STYLE: An Automated Program Style Analyzer for Pascal

Al Lake
Curtis Cook
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

88-60-21

STYLE: An Automated Program Style Analyzer for Pascal

Al Lake and Curtis Cook
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

INTRODUCTION

Programming style plays an important role in program understanding and maintenance. Studies [Par83] have shown that as much as one-half of a maintenance programmer's time is spent in activities related to understanding the program. Program understanding is also important for testing and debugging. Programming style embellishes the readability of a program and hence improves its understandability.

Little time is spent on programming style in programming textbooks and in introductory programming courses which concentrate on teaching the syntax of a particular programming language and the use that programming language in solving problems. There is little space in the book and little class time for other than a superficial treatment of programming style. Programming assignments are graded on how well the program solves the problem; that is, the cleverness or efficiency of the algorithm. A small part, if any, of the program grade is based on style and readability.

Difficulty, consistency, subjectivity, and time are the major reasons program style is not graded more. To assist in this task two types of automated style grading programs have been developed. The first type gives a style score between 0 and 100. Its score is based on a set of style factors and is a weighted sum of the factors. The factors, the computation of the value for each factor, and the weights of each factor in the sum are set by the developer who based them on this intuition and experience. The second type of style grading program computes values for a battery of measures and leaves their interpretation to the user. The measures in the battery are set by the developer and no guidelines about the relative contribution of the factors is given.

STYLE, the Pascal style analyzer described in this paper, does not assign a style grade or score to a program. Instead it outputs meaningful and nontechnical messages about the programming style for each module. It is modeled after an writing teacher who writes constructive comments on a student compositions. Hence the goal of *STYLE* is to assist a student in developing an awareness of style and in improving his or her programming style. *STYLE* analyzes a Pascal program and outputs meaningful, non-technical messages about any programming style deficiencies it finds in the program. Comments from students who have used *STYLE* have been very positive.

In the next section we describe programming style analyzers and the style principles on which *STYLE* was based. The user interface and an example of how to use *STYLE* are given the sections three and four. A description of the implementation of *STYLE* is given in section five.

PROGRAMMING STYLE AND STYLE ANALYZERS

Programming style is an elusive yet intuitive quality of a program. It is difficult to define programming style and defining 'good' style that will produce more readable programs is even more difficult. A common approach to programming style is to formulate a set of principles or rules and use them as a yardstick to measure the style of the program. However, the principles or rules are subjective and in many instances difficult to quantify. A number of books and articles present rules for good programming style [Ker74, Led75], as well as rules for particular languages (Pascal [Ree82, Mee83], FORTRAN [Red86], C [Ber85]).

Even though there is no clear definition of programming style, the intent of programming style is to "produce code that is clear and easily understood without sacrificing performance" [Oma87]. Therefore, from a programmer's point-of-view, we define programming style as the effective structuring and arrangement of programs to increase readability and maintainability without degrading performance.

Several automated programming style analyzers/graders have been developed that attempt to measure style. They calculate a single style score between 0 and 100 that is a weighted sum of the counts of various program characteristics. Automated programming style analyzers have been developed for Pascal [Ree82, Mee83], FORTRAN [Red86], and C [Ber85]. Rees' Pascal source code grader [Ree82] was based on ten factors: average line length, comments, indentation, blank lines, embedded spaces, modularity, variety of reserved words, identifier length, variety of identifier names, and the use of labels and GOTOs. Each of the ten factors was quantified and assigned a weight. A trigger-point scoring scheme was used to quantify each factor. In this scheme an interval is established for each factor. If the factor is within the interval a linear interpolation scheme is used to calculate its value. Its value is zero if it is outside the interval. The style factors were selected on an intuitive basis and experience. The weights and trigger-points were selected by adjusting them until the analyzer awarded "A" grades to good programs. Rosenthal [Ros83] and Meekings' [Mee83] Pascal published style checkers based on the same style factors as Rees; however, the way they calculated the factors was slightly different and they omitted the "variety of identifiers" factor.

