Journal of the
Military Operations Research
Society of Korea, Vol.10, No.1
June, 1984

Program Family for Extended Pretty Printer

Ahn, Tae Nam*

Abstract

This thesis presents a design and partial implementation of a program family of extended pretty printers. Factors that influence the readability (perception) and understandability (cognition) of computer programs are indentified, previous work is reviewed, and new solutions are suggested. Extensions to the previous pretty printer designs include a capability to selectively display levels of control of a program. In order to accommodate different computer languages and to allow for several secondary functions, a family of pretty printers is designed. This design facilitates easy extension, contraction and modification.

1. INTRODUCTION

Programs are written to be read and understood by people. The textual representation should be such that it is easy to read. That is, the representation of the program should be such that it reduces the visual burden on the user and allows him to develop and exploit visual clues to aid in reading. In addition, the text of the program should be designed so that it is easy for the reader to

* Korea Military Academy

grasp the meaning of the program: that is, the representation of the program should help the reader understand the program.

Fifteen years age Dijkstra argued that "... our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible." [Ref. 16]. There is an additional conceptual gap between the program spread out in text and how we represent and manipulate the static program and its dynan process in our minds. Here also we should try to narrow the conceptual gap so that the program is easy to read and to understand.

In the computer science literature, readability and understandability are often used interchangably. Readability is related to physical conditions, for instance, the size, type font, color, and clarity of characters, proper indentations, and the spacing between lines. Understandability is related to psychological conditions, for instance, pattern, memory, logic, and repetition learnings. Precisely speaking, readability means good perception and understanc ability means good cognition. The system that will be designed in this thesis will seek to improve both readability and understandabil by means of reformating computer programs and presenting the user wi alternative representations to aid understanding.

There is evidence to show that readability and understandabilit of computer programs is an important issue that is directly related to programmer productivity. Although this has been recognized for some time, further improvements in the textual representation of con puter programs are possible. This thesis will review the previous work, analyze the remaining problems, and propose new solutions.

II. EXTENDED PRETTY PRINTER

BACKGROUND

In a study of commercial programming practices, Elshoff [Ref. 5] ıd that most programs were poorly written. They were very large, remely difficult to read and understand, and more complex than essary. Furthermore, the study determined that programming language ge was poor and inconsistent. The results of the survey by Lientz :. 6] show that the quality of programming is a generally perceived blem. There has been a major effort to improve programming prac- es. But there still exist many programs that are difficult to d and understand and yet they must regularly be corrected and/or ified.

There are many factors connected with the readability and under- ndability of a computer program. The reader's familiarity with program, knowledge of the application area, and own programming le are important factors that are mostly independent of the pro- m [Ref. 4]. This thesis is concentrated on the representation of gram text that impacts its readability and understandability. eadable program always seems to exhibit a common set of properties f. 8] [Ref. 9] [Ref. 10]. The program is well commented. The ical structure of the program is constructed from a common set of gle-entry single-exit flow of control units. Variable names are onic and reference to them localized. The program's physical lay- makes the salient features of the algorithm that is implemented nd out [Ref. 14].

Since abstraction is an important mechanism that people use to erstand programs, the suppression of details in a program can aid erstanding. Modern design methodologies include top down design ng stepwise refinement. In this methodology, the programmer de- ns successive levels of the program. These levels are visible ing the design but are often not visible in the final program.

The understandability of a program can be improved by making the levels of the program structure visible. It is true that a program may have all these properties and still be unreadable and not understandable; however, the readability and understandability of a progr are certain to suffer when it lacks one or more of the properties [Ref. 14].

B. DEFINITION

Rubin [Ref. 14] defined a pretty printer as follows: "It is a software tool to format programs to make them easier to read and understand." The extended pretty printer can be defined as: a soft ware tool to improve readability and understandability by adding level documentation, commenting and reformatting. These additional extensions to pretty printers will aid people in understanding the program by making more visible the structure of the program and supporting the viewing of the levels of the program. The primary function of an extended pretty printer is to add some level documentati and comments, to insert spaces and linefeeds between tokens - charac ter strings - and to decide where and how to break lines that are t long to fit on the output medium.

