STYLE

Reference Manual

An Automated Program Style Analyzer for Pascal Programmers

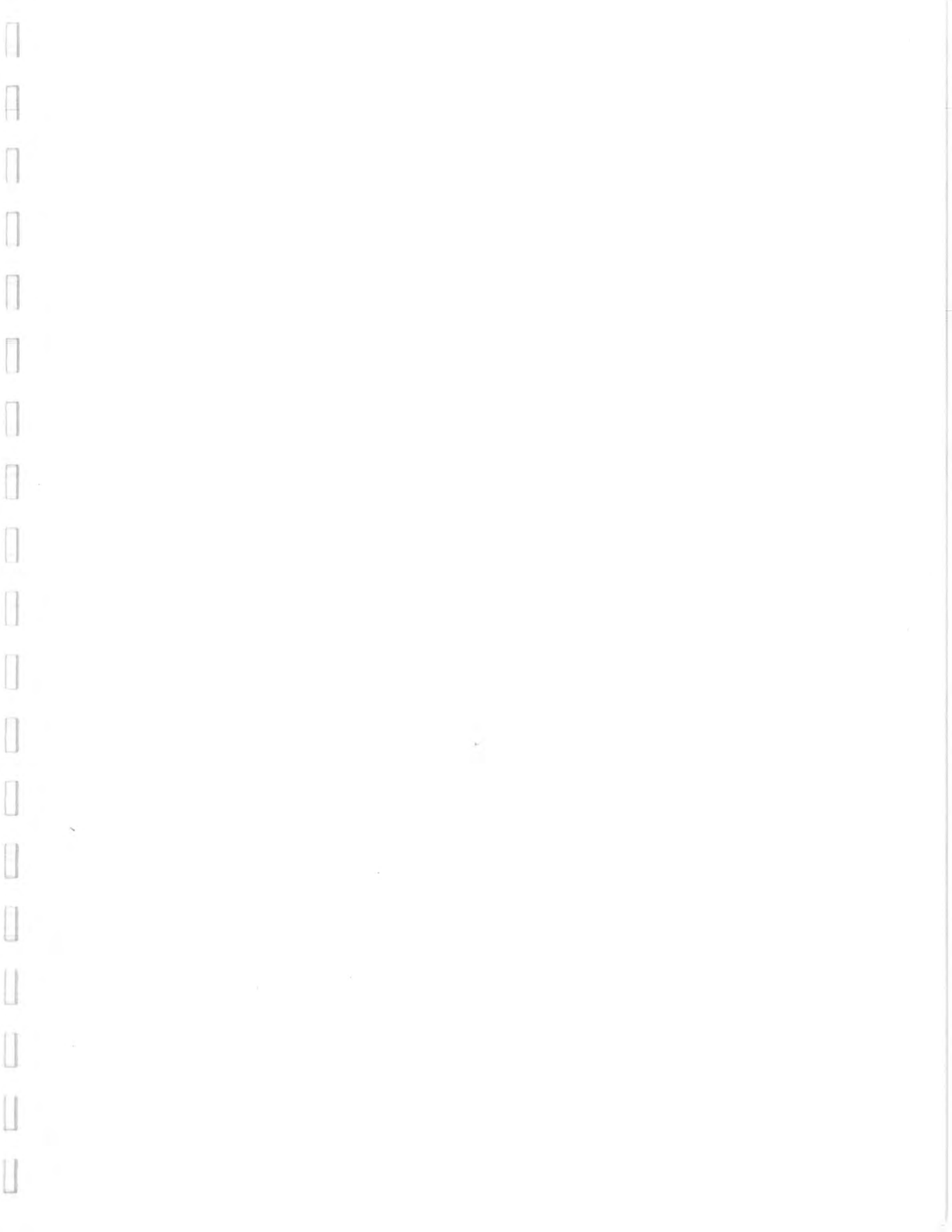## A Tool for Evaluating Programming Style

STYLE

Reference Manual

An Automated Program Style Analyzer for Pascal Programmers

## A Tool for Evaluating Programming Style

CONTENTS

Chapter 1

INTRODUCTION TO STYLE

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.

Understandability (or readability) and maintainability are two of the most important attributes of a program. Putnam estimated [Put80] that the total development work on a software project would only be 40% of the total effort. Therefore, modification and enhancement work was at least 50% of the life-cycle effort, (the systems definition and functional design specifications are the other 10%). The readability of a program directly affects the maintainability of a program. It is estimated that a maintenance programmer spends nearly half of his/her time attempting to understand a program, [Par83], as shown in Figure 1.

