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 because they concentrate on teaching the syntax of a particular programming language and the use of that programming language in solving problems. There is little space in textbooks and little time in classes 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 programming style is not given greater emphasis. 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. The style score is based on a set of style factors and is a weighted sum of these factors. The factors, the computation of the value for each factor, and the weights of each factor are set by the developer based on the developer's 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 are given.

STYLE, the Pascal style analyzer described in this paper, does not assign a style score to a program or provide a battery of numbers. *STYLE* outputs meaningful and nontechnical messages about the programming style for each module. *STYLE* is modeled after a writing teacher who writes constructive comments on a student composition. Hence, the goals of *STYLE* are to assist a student in developing an awareness of style and to improve his or her programming style. *STYLE* analyzes a Pascal program and outputs messages about any programming style deficiencies found in the program. Comments from students who have used *STYLE* have been very positive.

In the section entitled Programming Style Analyzers we describe programming style analyzers and the style principles on which *STYLE* was based. The section entitled USER INTERFACE provides a description of the implementation of *STYLE* and examples of the user interface. Our conclusions and references are located in the last section.

PROGRAMMING 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 readable programs is even more difficult. 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.

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 [Ker78, Led75], as well as rules for particular languages such as Pascal [Ree82, Mee83], FORTRAN [Red86], and C [Ber85].

Several automated programming style analyzers/graders have been developed that attempt to measure style by calculating a single style score between 0 and 100. This score is a weighted sum of the counts of various program characteristics.

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 then a linear interpolation scheme is used to calculate its value. The value is zero if the factor is outside the interval. The style factors were selected on the basis of the programmer's intuition 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] published Pascal style checkers based on the same style factors as Rees; however, they calculated the factors differently and 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 are 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 Rees' trigger-point scheme for each factor. The style score is the weighted sum of the factors.

All of these style graders compute a single style score based on a weighted sum of subjectively selected 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 includes 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 or one of several positive congratulatory messages if there are 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 found in the literature. 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.

The next step was to reduce the guidelines to the most basic level which are called "style principles".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 two 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. 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. The rule " Use a simple or straightforward algorithm" [Ker78] cannot be quantified. Only those rules rated as accurate or estimated were considered for implementation.

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. In addition, our style analyzer is based on programming language independent concepts.

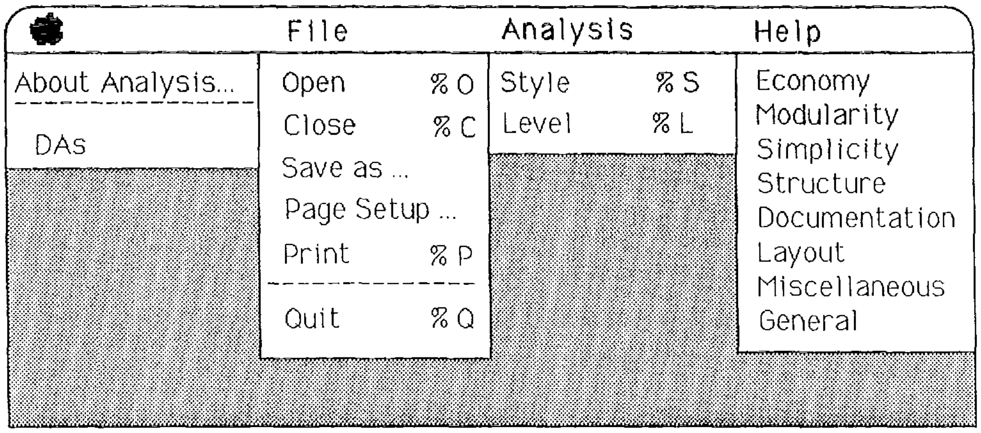
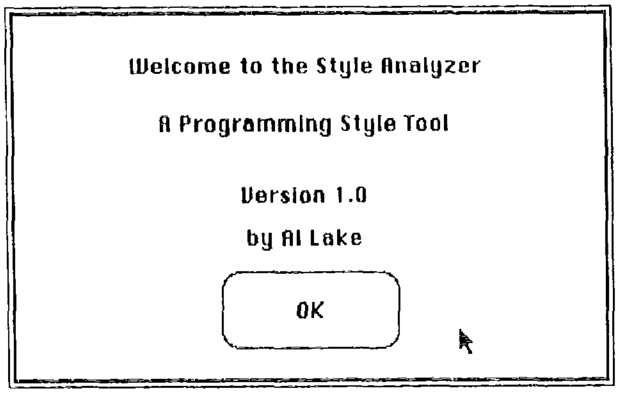


Figure 1 Style Desktop

USER INTERFACE

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

The **About Analysis** provides the author's name and version number of *STYLE*, and is shown in Figure 2


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 MacPascal™ and LightSpeed Pascal™ files for selection.

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.

