AD-A231 805

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1990	Thesis/Dissertation

4. TITLE AND SUBTITLE
Increasing the Readability and Comprehensibility of Programs

5. FUNDING NUMBERS

6. AUTHOR(S)
Thomas Michael Schorsch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Student at: University of Colorado

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA -90-141

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFIT/CI
Wright-Ptatterson AFB OH 45433

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release IAW AFR 190-1
Distribution Unlimited
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS

15. NUMBER OF PAGES
75

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED			

INCREASING THE READABILITY AND COMPREHENSIBILITY OF PROGRAMS

by

THOMAS MICHAEL SCHORSCH

B. A., United States Air Force Academy, 1985

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

1990

This thesis for the Master of Science degree by

Thomas Michael Schorsch

has been approved for the

Department of

Computer Science

by

Benjamin Zorn

Michael Main

William Waite

Date 30/Nov/90

Schorsch, Thomas Michael (M. S., Computer Science)

Increasing the Readability and Comprehensibility of Programs

Thesis directed by Professor Benjamin Zorn

Source code must be readable and understandable in order to be debugged, used, and maintained efficiently and effectively. Creating programs that are easily read and understood is important, but not as simple as one might think. This thesis surveys the ways in which programming style, programming language design and program presentation influence a person's ability to read and understand program text. It is concerned with solving small scale problems associated with individual functions or files and not with problems that pertain exclusively to large software systems. Software engineering issues, while important, are not included in this thesis.

The programming style chapter examines the changes a programmer may make to source code without modifying the content or meaning of the code. Two of the main sections cover code reformatting and identifier naming conventions. The chapter on language design discusses the selection of basic tokens (symbols and keywords) and the structure of language statements. This chapter also categorizes the subtle errors that occur when the meanings of a code segment when read and when executed are not the same. The program presentation chapter presents methods for enhancing completed source code making it easier to read and its true meaning easier to discern. Restructuring/transforming the code, graphic design principles, and introducing supplemental information are some of the main sections. Knuth's literate programming paradigm is examined in this chapter as well. One of the underlying themes of the thesis is that an automated proofreader program could greatly aid programmers to create readable, understandable programs.

DEDICATION

To SMS (both).

CONTENTS

FIGURES

FIGURE

CHAPTER 1

INTRODUCTION

This thesis examines the role that coding style, programming language design, and source code presentation have in increasing a person's ability to read and understand programs.

1.1 Motivation

Writing a syntactically correct program (one that a compiler will compile) is necessary but hardly sufficient. Compilers, by definition, are good at translating code and determining its meaning where as people are not. As an extreme example, the code in figure 1.1 is *syntactically correct* and presents no problems to the compiler; the difficulty it causes humans is obvious. People require a more readable style of coding in order to understand a program. Even then it is easy to misinterpret what a program will do, especially if the language in which it was written has ambiguous constructs (a compiler will always interpret a statement the same way, a human reader may interpret the meaning of the statement differently).

```
float      A(          x)   float
 x      ;     {if(x     <0)    return(-
  x      );return    (x       );}float sr(
   x     )float x    ;{ float g,   a;a   =0.00001;
    if   (        x
     <  0){                          printf(
    "error"     );return             (-1.0
    )    ;}while  (A       (g      *g -
   x)    >=a      )g      =(x     /g+
  g )/2.0       ;return(       g);}
```

Figure 1.1: Syntactically correct but otherwise unreadable code

The term 'coding' originated from the process "programmers" went through to translate their high level algorithms into a series of numeric codes or assembly language codes which the computer could understand (pre-Fortran days). Much of the code written was 'write once read never'; it was often easier to recode a small program then to try and modify it. Much has changed since then; now programmers spend an enormous amount of their time reading 'code', especially during debugging and testing. Some things have remained the same though; Fjeldstad and Hamlen, in a study on program maintenance, report that understanding the intent and style of another person's code was recorded as the major difficulty in making a change[21]. They also report that nearly half of a maintenance programmer's time is spent analyzing the program text and documentation. In view of the preponderance of time and energy the maintenance task consumes for a typical software system (Between 50% and 75% [58]) and the time spent coding and debugging the program in the first place, the additional requirement that programs be readable and understandable cannot be stressed highly enough.

People must be able to read and understand a program in order to use, debug, maintain, and modify it efficiently and effectively. The easier it is to comprehend a segment of code, the easier it is to be assured that the code is correct and reliable, or if it is not, to make it so. The degree to which a program can be comprehended is influenced by the style of its author, the language in which it is written, and the form in which it is presented to the reader. For these reasons, programmers and language designers need to be more aware of the choices available to them that will directly affect the creation of comprehensible software systems.

1.2 What this thesis is and is not

The programming languages discussed in this thesis are for the most part widely-used imperative, programming languages (Ada, C, Cobol, Fortran, Pascal,

PL/1, Modula-2), with a few digressions into Lisp and APL. No attempt has been made to include the more esoteric imperative programming languages, functional programming languages, or the object-oriented versions of existing mainstream languages. However, in spite of this bias, most of what is presented should be applicable to all programming languages.

The first chapter in the body of this thesis covers aspects of programming style. It contains a plethora of postulates and principles on how programs can be written to be more readable. In many cases the syntax and semantics of a language affect the style of programming that is most effective. In these cases the tradeoffs in readability of one style over another are discussed in terms of several specific language syntaxes. The arguments why, and uses when, one programming style is superior over another are presented.

The second chapter examines how to design a programming language that facilitates creating readable programs. This inquiry takes the bottom up approach. It scrutinizes first the basic tokens that make up a programming language, continues with the structure of a basic programming language statement, and ends with the design of programming language features. Individual decisions as to what syntactic form a language construct should take are weighed in terms of the tradeoffs in readability.

The last chapter in the body of the thesis examines how the readableness of a program can be increased by changing how it is presented. In particular, this section examines how reformatting the program, using graphic design principles, transforming the code, and supplying additional information about the program can greatly enhance the readability of the program. Knuth's "literate programming" paradigm is examined as well.

This thesis is mainly concerned with readability on a small scale. Programming styles are surveyed and analyzed but program design is not. The use of

structured systems design and top-down, bottom-up, data-oriented implementation techniques all lead to more readable code. Concepts like module coupling and cohesion are also important. These topics and others like them, while important in creating readable code, are beyond the scope of this thesis.

Similarly, this thesis looks at programming language design on a small scale instead of the design of a language as a whole. Language designs that support abstract data types, modules, encapsulation, and information hiding can be helpful for creating readable programs, these higher level language design issues are also outside the scope of this thesis. The search for the most readable language constructs and the most readable programming language depends too much on the problem being solved. Higher level language design issues such as what control structures and data structures are the most readable are not covered either. It is fairly easy to point to language constructs which impair readability, either because they are amenable to being used in an unreadable manner (e.g. the `COME FROM` statement [14]) or because the construct has an unreadable syntax. This thesis does discuss how to make a given control structure more readable. It also identifies several undesirable features that programming language constructs can have and offers suggestions to help remove or mitigate the effects of those features.

CHAPTER 2

PROGRAMMING STYLE

What is written without effort is in general read without pleasure
—Samuel Johnson

Beginning programmers believe they are writing code for a machine to read; more experienced programmers know they are writing programs for people to read. Programs written without readability as a goal require substantial extra effort to be understood. Programming specifically for readability is possible; Weinberg and Schulman demonstrated that programmers can write substantially different solutions to a problem depending upon whether program efficiency, size, or readability was stressed [70]. With current memory technology the way it is, the size of programs is rarely an important issue. The case against efficiency cannot be expressed better than Wulf's statement: "More computing sins have been committed in the name of efficiency (without necessarily achieving it) than for any other reason — including blind stupidity" [78].

Readability is clearly important as any given piece of code is read many times by the original author during the writing, debugging, and testing phases and by any maintenance programmers that succeed her. One of the main influences on the readability of a program is the style of its author. This chapter explores the aspects of programming style which can enhance the ability of a program to be read and understood independent of the language in which it was written.

2.1 lowercase versus UPPERCASE

There have been numerous experiments [54, 10, 55, 57], many of them predating the existence of computers, that prove that UPPERCASE BY ITSELF IS LESS READABLE THAN LOWERCASE. Text consisting only of uppercase characters is harsh and jolting to the eye. The lowercase character set contains 'distenders' that enable readers to read and recognize lowercase words easier then the same uppercase words [55, 49]. A distender is that portion of an individual character symbol which protrudes from the rest of the character string. For example, the uppercase character string, PRETTY, has no distenders while the lowercase character string, pretty has four: p, two t's, and y. The above citations agree that lowercase and mixed upper and lowercase, Pretty, is read 10–20% faster and comprehended better than uppercase alone.

In the past, programs were written using only capital letters due to the limited input and output devices of the early computers. Two of the oldest languages, Fortran and Cobol, have yet to escape their roots. Even though uppercase characters are not required by the language definitions, textbooks, journals, and new defining documents for Fortran and Cobol continue to use only uppercase characters in programming examples. Only a few textbooks can be found for both languages that use lowercase and mixed case programming examples [31, 65]. A later language, Pascal, was transitional in that its defining documents used lowercase examples [74, 34], yet many computing devices still could not support lowercase. Strangely enough, textbooks on Pascal can still be found which use only uppercase programming examples [7].

More recent languages, Ada, C, Fortran 8x, and Modula-2 have taken very different approaches. C considers upper and lowercases to be syntactically distinct and has defined all keywords to be lowercase [38]. C programming style encourages all identifier names (except macros) to be lowercase and to capitalize the first

character of important words in identifiers. Modula-2 on the other hand requires that keywords be in uppercase [77], but encourages user defined identifiers to be lowercase or capitalized like C's style. Ada does not require keywords to be of any case, but Ada's defining document has all keywords in lowercase and the user defined identifiers in uppercase [1], the opposite of Modula-2. Fortran 8x, like its Fortran predecessors, uses only uppercase characters [2].

The programming style that is used most for a given language is often the one that appears in the defining documents for that language. Figure 2.1 illustrates the four different styles of using character case while programming. The readability of the four versions of the program is affected by the familiarity of a programmer with a particular coding style, and the syntax of the language in which the example was written. The readableness of the code must be offset by the need to visually distinguish keywords, identifiers, and comments. The original version of the code was taken from a textbook on Fortran 8x programming [47, pp. 299].

The Modula-2 style is the best style to use where readability is concerned; the user defined identifiers, which make up the bulk of the text, are in the easy to read lowercase. The individual words in an identifier are capitalized to make each word standout better. The keywords are written using only capital letters which visually sets them apart from the rest of the code. Also keywords often need to be searched for and found quickly; studies have shown that uppercase characters can be used to draw attention to a limited subset of the text and to aid in searching [68]. Coming in a close second is the Ada style. Reserved words and identifiers are distinguished, but the bulk of the text is capitalized. Since comments are normally written in lowercase, to give them as little visual impact as possible, this style differentiates code and comments very well. The C style is probably third best in terms of readability, key words and identifiers are not distinguished by case, nor are the comments. However, important identifiers are capitalized which helps slightly. Fortran style, all capital

```
! Randomize the order of a sorted
! deck of cards
SUBROUTINE SHUFFLE(CARDS)
   INTEGER, DIMENSION(52) :: CARDS
   INTEGER LEFT, CHOICE, I, TEMP, R
   CARDS = (/ (I, I=1,52) /)
   DO LEFT = 52,1,-1
   CALL RANDOM(R)
      CHOICE = R*LEFT + 1
      TEMP = CARDS(LEFT)
      CARD(LEFT) = CARDS(CHOICE)
      CARDS(CHOICE) = TEMP
   END DO
END
```

Fortran 8x style

```
! Randomize the order of a sorted
! deck of cards
subroutine Shuffle(Cards)
   integer, dimension(52) :: Cards
   integer Left, Choice, I, temp, r
   Cards = (/ (I, I=1,52) /)
   do Left = 52,1,-1
   call Random(r)
      Choice = r*Left + 1
      temp = Cards(Left)
      Card(Left) = Cards(Choice)
      Cards(Choice) = temp
   end do
end
```

C style

```
! Randomize the order of a sorted
! deck of cards
subroutine SHUFFLE(CARDS)
   integer, dimension(52) :: CARDS
   integer LEFT, CHOICE, I, TEMP, R
   CARDS = (/ (I, I=1,52) /)
   do LEFT = 52,1,-1
   call RANDOM(R)
      CHOICE = R*LEFT + 1
      TEMP = CARDS(LEFT)
      CARD(LEFT) = CARDS(CHOICE)
      CARDS(CHOICE) = TEMP
   end do
end
```

Ada style

```
! Randomize the order of a sorted
! deck of cards
SUBROUTINE Shuffle(Cards)
   INTEGER, DIMENSION(52) :: Cards
   INTEGER Left, Choice, I, temp, r
   Cards = (/ (I, I=1,52) /)
   DO Left = 52,1,-1
   CALL Random(r)
      Choice = r*Left + 1
      temp = Cards(Left)
      Card(Left) = Cards(Choice)
      Cards(Choice) = temp
   END DO
END
```

Modula-2 style

Figure 2.1: Comparison of four character case programming styles

letters, is the worst style to read. The fact that lowercase comments are easily distinguished from the code itself is its one advantage.

