; pp 133-144.

[DLSo81] "A Plagiarism Detection System"; Donaldson, John; Lancaster, Ann-Marie; Sposato, Paula; ACM SIGCSE ; Vol.13; No.1; Feb 81; pp 21-25.

[Fole83] "Program Documentation at Wichita State University" Foley, David; ACM SIGCSE ; Vol.15; no.1; Feb 83; pp 133-136.

[FoWi65] "Automatic Grading Programs"; Forsythe, George; Wirth, Niklaus; Comm of the ACM ; Vol.8; No.5; May 65; pp 275-278.

[Grie81] "A Tool that Detects Plagiarism in Pascal Program"; Grier, Sam; ACM SIGCSE ; Vol.13; No.1; Feb 81; pp 15-20.

[Grog79] "On Layout, Identifiers and Semicolons in Pascal Programs"; Grogono, Peter; ACM SIGPLAN ; Vol.14; No.4; April 79; pp 35-40.

[Gust79] "Some Practical Experiences Formatting Pascal Programs" ; Gustafson, G. G.; ACM SIGPLAN ; Vol 14; No.9; Sept 79; pp 42-49.

[Hals77] Elements of Software Science ;Halstead, Maurice; Elsevier North-Holland, Inc; New York; 1977.

[HeNo79] "A One-Pass Prettyprinter"; Hearn, Anthony; Norman, Arthur; ACM SIGPLAN ; Vol.14; No.12; Dec 79; pp 50-58.

[HeUl75] "Global Data Flow Analysis Problems"; Hecht, Matthew; Ullman, Jeffrey; SIAM Journal on Computing ; Vol.4; No.4; Dec 75; pp 519-532.

[HHRT83] "A Tool for Program Grading: The Jacksonville University Scale" ; Hamm, R.; Henderson, Kenneth; Repsher, Marilyn; Timmer, Kathleen; ACM SIGCSE ; Vol.15; No.1; Feb 83; pp 248-252.

[Holl60] "Automatic Grader for Programming Classes"; Hollingsworth, Jack; Comm of the ACM ; Vol.3; No.10; Sept 1960; pp 528-529.

[Holl83] "The Grading of Students Computer Homework"; Hollingsworth, Jack; 1983 ASEE Annual Conf Proceedings ;1983; pp 755-756.

[Howd78] "Theoretical and Empirical Studies of Program Testing"; Howden, William; IEEE Tran on Software Eng ; Vol.SE-4; No.4; July 78; pp 293-298.

[Hwan82] "Preventing the Plagiarism of Programming Assignments"; Hwang, C. J.; ACM SIGCSE ; Vol.14; No.1; Feb 82; pp 262-264.

[HwGi82] "Using an Effective Grading Method for Preventing Plagiarism of Programming Assignments"; Hwang, C. Jinshong; Gibson, Darryl; ACM SIGCSE ; Vol.14; No.1; Feb 82; pp 50-59.

[King75] "A New Approach to Program Testing"; King, James; ACM SIGPLAN ; Vol.10; No.6; June 75; pp 228-233.

[Knut71] "An Empirical Study of FORTRAN Programs"; Knuth, Donald; Software-Practice and Experience ; Vol.1; No.2; Feb 71; pp 105-133.

[Lask80] "A Hierarchical Approach to Program Testing"; Laski, Janusz; ACM SIGPLAN ; Vol.15; No.1; Jan 80;

pp 77-85.

[LeHu77] "An Automatic Formatting Program for Pascal"; Ledgard, Henry; Hueras, Jon; ACM SIGPLAN ; Vol.12; No.7; July 77; pp 82-84.

[LHSi77] "A Basis for Executing Pascal Programmers"; Ledgard, Henry; Hueras, Jon; Singer, Andrew; ACM SIGPLAN ; Vol.12; No.7; July 77; pp 101-105.

[MaMi76] "A Simple, Machine Independent Tool for Obtaining Rough Measures of Pascal Programs"; Matwin, S.; Missala, M.; ACM SIGPLAN ; Vol.11; No.8; Aug 76; pp 42-45.

[Marc81] "Some Pascal Style Guidelines"; Marca David; ACM SIGPLAN ; Vol.16; No.4; April 81; pp 70-80.

[Meek83] "Style Analysis of Pascal Programs"; Meekings, B. A.; ACM SIGPLAN ; Vol.18; No.9; Sept 83; pp 45-54.

[MiHo78] Tutorial: Software Testing and Validation Techniques ; Miller, Edward; Howden, William; IEEE Computer Society; New York; 1978.

[Mill81] "Plagiarism in Computer Sciences Courses"; Miller, Philip; ACM SIGCSE ; Vol.13; No.1; Feb 81; pp 26-27.

[Mohi78] "Prettyprinting Pascal Programs"; Mohilner, Patricia; ACM SIGPLAN ; Vol.13; No.7; July 78; pp 34-39.

[Morg82] "Evaluating Students' Computer Programs"; Morgan, George; Balance Sheet ; Vol.63; No.4; Feb 82; pp 207-209.

[MoRo82] Tools and Techniques Computer Performance Evaluation for Effective Analysis ; Morris, Michael; Roth, Paul; Von Nostrand Reinhold Company; New York; 1982.

[MiPe80] "A Method for Evaluating Students Written Computer Programs in an Undergraduate Computer Science Programming Language Course"; Miller, Nancy; Peterson, Charles; ACM SIGCSE ; Vol.12; No.4; pp 9-17.

[Naur64] "Automatic Grading of Students' Algol Programming"; Naur, Peter; Bit 4 ; 1964; pp 177-188.

[Newc80] "Use of Program Generator to Improve Student Productivity in a Small-Computer Lab"; Newcomer, Larry; ACM SIGCSE ; Vol.12; No.2; July 80; pp 40-42.

[Otte77] "An Algorithmic Approach to the Detection and Prevention of Plagiarism; Ottenstein, Karl; ACM SIGCSE ; Vol.8; No.4; 1977; pp 30-41.

[Pete77] "On the Formatting of Pascal Programs"; Peterson, James; ACM SIGPLAN ; Vol.12; No.12; Dec 77; pp 83-86.

[Rams79] "Prettyprinting Structured Programs with Connector Lines"; Ramsdall, John; ACM SIGPLAN ; Vol.14; No.9; Sept 79; pp 74-75.

[Rees82] "Automatic Assessment Aids for Pascal Programs"; Rees, Michael; ACM SIGPLAN ; Vol.17; No.10; Oct 82; pp 33-42.