C. GOALS

The methods for improving the readability and understandabilit of a program use a set of specific transformations that can be appl to the program text. The following program transformations can be done by an extended pretty printer.

1. Reformat

The consistent formating of programs is very important. Elsho [Ref. 14] said "Just paragraphing and sectioning help written Engli so can indentation, keyword positioning, and logical grouping aid a programming language." Those jobs can be done automatically by a pretty printer. It will allow the program to be read more easily.

. Add Level Structure Documentation

In writing about his experiments on program comprehension, Shneimann [Ref. 17] said "Instead of absorbing the program on a charac--by-character basis, programmers recognize the function of groups statements and then piece together these chunks to form ever larger nks until the entire program is comprehended." This experiment gests that the level documentation (chunks) of a program will help understandability of the program.

. Standardization

Standardization contributes understandability of a program. To erstand this, it is helpful to know the source of the expert promming's capacity. The primary piece of direct behavioral evidence this is Shneiderman's replication [Ref. 26] for programming of se and Simon's classic study on memory for cheses position [Ref. . In both these studies, the experts in a particular domain could orize information from that domain (i.e. a program or a chess posin) far better than novices, provided that the information was ropriately structured. If the structure was made random (by shuffg the statements of the program or rearranging the chess pieces), advantage of the expert would be greatly reduced. That means that expert has no better memory than the novice, but rather an elaboe knowledge structure in terms of which correspondingly structured ms can be very efficiently enceded [Ref. 15].

If this result is extended to programming, it suggests that the ert programmer gets his better knowledge of programs from visible gram structure. As noted above, if the textual representation is structured (e.g. random), the expert programmer will lose part of capability. People understand something better when they can egrate it with what they already know. From this view, standardiion helps people to understand other people's programs more quickly visual cues are important in order to unburden the program reader. final objectives of computer program standards are to ensure con-

sistency, reduce program development and testing time, improve mair tainability of programs, and improve changeability of programs [Ref 12]. Programming standards are not intended to stifle the imaginat of programmers. Experiments of Godfrey [Ref. 12] have shown that standards simply remove the drudgery of coding and allow programmer to concentrate more on the problem at hand. It should be noted tha the establishment of standards is a costly process. It should be kept in mind that programming standards are not a panacea for elimi nating all poorly written programs. Adherence to these standards will not automatically produce 'good' code [Ref. 12].

There are multiple levels of understanding a program. It is possible to follow each line of code without understanding the over all program function. It may also be possible to understand the program function but not understand each of the steps. There is also a middle level of understanding concerning control structures, module design, and data structures [Ref. 17]. Skimming for a top down view is to suppress detail until the overall program is understood. Then the program is read selectively and understood in more detail.

III. SOME APPROACHES AND VARIOUS OBJECTS

A. SOME APPROACHES

There have been many attempts to improve understandability and readability. The following are typical examples.

1. Neater2

Neater2 accepts a PL/I source program and operates on it to produce a reformatted version. When in the LOGICAL mode, it indica the logical structure of the source program in the indentation patt of its output. A number of options are available to give the user full control over the output format and to maximize its utility. [R 19]

2. Prettyprint

It takes as input a Pascal program and reformats the program according to a standard set of pretty printing rules. The pretty printing rules are given i.e., fixed. [Ref. 22]

3. Pascal Program Formatter

Format is a flexible pretty printed for Pascal programs. It takes as input a syntactically-correct Pascal program and produces output an equivalent but reformatted Pascal program. The resulting program consists of the same sequence of Pascal symbols and comments, but they are rearranged with respect to line boundaries and columns for readability. [Ref. 20]

The flexibility of Format is accomplished by allowing the user to supply various directives(options) which override the default values. Rather than being a rigid pretty printer which decides how a program is to be formatted, the user has the ability to control how formatting is done, not only prior to execution but also during execution through the use of pretty printer directives embedded in the program. [Ref. 20]