On a dissenting note, the superiority of lowercase over uppercase is affected by the syntax of the language. Falkoff argues that identifiers should be uppercase in APL because there is a greater potential for lowercase characters to be confused with the many specialized operator symbols [20]. Uppercase characters provide a greater contrast with the operator symbols in APL than do the smaller lowercase characters; this makes the operator symbols easier to spot.

2.2 Formatting the Code

Programming languages are for the most part extremely lenient with how program text can be positioned. An arbitrary number of spaces, tabs, and carriage returns (white space) can usually be placed between any two programming language constructs. This freedom presents an opportunity for the programmer to format the code in a readable manner.

2.2.1 Indentation. Proper indentation of programs has a long history of research. Many of the later studies contradict earlier findings that indicated code indentation did not improve readability [71, 48]. Some of the earlier, controversial studies on indentation were flawed as non-block structured languages were used (Cobol and Fortran) or many `GOTO` constructs were used. Both led to test code that was non-structured and therefore any indentation was of little help. It is generally agreed now that indentation provides perceptual clues that aid the reader in understanding the block structure of a program. The fact that there was a controversy over the benefits of indentation in the first place indicates that many aspects of indentation were not understood. One of the better studies, done by Miara and others, shows that two to four spaces of indentation are optimal and that six spaces of indentation is below optimal, but is still better than no indentation at all [48].

In figure 2.2 the words **start** and **finish** represent opening and closing keywords (or symbols) of a control statement, and **block** represents one or more statements nested within the control statement. The three indentation styles differ only in the placement of the **finish** keyword. Again, styles attributed to a programming language were taken from defining documents of that language, and are usually the most common style used.

```
start                start                start
    block                block                block finish
finish               finish
```

Ada PL/1 Lisp

Figure 2.2: Indentation styles

The first indentation style enables the reader to quickly scan to the left and downward to find the closing keyword. This style is particularly well suited to Ada which has matching paired keywords: **if** ... **end if**, **case** ... **end case**, and **loop** ... **end loop**. The second indentation style, used by PL/1, enables the reader to quickly find the statement following the indented block. The closing keyword is 'hidden'. PL/1, in contrast with Ada, has only one closing keyword, **end**. It adds little information to understanding the code beyond the obvious fact that it closes a control statement so there is no need to draw attention to it. Modula-2 also has **end** as its only closing keyword but, because of their common roots in Pascal, Modula-2 uses the same indentation style as Ada.

The last indentation style is used mainly to keep the closing symbol (or keyword) as far away from the reader's main area of concentration as is possible. Lisp uses parenthesis,(...), as opening and closing symbols, and it is not uncommon to find five or six contiguous closing parenthesis. Because there can be so many parenthesis, so close together, the closing symbol in Lisp provides even less information to the reader than the closing **end** did in Modula-2. Many programmers

rely on their editor to briefly display the matching beginning parenthesis whenever the closing parenthesis is typed in. Unlike many other languages, the reader of Lisp code is uniquely forced to use indentation alone to determine block structure and only attempts to match the multitude of pairs of parenthesis as a last resort (usually when the code was indented improperly).

Code is indented for the benefit of the writer and the reader. Kernighan and Plauger, in their book *The Elements of Programming Style* offer the following warning: "Indentation must be done carefully, however, lest you confuse rather than enlighten." [39, pp. 146]. To emphasize the point they present a programming example with ill-chosen indentation followed by two improved versions. Figures 2.3a and 2.3b contain the same code, they differ only in the amount of white space used and the indentation method. Both figures use an acceptable method for formatting nested `if` statements. Figure 2.3b, however, is much more readable. The structure of the program is evident and the programmer can then see that changing the code slightly will create figure 2.3c, which is even more readable than the first two. Oman and Cook tested the three versions of code and found that the modifications to style and structure significantly increased the comprehension of the revised versions [52]. This example demonstrates that indentation rules should not be followed blindly. Each portion of the text must be examined and indented to bring out the similarities in the textual form and to reveal the underlying structure of the code.

2.2.2 Visual Blocks. It is interesting to note that these perceptual cues which the reader depends on are completely ignored by compilers. If a program's indentations do not match the underlying code it is possible that the compiler and programmer are reading two different meanings. Bhujade proposes that languages should be extended to support "visual block specifications" [8]. Statements with the same indentation level form an implicit block. The traditional form of creating a block, `start ... finish`, would override the implicit intent of the visually specified

```
IF A > B
   THEN S := 1
   ELSE IF A = B
      THEN IF C > D
        THEN S := 2
        ELSE S := 3
      ELSE IF C > D
        THEN S := 4
        ELSE IF C = D
          THEN S := 5
          ELSE S := 6;
```

Original code, 2.3a

```
IF      A>B THEN S := 1
ELSE IF A=B THEN
     IF C>D THEN S := 2
            ELSE S := 3
ELSE IF C>D THEN S := 4
ELSE IF C=D THEN S := 5
ELSE             S := 6;
```

Typographically modified code, 2.3b

```
IF       A>B             THEN S := 1
ELSE IF (A=B) AND (C>D) THEN S := 2
ELSE IF  A=B             THEN S := 3
ELSE IF            C>D  THEN S := 4
ELSE IF            C=D  THEN S := 5
ELSE                         S := 6;
```

Structurally modified code, 2.3c

Figure 2.3: Readability of nested `if` statements

block when necessary. Leinbaugh goes even further and argues that indentation alone is sufficient to represent a programs block structure and that compound statements and closing keywords are not necessary [45].

Using visual blocks to connote a program's structure makes proper indentation compulsory for the programmer and forces the program writer, reader, and language compiler to use the same syntax when parsing the program text. This form of block structuring will work well for small segments of code. If the blocks of code are very large, the nesting level becomes very deep, or the nesting level changes frequently, then the traditional method of block structuring may be more readable.

2.2.3 Horizontal Alignment. Another common method of formatting the program is to align code horizontally as well as vertically. When several contiguous program statements have the same syntactic forms, readability is increased by placing like operands or operators in columns. Figure 2.4 compares a segment of code, taken from the *Pascal User Manual and Report* [34, pp 60], and a horizontally aligned version. The aligned version is obviously easier and more pleasant to read. Figure 2.3 also contains good and bad examples of horizontally aligned code.

2.2.4 Code spacing. Another method that improves readability by restructuring the code is code spacing. For example, statements that perform a single logical action can be separated from the rest of the code by surrounding them with one or more blank lines. The reader can assign a single meaning to a group of statements and need not examine each statement individually to determine its purpose. If all statments are contiguous the reader has no visual break point to help separate and define the different actions that are taking place.

Similarly, statements that have a common property should be grouped together and separated from other statements. When constructs with a similar property are grouped together the reader can determine that property for one or more of

```
var ch: char;
    count: array['a'..'z'] of integer;
    letter: set of 'a'..'z';
begin  letter := ['a'..'z'];
   for ch := 'a' to 'z' do count[ch] := 0;
```

Original code

```
var ch    : char;
    count : array ['a'..'z'] of integer;
    letter: set of 'a'..'z';
begin
    letter := ['a'..'z'];
    for ch := 'a' to 'z' do count[ch] := 0;
```

Aligned code

Figure 2.4: Readability and horizontal code alignment

the constructs and immediately infer that the rest of the constructs have that same property. As an example, when global variable declarations are separated from local variable declarations the reader can easily see which variable has what properties. If the two types of declarations are intermixed the readers job becomes harder.

Code spacing can also affect how readable a single statement is. Figure 2.5 demonstrates the difference code spacing can have on readability by showing three versions of the same statement. The first version leads the reader to misinterpret the code. The second, while not misleading, is nonjudgemental as to the correct interpretation. The last version increases the likelihood that the reader will perceive the correct meaning.

`42+A / B-C`	Incorrect spacing
`42 + A / B - C`	Correct but nonjudgemental
`42 + A/B - C`	Proper spacing

Figure 2.5: Code spacing improves readability

Some languages, such as Cobol, require that the programmer place at least one space between operators and the operands. Generally features that try to force a particular style of writing upon a programmer are more burdensome than they are successful. An enhancement to the compiler, or some type of style checker, could easily search for such inconsistencies in the code and warn the programmer.

2.3 Naming Conventions

User defined identifiers usually represent the largest portion of any program text. If these identifiers are named poorly a person's ability to read and comprehend the program is greatly impaired. Conversely, if the names are well suited for the identifiers the readability of the program is enhanced. Knowing something's true name, as opposed to a name that does not fit it well, gives you power over that thing (this basic idea is often used in fairy tales and folklore; knowing a persons true given name gives you power over that person).

Textbooks on programming often contain very little information on the proper naming of identifiers; usually the most said about the subject is "names for identifiers must be descriptive". Programmers spend considerable time deciding on appropriate names for their identifiers but have very few guidelines to help them. Sometimes the languages themselves hinder the programmers efforts by limiting how long names could be; at one time Fortran names had a maximum length of six characters and Basic names could only be two characters in length. This limit on the length of identifier names is one reason why acronyms and abbreviations became so prevalent.

Wirth, in a paper on structured programming, notes "Our most important mental tool for coping with complexity is abstraction." [75]. The name of an identifier should be an abstraction of its purpose and use. Suitable abstractions remove the need to remember information about an identifier; anything worth knowing is contained in, or made obvious by, the identifier's name.

2.3.1 Abbreviations. An abbreviation is a new word that has the same meaning as an existing word. The new word is invariably shorter, but less readable than the word for which it stands. Some of the more common methods of creating an abbreviation are listed below:

Truncate	Eliminate letters from the end of a word.
Drop letters	Eliminate letters starting with the most frequently used English letters (vowels first).
Phonetic	Rewrite the word using a phonetic spelling.
Concatenate	Retain first and last letter(s) and eliminate letters from the middle of a word.
Natural method	Use any abbreviation method (usually one of the above) that appears to abbreviate the word the best.

Programmers often overuse (abuse) abbreviations, usually in an attempt to shorten the amount of typing they have to do when keying in a program. Many programmers succumb to the TLA (Three Letter Abbreviation) mind-set. Even words that have only four or five letters to begin with are commonly shortened to

three letters. The savings in typing time for most abbreviations is minuscule, and is more than likely lost to increased disciphering time later as well as the time it takes to decide upon the appropriate abbreviation to begin with.

Newsted phrased it well when he wrote "One programmer's mnemonic is another's gobbledegook" [51]. Often the programmer who created the abbreviation may not be able to remember what it stands for later. Most editors allow arbitrary character strings to be searched for and replaced; so even if an abbreviation is used to "save typing", it can and should be replaced with its more substantial, palatable namesake as soon as possible.

The reader of a program filled with abbreviations literally has to learn a new vocabulary in order to understand the program. New vocabulary words and abbreviations differ in content though. The new vocabulary word usually has a meaning different from other known words, enables certain concepts to be expressed better, and thus provides its own incentive to be learned and remembered. The abbreviation on the other hand is merely a poor replacement for the original word, imparts no additional meaning, and provides its own frustration whenever used and not understood.

Ehrenreich examines and synthesizes the results of over 20 different studies on 11 different abbreviation techniques [19]. Most of the studies dealt with encoding (creating) abbreviations as opposed to decoding (understanding) abbreviations. He came to the following conjectures which are supported by the studies:

- People do not encode words in the same manner even when using the same abbreviation technique.
- The 'Natural method' of abbreviation is not natural as there is little agreement on the correct encoding for any given word.
- Rule generated abbreviations are generally superior to the 'Natural method' when encoding.
- The simpler rule techniques are better than more complex ones when encoding.

- Truncation is as good or better than any other abbreviation technique when encoding.
- No abbreviation technique has been found to be superior for decoding.

Ehrenreich's conjectures can be summarized and restated so they have greater relevance to program readability.

- If abbreviations are not standardized different programmers will invariably encode the same word in several different ways thus decreasing readability even further.
- The 'Natural method' of abbreviation used by most people may not be the best technique to use, a simple rule method such as truncation, or dropping the vowels may be better.
- No known abbreviation technique will enhance readability over any other method; the best way to be clear and unambiguous is to not abbreviate at all.

If abbreviations are to be used, they should be used sparingly. Multiple abbreviated words per identifier and multiple abbreviated identifiers per statement read like a cryptographers nightmare. Carter suggests creating a list of approved identifiers and abbreviations before starting to program [13]. This will limit the number of abbreviations that could be used in a program, provide the reader with a look up table that matches abbreviations with their parent word, and aid in maintaining the consistency of abbreviations across multiple programming efforts. Each abbreviation placed in the list should be examined carefully. If an abbreviated word appears difficult to decode imagine how hard it will be for a person who did not write it.

Carter also offers several guidelines on choosing abbreviations.

- Use only one abbreviation, if any, per word.
- If an abbreviation is defined then use it consistently, and not the full word.
- Drop letters from the end of the word, and not the middle (truncation).
- An abbreviation must be at least three letters shorter than the parent word.
- Abbreviations for two different words should not be the same.
- An abbreviation for one word should not be a possible abbreviation of another word.
- An abbreviation for one word should not be the valid full name of another word.