Berry and Meekings [Ber85] modified Meekings' style analyzer for C. They added a count of the included files and the "percentage of constant definitions" and slightly modified the manner in which the other factors were calculated. Redish and Smyth [Red86] used 33 factors in their FORTRAN77 style analyzer. Their 33 factors can be grouped into categories: commenting (4), indentation (1), block sizes (2), statement labels and formats (7), counts of names and statements (6), array declarations (2), control flow and nesting measures (7), blank lines (1), operator count (1), operand count (1), and parametrization (1). Their AUTOMARK program uses the trigger-point scheme of Rees for each factor. The style score is the weighted sum of the factors.

All of the style graders compute a single style score based on a weighted sum of subjectively (intuition and experience) selected set of factors (e.g. program characteristics), factor weights and trigger-points for each factor. With one minor exception they provide no non-technical feedback, justification, or guidance to the user about the style factors, weights, or trigger-points selected. The one exception is the AUTOMARK and ASSESS programs [Red86] for FORTRAN. AUTOMARK output include a brief semi-technical description of each factor. The ASSESS program provides a Low-Average-High evaluation for 10 factors and some specific comments on indentation, commenting, and label usage. It is interesting to note that although AUTOMARK uses 33 factors, their FORTRAN syntax checker actually computes 376 measurements. The authors state that they expect this set to evolve to about 100. They also hope to "validate" various sets of factors in the future.

Our programming style analyzer, *STYLE* , does not assign a grade or give a battery of numerical metrics to the user. Instead it analyzes each module and outputs descriptive non-technical messages about any style deficiencies it found or one of several positive congratulatory messages if it found no deficiencies. The messages are provided to the user in a non-threatening manner, much like an English teacher writing comments on a student's paper. Hence running our style analyzer is like having an expert evaluate the program code and provide comments about the style.

Our approach to quantifying program style was to first formulate widely accepted and general principles of style that include all of the commonly accepted programming style guidelines. We adopted principles based on six "desirable qualities" of style in Redish and Smyth [Red86]. The six qualities are defined as:

1. Economy - the careful or thrifty measures taken to provide the code in as concise a manner as is possible and practical.
2. Modularity - to regulate the standard structural component as a unit of measurement of program source code.
3. Simplicity - the state or quality of being simple, the absence of complexity, intricacy, or artificiality.
4. Structure - the organization of elements, parts, or constituents in a complex entity.
5. Documentation - supporting references explaining the process of the program, the degree of self-descriptiveness of an application.
6. Layout - the arrangement, plan or formatting of the program.

These principles form the framework for our programming style rules. Rather than grouping all the program characteristics we could compute or think of under the style principles, we listed all of the applicable programming style rules from the most popular books on programming style [Ker78, Led75] under each principle. These rules provide more detailed information about the principles and the basis for the meaningful comments output to the user.

The last step in our approach was to quantify each of the style rules through measures of program characteristics. Because of the nature of these rules our measurements were rated as either accurately quantified, estimated, or unable to quantify. For example one part of an accurate quantification of the rule "Avoid superfluous actions or variables in the program" [Ker78] is to determine whether every variable declared is used in the program. The rule "Use meaningful variables names" [Ker78] can be estimated by average length of variable names and the rule " Use a simple or straightforward algorithm" [Ker78] cannot be quantified. Only those rules rated as accurate or estimated were considered for implementation. A more complete description of the principles, the rules, and the quantification of the rules is given in Appendix A.

Through our approach we tried to be as objective as possible. We did not want our selection of style factors to be overly influenced by what program characteristic measurements were easily obtainable from the program. Since our style analyzer was to output meaningful messages, we wanted it to be based on a set of well established and accepted principles of programming style which would form the basis for our messages. In addition, our style analyzer would be based on programming language independent concepts.

USER INTERFACE

The user interface for *STYLE* is the desktop and uses the Apple™ Macintosh™ menu bar. See Figure 1 below. This figure shows all of the menus of the application extended.