[RiGr75] "Tools for the Measurement of SNOBOL4 Programs"; Ripley, G. David; Griswold, Ralph; ACM SIGPLAN ;Vol.10; No.5; May 75; pp 36-52.

[Rose83] Letter to the Editor; Rosenthal, David; ACM SIGPLAN ; Vol.18; No.3; March 83; pp 4-5.

[RoSo80] "An Instructional Aid for Students Programs"; Robinson, Sally; Soffa, M. L.; ACM SIGCSE ; Vol.12; No.1; Feb 80; pp 118-129.

[RoTo77] "The Automatic Measurement of the Relative Merits of Student Programs"; Robinson, S. K.; Torsun, I. S.; ACM SIGPLAN ; Vol.12; No.4; April 77; pp 80-93.

[Shaw80] "Cheating Policy in a Computer Science Department"; Shaw, Mary; ACM SIGCSE ; Vol.12; No.2; July 80; pp 72-76.

[Site78] "Programming Tools: Statement Counts and Procedure Timings"; Sites, Richard; ACM SIGPLAN ; Vol.13; No.12; Dec 78; pp 98-101.

[Tane76] "In Defense of Program Testing or Correctness Proofs Considered Harmful"; Tanenbaum, Andrew; ACM SIGPLAN ; Vol.11; No.5; May 76; pp 64-68.

[Yuva75] "Gathering Run-Time Statistics Without Black Magic"; Yuval, G.; Software-Practice and Experience ; Vol.5; No.1; Jan 75; pp 105-108.