4. Contour

It is a program whose purpose is to graphically illustrate a program's structure. It operates by bounding the scope of loops and conditionals by solid (or nearly solid) lines. When compound statements are embedded in other compound statements, one obtains, rather than confusion, a rather pleasant display reminiscent of the contour lines of a topographical map. [Ref. 22]

5. Syntax-Directed Pretty Printer

It is a language independent pretty printer. It is divided into two phases: the grammer processing phase and the program processing phase. A language grammar for the specific language must be provided It is much easier and quicker to write a grammar for a language than

to code a new pretty printer for a specific language. It can work for all structured programming languages, and with minor modificatioı can work for other languages. It can handled such problems as comments and error recovery.

6. Others

The recent availability of low cost, high quality computer prinı ers allows additional opportunities to improve readability and under· standability. Important characters or words can be represented with different fonts: for instance, the keywords can be represented by bold characters or be underlined to be recognized more easily than other words. This can improve the readability of program.

B. VARIOUS OBJECTIVES

Although the final objective of all approaches is to improve the readability and understandability of the program, there are many secondary objectives. The following are typical examples of them:

Teaching structure: An automatic system that checks structure and indentations can help beginning students learn good programming practice. A system that gives clear corrections to mistakes can provide a student with quick feedback. Such a system helps a studenı to learn structured programming and to learn a set of programming standards.

Standardization in a programming organization: For large software projects with many programmers, program standardization is neces sary to help in communication among programmers.

Reformatting for maintenance: There are many programs that are very difficult to read. The maintenance process can be helped if programs can be transformed into a form that is familiar to the maintenance programmers. The scoping capability of an extended pretty printer as described above can also help programmers understand programs they are correcting and modifying.

Automatic corrections: An extended pretty printer can check e indentation of programs, correct indentation errors, and give e user messages explaining the errors.

From the above observations, several common parts of the exist- g approaches can be found. First, most of the systems are for a ecific programming language, for another programming language they uld have to be written again. The one exception is the syntex rected pretty printer: for each new language it requires a grammar r each the language. Defining a correct grammar is not an easy sk. Second, most of the systems try to make the pretty printer exible, but the flexibility is limited to a few options and it is : easy to extend the requests. Most constructs of the pretty print- s are fixed, but the constructs themselves can be changed e.g. ex- ıded or contracted. New structures for indentation can be generated

IV. PROGRAM FAMILY

DEFINITION

Program families are defined by Parnas [Ref. 13] as sets ot pro- ms whose common properties are so extensive that it is advantageous study the common properties of the programs before analyzing indi- ual members. Program families are analogous to the hardware fami- s promulgated by several manufacturers. Although the various els in a hardware family might not have a single component in com- , almost everyone reads the common 'principles of operations' ual before studying the special characteristics of a specific el [Ref. 13].

DESIGN METHODOLOGY

Parnas [Ref. 13] shows how module specifications define a family s is an important guide for selecting the design method. Members

of a family of programs defined by a set of module specifications can vary in three principal ways.

1. Implementation Methods Used within the Modules

Any combination of sets of programs which meet the module specifications is a member of the program family. Subfamilies may be defined either by dividing each of the main modules into submodules in alternative ways, or by using the method of structured programmin{ to describe a family of implementations for the module.

2. Variation in the External Parameters

The module specifications can be written in terms of parameters so that a family of specifications results. Programs may differ in the values of those parameters and still be considered to be members of the program family.

3. Use of Subsets

In many situations one application will require only a subset of the functions provided by a system. We may consider programs whi(consist of a subset of the programs described by a set of module spe(fications to be members of a family as well.

As discussed above, there are many primary and secondary objectives for a pretty printer. One approach to these various demands would be to design a large program with many options. Such an approach has several drawbacks: first, the resulting program would be large and necessarily complex, second, for each specific use of the program the unneeded options will most likely impose an unnecessary computational burden. The notion of a program family offers an alte: native design. A separate program will be written for different demands, however, all these programs will share a common design and many modules will be common to several family members.