**Figure 1.**

Activities of Maintenance Personnel
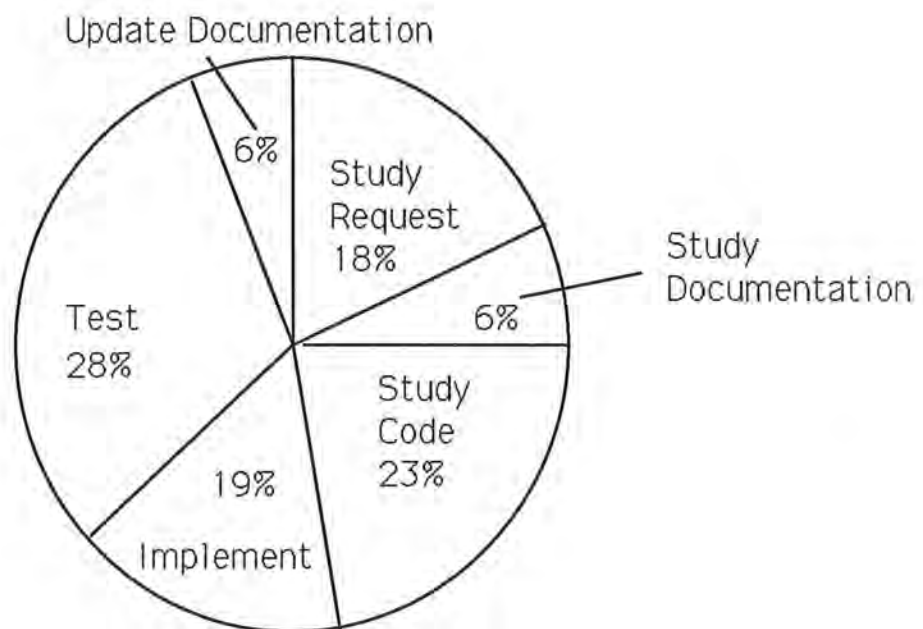
Hence, the easier a program is to read the easier it will be to maintain.

The cost of software maintenance has increased steadily during the past 20 years. The Figure 2 illustrates the past, current, and projected percentage of overall software budget expended on maintenance of existing software [Pre82].

Maintenance Cost
As a Percent of Software Budgets

Figure 2.

Maintenance Cost
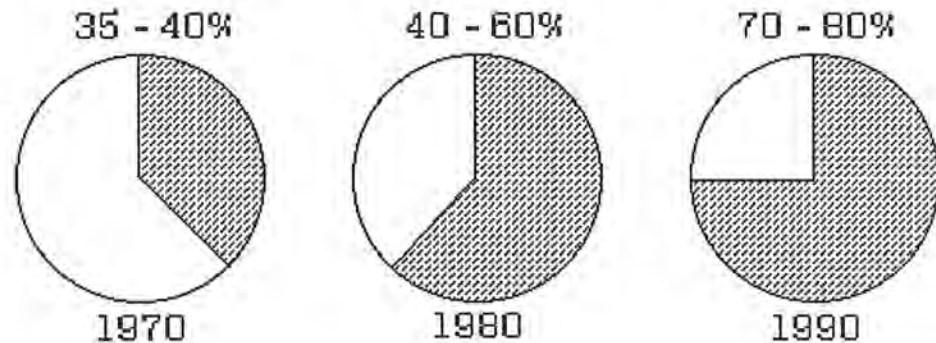
Because of the possibility of modification, the percentage of the software budget which is maintenance cost has steadily increased from about 35-40 percent in 1970 to 70-80 percent in 1980 [Pre82]. "That the issues of testability and maintainability are important is borne out by the fact that we often spend half of the development time in testing and can spend most of our dollars maintaining systems" [McC76]. "The maintainability of an application is more important to the developer than the user" [Lew87]. As such, software should be developed and coded with ease of maintenance and readability in mind. Following good programming style rules improves the readability and maintainability of the program.

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.

2

Chapter 2

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 mor 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 [Ker71, 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.

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 *STYLE* is like having an expert evaluate the program code and provide comments about the style.

The *STYLE* approach to quantifying program style is to first formulate widely accepted and general principles that include all of the commonly accepted programming style guidelines.