[VanT78] Program Style, Design, Efficiency, Debugging, and Testing ; Van Tassel, Dennis; Prentice Hall, Inc; New Jersey; 1978.

```
/*************************************************************************/
/*      filename:       str.com.c                                        */
/*      author:         Kathleen Muller                                  */
/*      date:           May, 1984                                        */
/*      purpose:                                                         */
/*                      This C program will remove all comments from a   */
/*                      Pascal program.  This means anything between     */
/*                      {} and (* *) is not output.  Anything between    */
/*                      a single or double quote is also removed from    */
/*                      the program. A count of the number of comment    */
/*                      lines in a program is also kept.                 */
/*      input file:     student's program                                */
/*                              created by   : the student               */
/*                              input        : the program               */
/*      output file:    standard output                                  */
/*                              used by      : style.c                   */
/*                              output       : uncommented               */
/*                                             program                   */
/*                      commfile                                         */
/*                              used by      : style.c                   */
/*                              output       : count of comment          */
/*                                             lines in the              */
/*                                             program                   */
/*      procedures:     none                                             */
/*************************************************************************/
#include <stdio.h>
main()
{
int c;                                  /* character read and printed */
int cnt_comment;                        /* count of comments */
char *commfile = "commfile";
                                        /* definitions for commfile */
FILE *fopen(), *fp;
cnt_comment = 0;                        /* initialise comment count */
/* if not EOF will process the character using a case process */
        while ((c = getchar()) != EOF)
        {
                switch (c)
                {
/* if character is ' print everything up to and including next ' */
                case '\'':
                        {
                        while ((c = getchar()) != '\'')
                                ;
                        break;
                        }
/* if character is " print everything up to and including next " */
                case '"':
                        {
                        while ((c = getchar()) != '"')
                                ;
                        break;
                        }
/* if character is { ignore everything up to and including next } */
/* increment comment count if at end of line */
                case '{':
```

```
                        {
                        while ((c = getchar()) != '}')
                                {
                                if (c == '\n')
                                        ++cnt_comment;
                                else
                                        ;
                                }
                        ++cnt_comment;
                        break;
                        }
                case '(':
                        {
/* if character is ( and next character not * print both characters */
/* if character is ( and next character is ' or " treat like a single
   or double quote and disregard anything up to and including the
   next ' or " */
                        if ((c = getchar()) != '*')
                                {
                                if (c == '\'')
                                        {
                                        putchar('(');
                                        while ((c = getchar()) != '\'')
                                                ;
                                        }
                                else if (c == '"')
                                        {
                                        putchar('(');
                                        while ((c = getchar()) != '"')
                                                ;
                                        }
                                else
                                        {
                                        putchar('(');
                                        putchar(c);
                                        }
                                }
                        else
/* if character is ( and next character is * then ignore ( * and all */
/* up to and including * ) */
/* increment comment count at the end of line */
                        {
                        while (((c = getchar()) != '*') ||
                               ((c = getchar()) != ')'))
                                {
                                if (c == '\n')
                                        ++cnt_comment;
                                else
                                        ;
                                }
                        ++cnt_comment;
                        }
                        break;
                        }
/* if character is not a ', ", { and }, or (* and *) then print characters */
                default:
```

```
                        {
                        putchar(c);
                        break;
                        }
                }                                   /* for switch */
        }                                           /* for while */
/* output comment count into file commfile */
        if ((fp = fopen(commfile,"w")) == NULL)
                printf("error:comment file can not be opened\n");
        else
                {
                fprintf(fp,"%10d\n", cnt_comment);
                fclose(fp);
                }
}
```

```
%{
/*****************************************************************************/
/*      filename:       token.l                                              */
/*      author:         Kathleen Muller                                      */
/*      date:           May, 1984                                            */
/*      purpose:                                                             */
/*                      This program is written in lex.  Its function       */
/*                      is to take a student's Pascal program and break      */
/*                      it into operator, operand, or reserved word          */
/*                      tokens.  When token.l is called from style.c it      */
/*                      looks for the appropriate token to return to         */
/*                      style.c.  When a token is found the type of          */
/*                      token is returned using the defines in token.h,      */
/*                      and the actual token, if needed, is returned         */
/*                      in lex_text.                                         */
/*      input files:    none.                                                */
/*      output files:   none.                                                */
/*      procedures:     none.                                                */
/*****************************************************************************/

#include <stdio.h>
#include "token.h"                      /* a file of the defines */
extern char *lex_text;                  /* text from the file token.l */
/*
        The following are the different types of tokens possible in a
        Pascal program.  The types of tokens are seperated into operators,
        operand, and reserved words.
*/
%}
INT     [0-9][0-9"."]*                  /* definition of integer or real */
ID      [A-Za-z][A-Za-z0-9"^""."_"]*    /* definition of identifier */
MLINE   ["\n"]["\n"]*                   /* definition of multiple newlines */
%%
{MLINE} return(NEWLINE);
[Ee][Nn][Dd]" "*"."  return(LASTEND);
[Pp][Rr][Oo][Cc][Ee][Dd][Uu][Rr][Ee]" "{ID}
        {
        lex_text = yytext;
        return(PROC);
        }
[Ff][Uu][Nn][Cc][Tt][Ii][Oo][Nn]" "{ID}
        {
        lex_text = yytext;
        return(FNC);
        }
[Aa][Nn][Dd]  return(LOG_AND);
[Oo][Rr]  return(LOG_OR);
[Nn][Oo][Tt]  return(LOG_NOT);
[Dd][Ii][Vv]  return(INT_DIVIDE);
[Mm][Oo][Dd]  return(MODULUS);
[Ii][Nn]  return(SET_MEMBER);
{ID}
        {
        lex_text = yytext;
        return(IDENTIFIER);
        }
```

```
{INT}
        {
        lex_text = yytext;
        return(INTEGER);
        }
":="    return(ASSIGNMENT);
"**"    return(POWER);
"*"     return(MULT);
"+"     return(ADD);
"-"     return(SUB);
"/"     return(DIVIDE);
"<>"    return(UNEQUAL);
"<="    return(EQ_LESS_THAN);
">="    return(EQ_GRT_THAN);
"="     return(EQUAL);
"<"     return(LESS_THAN);
">"     return(GRT_THAN);
.       {lex_text = yytext;
        return(ALL_ELSE);}
```

```
/*****************************************************************************/
/*      filename:       token.h                                              */
/*      author:         Kathleen Muller                                      */
/*      date:           May, 1984                                            */
/*      purpose:                                                             */
/*                      This file contains the defines used in the           */
/*                      programs token.l and style.c.  These defines         */
/*                      represent the different types of Pascal tokens       */
/*                      These tokens are seperated from a program by         */
/*                      token.l and processed by the program style.c         */