The concept of program families provides one way of considering program structure more objectively. For any precise description of

ı program family (either an incomplete refinement of a program or a ›et of specification or a combination of both) one may ask which pro- ;rams have been excluded and which still remain [Ref. 13]. The :riteria of defining modules can be a way to select or distinguish ·ome design methodologies [Ref. 3].

. PROGRAMMING LANGUAGE FOR OBJECT ORIENTED DESIGN

A design methodology alone is not sufficient to create computer olutions [Ref. 3]. Some features of a programming language can also elp in creating good software. In the following table, P. Wegner as categorized some of the most popular languages into generations, long with some of

TABLE I

Programming Language Generation Table

Generation	Languages	Period
1ST	FORTRAN I, ALGOL58	1954 - 1958
2ND	FORTRAN II, ALGOL60 COBOL, LISP	1959 - 1961
3RD	PI/I, ALGOL68, PASCAL	1962 - 1970
GAP		1970 - 1980

he language features they introduced.

ADA was developed at the end of the language generation gap, and o has been influenced by contemporary software methodologies.

The following key features of ADA will support the tools for mplementing the object oriented design [Ref. 23].

1. Programming in the Large

Mechanisms for encapsulation, separate compilation, and library ɪanagement are necessary for the writing of portable and maintainable programs of any size.

2. Exception Handling

Large programs are rarely correct. It is necessary to provide a means whereby a program can be constructed in a layered and parti tioned way so that the consequences of errors in one part can be contained.

3. Data Abstraction

Extra portability and maintainability can be obtained if the details of the representation of data can be kept separate from the specifications of the logical operations on the data.

4. Tasking

For many application it is important that the program be con- ceived as a series of parallel activities rather than just as a single sequence of actions. Building appropriate facilities into a language rather than adding them via calls to an operating system gives better portability and reliability.

5. Generic Units

In many cases the logic of part a program is independent of th types of the values being manipulated. A mechanism is therefore necessary for the creation of related pieces of program from a sing template. This is particularly useful for the creation of librarie

V. MY SOLUTION

A. PROBLEM AND SOLUTION

As shown above, most traditional approaches to pretty printer are for a specific programming language. A recent development is the syntax directed pretty printer that can be used for different languages by providing a grammar of the language. The requirement to provide a language grammar represents a non-trivial task. Ther

re many different secondary objectives for a pretty printer for ifferent users. The functions of a traditional pretty printers are ot enough to improve both the readability and understandability .g. the program level construct documentation that traditional ap- oaches do not support is needed to help to understand a given rogram. In short, there are many programming languages and many urposes, but there is not a system that satisfies all those requests nd can be modified easily.

In the previous section, the concept of a program family was iscussed. The best way to solve the various demands and many pro- ramming languages is to construct a program family for the extended retty printer. The characteristics of program family will permit asy change, easy extension, and easy contraction. Each programming anguage will have a module for itself and data abstraction and pro- edural abstraction will be used to hide design decisions that will iffer among the members of the program family. Data and procedural ostraction will also allow some modules to be used by all program amily members. For example, the blank operations are a important ata abstraction. These operations can be used for all programming anguages and objectives.

GENERALIZED PROGRAMMING LANGUAGE CONSTRUCT

For generalized indentation and level documentation, an general nternal representation of program structure is required that is ndependent of any particular programming language. Let us call it generalized formatter structure. Since there are many programming anguage constructs in the many different programming languages, it s too difficult to define a perfect universal programming language ormatter construct. So, we define here a generalized programming anguage formatter construct that can cover only a limited number of rogramming languages - structured FORTRAN, PASCAL and some other tructured programming languages. For simplicity, the detailed re-

prepresentation of a simple statement will be omitted.

The structure of the program will be shown by indenting the constructs. First, the control structure will be considered. Dijkstra argued that control flow should be limited to three basic structures - linear sequence, structured selection, and structured iteration. But many programmers use the following five structures - if, case, while, until, do for. Also the block can be a element of the structure. Second, most program units are divided into two parts: a declarative part and imperative part. This is also important for the indentation.