Do not abbreviate words that occur seldom in a program; nothing is gained by the abbreviation. If a word is used in several identifiers, is used frequently throughout a program, and its abbreviation is easy to decode, perhaps the readability lost by using an abbreviation can be reclaimed in other areas. Using the same abbreviation throughout a program may significantly reduce the amount of code that must be scanned and make certain expressions and statements more concise. Perhaps some multi-line statments can be eliminated and some awkwardly long expressions can be shortened. The reader, spotting the same abbreviated word over and over, will decode it easier each time.

Some abbreviated words have become "standard" through their repeated use in programs and can thereby be safely used. The following example abbreviations fit that category, they are very easy to decode: `max`, `min`, `ptr`, `pos`, `col` (especially when used in conjunction with `row`), and `num`. Use an abbreviated word only if the factors in the two paragraphs above will offset the reduced readability caused by using the abbreviated word in the first place.

2.3.2 Meaningful Names. The name of an identifier should reflect its purpose; its usage and scope should act as modifiers to the name. An identifier's name must quantitatively differentiate it from all other identifiers active in the same scope. Its meaning must be clear and unambiguous given the context in which it is used. If an identifier's scope is global to a program, or it is "exported" from a module, its name must be very descriptive. If an identifier is local to a single subroutine a shorter, less descriptive name can be used safely.

The best name for an identifier is both descriptive and short. Usually the longer the name, the more descriptive it is; but longer names are also more awkward to use. Obviously, using shorter names will make the code more compact, but shorter names are invariably less descriptive. If an identifier is used mainly by itself, or with a few other identifiers, its length does not matter as much. An identifier that is often

combined with many other identifiers should be as short as possible. Mathematical functions like tangent and cosine are usually abbreviated as `tan` and `cos` because they are often combined with many other identifiers in expressions; the shorter names make them easier to read.

Figures 2.6a and 2.6b compare the readability of two different names for the same identifier: `I`, and `AircraftCounter`. In this example `I` is the more readable variable name, it is descriptive of its purpose within the program — to be a counter variable. Under different circumstances `AircraftCounter` may be the better choice, it is more descriptive in terms of the problem domain. The context in which an identifier appears can be used to eliminate redundant information [44]. The fact that aircraft are on radar, are arriving, and are being placed in a holding pattern adds no new information that could not be easily discerned by the nature of the program. Figure 2.6c gives a more readable representation by removing superfluous words.

Daniel Keller, in an article on naming conventions, offers several guidelines to follow when choosing names [37]. Keller believes that the names for type identifiers should be chosen first as the type name becomes the core of all other identifier names. Type names should be short, simple and generic, preferably nouns. Procedures imply action and their name should include a verb. The form "verb + type name" can be used if the procedure acts upon a specific type. Variables and function's should be named "adjective + type name" as they reference a specific value. Figure 2.7 contains examples of good names using Keller's naming conventions.

When a type forms the basis for variable, function, and procedure names it is easier to spot the connections between these different entities. Variables with the same type hold related values; the adjective modifiers in the names of the variables should specify that relationship. The type that is returned from a function is im-

```
FOR AircraftCounter = 1 TO NumberOfAircraftOnRadar DO
    IF AircraftIsArriving( AircraftOnRadar[AircraftCounter] ) THEN
        PlaceAircraftInHoldingPattern( AircraftOnRadar[AircraftCounter] );
    END IF
END FOR
```

Figure 2.6a, Long unwieldy identifier names,
each specified in terms of the problem domain

```
FOR I = 1 TO NumberOfAircraftOnRadar DO
    IF AircraftIsArriving( AircraftOnRadar[I] ) THEN
        PlaceAircraftInHoldingPattern( AircraftOnRadar[I] );
    END IF
END FOR
```

Figure 2.6b, 'I' has meaning in terms of the program,
not the problem domain

```
FOR I = 1 TO NumberOnRadar DO
    IF Arrival( OnRadar[I] ) THEN
        PlaceInHoldingPattern( OnRadar[I] );
    END IF
END FOR
```

Figure 2.6c, Removing redundant information from an identifier's name

Figure 2.6: Different aspects to consider when selecting an identifier's name

Types	**Procedures** *Verb* + **TypeName**	**Variables & Functions** *Adjective* + **TypeName**
Table	*Print*Table	*SymbolName*Table
Window	*Delete*Window	*Help*Window
Page	*Display*Page	*Title*Page
Address	*Change*Address	*Home*Address
FileName	*Open*FileName	*Log*FileName
MachineState	*Check*MachineState	*Current*MachineState

Figure 2.7: Sample identifier names derived from type names

mediately apparent from the functions name; the modifiers in the name can explain how the data being returned was derived.

In an earlier article on naming conventions, N. Anand states that the most difficult task in understanding a large program is the need for the reader to remember an immense amount of information about that program [3]. He argues that all identifiers in a program should be given a "functional description" (which is merely a statement as to the identifier's purpose in the program). If all identifiers in a program are then named according to their "functional description", the need to remember the purpose and usage of these entities is eliminated or at least significantly reduced. The program can be understood as it is being read.

Anand offers the following guidelines when naming an identifier after its functional description.

- All identifiers should be given a good functional description, if it is impossible to do so the identifier's purpose should be changed.
- Each identifier should perform only one specific action. If the functional description is a compound sentence, the identifier is performing multiple functions and should be changed.
- All relevant information concerning the identifier should be 'predictable' given its functional description, and the kind of identifier it is.
- A functional description should reflect what the identifier does, not how it does it, or the context in which it is done.
- The functional description should be clear and concise without violating the previous criteria.

The functional description of an identifier can be stated in terms of its purpose within the program or its purpose within the problem domain. In lower level subroutines that perform generic functions necessary in several different contexts, the functional description can be stated in terms of what actions the code is accomplishing. Similarly, counter variables that are used within small loops can be named 'I' and 'J' because their purpose within the program is very clear independent of the problem being solved. In higher level subroutines the functional description must be thought of in terms of the problem domain. The actual code may perform several

seemingly unrelated tasks when taken in isolation but in terms of the problem domain only one specific action is being accomplished. Similarly a record structure may hold several different pieces of information, its name should reflect an abstraction in the problem domain.

2.4 Comments

Comments are sometimes considered unimportant by programmers for a variety of reasons. The reasons given include:

- Comments have no more significance than white space; they do not affect the execution speed of a program or the accuracy of the results.
- This program is a test program (or a prototype) and will be used only once so comments are unnecessary.
- Comments obscure the structure of the code making the program harder to read.
- The program is written in Ada (or Cobol or Pascal or ...) which is a self documenting language so comments are unnecessary.
- Comments can always be added later when and if there is time; the main goal is to create a working program.

All these reasons contain just enough truth to convince the uninitiated. The following counterarguments, also true, should be more convincing.

- Programmers vastly overrate their ability to remember crucial details of a program.
- The time saved writing a program is often lost trying to understand it later (or rewriting it because it couldn't be understood).
- What is clear to the author of a program may not be as easily understood by the maintenance programmer.
- Programs have a tendency to remain in use longer than planned.
- Comments do not affect the meaning of a program, but they may aid the reader to ascertain the intended meaning.
- There will never be time enough to add comments later, so add them as the program is being written.

Programmers that do consider comments desirable can often overuse or abuse them. The code itself, sans comments, should be as self-documenting as possible. When numerous comments are needed to explain a segment of the program the code may be overly (and usually needlessly) complex; profusely commenting the

program will not make bad code any better. Before a comment is written, the code being documented should be examined to see if it can be rewritten using a clearer or different solution; often (as was seen in section 2.3) changing the names of one or more identifiers can improve readability enough to make a comment superfluous.

Every comment placed in the program should be there for a good reason just as every line of code in the program should be there for a good reason. The attention placed on comments should be the same as that placed on the code as incorrect comments are potentially as damaging as incorrect code. This implies that comments should be checked and tested for correctness when possible. Some languages have assertion commands (ANSI C) which act as runtime comments and are checked during the execution of the code. At the very least all comments should be desk checked.

Comments aid readability only when they add information that cannot easily be derived from the program text. Comments that paraphrase the source code distract the reader from the code, and can often hide the underlying structure of the code. Good usages of comments can include the following:

- Summarize the purpose of functions, procedures and large blocks of code.
- Highlight the logical content of a group of statements.
- Emphasize a crucial step in the code.
- Describe the precise meaning of an identifier or constant.
- Document otherwise hidden dependencies in the code.
- Inform the reader of design decisions and design rationale.
- Document the date and reason particular modifications were made.
- Clarify particularly obscure sequences of code.

Also necessary for readability is the visual separation of comments from code so that one can easily be distinguished from the other [79]. This has traditionally been accomplished by placing comments in separate blocks above, or to the right of, the code being documented. In both cases there should be sufficient white space between the comments and the code to visually separate them. In cases where white space is not used, comments that are sandwiched between lines of code or that are

aligned with the start of the next or previous line of code are especially distracting. Ledgard suggests using two dashes, --, in addition to the normal start of comment syntax, to clearly distinguish the comment from the code [44].

Sometimes during implementation and debugging, or when documenting modifications to a program, sections of code are commented out. If each line is commented out, individually, it is easy to determine what code will be executed and what code will not. However, if blocks of code are commented out using start and finish comment symbols like /* ... */, or { ... }, it may be difficult determine that code has been commented out; especially if the section of code is large.

If code is to be commented out for a long time, the writer of the code should make a greater effort to distinguish the commented out code from code that will be executed. Each line of code should have prepended to it some symbol marking it as being commented out. Most programming editors can be enhanced with a macro that will enable the easy placement and removal of these symbols.

2.5 Odds and Ends

2.5.1 To default or not to default. Every programming language allows defaults of one kind or another. Most defaults have to do with the properties a variable can have or data conversion rules between different types. It would be extremely unwieldy to program without these kinds of defaults. Fortran, for example, needs five pieces of information to fully declare a numeric variable [72]. Picture a programmer having to specify the following five attributes each time a numeric variable is declared:

- REAL, INTEGER, or COMPLEX
- DIMENSIONed or not
- COMMON or not
- EQUIVALENCEd or not
- PRECISION: single, DOUBLE or extended

Luckily, Fortran does not even allow the programmer to specify that something is not dimensioned, not common, not equivalenced, or single precision. Not having the above properties and being single precision are the defaults; this makes perfect sense as the majority of the variables declared would not need those properties. Not having to declare all those attributes saves the programmer time, reduces possible errors on input and makes the code more concise and readable.

In some languages, though, the defaults are not as sound. PL/1 has a multitude of data types and data type attributes; the rules in PL/1 for default type declarations and default type conversions are truly strange. Figure 3.12 on page 44 lists several unintuitive default type conversions that PL/1 performs. Imagine if the type conversions listed were what was actually wanted by the programmer; the difficulty for someone reading such a program should be obvious.

PL/1 also has some irregular rules for defaulting variable attributes. In PL/1 an undefined variable that starts with the letters 'I' through 'N' is defaulted to have a "scale" of FIXED and a "base" of BINARY; all other undefined variables are defaulted to have a "scale" of FLOAT and a "base" of DECIMAL. There are three ways to make the variable 'I' have the attributes FLOAT and DECIMAL: Explicitly declare 'I' to have those attributes, declare 'I' to have a scale of FLOAT and the base will default to DECIMAL, or declare 'I' to have a base of DECIMAL and the scale will default to FLOAT. Notice that to change only one of the original default attributes of 'I' the programmer must declare both attributes specifically. Taking advantage of strange, unintuitive defaults makes reading and understanding the code much harder.

Using the more intuitive defaults saves the programmer time and makes the resulting code more concise with little loss in readability due to information not being present for the reader. Using ill-conceived defaults hides the true meaning of the program from the reader. Whether to use the defaults a given language provides is a decision that must be made based on the reasonableness of the defaults. It should

be possible to construct a translator program, for languages that have particularly troublesome defaults, which when given code as input returns a listing of all the defaults used in the code. The places where defaults are especially unintuitive can then be made more explicit by the programmer. In addition when using such a system, it is possible that many subtle errors may be found in the code that are caused by unusual default actions which would otherwise remain unfound, or be found later at great cost.

2.5.2 Magic numbers. Magic numbers are numbers that appear in the executable portions of the program text and have some influence on the computation. These numbers usually represent some logical limit, constant, or condition such as the maximum size of an array, the number of characters allowed for an input string, or an error condition like file not found. The numbers are 'magical' because the code they are in continues to work even when the meaning behind the numbers has been forgotten (What is this number for and how does it make the program work? I don't know, it's magic!).

It has long been recognized that the use of magic numbers is a poor programming habit. When a constant needs to be increased or decreased, it can be difficult to find all occurrences of that number. The number may occur in several different files, or may be one greater or smaller than normal because of the context in which it is used. Often the same number may represent different constants causing additional difficulties when trying to modify one constant and not the other. If any number other than 0 or 1 appears in the body of the code it should be scrutinized closely to see if it can be removed.