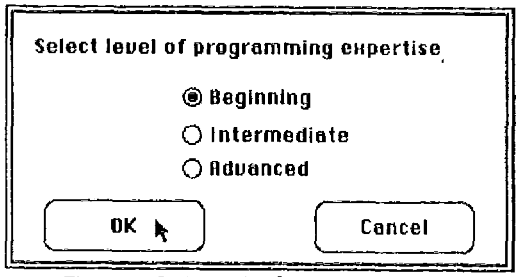
Print - prints the style analysis report on the selected printer.

Quit - quits operation of *STYLE*.

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

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

Level - Sets the user expertise level as either beginning, intermediate, or advanced. The


Select level of programming expertise

[✓ Beginning]

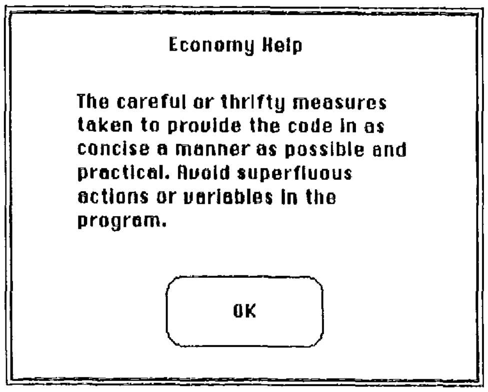
Intermediate

Advanced

Figure 3. Level of Programming Expertise Dialog

level will determine the acceptable range of values for measuring. The assumption is that beginning programmers do not have programming skills which are as well developed as advanced programmers and cannot manage the greater levels of nesting, complexity and other problems associated with advanced programming. As a result, choosing the Beginning level will generate more messages than choosing the Advanced level.

Help provides a brief description of the six different style qualities. All **Help** information is displayed in a modal dialog. The Economy Help dialog screen, shown in figure 4, is an example of the type of dialogs


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

used to provide the user with information about the desirable qualities of style. 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 option, **Page Setup**, is always available to modify the description of the printed page.

To open a file for analysis, select from the **Open** option the **File** menu. The open dialog, shown in figure 5, will be displayed, filtering out all but the MacPascal™ and LightSpeed Pascal™ files. No special file names are necessary.

If the user selects **Save As...** or tries to exit the program without saving the style analysis report, a "save dialog" will be displayed, as in the following figure, giving the user the option to name the file.

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.

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. Figure 7, Report Window, shows an example of the report window. The information displayed in the report window begins with the program name followed by style messages for each of the