C. ANALYSIS AND DESIGN

1. Analysis

The extended pretty printer has two basic functions. The first is to reformat the source program e.g. indent, insert spaces and line feeds between tokens and to decide where and how to break lines that are too long to fit on the output medium. The second is to produce level structure documentation of the source program. The basic requirement of the total system is that it has to be easy to change, easy to extend, easy to contract, e.g. it should be independent of the programming language and should be able to fulfill a variety of purposes.

Every structured programming language can be represented as English is - character, word, statement, compound statement(paragraph unit program(a paper). What is of interest is the way to represent these component as lines. The relationship of these components and lines is very important for the extended pretty printer. The following table represents the relationship of line and statement. The other components have some relation with the statements. So, every component can be represented by lines.

Each level is represented by the source program structures.

he structures are represented by statements. So, each statement can ave a level degree.

2. Design

As noted in the section on program families, the most important spect of this system design is to identify the objects. For the in-entation, the line and statement are basic elements. Blark is other mportant object. For the construct representation, level has to be object.

TABLE II

elationship Table

Line	Statement
one	one
one	many
one	part
one	part and one/many

("part" means part of a statement)

The heavily dependent parts should be encapsulated in a module o allow for easy change. The indentation policy can be changed ariously, it needs to be manipulated independently. To manipulate he input programming languages independently, the program should be independent module. The program module needs some data structures STACK, QUEUE -, Keywords table, and some statement operations. he files - input source file and output file - and their format can e changed easily. So, the input/output files manipulations need be eparated from other modules.

For convenience, the module will be divided into two kinds. ne is passive modules that are used by other modules but that do ot use other modules, for example, clank, level, stack, queue and

line. The other kind is active modules that use the other modules, for example, input, output, program and so on. ADA will be used for the detailed design of the system.

a. Passive Modules

(1) Stack Module. This module provides some stack operations. And it provides the following procedures for other modules that use them [Ref. 24].

(2) Queue Module. This module provides some QUEUE operations. And it provides the following procedures for other modules that use them [Ref. 24].

(3) Blank Module. This module provides all blank operations that insert, remove, count and so on for other modules tha need the blank operations.

(4) Level Module. This module will provides the level operations for other modules that need them. The operations are:

(5) Line Module. This module manages the line object. It provides a set of procedures available to other modules that use the line.

(6) Symbol Table Module. This module will manage a symbol table. It is designed for general symbol manipulator.

b. Active Modules

(1) Input Module. This module hides the input format. It reads the original lines from the input media and calls procedures provided by the line module to stone the lines inside of the line object.

(2) Output Module. This module will hide the outfile media. And it will output the indented results, the construct for of the input program and the input using other modules - indent, line and so on.

(3) Statement Module. This module manages the statement object and also provide a set of procedures available to other modules that use the statement object by using line module procedures.

(4) Indent Module. This module will indent each line using the line module, statement module and blank module. And the indentation policy can be decided here e.g. the size of each level, the treatment of blanks, and so on.

(5) Program Module. This module will hide the program characteristics. It should be highly dependent on each programming language. It have two procedures - scanner and parser

(6) Master Module. This module will control all above modules. The above figure explain the interfaces of each module. The arrow direction indicates using module.