/*      input files:    none.                                                */
/*      output files:   none.                                                */
/*      procedures:     none.                                                */
/*****************************************************************************/
char *lex_text;                 /* characters of matched in lex */
#define IDENTIFIER      1       /* an identifier */
#define INTEGER         2       /* integer identifier */
#define NEWLINE         3       /* new line control character */
#define PROC            4       /* a procedure name */
#define FNC             5       /* a function name */
#define ASSIGNMENT      6       /* an assignment operator := */
#define MULT            7       /* multiplication operator * */
#define ADD             8       /* addition operator + */
#define SUB             9       /* subtraction operator - */
#define DIVIDE         10       /* division operator / */
#define INT_DIVIDE     11       /* integer division operator div */
#define MODULUS        12       /* modulus operator mod */
#define POWER          13       /* power operator ** */
#define UNEQUAL        14       /* unequal operator <> */
#define EQ_LESS_THAN   15       /* less than and equal operator <= */
#define EQ_GRT_THAN    16       /* greater than and equal operator >= */
#define EQUAL          17       /* equal operator = */
#define LESS_THAN      18       /* less than operator < */
#define GRT_THAN       19       /* greater than operator > */
#define LOG_AND        20       /* logical and operator and */
#define LOG_OR         21       /* logical or operator or */
#define LOG_NOT        22       /* logical not operator not */
#define SET_MEMBER     23       /* set member operator in */
#define LASTEND        24       /* programs last end */
#define ALL_ELSE       25       /* any thing not accounted for above */
```

```
/*****************************************************************************/
/*      filename:       style.c                                              */
/*      author:         Kathleen Muller                                      */
/*      date:           May, 1984                                            */
/*      purpose:        This C program accepts a student's                   */
/*                      Pascal program immediately after the                 */
/*                      comments have been stripped by                       */
/*                      stripped by str.com.out. This program performs       */
/*                      the following functions:                             */
/*                      - calls token.l and token.h to return token          */
/*                        values to this program,                            */
/*                      - the token values returned are evaluated and        */
/*                        handled as operators, operands, reserved           */
/*                        words or comments,                                 */
/*                      - a table of operands, reserved words,               */
/*                        and constants is created using a hash function     */
/*                      - the information collected about the program        */
/*                        is used to produce a style report, style grade     */
/*                        and information for a plagiarism report.           */
/*      input files:    standard input                                       */
/*                              created by : str.com.c                       */
/*                              input      : uncommented                     */
/*                                           program                         */
/*                      commfile                                             */
/*                              created by : str.com.c                       */
/*                              input      : comment count                   */
/*      output files:   plagfile                                             */
/*                              used by    : plag.c                          */
/*                              output     : plagiarism info                 */
/*                      standard output                                      */
/*                              used by    : student, professor              */
/*                              output     : style report                    */
/*      procedures:     build_tbl                                            */
/*                              called by : main                             */
/*                              calls     : yylex                            */
/*                                          lookup                           */
/*                                          ins_tbl                          */
/*                                          strip_p_f                        */
/*                      lookup                                               */
/*                              called by : build_tbl                        */
/*                              calls     : hash                             */
/*                                          strcmp                           */
/*                                          cnt_declaration                  */
/*                                          cnt_structure                    */
/*                                          cnt_i_o                          */
/*                      hash                                                 */
/*                              called by : lookup                           */
/*                      ins_tbl                                              */
/*                              called by : build_tbl                        */
/*                              calls     : strsave                          */
/*                      strsave                                              */
/*                              called by : ins_tbl                          */
/*                              calls     : strcpy                           */
/*                      strip_p_f                                            */
/*                              called by : build_tbl                        */
/*                              calls     : strsave                          */
```

```
/*                      cnt_operators                                   */
/*                              called by : main                        */
/*                      cnt_res_wd                                      */
/*                              called by : main                        */
/*                              calls     : cnt_declaration             */
/*                                        : cnt_structure               */
/*                                        : cnt_i_o                     */
/*                      cnt_declaration                                 */
/*                              called by : cnt_res_wd                  */
/*                              calls     : lookup                      */
/*                      cnt_structure                                   */
/*                              called by : cnt_res_wd                  */
/*                              calls     : lookup                      */
/*                      cnt_i_o                                         */
/*                              called by : cnt_res_wd                  */
/*                              calls     : lookup                      */
/*                      cnt_id                                          */
/*                              called by : main                        */
/*                      grade                                           */
/*                              called by : main                        */
/*                      print_out                                       */
/*                              called by : main                        */
/*                      plagiarism                                      */
/*                              called by : main                        */
/*                                                                      */
/************************************************************************/
#include <stdio.h>
#include "token.h"                      /* a file of the defines */

extern char *lex_text;                  /* text info from file token.l */

#define TBL_SIZE 509                    /* table size */
#define NO_ANALYSIS 9                   /* number items in style grade */

struct tbl_entry                        /* input for a table entry for
                                           identifiers, reserved words,
                                           and operands */
{
        char *word;                     /* table identifier name */
        int  word_cnt;                  /* number of identifiers counted */
        char type;                      /* type of identifier */
};
static struct tbl_entry entry[TBL_SIZE]; /* table for entries */
                                        /* the following are the variables
                                           for the operator counts */

int cnt_comment;
int cnt_newline;
int cnt_assignment;
int cnt_mult;
int cnt_add;
int cnt_sub;
int cnt_divide;
int cnt_int_divide;
int cnt_modulus;
int cnt_power;
int cnt_unequal;
```

```
int cnt_eq_less_than;
int cnt_eq_grt_than;
int cnt_equal;
int cnt_less_than;
int cnt_grt_than;
int cnt_log_and;
int cnt_log_or;
int cnt_log_not;
int cnt_set_member;
                                        /* the following are the variables
                                           for the specified type, both
                                           for the unique count and the
                                           total count */
                                        /* the operator count */
int oper_u;
int oper_t;
                                        /* the declaration counts */
int dec_u;