The style principles used in *STYLE* are based on six "desirable qualities" of style in Redish and Smyth [Red86].

The six qualities are defined as:

ECONOMY
- Economy - the careful or thrifty measures taken to provide the code in as concise a manner as is possible and practical.

MODULARITY
- Modularity - to regulate the standard structural component as a unit of measurement of program source code.

SIMPLICITY
- Simplicity - the state or quality of being simple, the absence of complexity, intricacy, or artificiality.

STRUCTURE
- Structure - the organization of elements, parts, or constituents in a complex entity.

DOCUMENTATION
- Documentation - supporting references explaining the process of the program, the degree of self-descriptiveness of an application.

LAYOUT
- Layout - the arrangement, plan or formatting of the program.

These principles form the framework for the 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 the *STYLE* 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 de-

5

clared is used in the program. The rule "Use meaningful vari-
ables 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.

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 mes-
sages. In addition, our style analyzer would be based on pro-
gramming language independent concepts.

Chapter 3

USER INTERFACE

The user interface for *STYLE* is the desktop and uses the Apple™ Macintosh™ menu bar. See Figure 3 below for an example, this shows all of the menus of the application extended.

Figure 3
Style Desktop

●	File		Analysis		Help
About Analysis...	Open	% O	Style	% S	Economy
	Close	% C	Level	% L	Modularity
DAs	Save as ...				Simplicity
	Page Setup ...				Structure
	Print	% P			Documentation
					Layout
	Quit	% Q			Miscellaneous
					General
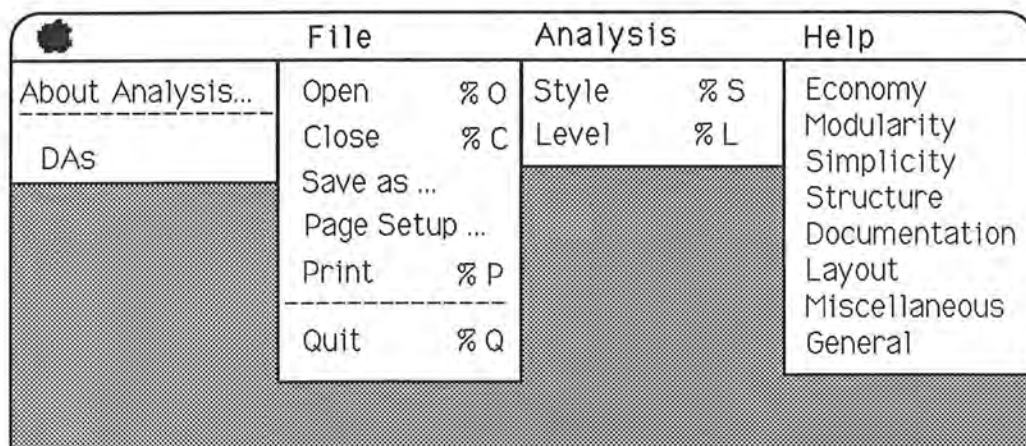
The **About Analysis** provides the author's name and version number of *STYLE*, and is shown below, in Figure 4.

Figure 4
About Analysis...

Welcome to the Style Analyzer

A Programming Style Tool

Version 1.0

by Al Lake

OK

File   **File** provides all of the file handling operations:

    ⌘O   **Open** - displays all files of type MacPascal™ or Light-Speed Pascal™, so that one can be selected.

    ⌘C   **Close** - closes the current work file.

    ⌘S   **Save as...** - saves the style analysis output to a text report file of TeachText format.

         Page Setup - performs page setup.

    ⌘P   **Print** - prints the style analysis report on the selected printer.

    ⌘Q   **Quit** - quits operation of *STYLE*.

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

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

    ⌘L   **Level** - Sets the user expertise level: either beginning, intermediate, or advanced. This level will determine the acceptable range of values for measuring. Beginning programmers do not have programming skills as well 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.

Help   **Help** provides a brief descriptions of the different principles and other information. All **Help** information is displayed in a modal dialog about the six style guidelines, a miscellaneous options, and any general information.

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.