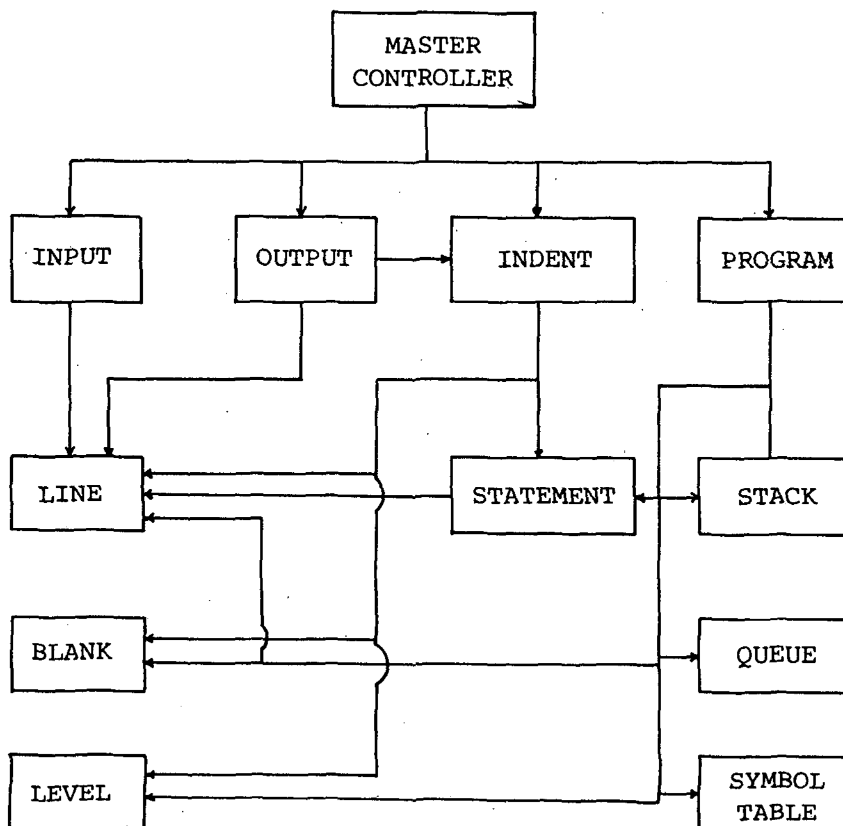

Figure 5.1 Module Interface

D. EXAMPLE (FORTRAN)

1. Standard Form

There have been many attempts to standardize the FORTRAN program ming language. Here, the standard form will follow the concept of COMPATIBLE FORTRAN [Ref. 1].

2. Structured Form

The algorithm language [Ref. 18] is convenient for representing the generalized construct structure. So, to represent the structure FORTRAN form, it will be compared with the algorithm language.

3. Format Grammar

This grammar represents the construct format of structured FORTRAN. It is a subset of the generalized format structures. The control structure is limited to 5 structures - if, case, while, until, and do.

4. Implementation

a. Limitations

An ADA complier was not available for this work. So, the PASCAL programming language was used to implement the system This implementation is a little different from the design of the previous section because PASCAL does not support all the ADA programming features. In order to simply the implementa tion, just a subset of the system was implemented, i.e. the UNTIL construct is omitted.

Also the implemented system does not cover all standard FORTRAN - it does not include some keywords like PAUSE, RE-WIND and so on. The other limitations of this are the follo ing: 1. All input programs should be syntatically correct to get proper indentation and the level documentation.

2. All input FORTRAN programs should be conform to the standard structured form mentioned in previous sections.
3. The input lines should be short enough to indent without being extended onto the next line. That is the implemented system does not have the line break function.

b. Internal Data Structure

(1) Line Data Structure. The input line and output line are represented as an array of characters. Normally, programming languages use 80 column per line. In actual programs, most lines do not use all of the columns; the mean of programming line size is 34 [Ref. 2]. If the maximum array is assigned for one line, space is wasted. So to save memory and make the line flexible, a double linked data structure was used for the internal line structure. Also, a sentinel node will be used. It allows an easy check of an empty input file.

(2) Statement Data Structure. As shown above, the relationship of line and statement is one to one or many to one. Clearly, the statement can be represented by the line data structure. So, a line record will have information about statements. Comment statements will be ignored for statement representation.

(3) Construct Data Structure. The construct will have some relationship with the statements e.g. one to one for simple statements, one to many for others. The statements can have the information of the construct, since every construct can be separated into statements. For example, the DO construct consist of DO_COND statement, compound statement and END_DO statement. But here, the line also will have the construct information. It is possible since the relationship of line and statement also one to one and many to one.

c. The Program and Example Input/Output

Anyone interested in obtaining a copy of the program should contact the author directly or the Computer Science Department at the Naval Postgraduate School, Monterey, California.