The most common solution to these problems (for those languages that have global constants) is to use a named constant in place of the magic number. Changes to the value of the named constant is then limited to a single place and the meaning of the magic number is contained in its name. Some languages have special

features that eliminate the need for certain types of magic numbers. Languages that allow the creation and use of enumeration types can use them in place of magic numbers when the actual value of the number does not matter. For example, a meaningful error such as `FileNotFound` can be reported using an enumerated value instead of an error code number such as '36'. Ada has an apostrophe operator that accesses attributes of a type and can eliminate another category of magic numbers. For example, the commands *ArrayType*`'first` and *ArrayType*`'last` represent the upper and lower bounds of a given array type; thus named constants for a array boundaries need not be used in Ada.

A byproduct of the use of named constants, enumeration types, and type attributes is the increased readability of the source text. A reader should never have to unravel the meaning of a magic number. Using the above methods, the exact value of the number is placed elsewhere while the meaning of the number, contained in its name, remains part of the program text.

CHAPTER 3

PROGRAMMING LANGUAGE DESIGN

There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs. — **Lawrence Flon**

The readableness of a programming language or a program is often influenced by personal bias. Is Cobol readable? Is Lisp or APL? Terseness should not be associated with unreadableness just as verbosity should not be associated with readableness. How readable the programs from a particular language are depends upon who you ask and how long they have spent reading listings in that language. A person who is familiar with programming, but unfamiliar with a particular language, may be in a better position to judge.

Certain design guidelines and principles are universal and language designers can use them to improve their language. The syntax of a language should be suggestive of the semantics. Syntax features should aid in the recognition of basic program concepts, not hinder the reader by obscuring the meaning. Similarly, the semantics of a programming language should help aid the understanding of programs, and not hinder it. Unfortunately the syntax and semantics of a language are often looked upon as a convenient device to structure code for the translator to parse, and not as a tool to aid in reading and understanding the code. This chapter examines how the design of a programming language can influence how readable the programs written in that language are.

3.1 Basic Language Tokens

The keywords and symbols of a language are the building blocks with which programs are built; it is difficult to construct an understandable program with uncomprehensible basic tokens. Because of this, and the fact that the syntax cannot be changed later, the language designer must make a great effort to insure that the keywords and symbols of a new programming language are clear and unambiguous. The semantics of a construct, the context in which it is used, how often it will be used, other keywords and symbols in the language, and even other programming languages all affect how readable a particular keyword or symbol is.

3.1.1 Keywords. Many of the same criteria given earlier in section 2.3 for naming identifiers should be applied to naming the keywords of a language. Keywords should not be abbreviations, or have abbreviations. As an example, figure 3.1 contains a list of PL/1 keywords and their abbreviations as defined in the language [72]. The two columns of words have deliberately been mixed up. It will be difficult for someone unfamiliar with PL/1 to connect the parent words to their abbreviations on their first try. The choice of abbreviations used in the PL/1 language follows no discernible pattern that would aid the reader in decoding them. Two of the abbreviations, `DEF` and `DEC`, could match several different keywords. The ability to abbreviate the keywords of a programming language is a useless feature which hinders readability. In addition, seeing and using abbreviated keywords may influence the programmer to use more abbreviations in her code as well.

Assembly languages are also notorious for having cryptic abbreviated command instructions. This is a byproduct of having a large number of similar instructions and historical precedence. Some abbreviations, such as using `JMP` instead of `JUMP` and `MOV` instead of `MOVE`, are fairly benign abbreviations and easy to decode, but it is difficult to justify the removal of just a single character. Many other languages abbreviate their keywords also. C saves three characters by using `struct`

OVERFLOW	DEF
CHARACTER	ATTN
DEFAULT	CPLN
DECLARE	OFL
COMPLEX	DEC
DEFINED	DFT
COMPLETION	CPLX
ATTENTION	DCL
DECIMAL	CHAR

Figure 3.1: PL/1 reserved words and abbreviations (Not matched)

instead of `structure`, and saves two by using `extern` instead of `external`. Pascal abbreviates the word 'character' as `chr`, an intrinsic function that returns a character given an integer, and `char`, the character data type. In addition, Pascal has two intrinsic subroutines, `eoln` and `readln`, where `ln` decodes to `line`; what does the math function `ln` decode to?

Keywords, like identifiers, should be named after the abstraction they represent, and not its implementation. The Lisp commands `car` and `cdr` stand for 'Contents of the Address part of Register number' and 'Contents of the Decrement part of Register number' respectively [46, pp. 175]. Not only are the commands named by how tasks are being accomplished instead of what is being accomplished, the command names are tied to a particular machine's instruction set. Very few people know the original meanings of `car` and `cdr`. The preferred names for these commands, `head` and `tail`, describe the functionality better. The names for these functions in Common Lisp, `first` and `rest`, are even more intuitive.

Using the reverse of a keyword as an additional keyword was a 'clever' idea on the part of the Algol68 designers, but does nothing to enhance readability. Algol68 has the following keyword pairs: `if ... fi`, `do ... od`, and `case ... esac` [67]. These are nonsense words which have no meaning. How do the following sound: `for ... rof`, `while ... elihw`, `begin ... nigeb`, `function ... noitcnuf`, and

`procedure ... erudecorp`? The ACM Sigplan journal had a series of articles and letters to the editor in which the merit of reverse keywords were debated [27, 26, 35]. The consensus of these articles can be summed up by the commented program text handed in by one of Knuth's students: "`esac; comment bletch tnemmoc;`" [40].

A much clearer syntax that can be used in place of reversed keywords is that used by Ada: `if ... end if`, `loop ... end loop`, and `case ... end case`. The closing keywords are longer, but have meaning. Modula-2 took a step backward by generalizing all the closing keywords into the single keyword `end`. This specific generalization removes information that is helpful to the reader and offers only the savings of not having to type in an additional two to four characters (six if you count `end record`).

In many programming languages the keywords are not reserved and the same keyword word can be used as an identifier as well. In a few languages textually distinct classes of identifiers can also share the same word. These are useless features that cannot be justified; of all the thousands of words possible, why decrease readability by assigning two or three distinct meanings to the same word. As an extreme example, the freedom to overload identifiers and keywords permits barbarisms such as the one depicted in figure 3.2 [66, pp. 13].

```
if if = then                          DO 5 I6 = 1.34
   then then = else;            14    FORMAT(I6) = I
        else else = else;        5    END = K*I
          PL/1                            Fortran
```

Figure 3.2: Overloaded identifiers and keywords

3.1.2 Symbols. Symbols should be used in place of keywords only when readability is enhanced. This occurs when the symbol is an intuitive match for its meaning within the program or when the symbol is used so often that the textual brevity of the symbol makes the code more readable than a longer, more

understandable keyword would. If either of the above conditions does not exist then a keyword should be used instead.

Symbols have traditionally been used as mathematical functions, separators within statements, and statement terminators or separators; the use of symbols within statements is covered in the next section. A symbol that represents a mathematical function is most readable when the symbol is in some way analogous to that function. For example, the math functions "plus", "minus", and "less than" are easily decoded from "+", "-", and "<" since the relationships between the symbols and the functions were learned by most people in grade school and have become second nature to them. It is unfortunate that the computer keyboard and the ASCII character set are derived from the old fashion typewriters as the usual symbols for multiply and divide, "×" and "÷", are not available. Their 'replacement' symbols, "*" and "/", have become generally accepted though. One can argue that they are better choices because "×" might easily be mistaken for the characters "X" or "x", and "/" is highly reminiscent of the horizontal bar in fractions like $\frac{1}{2}$ (people often write fractions using a "slash", 1/2, also).

Because there is a general scarcity of available symbols many symbols are overloaded and often two or more symbols are combined to represent new functions. The equals sign, for example, is often used both for assignment and for testing equality. Pascal and Ada use "=" for equality and ":=" for assignment while C uses "==" and "=" respectively. In C where assignments can be included in expressions, the similarity of the two forms often cause many errors which are not detectable by the compiler. Better choices for an assignment symbol are "←", or "⇐", had they been available.

Some symbols, through repeated use in many programming languages, have become all but synonymous with a specific operation. Whether or not these symbols are the best ones to use for those operations is no longer an issue. If a new language

uses a different symbol for the same operation it runs the risk of decreasing readability (and writability) by going against the de facto standard. The dot symbol, for example, is used by many languages (Ada, C, Modula-2, Pascal) to signify a particular field within a record structure (*Record.Field*). Fortran 8X's designers choose to ignore 'tradition' and use the percent symbol, (*Record%Field*). Their decision might have been influenced by the fact that the dot symbol is already used in the logical operators (`.GT.`, `.LT.`, `.GE.` ...) and overloading it might have caused readability problems as well. Fortran 8x was also extended to allow programmers to use the more traditional logical operators (`>`, `<`, `>=` ...) so the argument that readability would have been decreased had the dot symbol been used is not as strong.

New languages should depart from existing languages if the new features will improve the language (otherwise why design new languages). Herriot, in an article titled "Towards the Ideal Programming Language", argues that readability would be enhanced if the "`'s`" operator or the "`OF`" operator were used instead of the dot symbol [28]. Figure 3.3 compares the four different 'field' operators discussed above.

`Flight.DepartingTime.Hour`	Traditional
`Flight%DepartingTime%Hour`	Fortran 8x
`Flight's DepartingTime's Hour`	`'s` operator
`Hour OF DepartingTime OF Flight`	`OF` operator

Figure 3.3: Comparison of four different 'field' operators

The APL language is the ultimate symbol language. It was originally designed to be a unifying mathematical notation, and was only implemented on a computer as an afterthought [33]. APL is one of the few languages to free itself from the confining ASCII character set, it uses almost 70 different symbols in addition

to the alphanumeric symbols. To complicate matters further, many of the symbols represent two functions, a monadic and a dyadic. Taylor argues that APL is hard to learn because mathematics is hard to learn and that the individual and not the language is the reason people do not become APL literate [63]. If a person understands the mathematics behind APL it should be easy for him to read and write code.

The problem with understanding APL is that associating the symbols with the appropriate functions is difficult; many of the symbols and functions match each other well, but many others do not. Perhaps if some of the lesser-used and some of the less intuitive symbols were replaced with keyword commands the end result would be more readable. For example, some languages such as C use the percent symbol, "%", as the modulus function, while others such as Ada have an intrinsic keyword function defined.

When obscure keywords and symbols are combined with cryptic identifiers, programs are even more difficult to comprehend. The arguments that other people have learned the idioms of a language and that people who want to program in it will continue to learn the language are not sufficient reasons for having produced a poor language design. The cryptic command language of the UNIX operating system is a prime example of this shortsighted mindset. At one time many people argued against the use of decimal numbers in place of octal numbers as constants in programs for similar reasons [5, pp. 129].

3.2 Statement Structure

A statement is the basic unit of information within a programming language. Some statements are declarations, some are executed, and some contain other statements within them. One can consider an entire subroutine to be the declaration statement for that subroutine. One of the decisions that needs to be made concerning statement syntax is how to distinguish when one statement is finished,

and the next statement starts; this is important to the compiler, the programmer and the reader. Two of the most common methods of marking statements are the statement terminator and the statement separator.

3.2.1 Statement terminators versus statement separators. A statement terminator is a symbol which ends a statement. A statement separator is a symbol which lies between two statements, similar to how a comma separates the items in a list. Statements that end in a keyword should not have to require an additional termination or separation symbol as the closing keyword can serve the same function (PL/I choose to ignore this and requires many unnecessary termination symbols). Languages using either method of marking statements usually allow multiple statements per line of text as well as multiple lines of text per statement. By convention, statement terminators are placed at the end of a statement (or on a line by themselves if a keyword) and statement separators are placed at the end of the first of two statements. Because the two methods are so similar (they often use the same symbol, ";", as well), it is easy to confuse them.

While the rules for both methods are easy to grasp, programmers have a much harder time programming with statement separators. In a study comparing language design issues, statement separator compiler errors were shown to occur an order of magnitude more frequently then statement terminator compiler errors [24]. Pascal uses statement separators; placing a semicolon at the end of a statement in certain contexts will cause a syntax error. Modula-2 (which could have been called Pascal-2) also uses statement separators, but fixes the problem above by allowing statements to consist of no symbols at all. Thus statement separators can be placed after every statement in Modula-2 just as if statement terminators were required. An error will still occur, in both Pascal and Modula-2, if a statement with no separator symbol following it is moved from a context where a separator symbol is not required

to one where it is. Thus statement terminators appear to be the better of the two methods as they are always used in a context independent manner.

A third method of distinguishing statements, differing in philosophy than the two methods seen so far uses no symbols to terminate or separate statements. Statements are terminated implicitly by the end of the line. This method does not allow for multiple statements per line or multiple-line statements unless specific symbols for these constructs are used. For example, in Fortran 8x a semicolon is placed between multiple statements on a line to act as a separator and an ampersand character, "&", is placed at the end of each line being continued. It would be interesting to compare this method and statement terminators to see which is the most readable.