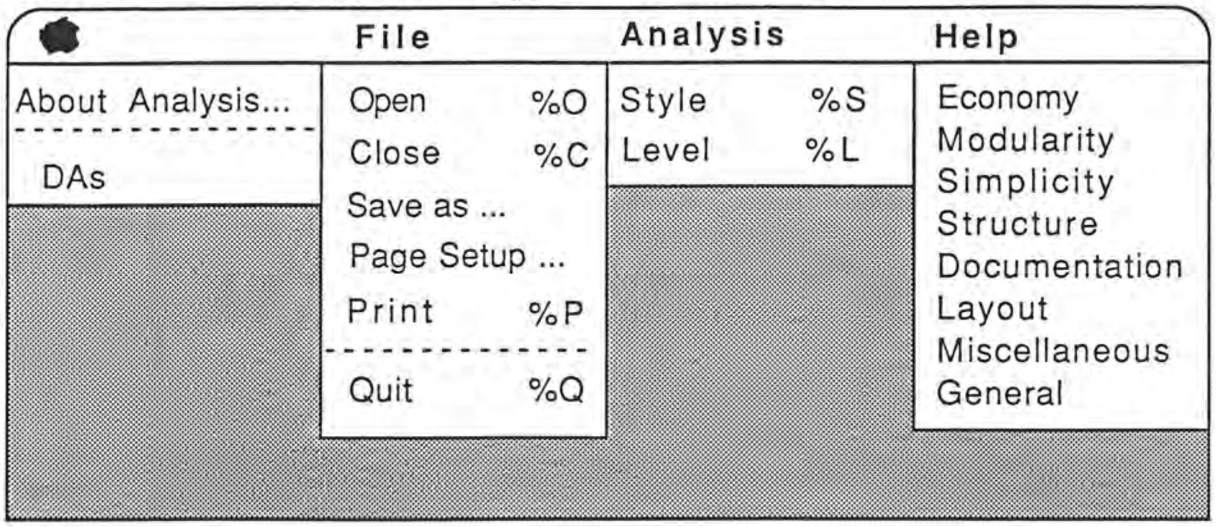
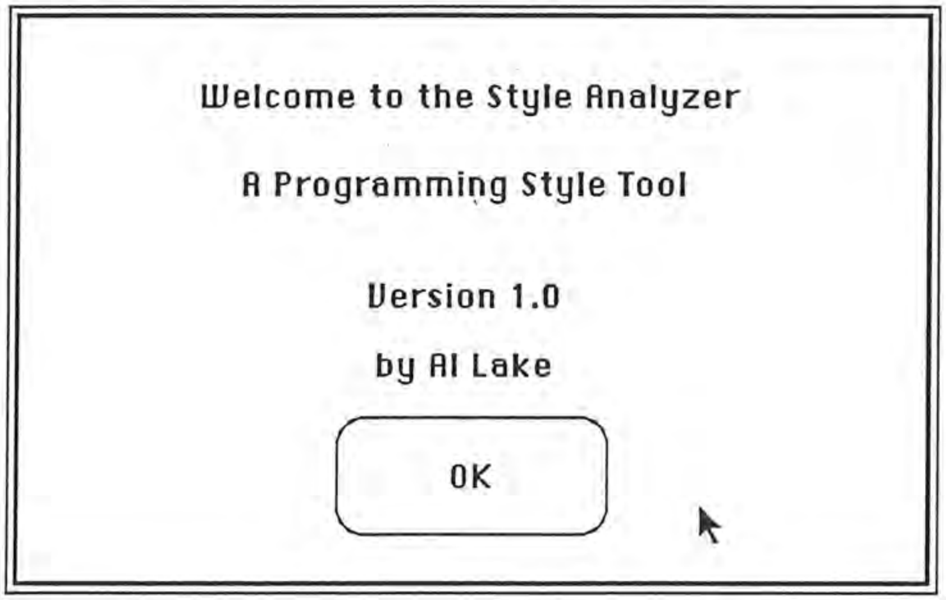


Figure 1 Style Desktop

The **About Analysis** provides the author's name and version number of *STYLE*, and is shown below.


Welcome to the Style Analyzer

A Programming Style Tool

Version 1.0

by Al Lake

Figure 2. About Analysis...