int dec_t;
                                        /* the control structures counts */
int cont_str_u;
int cont_str_t;
                                        /* the input/output counts */
int i_o_u;
int i_o_t;
                                        /* the reserved word counts */
int res_wd_u;
int res_wd_t;
                                        /* the procedure counts */
int proc_u;
int proc_t;
                                        /* the function counts */
int fnc_u;
int fnc_t;
                                        /* the numeric constant counts */
int num_cons_u;
int num_cons_t;
                                        /* the identifier type counts */
int i_type_u;
int i_type_t;
                                        /* the actual identifier counts */
int id_u;
int id_t;
                                    /* information to update for style output */

static int max_score[NO_ANALYSIS] =     { 11, 11, 11, 11, 11, 11, 11, 11, 12 };
/* variables used for the BIG PASCAL program */

static int low_no[NO_ANALYSIS] =        {  1,  1,  1,  1,  1,  1,  1,  1,  1 };
static int low_max[NO_ANALYSIS] =       {200, 20,300, 25, 60, 15,180, 10,250 };
static int high_max[NO_ANALYSIS] =      {250, 40,400, 45, 85, 35,190, 15,300 };
static int high_no[NO_ANALYSIS] =       {300, 50,450, 50, 90, 45,200, 50,400 };


/* variables used for the PUNCH and RUN programs */
```

```
/*
static int low_no[NO_ANALYSIS] =        {  5,  1,  0,  5,  0,  0,  5, 50,  0 };
static int low_max[NO_ANALYSIS] =       { 20,  4,  1,  9,  0,  0, 10, 60,100 };
static int high_max[NO_ANALYSIS] =      { 30,  7,  3, 14,  0,  0, 15, 70,300 };
static int high_no[NO_ANALYSIS] =       { 40,  8,  5, 20,  0,  0, 20, 80,400 };
*/

/* variables used for the GROUP programs */
/*
static int low_no[NO_ANALYSIS] =        { 30,  5, 30,  5,  0,  0, 20, 90,  1 };
static int low_max[NO_ANALYSIS] =       { 40, 10, 40, 10,  5,  0, 25,100,250 };
static int high_max[NO_ANALYSIS] =      { 50, 20, 55, 20, 10,  0, 40,150,300 };
static int high_no[NO_ANALYSIS] =       { 60, 30, 65, 25, 15,  0, 55,200,400 };
*/
/* variables used for the CRAPS programs */
/*
static int low_no[NO_ANALYSIS] =        { 25,  5, 30,  5,  0,  5,  5, 50,  1 };
static int low_max[NO_ANALYSIS] =       { 30, 10, 40, 10,  5, 10, 10, 60,250 };
static int high_max[NO_ANALYSIS] =      { 40, 20, 55, 20, 10, 25, 20, 90,300 };
static int high_no[NO_ANALYSIS] =       { 50, 30, 65, 25, 20, 30, 30,100,400 };
*/
static char *style_par[] = { "    operators          ",
                             "    declarations       ",
                             "    control structures ",
                             "    input/output       ",
                             "    procedures         ",
                             "    functions          ",
                             "    numeric constants  ",
                             "    identifiers        ",
                             "    no. of comment lines" };
static float param_id[NO_ANALYSIS];
static float style_score[NO_ANALYSIS];
float tot_score;
char *commfile = "commfile";
FILE *fopen(), *fp;

/*****************************************************************************/
/*
        This is the main procedure which controls the whole program
*/
main()
{
build_tbl();
cnt_operators();
cnt_res_wd();
cnt_id();
grade();
print_out();
plagiarism();
}

/*****************************************************************************/
/*
        This procedure will take the information provided by the lex
        program (token, and text) and handle the information according
        to the following rules:
```

```
        operators: counts for each are incremented by one.
        identifier, numeric constants: are added to the table with a type
                of 'I' or 'N' respectively if they are not in the table,
                and the count is incremented by one.
                If they are in the table only the count is incremented by
                one for that identifier.
        procedure, function: names are added to the table with a type of
                'P' or 'F' respectively if they are not in the table,
                and the count is incremented by one.  If they are in
                the table only the count is incremented by one
                for that procedure or function name.
*/
build_tbl()
{
int i;                                  /* table index */
int token_type;                         /* lex token type */
char *p_f_name;                         /* procedure or function name */
char *strip_p_f();                      /* procedure for obtaining
                                           procedure or function name */
                                        /* initialize operator counts */

        cnt_newline = 0;
        cnt_assignment = 0;
        cnt_mult = 0;
        cnt_add = 0;
        cnt_sub = 0;
        cnt_divide = 0;
        cnt_int_divide = 0;
        cnt_modulus = 0;
        cnt_power = 0;
        cnt_unequal = 0;
        cnt_eq_less_than = 0;
        cnt_eq_grt_than = 0;
        cnt_equal = 0;
        cnt_less_than = 0;
        cnt_grt_than = 0;
        cnt_log_and = 0;
        cnt_log_or = 0;
        cnt_log_not = 0;
        cnt_set_member = 0;
                                        /* initialization of the table */
        i = 0;
        while (i < TBL_SIZE)
        {
                entry[i].word = NULL;
                entry[i].word_cnt = 0;
                entry[i].type = NULL;
                i++;
        }
                                        /* code to process the lex tokens
                                           and provided text */
        while ((token_type = yylex()) != LASTEND)
        {
        switch(token_type)
        {
        case IDENTIFIER:                {
                                        i = lookup(lex_text);
```

```
                        if ((entry[i].word) != NULL)
                                ++entry[i].word_cnt;
                        else
                        {
                        ins_tbl(lex_text,i);
                        entry[i].type = 'I';
                        }
                        break;
                        }
        case INTEGER:   {
                        i = lookup(lex_text);
                        if ((entry[i].word) != NULL)
                                ++entry[i].word_cnt;
                        else
                        {
                        ins_tbl(lex_text,i);
                        entry[i].type = 'N';
                        }
                        break;
                        }
        case NEWLINE:   {
                        ++cnt_newline;
                        break;
                        }
        case PROC:      {
                        p_f_name = strip_p_f(lex_text);
                        i = lookup(p_f_name);
                        if ((entry[i].word) != NULL)
                                ++entry[i].word_cnt;
                        else
                                {
                                ins_tbl(p_f_name,i);
                                entry[i].type = 'P';
                                }
                        break;
                        }
        case FNC:       {
                        p_f_name = strip_p_f(lex_text);
                        i = lookup(p_f_name);
                        if ((entry[i].word) != NULL)
                                ++entry[i].word_cnt;
                        else
                                {
                                ins_tbl(p_f_name,i);
                                entry[i].type = 'F';
                                }
                        break;
                        }
        case ASSIGNMENT:{
                        cnt_assignment++;
                        break;
                        }
        case MULT:      {
                        cnt_mult++;
                        break;
                        }
```

```
        case ADD:               {
                                cnt_add++;
                                break;
                                }
        case SUB:               {
                                cnt_sub++;
                                break;
                                }
        case DIVIDE:            {
                                cnt_divide++;
                                break;
                                }
        case INT_DIVIDE:        {
                                cnt_int_divide++;
                                break;
                                }
        case MODULUS:           {
                                cnt_modulus++;
                                break;
                                }
        case POWER:             {
                                cnt_power++;
                                break;
                                }
        case UNEQUAL:           {
                                cnt_unequal++;
                                break;
                                }
        case EQ_LESS_THAN:      {
                                cnt_eq_less_than++;
                                break;
                                }
        case EQ_GRT_THAN:       {
                                cnt_eq_grt_than++;
                                break;
                                }
        case EQUAL:             {
                                cnt_equal++;
                                break;
                                }
        case LESS_THAN:         {
                                cnt_less_than++;
                                break;
                                }
        case GRT_THAN:          {
                                cnt_grt_than++;
                                break;
                                }
        case LOG_AND:           {
                                cnt_log_and++;
                                break;
                                }
        case LOG_OR:            {
                                cnt_log_or++;
                                break;