3.2.2 Closing Keywords. Some statement constructs, like the `if` and `while` statements in Pascal, have no closing keyword. This leads to several problems when writing and reading programs using those statements. The first is that a second statement cannot be added to the body of the construct by itself. The addition of a second statement requires that both statements be surrounded by a `begin ... end` block, as in figure 3.4. In an article that rationalizes many of Modula's design decisions, Wirth states that he choose to include closing keywords to eliminate ever needing additional bracketing keywords when inserting a statement into a program [76].

```
if <expression> then
    <statement>
```

```
if <expression> then
    begin
        <statement>;
        <statement>
    end
```

Figure 3.4: `Begin ... end` must be added with a second statement

A second problem caused by `begin ... end` pairs is that they tend to proliferate in most programs. The `begin ... end` pairs act as syntactic 'noise'; the numerous occurrences of the keywords quickly become distracting and can obscure

the flow of control from the reader. Deciding upon useful, uniform indentation and keyword placement conventions for the multitude of **begin ... end** pairs is also difficult. One style of programming espoused by Pascal programmers is to always include the **begin ... end** pairs and comment the **end** keyword. This style of programming simulates a closing keyword, and mitigates the unwieldiness of the **begin ... end** pairs as well as providing additional helpful information for the reader. This style is less helpful when the **else** clause is present, as in figure 3.5.

```
if <expression> then begin
    <statement>
end {if}
```

```
if <expression> then begin
    <statement>;
    <statement>
     end {then}
else begin
    <statement>
end {if}
```

Figure 3.5: **Begin ... end** pairs increase the syntactic 'noise'

A third problem is illustrated by the difficulty beginning programmers have in understanding why it is not legal to include multiple statements between the **then** and **else** keywords without enclosing them in a block. When read, the program looks correct to them, why does the compiler object? The problem is that without a closing keyword the compiler, and the reader, can never be sure that an **else** will not occur later in the program. Thus, until the end of the current block is reached it would be unknown whether one statement, or all intervening statements, need to be executed when the expression in the **if** statement is true.

```
if <expression> then
    if <expression> then
        <statement>
else
    <statement>
```

```
if <expression> then
    if <expression> then
        <statement>
    else
        <statement>
```

Figure 3.6: Dangling **else** problem

A fourth problem with not having a closing keyword is the dangling else problem: If there are two nested **if** statements, and a single **else** clause, to which **if** statement does the **else** clause belong? The version on the right in figure 3.6 is indented correctly according to Pascal semantics; the **else** clause belongs to the closest earlier unbound **if** statement which is at the same nested (block) level. However, a reader spotting the version on the left would most likely take it at face value and assume the **else** clause belongs to the wrong **if** statement. If the meaning implied by the version on the left is what is really wanted, an additional **begin ... end** block is needed even though it is enclosing only a single statement, as in figure 3.7.

```
if <expression> then begin
    if <expression> then
        <statement>
    end {if}
else
    <statement>
```

Figure 3.7: **Begin ... end** can sometimes be necessary around a single statement

A fifth problem with not using closing keywords is that it is easy to mistakenly execute a null statement. The two examples in figure 3.8 each have an extra semicolon (enclosed in a box) which change the meaning of the code drastically. An **if** statement with a null **then** clause and no **else** clause is meaningless. However, a **while** statement with a null do clause makes perfect sense if the **<expression>** portion of the **while** statement is a boolean function which if called enough times will eventually become false. The textual smallness of the extra semicolons, and the indentation of the code, obscures the true meaning of the code.

Lest you believe that all the benefits in readability lie on the side of having closing keyword, examine figure 3.9. Not having a closing keyword makes the code more concise when only a single statement follows the control construct. Attempts to place a construct with a closing keyword all on the same line in these cases is not as

```
while <expression> do;
    <statement>
```

```
if <expression> then;
    begin
        <statement>;
        <statement>
    end; {if}
```

Figure 3.8: Null statement problem

appealing, "`if <expression> then <statement>; end if`". The C programming language uses brackets, {...} , to mark a compound statement. These symbols are not as obtrusive as the `begin ...end` pairs; the "syntactic noise" of having a multitude of brackets is considerably less. All the other problems with not having a closing keyword still apply to the C syntax though.

```
if <expression> then <statement>;
```

```
while <expression> do <statement>;
```

Versus

```
if <expression> then
   <statement>;
end if
```

```
while <expression> do
      <statement>;
end while
```

Figure 3.9: Compactness of constructs that have no closing keyword

3.2.3 Intermediate keywords versus intermediate symbols. Intermediate keywords and symbols are a necessary notation that separate the different clauses within a statement. A language designer must choose whether to use keywords or symbols within a statement. Symbols take up very little space, but must be translated to have meaning. Keywords can be read as is, but occupy a larger area then a symbol would. Studies have shown that a reader is guided through the program by 'beacons' [12] in the language, readability is improved in most cases by using a notation that will catch the readers eye.

Figure 3.10 compares the use of intermediate keywords and intermediate symbols in two common control constructs, the **if** statement and the **while** statement. The statements on the left (C syntax with capitalized keywords) use intermediate symbols to separate the **<expression>** clause from the **<statement>** clause. The arguably more readable statements on the right use intermediate keywords which eliminate the need for the parentheses [42].

```
IF ( <expression> )                IF <expression> THEN
    <statement>;                       <statement>;

WHILE ( <expression> )             WHILE <expression> DO
    <statement>;                       <statement>;
```

Figure 3.10: Intermediate keywords versus intermediate symbols

A more convincing example, figure 3.11, compares a **case** statement using Modula-2 syntax and the author's own *semantically equivalent* **when** statement. The Modula-2 **case** statement uses a colon symbol to separate the item label from the item statements, and uses a vertical bar symbol, "|", to separate an item's statements from the next item. The **when** statement uses only keywords to separate the different statement clauses. A reader's eyes are immediately drawn to the capitalized keywords in the **when** statement where as the colon symbols and vertical bar symbols in the **case** statement are easily overlooked.

```
CASE <expression> OF               WHEN <expression>
    <item> : <statement>;              IS <item> THEN <statement>;
             <statement>|                             <statement>
    <item> : <statement>|              IS <item> THEN <statement>
    <item>,                            IS <item>,
    <item> : <statement>                  <item> THEN <statement>
END CASE                           END WHEN
```

Figure 3.11: **Case** statement with symbols versus **when** statement with keywords

3.3 Syntactic Subtleties

Every programming language has syntactic subtleties which cause even experienced programmers to make errors. These types of errors are syntactically correct, appear to accomplish what the programmer wanted, yet perform some other, wrong action. Programming mistakes that result in compiler errors are annoying, but relatively easy to find. Programming mistakes that appear to be correct, and that are syntactically correct, can be very difficult for a reader to spot. To a language compiler there are no language ambiguities; the compiler will compile error-free code, correctly (according to its definition), the same way each time. To the reader, a particular piece of code may be ambiguous; the reader may believe the code will perform a specific action while the compiler turns the code into something completely different. Often, a reader must be aware of these syntactic subtleties and be actively looking for them in order to find them; the compiler usually offers no support.

The number and type of these errors are endless. It would be virtually impossible to design a substantial language in which none of these types of errors could occur. The best a language designer can do is to be aware of the types of syntactic subtleties that have caused errors in other languages and, if possible, avoid them. While a complete list of the syntactic subtleties in existing languages is beyond the scope of this paper, the following paragraphs attempt to categorize the types of problems that can occur.

A common source for errors in many languages is when the addition or deletion of a single character changes the meaning of the code but still results in a well formed program. In languages where variables do not have to be explicitly declared (Fortran for one), every misspelled identifier means a new variable will be implicitly declared. Very few compilers or run time systems check to see whether a variable has been assigned before being used or is assigned but never used; therefore the program will compile and run but will not give the results the programmer

wanted. In C, the statement "`a / *b`" means divide "`a`" by whatever "`b`" is pointing to. The statement "`a /*b`" means "`a`" followed by a comment.

Two more examples of this type of error include the null statement problem described in section 3.2.2 and depicted in figure 3.8 and the problem of the missing comment delimiter. If a closing comment delimiter is forgotten, the comment statement extends itself through all intervening code until the next closing comment delimiter is found. As was mentioned previously in section 2.4, the most readable comment syntax is one that is terminated by the end of the line. Alternatively, the bracketed comment syntax could be modified to allow for nested comments and the compiler's error checking extended to flag any unbalanced comment delimiters in the same way that unbalanced parenthesis are flagged.

Another category of errors occurs when a programmer uses the freedom of form within a language to obscure the true meaning of a statement and make a different, wrong meaning appear to be correct. For example, most languages allow the programmer to use any type of indentation, and any amount of white space desired. The dangling `else` problem depicted in figure 3.7 is one case where improper indentation deceives the reader into believing the wrong meaning. As was mentioned in section 3.2.2, a partial solution to this problem would be for the `if` statement to use a closing keyword; this would at least make the improper indentation more obvious. A case where ill thought white space deceives the reader is depicted by the ease with which mathematical expressions can be perverted. Does "`47+38 * 22`" mean "`(47 + 38) * 22`" or "`47 + (38 * 22)`" ? The lack of white space around "`47+38`" deludes the reader into believing that the addition will take precedence over the multiplication. In both the above examples the compiler (or some other language preprocessing tool) could be extended to check for inconsistencies in formatting and warn the programmer that the code is suspect.

Another common source for errors occurs when two distinct constructs that are similar in form can be used in the same context. The similarity problem of the assignment symbol, =, and the equality symbol, ==, in C described in section 3.1.2 is one example. Possible solutions include changing the symbols used in each case, making them dissimilar; extending the compiler to offer a warning whenever an assignment symbol occurs in an expression; or changing the language by making assignment statements within expressions illegal. As an aside, many programming shops do not allow their programmers to use assignment statements within expressions, and many programmers have stopped using that feature on their own. The problems still occur though as there is usually no compiler (or lint) support to help them enforce these rules.

```
n * .8                    actually means n * .7875
   (Best possible representation of .8 in four bits)

25 + 1/3                  yields 5.33333333333
   (Operator precedence is strictly left to right)

.1**1                     does not equal .1***+1
   (Exponentiation operator has many other weird properties as well)

DO I = 1 TO 32/2 ;
  <statements>            is executed zero times
END;          (Type conversion problem)
```

Figure 3.12: PL/1 subtleties which cause errors

Many times, the programming language itself contains syntax and semantics which go against both traditional thinking and common sense. This type of language subtlety will automatically cause many errors as the syntax and semantics of the language combine to obscure the true meaning of the program from the reader. PL/1, as demonstrated in figure 3.12, is probably the definitive language for these types of errors. The first 'bad' example comes from the article "Language Features

that Aid Debugging" by Gimpel [25]; the rest of the examples come from the article "Teaching the Fatal Disease" by Holt [30]. There is no easy way to stop the reader from misinterpreting these language constructs (or the programmer from misusing them). The only sure solution is to change the definition of the language to get rid of these inconsistencies.

Another category of syntactic subtleties can be caused by non-uniformity in the programming language. One such problem occurred recently, 15 January 1990, when AT&T experienced software problems which disrupted phone services across the nation. The errant code, written in C, contained a `switch` statement which contained an `if` statement which contained a `break` statement [50]. The `break` statement was meant for the `if` statement but "broke" the `switch` statement instead. A `break` statement can only be used within an iteration construct (`do`, `for`, and `while`) and a `switch` statement [38]. The programmer's assumption that `break` statements also worked within `if` statements, while erroneous, was a simple, natural extension as other control statements can use it. If `break` statements (like `continue` statements) had been allowed only within iteration constructs, and a different keyword was used for terminating `switch` statements, the `break` statement would not have appeared to be so universal a construct. A different and more general solution would be to extend the syntax for the `break` statement to `break` *construct*, where *construct* could be `do`, `for`, `while`, `switch`, or `if`.

Perhaps the most famous error caused by a syntactic subtlety was the missing comma which (reportedly) caused America to lose its first space probe to Venus [29]. Figure 3.13 depicts the wayward line of code as the programmer wrote it, as the compiler interpreted it, and as the line should have been written. Two syntactic subtleties of Fortran combined to cause this error and make it undetectable to the reader: Fortran's ignorance of blank characters, and the fact that identifiers do not need do be declared in Fortran.

`DO 17 I = 1 10`	`DO17I = 110`	`DO 17 I = 1, 10`
(As Written)	(As Interpreted)	(As Desired)

Figure 3.13: Syntactic subtlety in Fortran code

Most errors caused by syntactic subtleties in a language can be eliminated, or at least substantially reduced, by the methods talked about above. Simply removing the offending 'feature' or modifying the language syntax to limit the occurrence of the error is not possible for existing, established languages. Only future languages may benefit by language designers taking those measures. Extending the compiler to offer a warning when the spurious situation occurs is also a possible but highly unlikely solution (unlikely because compilation speed and executable code efficiency is usually the compiler writer's main concern, not error diagnostics; that is what debuggers are for). In an ideal programming language, easily made errors would result in a syntactically incorrect program, and warnings would be offered by the compiler if the meaning of the code is suspect.