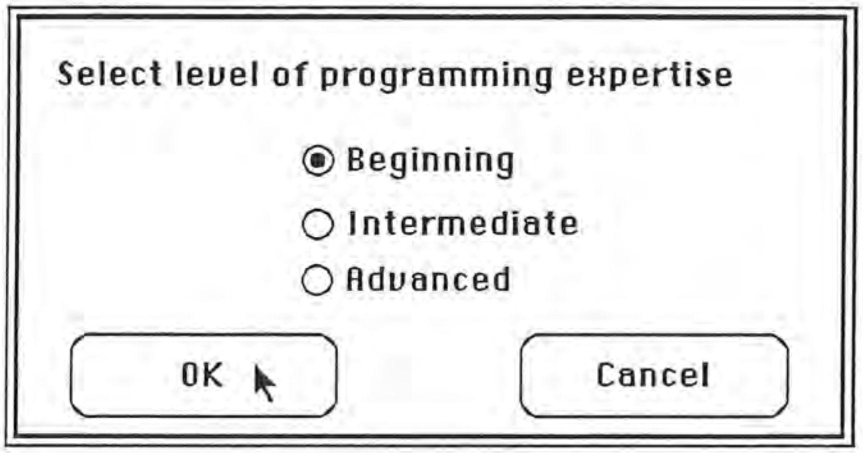
File provides all of the file handling operations:

- **Open** - displays all files of types MacPascal™ and LightSpeed Pascal™, so that one can be selected.
- **Close** - closes the current work file.
- **Save as...** - saves the style analysis output to a text report file of TeachText format.
- **Page Setup** - performs page setup.
- **Print** - prints the style analysis report on the selected printer.
- **Quit** - quits operation of *STYLE.*

With the **Analysis** menu the user can set the skill level (beginner, intermediate, or expert) for the analysis or invoke the analysis.

Style - Performs a style analysis of the selected program file.

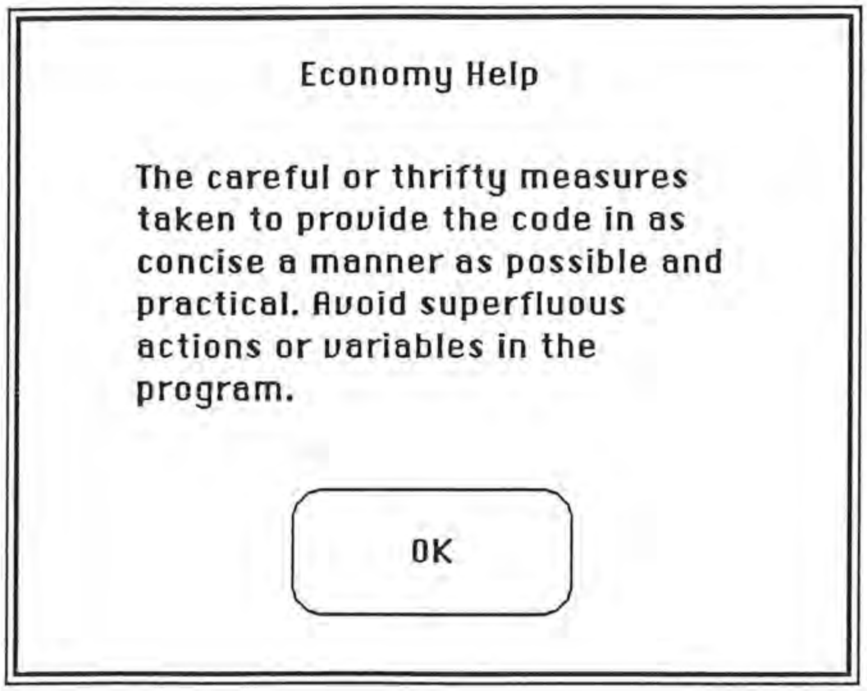
Level - Sets the user expertise level: either beginning, intermediate, or advanced. This level will determine the acceptable range of values for measuring. The assumption is that beginning programmers do not have programming skills as developed as advanced programmers and as such cannot manage the greater levels of nesting, complexity and other problems associated with advanced programming problems, so Beginning level will generate more errors than Advanced level.


Select level of programming expertise

[✓ Beginning] Intermediate Advanced

Figure 3. Level of Programming Expertise Dialog

Help provides a brief descriptions of the different principles and other information. All **Help** information is displayed in a modal dialog. The Economy Help dialog screen is shown as an example:


Economy Help

The careful or thrifty measures taken to provide the code in as concise a manner as possible and practical. Avoid superfluous actions or variables in the program.

Figure 4. Economy Help Dialog

These dialogs are meant to provide some additional information to the user about the analysis process and the methods used in providing the output.