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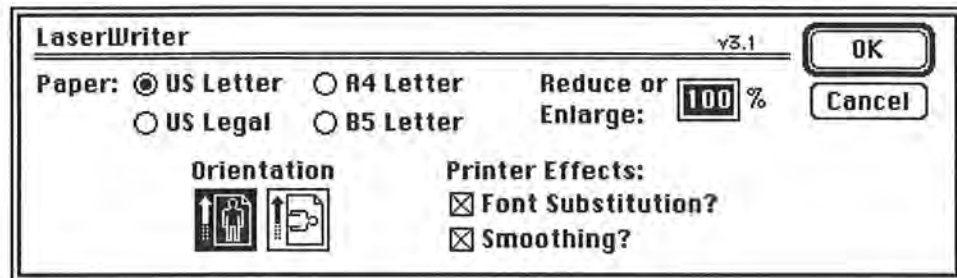
To open a file for analysis, select from the **File** menu the **Open** option. The following dialog will be displayed, filtering out all but the MacPascal™ and LightSpeed Pascal™ files. No special file names are necessary.

Figure 6
Open Input
File Dialog
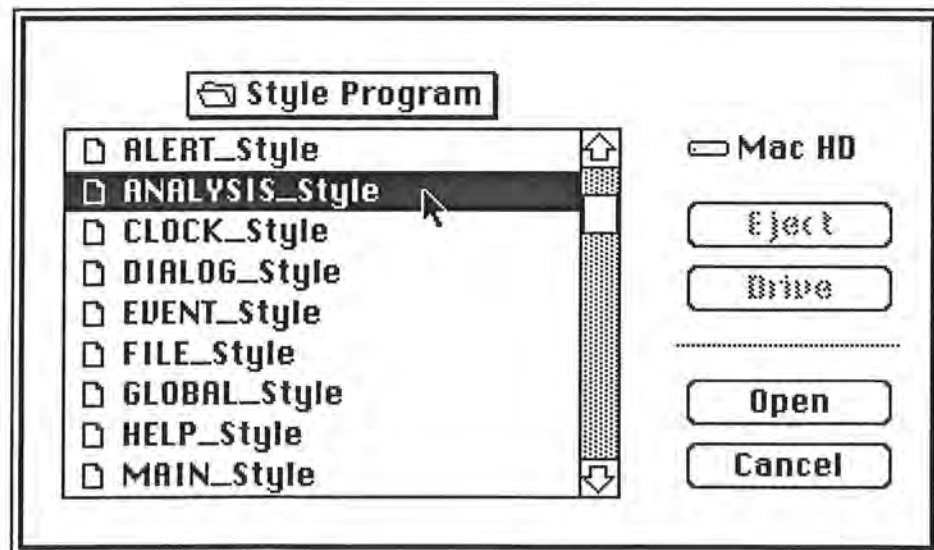
When the file is **Open**ed the program is read into a memory buffer. This allows the disk file to be closed and the program to operate more efficiently.