                                }
```

```
        case LOG_NOT:           {
                                cnt_log_not++;
                                break;
                                }
        case SET_MEMBER:        {
                                cnt_set_member++;
                                break;
                                }
        case LASTEND:           {
                                i = lookup("end");
                                entry[i].word_cnt++;
                                break;
                                }
        case ALL_ELSE:          {
                                break;
                                }
        default:                {
                                ;
                                break;
                                }
        }                               /* for switch */
        }                               /* for while */
                                        /* processing last end statement */
        i = lookup("end");
        if ((entry[i].word) != NULL)
                ++entry[i].word_cnt;
                else
                {
                ins_tbl(lex_text,i);
                entry[i].type = 'I';
                }
        i = lookup("End");
        if ((entry[i].word) != NULL)
                ++entry[i].word_cnt;
                else
                {
                ins_tbl(lex_text,i);
                entry[i].type = 'I';
                }
        i = lookup("END");
        if ((entry[i].word) != NULL)
                ++entry[i].word_cnt;
                else
                {
                ins_tbl(lex_text,i);
                entry[i].type = 'I';
                }
        ++cnt_newline;
}

/*****************************************************************************/
/*
        This procedure will use hash to give a table index.  If that
        entry in the table's name matches the name given, the procedure
        returns the index.  If the name does not match the index is
        incremented by one until either an empty table entry is found
```

```
        or the entry name is found.  In both cases the index is returned.
*/
lookup(string)
char *string;                                   /* name to look up */
{
int i;                                          /* table index */
        i = hash(string);
        while (entry[i].word != NULL)
                {
                if (strcmp(string,entry[i].word) == 0)
                        return(i);
                i++;
                if (i == TBL_SIZE)
                        {
                        printf("error:lookup:table size too small\n");
                        return(i);
                        }
                }
        return(i);
}

/*****************************************************************************/
/*
        This procedure provides the hash routine to make entries into
        the entry table.  The value returned is used as the index into the
        table.
*/
hash(string)
char *string;                           /* name to enter into the table */
{
int hashval;                            /* a sum used in the hash routine */
        for (hashval = 0; *string != '\0';)
                hashval += *string++;
        return(hashval % TBL_SIZE);
}

/*****************************************************************************/
/*
        This procedure inserts the name in the table using the index i,
        and increments the count for the name by one.
*/
ins_tbl(name,i)
char *name;                             /* name to be inserted in table */
int i;                                  /* table index */
{
char *strsave();                        /* procedure to store a string */
        if (entry[i].word == NULL)
        {
                entry[i].word = strsave(name);
                ++entry[i].word_cnt;
                return(i);
        }
        printf("error:ins_tbl:wrong index");
}

/*****************************************************************************/
```

```
/*
        This procedure stores the string passed to it and returns the
        pointer to the character string.
*/
char *strsave(string)
char *string;                           /* string to be stored */
{
char *ptr;                              /* pointer to the string */
        if ((ptr = (char *) malloc(strlen(string) + 1)) != NULL)
                strcpy(ptr,string);
        else printf("error: strsave out of storage");
        return(ptr);
}

/*****************************************************************************/
/*
        This procedure takes a procedure or a function along with its
        identifying name from the student's program, and passes back
        only the procedure or function identifier name.
*/
char *strip_p_f(string)
char *string;                           /* the complete name */
{
char *name;                             /* the identifier name returned */
char *strsave();                        /* procedure to save a string */
        while ((*string++) != ' ')
                ;
        name = strsave(string);
        return(name);
}

/*****************************************************************************/
/*
        This procedure totals the number of unique operators and the
        total number of operators that appeared in the student's program.
*/
cnt_operators()
{
        oper_u = 0;
        oper_t = 0;
        if (cnt_assignment != 0)
                        {
                        ++oper_u;
                        oper_t += cnt_assignment;
                        }
        if (cnt_mult != 0)
                        {
                        ++oper_u;
                        oper_t += cnt_mult;
                        }
        if (cnt_add != 0)
                        {
                        ++oper_u;
                        oper_t += cnt_add;
                        }
        if (cnt_sub != 0)
```

```
                {
                ++oper_u;
                oper_t += cnt_sub;
                }
        if (cnt_divide != 0)
                {
                ++oper_u;
                oper_t += cnt_divide;
                }
        if (cnt_int_divide != 0)
                {
                ++oper_u;
                oper_t += cnt_int_divide;
                }
        if (cnt_modulus != 0)
                {
                ++oper_u;
                oper_t += cnt_modulus;
                }
        if (cnt_power != 0)
                {
                ++oper_u;
                oper_t += cnt_power;
                }
        if (cnt_unequal != 0)
                {
                ++oper_u;
                oper_t += cnt_unequal;
                }
        if (cnt_eq_less_than != 0)
                {
                ++oper_u;
                oper_t += cnt_eq_less_than;
                }
        if (cnt_eq_grt_than != 0)
                {
                ++oper_u;
                oper_t += cnt_eq_grt_than;
                }
        if (cnt_equal != 0)
                {
                ++oper_u;
                oper_t += cnt_equal;
                }
        if (cnt_less_than != 0)
                {
                ++oper_u;
                oper_t += cnt_less_than;
                }
        if (cnt_grt_than != 0)
                {
                ++oper_u;
                oper_t += cnt_grt_than;
                }
        if (cnt_log_and != 0)
                {
```

```
                        ++oper_u;
                        oper_t += cnt_log_and;
                        }
        if (cnt_log_or != 0)
                        {
                        ++oper_u;
                        oper_t += cnt_log_or;
                        }
        if (cnt_log_not != 0)
                        {
                        ++oper_u;
                        oper_t += cnt_log_not;
                        }
        if (cnt_set_member != 0)
                        {
                        ++oper_u;
                        oper_t += cnt_set_member;
                        }
}

/*****************************************************************************/
/*
        This procedure calls the appropriate procedure to update the number
        of unique and total number of declarations, control structures, and
        input/output reserved words, that appear in the student's program.
*/
cnt_res_wd()
{
        dec_u = 0;
        dec_t = 0;
        cont_str_u = 0;
        cont_str_t = 0;
        i_o_u = 0;
        i_o_t = 0;
        res_wd_u = 0;
        res_wd_t = 0;
        cnt_declaration("array");
        cnt_declaration("Array");
        cnt_declaration("ARRAY");
        cnt_declaration("boolean");
        cnt_declaration("Boolean");
        cnt_declaration("BOOLEAN");
        cnt_declaration("char");
        cnt_declaration("Char");
        cnt_declaration("CHAR");
        cnt_declaration("const");
        cnt_declaration("Const");
        cnt_declaration("CONST");
        cnt_declaration("file");
        cnt_declaration("File");
        cnt_declaration("FILE");
        cnt_declaration("integer");
        cnt_declaration("Integer");
        cnt_declaration("INTEGER");
        cnt_declaration("label");
        cnt_declaration("Label");
```

```
        cnt_declaration("LABEL");
        cnt_declaration("packed");
        cnt_declaration("Packed");
        cnt_declaration("PACKED");
        cnt_declaration("program");
        cnt_declaration("Program");
        cnt_declaration("PROGRAM");
        cnt_declaration("real");
        cnt_declaration("Real");
        cnt_declaration("REAL");
        cnt_declaration("record");
        cnt_declaration("Record");
        cnt_declaration("RECORD");
        cnt_declaration("set");
        cnt_declaration("Set");
        cnt_declaration("SET");
        cnt_declaration("type");
        cnt_declaration("Type");
        cnt_declaration("TYPE");
        cnt_declaration("var");
        cnt_declaration("Var");
        cnt_declaration("VAR");
        cnt_declaration("varying");
        cnt_declaration("Varying");
        cnt_declaration("VARYING");
        cnt_structure("begin");
        cnt_structure("Begin");