In all cases the options available to the user at any time are limited to those which can logically be executed. For example, when the user begins execution of the program only the **Open**, **Quit**, and **Help** functions are available. When a file is opened the **Open** option is disabled and the **Close** option is enabled, since only one file can be open at a time. The **Save As...** and **Print** options are not enabled until the analysis is completed, since no analysis data can be saved or printed prior to the input source program being analyzed. The **Page Setup** option is always available.

To open a file for analysis, select from the **File** menu the **Open** option. The following dialog will be displayed, filtering out all but the MacPascal™ and LightSpeed Pascal™ files. No special file names are necessary.

When the file is **Open**ed the program is read into a memory buffer. This allows the disk file to be closed and the program to operate more efficiently.

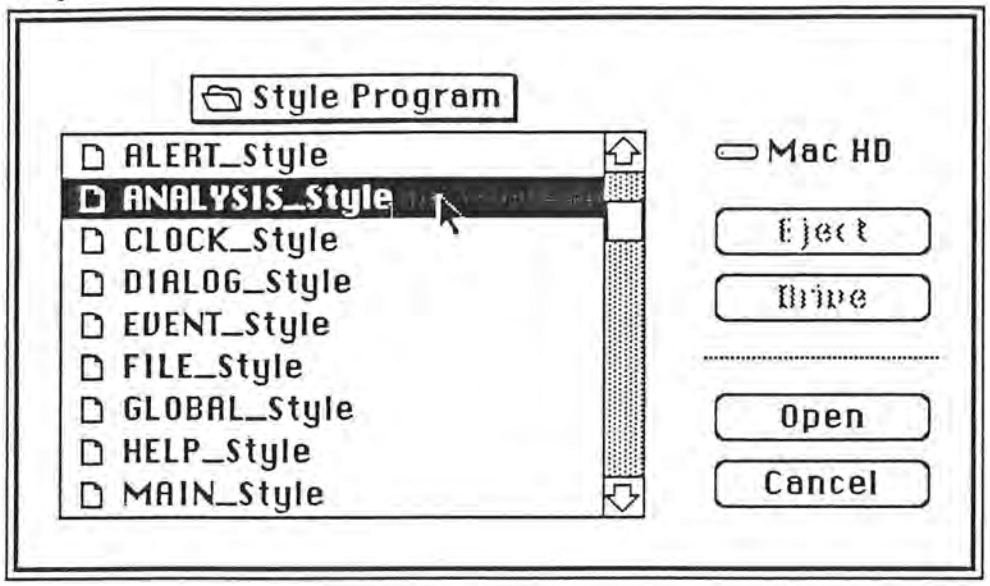


Figure 5. Open Input File Dialog

If the user selects **Save As...** or tries to exit the program without saving the style analysis, a save dialog will be displayed, like the following:

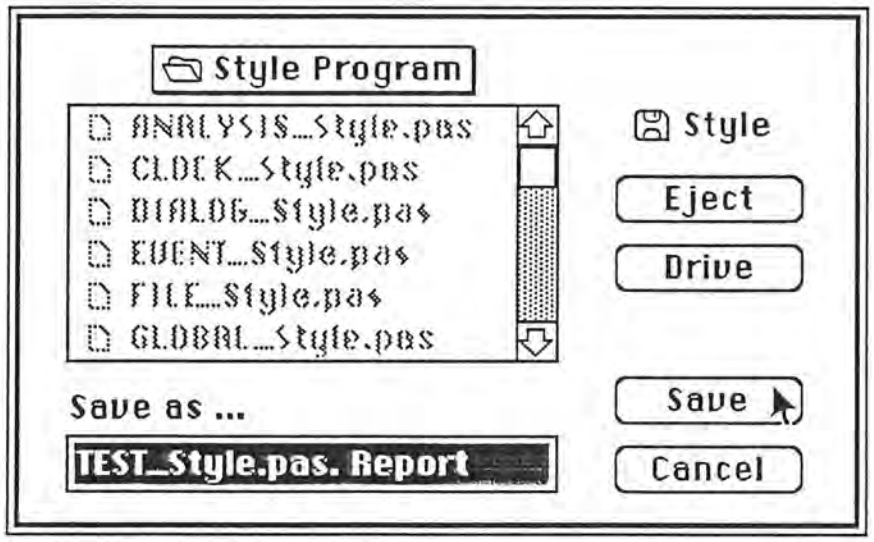
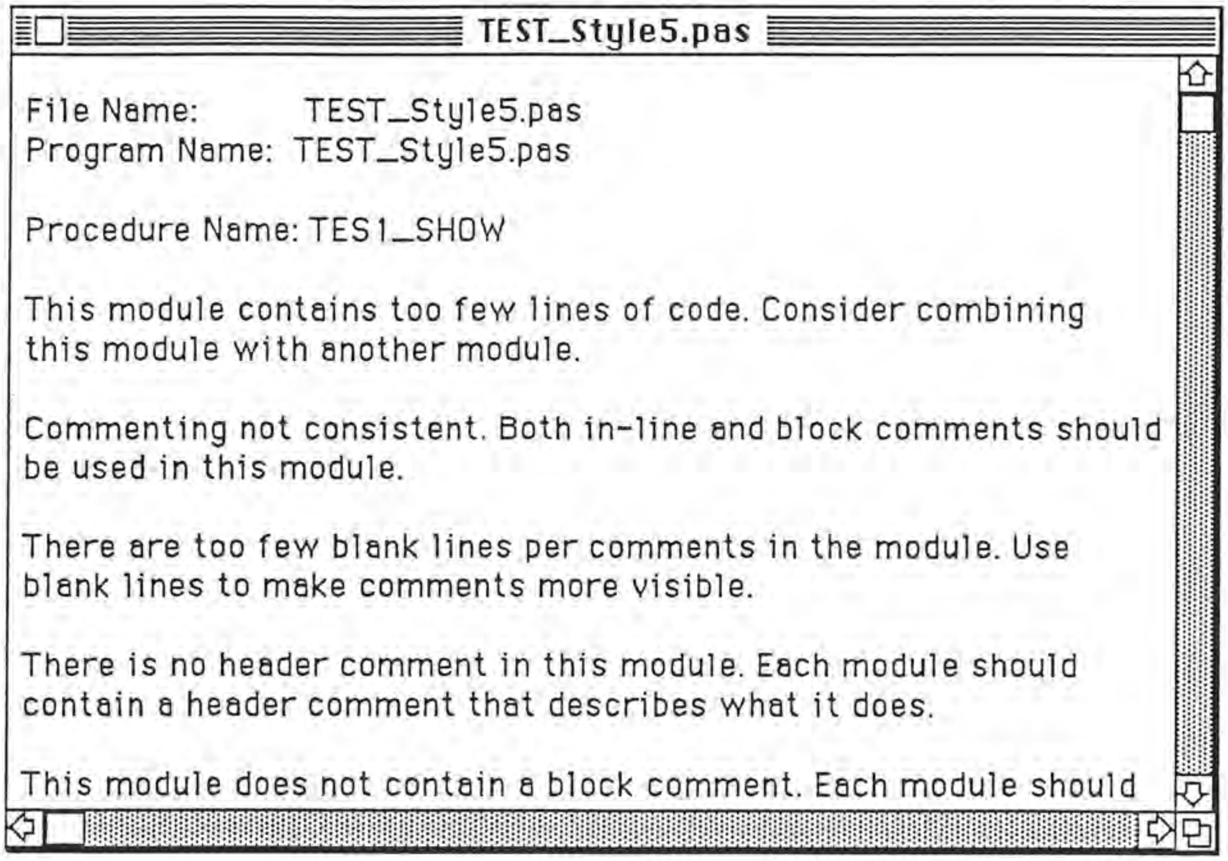


Figure 6. Save Dialog

The program will automatically suffix the file name with ".Report" to help keep track of the relationship between the program file name and the style analysis report file (see figure 7, Sample Window, for an example of a report file).

The information displayed in the analysis window begins with the program name followed by style messages for each of the subprograms in the physical order in which they occur in the program. This is illustrated in the sample window displayed below.



```
File Name:       TEST_Style5.pas
Program Name:  TEST_Style5.pas

Procedure Name: TES1_SHOW

This module contains too few lines of code. Consider combining
this module with another module.

Commenting not consistent. Both in-line and block comments should
be used in this module.

There are too few blank lines per comments in the module. Use
blank lines to make comments more visible.

There is no header comment in this module. Each module should
contain a header comment that describes what it does.

This module does not contain a block comment. Each module should
```

Figure 7. Sample Window

The Sample Window, above, displays a portion of a test file which has been analyzed by the style tool. The user can scroll horizontally or vertically (the messages are defined by the width of the screen so no horizontal scrolling is actually necessary). The information is segmented by module, (procedure or function).