9

Saving a Report

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

Figure 7
Saving a Report
Dialog

```
┌─────────────────────────────────────────────┐
│  ┌─────────────────────────────┐            │
│  │ 🗁 Style Program          │            │
│  └─────────────────────────────┘            │
│  ┌───────────────────────┐ ⬆  ┌──────────┐ │
│  │ □ ANALYSIS...Style.pas│    │ 💾 Style │ │
│  │ □ CLOCK...Style.pas   │    └──────────┘ │
│  │ □ DIALOG...Style.pas  │    ┌──────────┐ │
│  │ □ EVENT...Style.pas   │    │  Eject   │ │
│  │ □ FILE...Style.pas    │    └──────────┘ │
│  │ □ GLOBAL...Style.pas  │ ⬇  ┌──────────┐ │
│  └───────────────────────┘    │  Drive   │ │
│  Save as ...                  └──────────┘ │
│  ┌───────────────────────────┐ ┌──────────┐│
│  │ TEST_Style.pas. Report    │ │  Save ▸  ││
│  └───────────────────────────┘ └──────────┘│
│                                ┌──────────┐ │
│                                │  Cancel  │ │
│                                └──────────┘ │
└─────────────────────────────────────────────┘
```
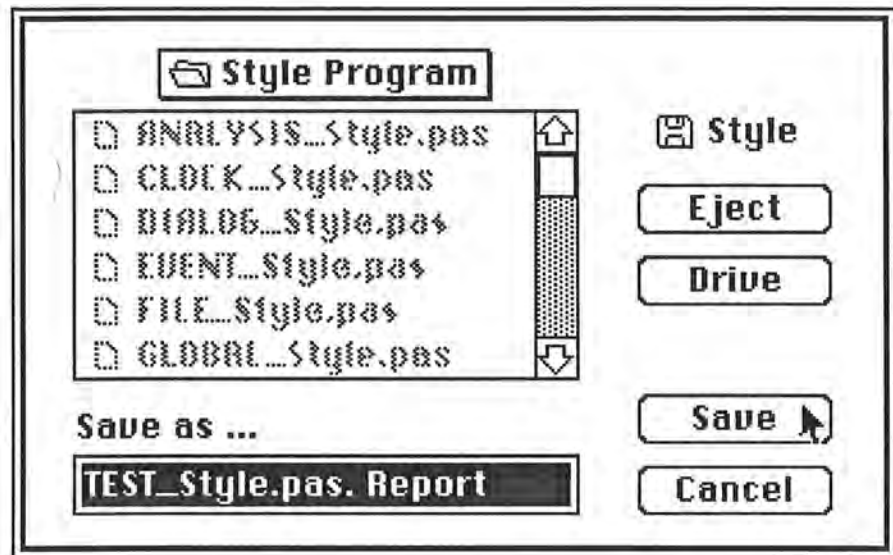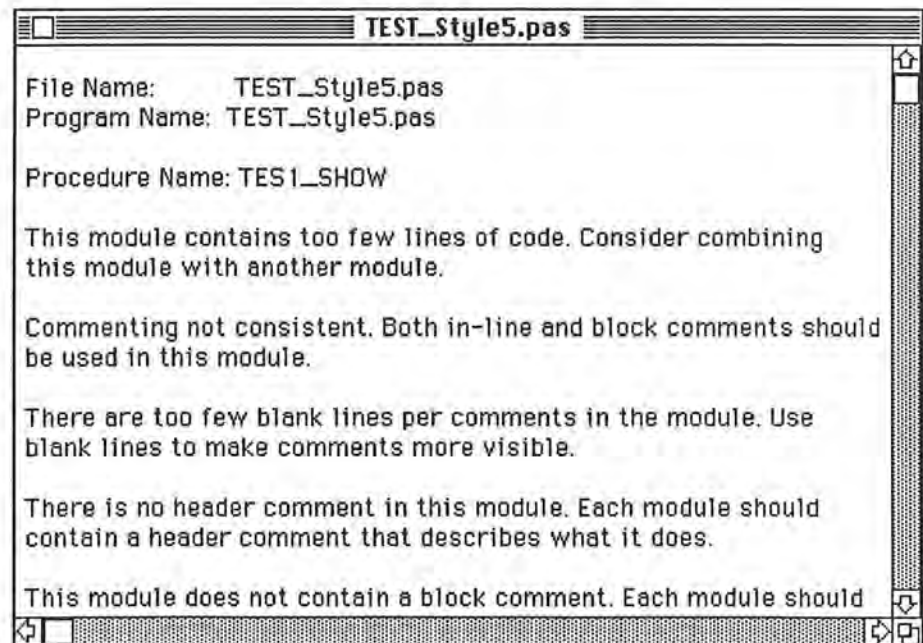
The program will automatically suffix the file name with ".Report" to help keep track of the relationship between the program file name and the style analysis report file, see the figure below for an example of a report file).

Figure 8

Sample Report
Window

```
┌═══════════════════ TEST_Style5.pas ═══════════════════┐
│ ■□■══════════════════════════════════════════════■  ⬆ │
│ 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 ⬇ │
│ ◁□▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▷ ⬓ │
└═══════════════════════════════════════════════════════┘
```
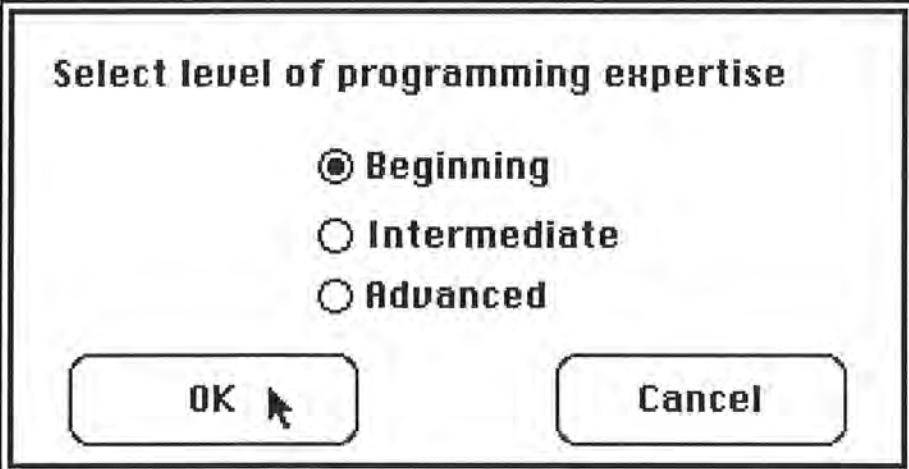
The information displayed in the analysis window begins with the program name followed by style messages for each of the subprograms in the physical order in which they occur in the program. This is illustrated in the sample window displayed below. The Sample Window, figure 8, displays a portion of a 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).