        cnt_structure("BEGIN");
        cnt_structure("case");
        cnt_structure("Case");
        cnt_structure("CASE");
        cnt_structure("do");
        cnt_structure("Do");
        cnt_structure("DO");
        cnt_structure("downto");
        cnt_structure("Downto");
        cnt_structure("DOWNTO");
        cnt_structure("else");
        cnt_structure("Else");
        cnt_structure("ELSE");
        cnt_structure("end");
        cnt_structure("End");
        cnt_structure("END");
        cnt_structure("for");
        cnt_structure("For");
        cnt_structure("FOR");
        cnt_structure("goto");
        cnt_structure("Goto");
        cnt_structure("GOTO");
        cnt_structure("if");
        cnt_structure("If");
        cnt_structure("IF");
        cnt_structure("of");
        cnt_structure("Of");
        cnt_structure("OF");
        cnt_structure("otherwise");
```

```
        cnt_structure("Otherwise");
        cnt_structure("OTHERWISE");
        cnt_structure("repeat");
        cnt_structure("Repeat");
        cnt_structure("REPEAT");
        cnt_structure("then");
        cnt_structure("Then");
        cnt_structure("THEN");
        cnt_structure("to");
        cnt_structure("To");
        cnt_structure("TO");
        cnt_structure("until");
        cnt_structure("Until");
        cnt_structure("UNTIL");
        cnt_structure("while");
        cnt_structure("While");
        cnt_structure("WHILE");
        cnt_structure("with");
        cnt_structure("With");
        cnt_structure("WITH");
        cnt_i_o("get");
        cnt_i_o("Get");
        cnt_i_o("GET");
        cnt_i_o("put");
        cnt_i_o("Put");
        cnt_i_o("PUT");
        cnt_i_o("read");
        cnt_i_o("Read");
        cnt_i_o("READ");
        cnt_i_o("readln");
        cnt_i_o("Readln");
        cnt_i_o("READLN");
        cnt_i_o("reset");
        cnt_i_o("Reset");
        cnt_i_o("RESET");
        cnt_i_o("rewrite");
        cnt_i_o("Rewrite");
        cnt_i_o("REWRITE");
        cnt_i_o("write");
        cnt_i_o("Write");
        cnt_i_o("WRITE");
        cnt_i_o("writeln");
        cnt_i_o("Writeln");
        cnt_i_o("WRITELN");
}

/*****************************************************************************/
/*
        This procedure processes the declaration reserved words by
        updating the unique and total declaration counts.
*/
cnt_declaration(name)
char *name;
{
int i;                                  /* table index */
int count;                              /* the count for the table entry */
```

```
        i = lookup(name);
        if ((count = entry[i].word_cnt) != 0)
                {
                ++dec_u;
                dec_t += count;
                ++res_wd_u;
                res_wd_t += count;
                }
}

/*****************************************************************************/
/*
        This procedure processes the control structure reserved words by
        updating the unique and total control structure counts.
*/
cnt_structure(name)
char *name;
{
int i;
int count;
        i = lookup(name);
        if ((count = entry[i].word_cnt) != 0)
                {
                ++cont_str_u;
                cont_str_t += count;
                ++res_wd_u;
                res_wd_t += count;
                }
}

/*****************************************************************************/
/*
        This procedure processes the input/output reserved words by
        updating the unique and total input/output counts.
*/
cnt_i_o(name)
char *name;
{
int i;
int count;
        i = lookup(name);
        if ((count = entry[i].word_cnt) != 0)
                {
                ++i_o_u;
                i_o_t += count;
                ++res_wd_u;
                res_wd_t += count;
                }
}

/*****************************************************************************/
/*
        This procedure reads through the table processing the appropriate
        types 'I', 'N', 'P', 'F'. The unique and total counts for each
        are updated.
*/
```

```
cnt_id()
{
int i;
char tbl_type;
        proc_u = 0;
        proc_t = 0;
        fnc_u = 0;
        fnc_t = 0;
        num_cons_u = 0;
        num_cons_t = 0;
        i_type_u = 0;
        i_type_t = 0;
        id_u = 0;
        id_t = 0;
        i = 0;
        while (i < TBL_SIZE)
                {
                tbl_type = entry[i].type;
                if (tbl_type == 'I')
                        {
                        ++i_type_u;
                        i_type_t += entry[i].word_cnt;
                        }
                if (tbl_type == 'N')
                        {
                        ++num_cons_u;
                        num_cons_t += entry[i].word_cnt;
                        }
                if (tbl_type == 'P')
                        {
                        ++proc_u;
                        proc_t += entry[i].word_cnt;
                        }
                if (tbl_type == 'F')
                        {
                        ++fnc_u;
                        fnc_t += entry[i].word_cnt;
                        }
                i++;
                }
                                        /* calculate actual identifier */
        id_u = ((i_type_u) - (res_wd_u));
        id_t = ((i_type_t) - (res_wd_t));
}

/*****************************************************************************/
/*
        This procedure produces the style grade for the student
*/
grade()
{
int i;
float inter_score;                      /* individual parameter scores */
float fact;                             /* multiplier for style score */
                                        /* reading comment count */
        if ((fp = fopen(commfile,"r")) == NULL)
```

```
                printf("error: comment file can not be opened for reading");
        else
                {
                fscanf(fp, "%d", &cnt_comment);
                fclose(fp);
                }
                                        /* definitions of parameters to be
                                           measured - may need to be updated
                                           if changes made to what is checked */
param_id[0] = oper_t;
param_id[1] = dec_t;
param_id[2] = cont_str_t;
param_id[3] = i_o_t;
param_id[4] = proc_t;
param_id[5] = fnc_t;
param_id[6] = num_cons_t;
param_id[7] = id_t;
param_id[8] = cnt_comment;
                                        /* calculation of style mark */
        tot_score = 0;
        for (i = 0; i < NO_ANALYSIS; i++)
                {
                /*
                 * the variable dmy is required becuase the BSD 4.2
                 * compiler doesn't typecast properly in if statments
                 */
                int dmy;
                inter_score = 0;
                dmy = param_id[i];
                if ((low_max[i] <= dmy) &&
                    (dmy <= high_max[i]))
                        inter_score += max_score[i];
                else if ((low_no[i] <= dmy) &&
                     (dmy < low_max[i]))
                        {
                        fact = ((param_id[i] - low_no[i])/
                                (low_max[i] - low_no[i]));
                        inter_score += max_score[i]*fact;
                        }
                else if ((high_max[i] < dmy) &&
                     (dmy <= high_no[i]))
                        {
                        fact = ((high_no[i] - param_id[i])/
                                (high_no[i] - high_max[i]));
                        inter_score += max_score[i]*fact;
                        }
                style_score[i] = inter_score;
                tot_score += inter_score;
                }
}
/*****************************************************************************/
/*
        This procedure produces the output for a student's program style.
*/
print_out()
{
```

```
int i;
float save;
printf("\n\n\n                                        ");
printf("STUDENT'S STYLE INFORMATION\n\n\n");
printf("                                number   unique/     ");
printf("total     total/    style\n");
printf("                                unique    lines      ");
printf("          lines      mark\n\n\n");
        i = 0;
        printf("%s", style_par[i]);
        save = ((float)(oper_u)/(float)(cnt_newline));
        printf("%9d %10.2f", oper_u, save);
        save = ((float)oper_t/(float)cnt_newline);
        printf("%9d %10.2f", oper_t, save);
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("%s", style_par[i]);
        save = ((float)(dec_u)/(float)(cnt_newline));
        printf("%9d %10.2f", dec_u, save);
        save = ((float)dec_t/(float)cnt_newline);
        printf("%9d %10.2f", dec_t, save);
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("%s", style_par[i]);
        save = ((float)(cont_str_u)/(float)(cnt_newline));
        printf("%9d %10.2f", cont_str_u, save);
        save = ((float)cont_str_t/(float)cnt_newline);
        printf("%9d %10.2f", cont_str_t, save);
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("%s", style_par[i]);
        save = ((float)(i_o_u)/(float)(cnt_newline));
        printf("%9d %10.2f", i_o_u, save);
        save = ((float)i_o_t/(float)cnt_newline);
        printf("%9d %10.2f", i_o_t, save);
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("%s", style_par[i]);
        save = ((float)(proc_u)/(float)(cnt_newline));
        printf("%9d %10.2f", proc_u, save);
        save = ((float)proc_t/(float)cnt_newline);
        printf("%9d %10.2f", proc_t, save);
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("%s", style_par[i]);
        save = ((float)(fnc_u)/(float)(cnt_newline));
        printf("%9d %10.2f", fnc_u, save);
        save = ((float)fnc_t/(float)cnt_newline);
        printf("%9d %10.2f", fnc_t, save);
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("%s", style_par[i]);
        save = ((float)(num_cons_u)/(float)(cnt_newline));
        printf("%9d %10.2f", num_cons_u, save);
        save = ((float)num_cons_t/(float)cnt_newline);
        printf("%9d %10.2f", num_cons_t, save);