STYLE also includes safeguards so that the user cannot lose work; such as, accidentally quitting without saving the work file. This action causes a **Save As...** menu to be displayed so that the report file can be saved. All menus have default file names and error checking to reduce the number of operating system errors which might occur, such as trying to save a file with no name.

CONCLUSION

STYLE was implemented in LightSpeed Pascal™ for Apple Macintosh™ computers. The program is a prototype since the goal of this project was to test the feasibility of developing a user friendly programming style analyzer that outputs meaningful non-technical comments about the style of a program. In limited class testing students gave *STYLE* high marks as they felt it gave them useful comments about their programming style.

The style tool will run on any Macintosh™ computer with a minimum of 128K, though this memory size will limit the user file to less than 50K. For the best results, the style tool should be used on a Macintosh Plus™ with 1 megabyte of memory.

When run on a Macintosh II™ the analysis window can be resized to fit the larger screen, i.e. *STYLE* will not limit to the user to the smaller Macintosh™ screen size when a larger work space is available. The printout procedure will work for any type of LocalTalk™-compatible network.

For further information about *STYLE* write to the authors at the address above.

REFERENCES

[Ber85] R. E. Berry and B. A. E. Meekings, "A Style Analysis of C Programs", *Communications of the ACM*, vol. 28(1), Jan. 1986, pp. 80-88.

[Ker78] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style.* McGraw-Hill, New York, 1978.

[Led75] H. F. Ledgard. *Programming Proverbs.* Hayden Book Company, Rochelle Park, New Jersey, 1975.

[Mee83] B. A. E. Meekings, "Style analysis of Pascal programs", *ACM SIGPLAN Notices* vol. 18(9), Sept. 1983, pp. 45-54.

[Oma87] P. W. Oman and C. R. Cook, "A Paradigm for Programming Style Research", Technical Report 87-60-7, Computer Science Department, Oregon State University, 1987.

[Par83] G. N. Parikh and G. N. Zvegintzov, Tutorial on Software Maintenance, *IEEE Computer Society Press*, 1983, p. 2.

[Ree82] M. J. Rees, Automatic Assessment Aids for Pascal Programs, *ACM SIGPLAN Notices*, Vol 17 (10), Oct. 1982, pp. 33-42.

[Ros83] D. Rosenthal, in correspondence from the members, *ACM SIGPLAN Notices* Vol. 18 (3), Mar. 1983, pp. 4-5.

APPENDIX A

WHAT *STYLE* IS CHECKING

Listed below are the definitions of the qualities of style in the actual values being quantified.

ECONOMY

Avoid superfluous variables - any variable that does not provide useful results, such as an intermediate variable that does not enhance the readability of the program. Superfluous variables are estimated from the ratio of the total number of variables to the number of executable lines of code.

Avoid overloading variables - the use of a variable name in more than one context. Variable overloading is estimated by counting the number of lines between uses of a variable. If the line count exceeds some constant value, then the variable is 'estimated' as being used for a different context.

Minimize the overall number of variables used - use the least number of variables possible. TOTAL VAR describes the total number of variables used in each module, if this value is greater than some constant, a message is issued.

Avoid unused labels - check for unused labels.

Avoid unused variables - check for unused variables.

Avoid unreferenced procedures and functions - check for any procedures or functions that have been defined, but not referenced.

MODULARITY

Long modules - check for modules with more than n lines of code,say 50, and less than m, say 10, lines of source code.

Module size - using McCabe's Complexity Measure, *V(G)*, check all modules for a complexity measure greater than 10. Count the number of conditional routines or functions, such as IF/DO WHILE/REPEAT/CASE.

More than one logical function in a module - check for functions that perform more than one logical function. This guideline is estimated by checking for I/O and arithmetic functions in the same module or multiple I/O in the same module.