3.4 Programming Language Abominations

Certain language features and constructs should not be used as the programs that result from using them are invariably an order of magnitude more difficult to read and understand. Banning certain constructs does not limit the *number* of problems that can be computed. Banning these constructs only limits the *way* they can be computed. But how things are computed does make a difference; some methods are easier to read and comprehend then others. Programs written in languages without the features mentioned in this section will be much more readable and understandable than programs which depend on those features.

The particular features being held up as 'bad' are actually less important than the reasons for which they are being held up. Understanding these reasons can help the language designer to avoid other similar language features and constructs

not mentioned here that have similar properties and/or drawbacks. If a construct is fraught with these problems but must be included in the language then the language designer should make as great an effort as possible to counter or minimize the effects of the problems.

One of the greatest mistakes a language designer can make is to design a feature whose only purpose is to save the programmer time. These features have a tendency to become major drawbacks as they often eliminate redundant information that is necessary for readability purposes or error checking by the compiler. For example, implicit variable declarations, used in Fortran and PL/1, attempt to save the programmer from having to declare the variables of his program. This small savings in typing time does not out weigh the problems created when a variables name is mispelt. Compilers and programming checking systems (lint) can help find some of these mispelt identifiers by warning the programmer if a variable was not assigned before it was referenced or was not referenced after it was assigned, but in general the burden of these types of errors falls on the programmer. Programming is inherently difficult as programs must be exactly correct and error free; features that loosen programming requirements must be examined closely in case subtle problems can result.

Following a thread of control through the program is one of the main ways in which a reader can understand what a program is doing. Language features that hinder the reader from following a thread of control are highly suspect. Using variables in place of labels, as PL/1 does, is one such construct. Finding the label that a `GOTO[B` construct jumps to is not very difficult when the label is explicit. If a variable is used instead though, the value that variable currently holds must be determined before the reader can discern where control passes to next. Using a variable to store a label allows a control decision to be made elsewhere in the

program than where the decision is implemented. This decreases the 'locality' of the decision and increases the coupling between two otherwise disjoint pieces of code.

Some language features, while extremely powerful, are almost impossible to use in a safe manner. The subtle nuances of the errors that can occur when using such a feature are sometimes not worth the abilities conferred by the feature. An example of this feature is the "by-name" parameter in Algol. It is impossible to write a swap routine (a routine that when given two numbers swaps their values) if the arguments are being passed in by-name [22]. Passing parameters by-name is an extremely potent ability which can be helpful in solving many problems. Passing a function to a procedure can solve that same class of problems in an arguably more readable fashion, without the side effects which by-name parameters can have.

Constructs which act differently depending on the contexts in which they are used are another source of programming problems. A reader of the program cannot determine what the construct does independent of some outside influences. The epitome of these type of constructs is the dynamic binding of variables that Lisp allows. Other more mundane examples are the overloaded symbols which have different meanings in different contexts.

Some languages attempt to be all things to all people, this leads to extremely large unwieldy languages. Any craftsman must be able to understand his tools completely in order to use them effectively; this means that programming languages should for the most part be kept simple [29]. PL/1 supports an excessively large number of data types each with an accompanying number of properties and type conversion rules and operator rules. In early versions of PL/1 arithmetic could be performed in pounds, shillings, and pence; in fact PL/1 finally eliminated sterling fixed-point constants only after the British empire did [56].

It is difficult to classify a language feature as being so troublesome that it should never be used in a programming language again. The examples given

previously are fairly safe to classify that way as none will probably ever be added to a new language again, especially in the forms as they exist in the languages cited. If a language feature is a true abomination it will disappear in time through natural selection (or de-selection as the case may be). Again, the point of this section was to categorize some of the problems caused by certain poor language features making programming language users and designers aware of them. The purpose was not to start a 'witch hunt' or continue the "<Insert Programming Language Feature here> Considered Harmful" thread of dialogue.

CHAPTER 4

PROGRAM PRESENTATION

You can't make a silk purse out of a sow's ear —**Jonathon Swift**

Programs typically are typed in as ASCII text, and the same text that is compiled and executed is read via a simple text editor or printed out and read as is. This does not have to be the norm. What if the computer itself were able to improve the "style" of the program and make it easier to read? What if other appearance aspects could be changed, or the code syntax modified for the better, or additional information added to supplement the code? Program presentation systems do just that. They modify the existing program in ways that make the code easier to read and the true meaning of the program clearer. The output of a program presentation system is meant specifically for a human reader.

4.1 Reformatting the Code

This aspect of improving the program presentation parallels the section titled "Formatting the Code" in chapter 2. The changes that are made to the code of a program are those that are allowed under the syntax rules of the programming language. Any of the formatting changes the programmer can make a reformatting program can make, and the output will still be compilable (unlike other presentation methods covered in succeeding sections which either change the syntax of the language or turn the program into a typeset document).

Program presentation systems which (for the most part) reformat program code according to some indentation and style standard are collectively known as

prettyprinters. Most of the benefits associated with reformatting programs for readability can be realized by these systems. In fact, in some ways reformatting programs surpass their human counterparts as prettyprinters can detect logic and syntax errors by flagging abnormalities [16].

One of the difficulties with implementing a prettyprinter is that determining the most readable format for a program, as was discussed in section 2.2, is not necessarily simple. There are many specific factors of the code being formatted that prettyprinters must take into consideration. For example, if code is indented too far to the right, longer lines may need to be split into two or more lines. A prettyprinter must decide if a line should be split or if the indentation scheme should be changed to make the line fit. If a statement is to be split, where should the break be made? How is the back half of the statement aligned beneath the start of the statement? Where should comments be moved to if they no longer fit on an indented line? When should white space be introduced into the middle of statements to make them more readable (figure 2.3b and figure 2.3c on page 12). More sophisticated prettyprinters need to make all the above decisions and more. Quite a few prettyprinters opt for a standard formatting scheme for a given piece of code or language construct rather then trying to determine the most readable one. This is not as bad as it seems as standardization has its own benefits in terms of readability.

Research is still being done on the best ways to format code. For example, Oman and Cook favor horizontal spacing over vertical spacing. Their program presentation software will modify the "strung out" code depicted in figure 4.1a to that depicted in figure 4.1b [53]. The traditional 80 column wide screen and printed listing made this kind of formatting unwieldy in the past due to the length of `<statement>`'s. This type of formatting is becoming more feasible now as the 80 character line length limit is rarely a programming language restriction anymore. Also printers and dis-

```
switch (<expression>) {
  case <value1>:
    <statement>;
    break;
  case <value2>:
    <statement>;
    break;
  case <value3>:
    <statement>;
    <statement>;
    break;
  case <value4>:
    <statement>;
    if (<expression>)
      <statement>;
    else
      <statement>;
    break;
}
```

Figure 4.1a, Vertical Spacing

```
switch (<expression>) {
  case <value1>:    <statement>;              break;
  case <value2>:    <statement>;              break;
  case <value3>:    <statement>;
                    <statement>;              break;
  case <value4>:    <statement>;
                    if (<expression>) <statement>;
                    else              <statement>;
                                              break;
}
```

Figure 4.1b, Horizontal Spacing

Figure 4.1: Comparison of vertical and horizontal spacing

play terminals have improved enough to handle more than 80 characters per line with limited loss of resolution.

The furor around prettyprinting systems has died down since their heyday in the late seventies and early eighties and only a few articles on these systems still appear [9, 36]. For the most part, prettyprinters were very simple programs and the differences between a programmer-formatted piece of code and a prettyprinter-formatted piece of code was not that great. Programmers have realized the benefits of indenting their own code. They can program faster and are much more efficient if they indent the code as they write it; seeing the structure of the code as they write it increases their understanding of the program and helps to reduces errors. Also, many program editors can aid the programmer in creating indented programs by anticipating the desired level of indentation based on the lines of code typed in previously and the syntax of the programming language.

4.2 Graphic Design

This approach heightens readability by using graphic design principles to distinguish and differentiate important language constructs, to make specific identifier classes visually explicit, and to lay out more appealing program listings. The text of the program remains the same, only the appearance and positioning of the text changes. Multiple type fonts, different size fonts, grey scaled tints, rules (lines which can act as demarcations), and variable spatial location of program elements on the page are some of the tools used to accomplish the above tasks [6].

The use of graphic design principles in program presentation is closely related to that of program reformatting. Both are concerned with the appearance of the code and providing easily recognizable visual clues to enable the meaning of the program to be discerned easier. While the output of a program reformatter can

still be compiled and run, the output from the graphic design approach changes a program into a typeset document.

Distinguishing different programming language constructs can be accomplished by using different type and size fonts. For example, keywords can be bolded, subroutine names can be written using a larger font, and character string data can be written using italics, figure 4.2a. Alternatively keywords can be italicized, subroutines names can be bolded, and character string data written using a smaller typewriter font, figure 4.2b. Many other combinations could be used as well. Using these different fonts for different language constructs provides an additional visual clue to the reader that makes it readily apparent which language construct a given word belongs in.

This is similar to, but on a much grander scale than, the notion that using uppercase characters for keywords and lowercase letters for identifier names distinguishes those two language constructs. The type and size fonts appropriate for distinguishing a given programming language construct depends on how important it is to emphasize the construct or to differentiate the construct from all the others. The only sure rule is that too many sizes and styles of fonts in use at the same time will overload the reader with information and cause confusion. One must also be careful that the typeset documents do not corrupt the indentation and alignment of the code due to the varying sizes of the fonts.

Baecker, in his program visualization system for the C language, uses a grey scaled tint to highlight stand alone comments. Comments that originally were on the same line as code are moved to the left margin of the page and written in a smaller font [6]. These two modifications visually separate the comments from the program more effectively than any amount of normal white space could. Baecker also uses rules to separate function definitions from function argument declarations and

```
float AbsoluteValue(x)
  float x;
{ if (x < 0) return(-x);
  return(x);}

float SquareRoot(x)
  float x;
{ float Guess,Accuracy;

  Accuracy = 0.00001;
  if (x < 0)
    {  printf("SquareRoot cannot take a negative argument");
       return (-1.0);}

  while (AbsoluteValue (Guess * Guess - x) >= Accuracy)
    Guess = (x / Guess + Guess) / 2.0;
  return(Guess);}
```

Figure 4.2a, bolded keywords, large function names, and italicized character string data

```
float AbsoluteValue(x)
  float x;
{ if (x < 0) return(-x);
  return(x);}

float SquareRoot(x)
  float x;
{ float Guess,Accuracy;

  Accuracy = 0.00001;
  if (x < 0)
    {  printf("SquareRoot cannot take a negative argument");
       return (-1.0);}

  while (AbsoluteValue (Guess * Guess - x) >= Accuracy)
    Guess = (x / Guess + Guess) / 2.0;
  return(Guess);}
```

Figure 4.2b, italicized keywords, bolded function names, and small smaller typewriter style character string data

Figure 4.2: Using different sizes and styles of fonts to accentuate constructs

function argument declarations from local variable declarations. This also clearly delineates one set of language constructs from another. The previous citation contains several excellent before and after pictures of Baecker's system in use; any attempt to reproduce them here would not do them justice.

Graphic design principles have been used mainly to vivify printed listings. There are a few program editors which use bolding and/or color to distinguish language constructs, usually keywords from identifiers, but little exists that approaches the sophistication of the program visualization systems described above. As bitmapped graphics terminals and windowing environments become the norm perhaps other graphic design elements will slip their way into program editors and programming environments as well.

4.3 Program Transformations

A program transformation is one which changes the actual text of the program. The change could be made according to the syntax rules of the language such as making default type conversions explicit, or the change could involve transforming the syntax of the program into a more readable (but not compilable) form. Both types of transformations preserve the original meaning of the program and hopefully make it easier for the reader to discern that meaning. The end result of a program transformation is that the text of the program is changed, hopefully for the better, making the true meaning of the program easier to discern when read.

Several trivial examples of program transformation are in wide use. For example, the pointer operator in Pascal is entered at the terminal as the caret symbol, "^". When Pascal programs are published, however, the pointer operator is often printed as the up arrow symbol, "↑". Another example in common use is to replace the assignment symbol, "="or ":=", with the left arrow symbol, "←", to avoid misinterpreting it with the equality symbol, "=" or "==". The syntax of a

programming language is locked in place by the language's definition; frequently it is constrained by the ASCII character set as well. This should not stop program presentation software from offering a different, more readable syntax.

Program transformations can be very successful for creating readable mathematical equations. In Fortran the raise-to-a-power operator is represented by two adjacent asterisks. Which is the more readable representation, `x ** (y + q)` or x^{y+q}? With a sufficiently powerful transformation system the unintuitive assignment statement in figure 4.3 becomes the easily recognizable quadratic roots equation.

```
x = (-b + sqrt(b*b - 4*a*c)) / 2*a;
```

$$x = \frac{-b+\sqrt{b^2-4*a*c}}{2*a}$$

Figure 4.3: Assignment statement versus mathematical equation

Many options are available once program transformations are used. Unintuitive keywords and symbols can be exchanged for more suitable ones. Intermediate symbols can be replaced with intermediate keywords. The entire character of the language can be changed. Baecker choose to remove all bracketing pairs, { ... }, in his enhanced C code and let indentation alone identify which statements belong to a control structure [6] (Bhujade's visual blocks, mentioned on page 11, have this property as well).