VI. CONCLUSION

One of today's software problems is the very high cost of developing and maintaining software. Much research has been devoted to solving this problem. One way to solve today's software crisis is to study software tools that can help people who serve in the software area.

This thesis designed and partially implemented a program family of extended pretty printers that can help to solve software problems by improving readability and understandability of programs.

The system will work for almost any structured programming language and for various secondary functions with only small changes in some modules. The design presented here is for a program family of pretty printers. The program implemented here is one member in this family. Other members of the program family remain to be implemented.

REFERENCES

1. Day, Colin A., Compatible Fortran, Cambridge University Press, 1978.
2. Bruell, S.C. and Schneider, G.M., Advanced Programming and Proble Solving with PASCAL, Wiley, 1981.
3. Bocch, G., Software Engineering with ADA, Benjamin/Cummings, 1983
4. Elshoff, J.L. and Marcotty, M., "Improving Program Readability to Aid Modification", Communication of the ACM, August 1982, Vol. 25, No. 8, pp.512-521.

5. Elshoff, J.L., "An Analysis of Some Commercial PL/I Programs", IEEE Trans. Software Eng., SE-2,2, pp.113-120.
6. Lientz, B.P. and Swanson, E.B., Software Maintenance Management, Addison-Wesley, Reading Mass., 1980.
7. Sheppard, S.B., Curtis, B., and Milliman, P., "Modern Coding Practices and Programmer Performance", IEEE Computer, Dec. 1979, pp.41-49.
8. Kernighan, B.W., and Pluger, P.J., The Elements of Programming Style, McGraw-Hill, New York, 1974.
9. Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, Englewood-Cliffs, N.J., 1975.
10. Myers, G.J., Software Reliability, John Wiley, New York, 1976.
11. Shooman, Martin L., Software Engineering, McGraw-Hill, 1983.
12. Lee, Godfrey and Others, "FORTRAN Programming Standards", SIGPLAN Notices, 15:2 (Feb. 1980), pp.52-63.
13. Parnas, David L., "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, pp.1-9, 1976.
14. Rubin, Lisa F., "Syntax-Directed Pretty Printing - A First Step Towards a Syntax - Directed Editor", IEEE Tran. Software Eng., Vol. SE. 9, No. 2, March 1983.
15. Sheil, B.A., "The Psychological Study of Programming", Computer Surveys, Vol. 13, No. 1, March 1981.
16. Dijkstra, E.W., "Go to Statement Considered Harmful", Communication of the ACM, Vol. 15, No. 10 (Oct. 1972), pp.859-66.
17. Shneiderman, Ben, Software Psychology, Winthrop Publishers, 1980
18. Graham, N., Introduction to Computer Science, West Publishing Co., 1982.
19. Conrow, K. and Smith, R.G., "Neater 2: A PL/I Source Statement Reformater", Communication of the ACM, 13, 11 (nov. 1970), pp. 669-675.
20. Condict, M.N., Marcus, R.L. and Mickel, A., "Pascal Program Formatter", Pascal News, No. 13 (Dec. 1978), pp.45-58.

21. Hueras, Jon F. and Ledgard, Henry F., "Prettyprint", Pascal News, No. 13 (Dec. 1978), pp.34-44.

22. Gimpel, James F., "CONTOUR - A Method of Preparing Structured Flowcharts", SIGPLAN Notices, Vol. 15, No. 10 (Oct. 1980), pp. 35-41.

23. Barnes, J.G.P., Programming in ADA, Addison-Wesley Publishing Co., 1982.

24. Maych, Brian, Problem Solving with ADA, John Wiley & Sons, 1982.

25. Gehani, Narain, ADA an Advanced Introduction, Prentice-Hall Software Series, 1983.

26. Shneiderman, B., "Exploratory Experiments in Programmer Behavior", Int. J. Comput. Inf. Sci., 5 (1976), pp.123-143.

27. Chase, W.G. and Simon, H.A., "Perception in Chess", Congnitive Psychol., 4 (1973), pp.55-81.