Parameter passing - minimize the number of parameters passed. Count the number of parameters being passed to determine if the number of parameters passed is greater than n.

SIMPLICITY

Write clearly -- don't be too clever and don't sacrifice clarity for efficiency - check for use of simple and straightforward algorithms. One way to quantitatively estimate the clarity of a program is to compare McCabe's Complexity Measure, *V(G)*, to a subjective value, such as 10, for the upper limit.

Parenthesize to avoid ambiguity - check extended lines of code for use of parenthesis. Any line of source code, either an assignment statement or logical function (IF statement), which contains more than n words, or more than m operators should contain parentheses.

Check for the number of operators in an expression to determine the number of parenthesis - there should be one set of parenthesis for every logical operator. Count the number of operators in each logical expression to determine if the number of parenthesis is sufficient.

Avoid unnecessary branches - an IF-THEN-ELSE statement with no executable statement on one of the alternatives. This check will look for empty IF-THEN-ELSE branches.

Avoid unnecessary GOTO's - check for the ratio of GOTO's to the rest of the code. Check for the ratio of GOTO statements to all source code (and total number of GOTO's. If the ratio is greater than 5 percent or the number of GOTO's greater than four for any module then print a message.

Check subprogram nesting - a deeply nested subprogram structure complicates the structure of the module. Count the number of embedded subprograms. There should be no more than four levels of nesting.

Average nested level - the average level of nesting for each LOC should not exceed a value, n. Count the nesting level of each line of code and take a weighted average. Check for an average nesting level greater than n.

Compute the maximum nesting level - find the maximum nesting level of any line in each module. Count the nesting level of each line of code to determine the maximum nesting level.

STRUCTURE

IF-THEN-ELSE statements with a null condition - do not allow null conditions in an IF-THEN-ELSE. Check for a null condition in IF-THEN-ELSE.

Check for ELSE GOTO and ELSE RETURN - control the use of a branch from an else condition and a return from an else condition. Check for a RETURN or GOTO condition in IF-THEN-ELSE.

The use of multiple GOTO's to replace a complex IF-THEN-ELSE - Use IF...ELSE IF...ELSE IF...ELSE... or a CASE statement to implement multi-way branches rather than using GOTO's to construct a logical path around. Check for complex IF-ELSE-IF-ELSE... conditions. Present a comment to the user about replacing the IF-ELSE clauses with a CASE statement.

DOCUMENTATION

Thorough and consistent documentation. This guideline can be estimated by checking for the consistent use of in-line versus block comments between modules. A logical value is returned depicting whether the module uses in-line or block and compared.

Use of a header block of comments after the beginning of a function or procedure - This guideline will only measure the existence of comments at the beginning of the module, it cannot measure the effectiveness of the comments.

Variables are described by comments - Ensure that all variables are properly and thoroughly documented. This guideline can be estimated by measuring the ratio of executable lines of code to comments. If the ratio is less than a percentage n, say 10%, or greater than a percentage m, say 80%, output a message.

Meaningful variable names - Check for meaningful variable names. This guideline is not directly measurable, but an estimate can be achieved by checking for variable names with a word length less than n, say 3, characters or greater than m, say 12, characters.

Effective and adequate comments - Check the estimated ratio of the number of words used in the comments to ensure adequate comments. If the ratio is less than a percentage n, say 10%, or greater than a percentage m, say 80%, output a message.

Don't use excessive comments - Overcommenting is a subjective measurement depending on the expertise of the maintenance programmer and the level of understanding of the program. This guideline is estimated by computing the average number of words in each comment.

LAYOUT

Effective use of programming space, both horizontal and vertical, to assist with program comprehension - The compliance with this guideline is estimated by the ratio of blank lines to comments on the page. If the ratio exceeds 50% a message is displayed.

Compute the average number of comments as an estimate to enhance clarity - This guideline is estimated by comparing the average number of words in comments with the number of executable lines of code.

Concise and effective use of space - Estimated by comparing the ratio of blank lines to the number of total lines.

Header comment - a header comment must be provided directly after the beginning of each program, procedure and function. This guideline monitors the inclusion of comments after the program, procedure or function verbs in the program.

Maximum number of blank lines - The maximum number of consecutive blank lines should not exceed some value, say 10. Check for any modules with more than 10 consecutive blank lines.