There is a danger in making too many program transformations especially of the kind described in the last paragraph. If too much of the syntax of the code is changed the original code and the transformed code become only tenuously connected. This may be fine in cases where the reader is unfamiliar with, or does not care what, the base programming language is. A maintenance programmer, however, would become very frustrated trying to maintain a program from listings that correspond only loosely to the text he is editing.

4.4 Additional Information

Part of the problem with trying to read and understand programs is that the program is usually the sole source of information about itself. All the information necessary to understand the program may be contained in the program itself but it is difficult for a reader to extract it. Questions like "Where is this global variable modified?" and "Where is this function used?" are hard for a reader to answer without the use of searching tools (grep) or cross reference lists and indices for the program. Much can be done in terms of mechanically scanning the code and supplying the reader with various access paths to the code as well as other helpful information.

Oman and Cook have experimented with supplying additional access paths to C code [53]. Their software generates a table of contents which lists all the source code files making up the program, the contents of those files by subroutine name, and the page number those subroutines are printed on. In addition an index lists all the subroutine names in alphabetical order, the page number on which the subroutine can be found, and the other subroutines that the subroutine calls and is called by. Additional types of information could have been added to the index as well; for example the names and types of the arguments to the subroutine and whether any global variables were used in the subroutine. Another language may have different information that could be placed in an index. Extracting all this information from the code and supplying it to the reader in a concise form is, for the most part, very easy for a program to do. And yet few of these types of systems exist; many are created 'in house' by the programmers themselves.

In some programming environments a source code analyzer is integrated with a program editor to provide these indices and cross references on-line. The reader can jump from subroutine to subroutine following a thread of execution.

moving into and out of various source code files seamlessly. All references and modifications of a global (or local) variable can be listed on the screen and each place where it is used or modified can be edited by selecting the appropriate line. In addition, a programmer can mark several locations in the code and skip from one to the other when needed; this is the electronic equivalent of holding a finger in several places in the listing and flipping from one to the other.

Having these additional access paths, either on-line or printed, help the reader to traverse the code easier. A reader will usually scan a program from top to bottom (front to back if a printed listing) or trace one or more execution paths through the code. Scrolling the program in a text editor or paging through a listing accomplishes the former way of reading the program. Until recently, though, there have been few tools or aids available for accomplishing the latter, more useful reading task.

When looking at any given page or screen of code in isolation, little is known about its relationship with the rest of the program. Access paths on a local scale can be helpful in determining a multitude of relationships. A reader should know the file name, module name, and subroutine name in which the viewed code resides. If one or more control structures span the page/screen break, the text on the next page should remind the reader what is being continued. Clifton, and separately Ramsdell, proposed using "connector lines" to connect the starting and ending parts of a control structure [15, 59]. This made matching the end of a control structure with the corresponding beginning trivial over long distances. Baecker's visualization program lists at the top of the page the control structures that are being continued. For example, if an `if` statement nested within a `while` statement continues to a second page, the text "`while... if...`" appears at the top of the second page [6]. An additional bit of information which could be useful to the reader is to list the

global variables that are referenced or modified on a particular page at the bottom of that page.

The ideas suggested above are just some of the ways in which information can be extracted from the program and presented to the reader. Different programming languages place different burdens on the reader. The information that can be extracted and presented to the reader may be different based on these burdens.

4.5 Literate Programming

"Literate programming" is the name given by Knuth to a generic programming language and documentation system that encourages and facilitates the concept that a program is a work of literature [41, pp. 97]. This approach is fundamentally different as compared to the other program presentation methods discussed so far, and yet, to a degree, includes all the other methods within it. Instead of starting with a program and transforming it into a document, literate programming starts with something that is neither, yet can be translated into both.

Knuth writes that a "practitioner of literate programming can be regarded as an essayist whose main concern is with exposition and excellence of style" [41]. The goal of literate programming is to write a program fit for human consumption that concentrates on explaining to a human reader how the program works yet can be compiled and executed by the computer as well. The 'literal' programmer becomes an author, expounding upon the code being written.

Knuth's implementation of literate programming is called Web, and has two translators, Tangle and Weave. The Web language combines the features of two other languages, TEX and Pascal, as well as a macro processor. Tangle takes a `.web` file as input and produces the Pascal code of the program as output. Weave takes the same `.web` file as input and produces a formatted TEX document as output. In principal any text processing/programming language combination could be used

as the base languages for Web. Other literate programming systems have been developed around C and troff, and Fortran 8x and TEX [64, 4].

Programmers need not learn a new language in order to use the literate programming paradigm. Ramsey created a table-driven literate programming system called Spider which can convert virtually any programming language into a literate programming system [60]. Programming languages which Spider has been used with so far include Ada, Awk, C, Fortran, Modula-2 and others. Programmers can continue programming in the languages they are most familiar with. In some literate programming systems no additional information needs to be added to the original code for it to be translated by the Web system [64]. This allows programmers to gradually use and become familiar with the features of the literate programming system.

Literate programming offers several additional benefits to the programmer and reader of the program besides the obvious documentation advantages. The programmer is able to to write programs as a collection of well documented code sections in whatever order is best for human comprehension. The rigid structure imposed by most programming languages need not be followed. If a new variable or function is needed, it can be declared at the spot it is used, thus improving the locality of the program elements. The program can be read in the order in which it was written, and not the order imposed by the base programming language. When a program is coded in a 'normal' language, the programmer continually traverses the text file adding a declaration in one place, creating a new function in another, and so on. In literate programming the code can be written in a "stream-of-consciousness" order with program elements appearing when needed. The connections between different program sections resemble a web as each may be related to several others in fairly obvious ways.

The web of relationships between program segments has other benefits as well. For example, often a simple one or two line subroutine grows to twenty lines in length because of the need for error handling. In normal programming languages, the subroutine appears to be an error handling routine rather than its true purpose. Many times a programmer will subconsciously reduce or eliminate necessary error checks because the additional code obscures the true meaning of the function. In literate programming, however, the error portions of the code can be abstracted away until a later time in favor of writing the real purpose of the function. Once that is accomplished, the programmer can concentrate on writing the best error handling code possible as that is the purpose of that particular code section. The benefits to the reader are obvious, the error code is spatially removed from the function code proper and need not be read unless the situation warrants it.

The output from the Web system that is meant for human consumption is completely typeset. All of the program presentation methods discussed in this chapter so far can be put to use in this typeset document, including extensive indexes and cross reference lists. The program could be sent to a journal and published as is without the usual cutting and rewriting and documenting of code that most be done in other language systems.

There are several problems with literate programming though, none of which appear to be insurmountable. Thimbleby reports that "The most subtle disadvantage of [Web] complements one major advantage: because it motivates programmers to document their programs, programmers will be more egoistic about their programs. This will result in all the problems which Weinberg illuminates so well" [64]. In fact, Knuth himself professed to being egoistic about his Web programs [41, pp. 109]. Weinberg details the problems associated with egoistic programmers and the need for egoless programming in his book *The Psychology of Computer Programming* [69].

The other problems Thimbleby found had more to do with the state of his Web implementation as opposed to any fundamental fault with literate programming itself. For example, the line numbers the compiler reported as having had errors had to be translated to the equivalent line numbers in the original `.web` file. Fortunately the compiler Thimbleby used allowed directives to be inserted that ensured the numbers reported reflected the original source. In general though the line number may not be as easy to solve without fully integrating the Web programming environment with the various compilers and translators. Also because of the lack of integration, the programmer has to contend with Tangle errors, Weave errors, base programming language errors and document formatting language errors (as well as algorithmic errors). Another problem was that small-scale debugging became harder because the program was spread out over a larger area. Thimbleby conjectures that a smarter editor, integrated in the Web way of doing things, would reduce that problem as well.

Many of the problems associated with literate programming are due to the literate programming system being a shell around an existing programming language. As was stated earlier, there is a great advantage in not having to learn a new programming language in order to become a literate programmer. However, there are problems associated with this approach as well. Unfortunately many programming languages, when placed into the literate programming paradigm, develop slight quirks that are awkward to program around. These quirks need not be there had the programming language been designed for the literate programming paradigm.

Literate programming has had very little impact on the programming profession to date despite the number of articles published about it. The journal *Communications of the ACM* occasionally has had a column concerning literate programming (July 1987, December 1987, December 1988, June 1989, September 1989, March 1990) but the column was discontinued. Interestingly, the trade magazine

Computer Language recently published an article on literate C++ [32]. Sewell published *Weaving a Language-Independent Web* which contains the source code (in Web) for the Tangle and Weave translators and a small tutorial on their use [62]. This above that literate programming is escaping the research world, but as yet there are no commercial systems available. It often takes fifteen to twenty years for any new technology to become accepted; look at object oriented programming for example. Whether literate programming becomes widely accepted in the future is yet to be seen.

CHAPTER 5

CONCLUSIONS

The following high-level conclusions are drawn from themes that are present through out the body of this thesis.

5.1 Readable Programs are Important

The idea that it is enough that a program be correct is untenable. The sheer number of times any given piece of code must be read and understood during its lifetime and the necessity to maintain the code should discount that belief immediately. A program must be interpreted each time it is read. A compiler will interpret the same code the same way each time. The likelihood that a person will interpret the same code the same way each time, let alone the same way the compiler does, depends on how readable the code is and how close the intuitive meaning matches the real meaning.

For all the benefits of readable programs to be realized, programmers must make code readable as they make it correct. It is easier to notice errors and ensure that the program will work correctly if this is done. Making a program more readable effectively decreases its complexity as well. The structure of the program is more apparent, relationships between different parts of the program are easily discerned, and the mental strain necessary for understanding the program is lessened. The time it takes to comprehend the code, and the program as a whole, is also reduced considerably.

5.2 Programming for Readability is Not a Simple Task

Programming clearly and concisely is achievable, but it is not easy. Just as good writing requires care, attention to detail, and practice, so does good programming. There are few magic formulas proffered by this thesis for writing readable code, designing programming languages that make writing readable code easier, and using presentation techniques that make the code more readable. Attempts to follow the suggestions put forth without knowing the reasoning behind them may do more harm than good.

Making code readable requires that a programmer understand and merge both aesthetics and functionality. The intention behind making a program readable is to make the program easier to understand and to ensure its meaning when read and meaning when executed are the same. A programmers style, the programming language used, and the way the program is presented all affect this goal. The readability tradeoffs of one programming style and language design over another are complex to determine and can vary greatly based on the interaction between the code under consideration and the particular programming style and programming language that are being used. The most readable design for a language construct depends on the way that construct will be used most often. The most readable programming style and program presentation method depends on the language being used and the particular code being improved. The bulk of this thesis discusses these tradeoffs and intricacies and attempts to shed some light on the subject.

One example that illustrates the complexity of programming for readability is choosing the name for an identifier. Selecting the most readable name for an identifier is very important, but many factors need to be examined and weighed before making a decision. Identifier names should be long enough to be meaningful but short enough to not be unwieldy. The midpoint where the balancing tradeoffs in readability offset each other changes based on how often, where, and in what

context the identifier in question will be used. The name should reflect the identifiers purpose in terms of the problem domain when possible or in terms of its use within the program when the former is not possible. The name should also make sense when read in the expressions and statements where it is used and thereby eliminate the need for explanatory comments. Consider the number of identifiers in the average program that must be given names and remember that naming identifiers is only one small aspect of creating readable programs.

5.3 Programming Languages Can be Designed to be More Readable

Few programming languages were designed with code readability and clarity in mind. Cobol and Ada are two of the best known attempts and in both cases many other considerations took higher precedence. The tools one uses directly influence the shape and form of the solution; special care needs to be taken to ensure that programming languages will facilitate creating readable code. It is much easier to create a readable program if the underlying structure of the code is readable.

Many language constructs can be improved just by changing their syntax, making them more readable, and less prone to errors of interpretation. In these cases the power and usefulness of the construct need not be changed, only its appearance. Syntactic subtleties of the language can often be removed by modifying the syntax as well. If changing the syntax is no longer an option, compiler warnings should be added, as part of the language definition, to aid the detection and elimination of these possible errors.

Particularly powerful language constructs that are prone to being abused should be examined closely. It is possible that the syntax and semantics of the construct can be changed to eliminate or at least limit the potential abuses. Designing the language construct so that it can be used only in the cases where it makes good sense is rarely an option, this can overly restrict the usefulness of the construct even

when it is possible. Changing the language construct to require the programmer to specify additional information that would otherwise not be necessary can make the construct's purpose clearer to the reader. Some powerful language constructs, such as global variables and the `GOTO` statement, should be used rarely and only for very specific purposes. Making constructs such as these difficult or unwieldy to use can help limit the number of times that they are used to those that the programmer deems are necessary in spite of the additional inconvenience.

Much thought goes into designing programming languages with new and more powerful features. Considerably less thought is spent on making those features readable and easy to use without causing errors. How many times is one particular piece of code compiled and run and re-compiled and tested and compiled again and debugged and so on? Designing languages that are more readable, have fewer syntactic subtleties, and are easier to use correctly can save an enormous amount time and energy in the long run.