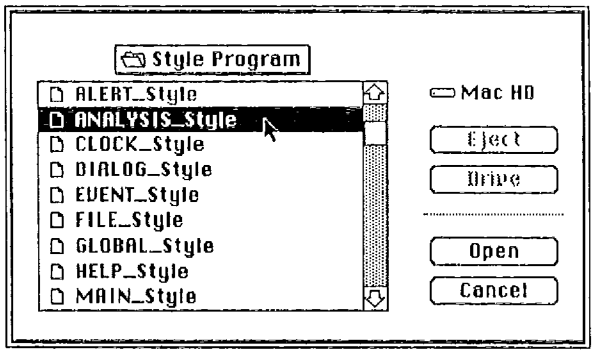


Figure 5. Open Input File Dialog

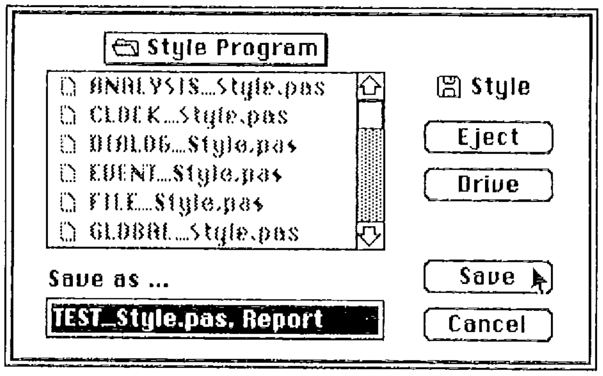
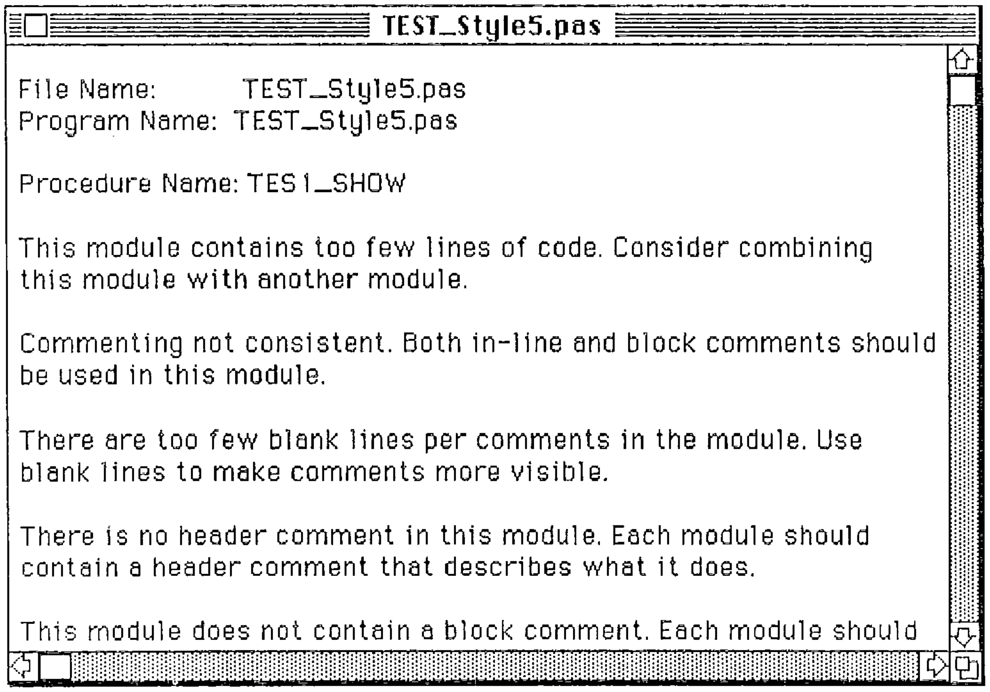
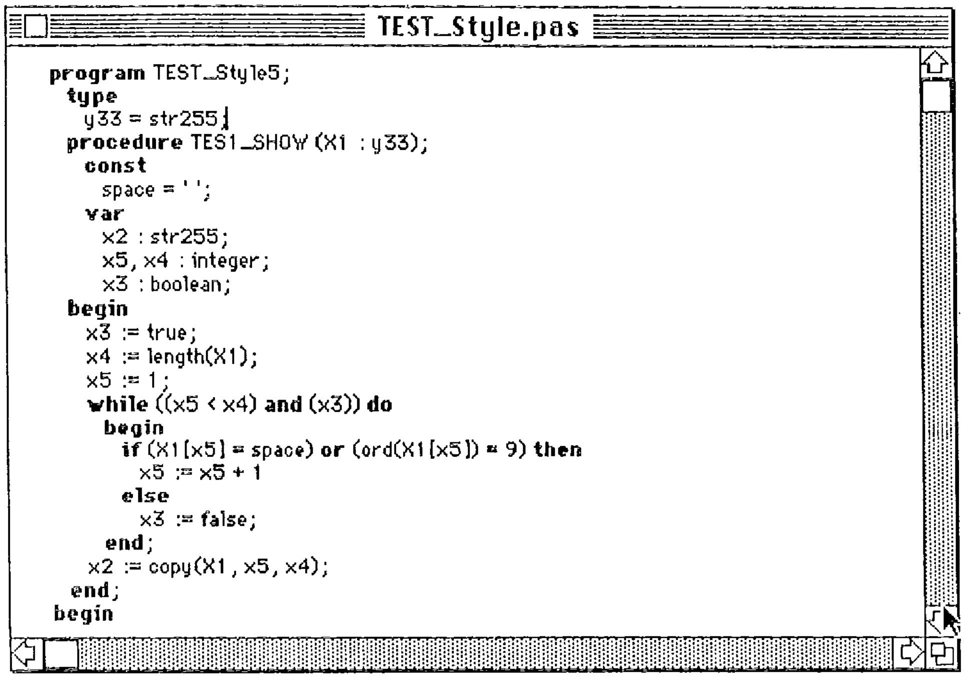


Figure 6. Save Dialog for Saving an Analysis Report File



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

Procedure Name: TEST_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. Report Window



```
program TEST_Style5;
  type
    y33 = str255;
  procedure TES1_SHOW (X1 : y33);
    const
      space = ' ';
    var
      x2 : str255;
      x5, x4 : integer;
      x3 : boolean;
  begin
    x3 := true;
    x4 := length(X1);
    x5 := 1;
    while ((x5 < x4) and (x3)) do
      begin
        if (X1[x5] = space) or (ord(X1[x5]) = 9) then
          x5 := x5 + 1
        else
          x3 := false;
      end;
    x2 := copy(X1, x5, x4);
  end;
begin
```

Figure 8. Sample Program

modules (e.g. procedures or functions) in the physical order in which they occur in the program. The user can scroll horizontally or vertically.

Figure 8, Sample Program, shows the Pascal program example used to generate the messages in Figure 7. As shown, the information is segmented by the program modules.

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

CONCLUSION

STYLE was implemented in LightSpeed Pascal™ for Apple Macintosh™ computers. The goal of this prototype 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, because 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 of memory, though this will limit the user file to less than 50K. For best results, the style tool should be used on a Macintosh Plus™ with 1 megabyte of memory.

When run on a larger screen, such as a Macintosh II™, the analysis window can be resized to fit the larger screen, because *STYLE* does not limit the user to the smaller Macintosh™ screen size when a larger screen work space is available.

The printout procedure will work on any LocalTalk™-compatible network or dedicated printer.

For further information about *STYLE*: An Automated Program Style Analyzer for Pascal, write to the authors at the address above or send e-mail to:

lake@mist.CS.ORST.EDU
or
cook@mist.CS.ORST.EDU

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.