The user can change the level of programming expertise, choosing 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 the same programming and cognitive skills as do advanced programmers, so that more errors will be displayed for beginning programmers than advanced programmers.

Figure 9

Expertise Level Dialog

Select level of programming expertise

⦿ Beginning

◯ Intermediate

◯ Advanced

OK ▶   Cancel
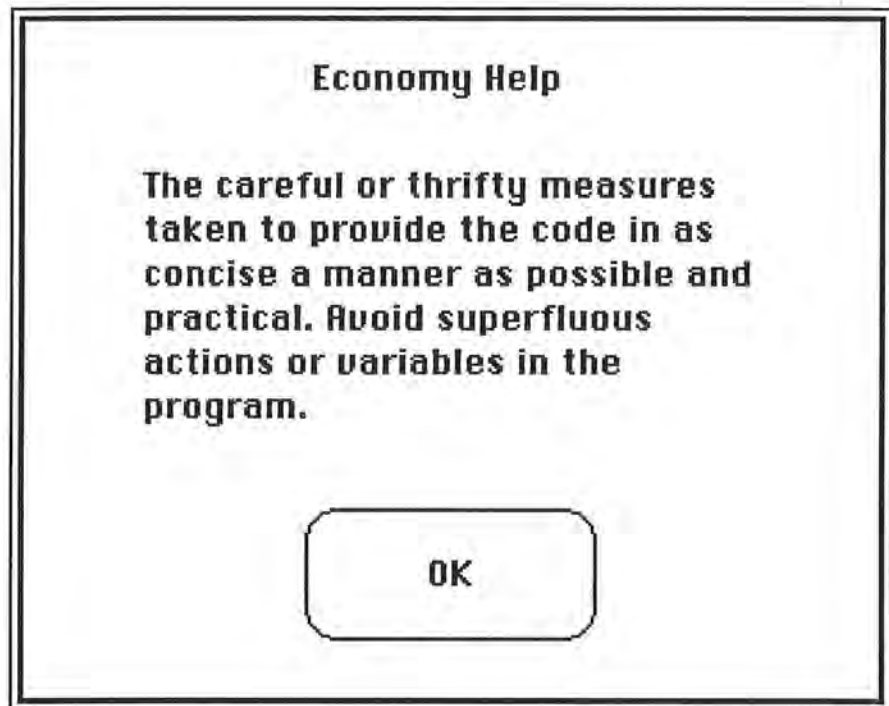
Help Dialogs

The Economy Help dialog screen is shown in figure 4 as an example of the type of dialogs 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.

Figure 10

An Example of the Economy Help Dialog

```
╔══════════════════════════════════════╗
║                                      ║
║          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.                           ║
║                                      ║
║            ╭──────────╮              ║
║            │    OK    │              ║
║            ╰──────────╯              ║
║                                      ║
╚══════════════════════════════════════╝
```

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.

Chapter 4

IMPLEMENTATION

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 the use of a computer with a limited memory size of 128K will limit the user file to less than 50K. For the best results, the *STYLE* 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, i.e. *STYLE* will not limit 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 or a dedicated printer.

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

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

Style is checking the six guidelines with quantifiable measures. These measures are described below.

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 (the nesting level times the number of lines at that level divided by the total number of lines in the module). 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. Maximum nesting level > 10.

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 RE-TURN 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 (RATIO LOC) 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

DOCUMENTATION
(Continued)

level of understanding of the program. This guideline cannot be effectively measured, but an estimate is made depicting the average number of words in each comment. With an upper and lower bound, so that n < AVG COMMENTS > m (for values like n = 50 and m = 3).

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.

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.

INDEX