5.4 Programming Languages Should have Automated Proofreaders

It is well known that the earlier an error is caught, the easier it will be to fix. Finding errors and correcting them statically (normally thought of as compile time) is much more efficient than finding them when running the program. Unfortunately compilers have traditionally given poor error diagnostics. Problems that could be found at compile time typically are not because the compiler is only concerned with turning away code that is not correct syntactically. Often the errors the compiler does report are phrased in terms of its problem rather than the programmers problem; reporting the error "Symbol table exceeded" instead of "Too many identifiers declared" is one such example.

Many of the 'errors' that trip up both beginning and experienced programmers alike are not treated as errors by the compiler as they are syntactically correct

by the rules of the programming language. For example, quite a few of the errors the UNIX "lint" program finds and reports are legal within the rules of the C programming language. More times than not the advice given by lint is taken because, syntactically correct or not, the code will not do what the programmer wanted.

A proofreading system could combine the best aspects of a program style system [43] and a program critic system (lint). A proofreader program could find all the syntactic errors, errors similar to those caught by lint, errors that take the form of syntactic subtleties similar to those discussed in section 3.3, and errors related to some some programming style issue. The proofreader's sole task is to warn the programmer of as many possible errors, inconsistencies, spurious code conditions, and readability problems as possible. Programmers can of course choose to ignore any advice given by such a system, but at least they will be made aware of situations where possible problems exist. By heeding the warnings of the proofreader system though, the resulting program can be made more readable and many errors can be caught and eliminated sooner.

5.5 Standard Disclaimer

Just as beauty is in the eye of the beholder, how readable something is depends on the outlook of the person doing the reading. Hopefully this thesis has not been biased too much towards the authors outlook. Writing this thesis has made me think more about the way I program and the way I look at programming languages and programming environments. If reading this thesis does the same for you then all has not been in vain.

BIBLIOGRAPHY

[1] American National Standards Institute, Washington, D.C.: U.S. Department of Defense. **Reference Manual for the Ada Programming Language**, 1983. ANSI/MIL-STD-1815A.

[2] American National Standards Institute, Washington, D.C.: U.S. Department of Defense. **Fortran X3.9-198x**, January 1986. X3J3/S8.

[3] N. Anand. Clarify function! **ACM Sigplan Notices**, 23(6):69–79, June 1988.

[4] Adrian Avenarius and Siegfried Opperman. Fweb: A literate programming system for Fortran 8x. **ACM Sigplan Notices**, 25(1):52–58, 1990.

[5] John Backus. Programming in America in the 1950's — Some personal impressions. In N. Metropolis, J. Howlett, and Gian-Carlo Rota, editors, **A History of Computing in the Twentieth Century**, pages 125–135. Academic Press, 1980.

[6] Ronald Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In **Proceedings of the 10th International Conference on Software Engineering**, pages 356–366, April 1988.

[7] Geneva G. Belford and Chung Laung Liu. **Pascal**. McGraw-Hill Inc., 1984.

[8] Moreshwar R. Bhujade. Visual specification of blocks in programming languages. **ACM Sigplan Notices**, 22(8):24–26, August 1987.

[9] G. Blaschek and J. Sametinger. User-adaptable prettyprinting. **Journal of Software Practice and Experience**, 19(7):697–702, 1989.

[10] Marian Kruse Breland and Keller Breland. Legibility of newspaper headlines printed in capitals and in lower case. **Journal of Applied Psychology**, 28:117–120, April 1944.

[11] Frederick P. Brooks, Jr. **The Mythical Man-Month**. Addison Wesely Publishing Company, January 1982.

[12] R. Brooks. Towards a theory of the comprehension of computer programs. **International Journal of Man-Machine Studies**, 18(6):543–554, June 1983.

[13] Breck Carter. On choosing identifiers. **ACM Sigplan Notices**, 17(5):54–59, May 1982.

[14] R. Lawerence Clark. A linguistic contribution to goto-less programming. **Datamation**, 19(2):62–63, December 1973.

[15] Mitchell H. Clifton. A technique for making structured programs more readable. **ACM Sigplan Notices**, 13(4):58–63, April 1978.

[16] Kenneth Conrow and Ronald G. Smith. Neater2: A PL/1 source statement reformatter. **Communications of the ACM**, 13(11):669–675, November 1970.

[17] Tom DeMarco and Tim Lister. Software development: State of the art vs. state of the practice. In **Proceedings of the 11th International Conference on Software Engineering**, pages 271–275, 1989.

[18] Edsger W. Dijkstra. The humble programmer. **Communications of the ACM**, 15(10):859–866, 1972.

[19] S. L. Ehrenreich. Computer abbreviations: Evidence and synthesis. **Human Factors**, 27(2):143–155, 1985.

[20] Adin Falkoff. The APL character set: Dual keyboards are better. **APL Quote Quad**, 20(2):28–32, December 1989.

[21] R. K. Fjeldstad and W. T. Hamlen. Applications program maintenance study: Report to our respondents. In G. Parikh and N. Zvegintzov, editors, **Tutorial on Software Maintenance**, pages 13–27. IEEE/CS Press, Silver Spring, Md., 1983.

[22] A. C. Fleck. On the imposibility of content exchange through the By-Name parameter transmission mechanism. **ACM Sigplan Notices**, 11(11), November 1976.

[23] Lawerence Flon. On research in structured programmining. **ACM Sigplan Notices**, 10(10):16–17, October 1975.

[24] J. D. Gannon and J. J. Horning. The impact of language design on the production of reliable software. **ACM Sigplan Notices**, 10(6):10–22, June 1975.

[25] James Gimpel. Language features that aid debugging. **Computer Language**, 5(4):41–45, April 1988.

[26] Richard Hamlet. A further note on symmetric keyword pairs. **ACM Sigplan Notices**, 15(6):7, June 1980.

[27] David Harel. do considered od odder than do considered ob. **ACM Sigplan Notices**, 15(4):75, April 1980.

[28] Robert G. Herriot. Towards the ideal programming language. **ACM Sigplan Notices**, 12(3):56–62, March 1977.

[29] C. A. R. Hoare. Hints on programming language design. **Computer Systems Reliability**, 20:505–534, 1974.

[30] Richard C. Holt. Teaching the fatal disease (or) Introductory computer programming using PL/1. **Sigplan Notices**, 9(5):8–23, May 1973.

[31] J. N. P. Hume and R. C. Holt. **Fortran 77 for Scientists and Engineers**. Reston Publishing Company, Inc., Reston VA, 1985.

[32] Marco S. Hyman. Literate c++. **Computer Language**, 7(7):67–79, July 1990.

[33] K. E. Iverson. **A Programming Language**. Wiley, New York, 1963.

[34] Kathleen Jensen and Niklaus Wirth. **Pascal: User Manual and Report**. Springer-Verlag, 1974.

[35] LeRoy Johnson. do considered obviously odd in three dimensions. **ACM Sigplan Notices**, 15(12):44, December 1980.

[36] M. Jokinen. A language-independent prettyprinter. **Journal of Software Practice and Experience**, 19(9):839–856, 1989.

[37] Daniel Keller. A guide to natural naming. **ACM Sigplan Notices**, 25(5):95–102, May 1990.

[38] B. W. Kernighan and D.M. Richie. **The C Programming Language**. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[39] Brian W. Kernighan and P. J. Plauger. **The Elements of Programming Style**. McGraw-Hill Book Company, 1978.

[40] Donald E. Knuth. Structured programming with go to statements. In Raymond T. Yeh, editor, **Current Trends in Programming Methodology**, pages 140–194. Prentice-Hall, Englewood Cliffs, NJ, 1977. Volume 1, Software Specification and Design.

[41] Donald E. Knuth. Literate programming. **The Computer Journal**, 27(2):97–111, 1984.

[42] T. A. Kovats. Program readability, closing keywords and prefix-style intermediate keywords. **ACM Sigplan Notices**, 13(11):30–42, November 1978.

[43] Al Lake and Curtis Cook. An automated program style analyzer for pascal. **ACM Sigcse Bulletin**, 22(3):29–34, September 1990.

[44] Henry Ledgard and John Tauer. **Programming Practice**, volume 2 of **Professional Software**. Addison-Weseley, 1987.

[45] D. W. Leinbaugh. Indenting for the computer. **ACM Sigplan Notices**, 15(5):41–48, May 1980.

[46] John McCarthy. History of Lisp. In Richard L. Wexelblat, editor, **History of Programming Languages**. ACM Sigplan, Academic Press, 1981. from the History of Programming Languages Conference, June 1–3, 1978.

[47] Michael Metcalf and John Reid. **Fortran 8x Explained**. Oxford University Press, 1989.

[48] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Schneiderman. Program indentation and comprehensibility. **Communications of the ACM**, 26(11):861–867, 1983.

[49] Carol Bergfeld Mills and Linda J. Weldon. Reading text from computer screens. **ACM Computing Surveys**, 19(4):329–358, December 1987.

[50] Peter G. Neumann. Risks to the public in computers and related systems. **Software Engineering Notes**, 15(2):3–23, April 1990. From "Gimme a 'Break' ...Or *would you rather* 'Switch'?" *submitted by a contributer.*

[51] Peter R. Newsted. Flowchart-free approach to documentation. **Journal of Systems Management**, 30(4):18–21, April 1979.

[52] Paul W. Oman and Curtis R. Cook. A paradigm for programming style research. **ACM Sigplan Notices**, 23(12):69–78, December 1988.

[53] Paul W. Oman and Curtis R. Cook. Typographic style is more than cosmetic. **Communications of the ACM**, 33(5):506–520, 1990.

[54] Donald G. Patterson and Miles A. Tinker. Influence of type form on speed of reading. **Journal of Applied Psychology**, 12:359–368, 1928.

[55] Donald G. Patterson and Miles A. Tinker. Readability of newspaper headlines printed in capitals and lowercase. **Journal of Applied Psychology**, 30:161–168, April 1946.

[56] P.J. Plauger. Programming on purpose. **Computer Language**, 5(4):17–22, April 1988.

[57] E. C. Poulton and C. Helen Brown. Rate of comprehension of an existing teleprinter output and of possible alternatives. **Journal of Applied Psychology**, 52:16–21, April 1968.

[58] Roger S. Pressman. **Software Engineering: A Practitioner's Approach.** McGraw-Hill, 1982.

[59] John Ramsdell. Prettyprinting structured programs with connector lines. **ACM Sigplan Notices**, 14(9):74–75, September 1979.

[60] Norman Ramsey. Weaving a language-independent Web. **Communications of the ACM**, 32(9):1051–1055, September 1989.

[61] Jean E. Sammet. **Programming Languages: History and Fundamentals.** Prentice-Hall, Inc., 1969.

[62] Wayne Sewell. **Weaving A Program: Literate Programming in WEB.** Van Nostrand Reinhold, 1989.

[63] Gregg Taylor. On being APL literate. **APL Quote Quad**, 20(1):17–19, September 1989.

[64] H. Thimbleby. Experiences of 'Literate Programming' using cweb (a variant of Knuth's web). **The Computer Journal**, 29(3):201–211, 1986.

[65] William H. Trotter, Jr. **Crystal Clear Cobol.** Prentice-Hall, Englewood Cliffs, NJ, 1989.

[66] Dennie van Tassel. **Program Style, Design, Efficiency, Debugging, and Testing.** Prentice-Hall, Englewood Cliffs, NJ, 1978.

[67] Aard van Wijngaarden, B. J. Mailloux, J. E. L.Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. T. T. Meertens, and R. G. Fisker. **Revised Report on the Algorithmic Language Algol 68.** Springer-Verlag, 1976.

[68] Allen G. Vartabedian. The effect of letter size, case, and generation method on CRT display search time. **Human Factors**, 13(4):363–368, August 1971.

[69] Gerald M. Weinberg. **The Psychology of Computer Programming.** Van Nostrand Reinhold, 1971.

[70] Gerald M. Weinberg and Edward L. Schulman. Goals and performance in computer programming. **Human Factors**, 16(1):70–77, February 1974.

[71] L. M. Weissman. Psychological complexity of computer programs: An experimental methodology. **ACM Sigplan Notices**, 15(6):25–36, June 1974.

[72] Richard L. Wexelblat. Maxims for malfeasant designers, or How to design languages to make programming as difficult as possible. In **Proceedings of the 2nd International Conference on Software Engineering**, pages 331–336, 1976.

[73] Richard L. Wexelblat, editor. **History of Programming Languages.** Academic Press, 1981.

[74] Niklaus Wirth. The programming language Pascal. **Acta Informatica,** 1(1):35–63, 1971.

[75] Niklaus Wirth. On the composition of well structured programs. **ACM Computing Surveys,** 6(4):247–259, December 1974.

[76] Niklaus Wirth. Design and implementation of Modula. **Software Practice and Experience,** 7(1):67–84, 1977.

[77] Niklaus Wirth. **Programming in Modula-2.** Springer-Verlag, 1983.

[78] William A. Wulf. A case against the goto. In **Proceedings of the 25th National ACM Conference,** pages 791–797.

[79] J. M. Yohe. An overview of programming practice. **ACM Computing Surveys,** 6(4):221–245, December 1974.