```

```
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("%s", style_par[i]);
        save = ((float)(id_u)/(float)(cnt_newline));
        printf("%9d %10.2f", id_u, save);
        save = ((float)id_t/(float)cnt_newline);
        printf("%9d %10.2f", id_t, save);
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("%s", style_par[i]);
        printf("                    ");
        save = ((float)cnt_comment/(float)cnt_newline);
        printf("%9d %10.2f", cnt_comment, save);
        printf("%10.2f\n\n\n\n", style_score[i]);
        i++;
        printf("    number of code lines            ");
        printf("%14d\n\n\n\n\n", cnt_newline);
        printf("    total style score       ");
        printf("                        ");
        printf("%20.2f\n\n",tot_score);
}

/*************************************************************************/
/*
        This procedure will output the plagiarism information
*/
plagiarism()
{
char *plagfile = "plagfile";
FILE *fopen(), *fp;

        if ((fp = fopen(plagfile,"a")) == NULL)
                {
                printf("error: plagiarism file can not be opened\n");
                }
        else
                {
                fprintf(fp, "%3d %3d", oper_t, dec_t);
                fprintf(fp, " %3d %3d", cont_str_t, i_o_t);
                fprintf(fp, " %3d %3d", proc_t, fnc_t);
                fprintf(fp, " %3d %3d\n", num_cons_t, id_t);
                fclose(fp);
                }
}
/*************************************************************************/
```

```
/*****************************************************************************/
/*      filename:       plag.c                                               */
/*      author:         Kathleen Muller                                      */
/*      date:           May, 1984                                            */
/*      purpose:        This C program reads in a file which contains        */
/*                      an entry for each student.  The entry includes       */
/*                      a string of values output by style.c.  These         */
/*                      values are compared on a student by student          */
/*                      basis calculating 2 values for each pair.            */
/*                      The two values are:                                  */
/*                      DIFFERENCE - the sum total of the weighted           */
/*                              absolute value of the difference between     */
/*                              two student's criteria.                      */
/*                      SIMILARITY - a total of the number of                */
/*                              criteria with a zero difference              */
/*      input files:    plagfile                                             */
/*                              created by : style.c                         */
/*                              input      : plagiarism info                 */
/*                                           for each student               */
/*      output files:   standard output                                      */
/*                              output     : difference and                  */
/*                                           similarity for a               */
/*                                           pair of students               */
/*      procedures:     read_input                                           */
/*                              called by  : main                            */
/*                      calculate                                            */
/*                              called by  : main                            */
/*                      print_out                                            */
/*                              called by  : main                            */
/*                                                                           */
/*****************************************************************************/
#include <stdio.h>

#define NO_ANALYSIS 9                   /* number of parameters analized */
#define NO_STUDENT 100                  /* number of students in system */
#define NOT_EOF 99                      /* beginning not EOF token */
#define MIN_DIFF 60                     /* min difference value allowed */
#define MIN_SIM 4                       /* min similarity value allowed */

struct plag                             /* table of all student's
                                           plagiarism info */
{
        int plag_cnt[NO_ANALYSIS];
};

static struct plag entry_in[NO_STUDENT];

struct plag_d_out                       /* table of pairs of students
                                           difference values */
{
        int plag_diff[NO_STUDENT];
};

static struct plag_d_out entry_diff[NO_STUDENT]; /* table of pairs of
                                                    students sim values */
```

```
struct plag_s_out
{
        int plag_sim[NO_STUDENT];
};

static struct plag_s_out entry_sim[NO_STUDENT];

                                        /* info to update for plagiarism
                                                        output */
static int imp[NO_ANALYSIS]  = {  1,  1,  1,  1,  1,  1,  1,  1,  1};
int filesize;                           /* file size - # of entries */

/*************************************************************************/
/*
        This is the main procedure which controls the whole program
*/
main()
{
read_input();
calculate();
print_out();
}

/*************************************************************************/
/*
        This procedure reads the plagiarism file into a table.
*/
read_input()
{
int i;                                  /* student index */
int j;                                  /* parameter index */
int ret;
char *plagfile = "plagfile";
FILE *fopen(), *fp;
        ret = NOT_EOF;
        i = 1;
        if ((fp = fopen(plagfile,"r")) == NULL)
                printf("error:plariarism file can not be opened");
        while (( i <= NO_STUDENT) && (ret != EOF))
                {
                j = 1;
                while ((j < NO_ANALYSIS) && (ret != EOF))
                        {
                        (ret = fscanf(fp, "%d", &entry_in[i].plag_cnt[j]));
                        j++;
                        }
                i++;
                }
        filesize = (i-2);
        fclose(fp);
}

/*************************************************************************/
/*
        This procedure calculates the difference and similarity information.
*/
```

```
calculate()
{
int i;                                  /* 1st student index */
int j;                                  /* 2nd student index */
int k;                                  /* parameter index */
int diff;                               /* difference between two parameters */
int abs_diff;                           /* absolute value of diff */
                                        /* initialize diff and sim tables */
        i = 1;
        while (i <= filesize)
                {
                j = i+1;
                while (j <= filesize)
                        {
                        entry_diff[i].plag_diff[j] = 0;
                        entry_sim[i].plag_sim[j] = 0;
                        j++;
                        }
                i++;
                }
                                        /* calculate diff and sim values */
        i = 1;
        while (i <= filesize)
                {
                j = i+1;
                while (j <= filesize)
                        {
                        k = 1;
                        while (k < NO_ANALYSIS)
                                {
                                diff = ((entry_in[i].plag_cnt[k]) -
                                        (entry_in[j].plag_cnt[k]));
                                abs_diff = abs(diff);
                                entry_diff[i].plag_diff[j] += abs_diff*imp[k];
                                if (abs_diff == 0)
                                        ++(entry_sim[i].plag_sim[j]);
                                k++;
                                }
                        j++;
                        }
                i++;
                }
}

/*************************************************************************/
/*
        This procedure outputs the plagiarism results for the
        difference and similarity values.
*/
print_out()
{
int i;                                  /* 1st student index */
int j;                                  /* 2nd student index */
int k;                                  /* parameter index */
int dif;                                /* diff value for the students */
int sim;                                /* sim value for the students */
```

```
        printf("\n\n\n                PLAGIARISM REPORT\n\n\n");
        printf("      STUDENTS     DIFFERENCE          SIMILARITY\n\n");
        i = 1;
        while (i <= filesize)
                {
                j = i+1;
                while (j <= filesize)
                        {
                        if ((dif = entry_diff[i].plag_diff[j]) < MIN_DIFF)
                                printf("      %3d %3d     %7d    **",i ,j ,dif);
                        else
                                printf("      %3d %3d     %7d     ",i ,j ,dif);
                        if ((sim = entry_sim[i].plag_sim[j]) > MIN_SIM)
                                printf("       %7d    **\n",sim);
                        else
                                printf("       %7d     \n",sim);
                        j++;
                        }
                i++;
                }
                printf("\n\n\n");
                printf("       ");
                printf(" * - represents values below the minimum allowed\n");
                printf("       ");
                printf("** - represents values above the maximun allowed\n");
}
/*************************************************************************/
```