Investigating Software Complexity: Knot Count Thresholds

Paula A. Hannan

A Research Paper submitted to

Oregon State University

in partial fulfillment of the requirements for the degree of

Master of Science

Presented October 17, 1989

Acknowledgements

I have been particularly fortunate to meet many helpful collegues and friends at Oregon State University. I would like to thank each and every one of these great people, but then the acknowledgements would be longer than the paper. Although it is a difficult job, I must single out a few particular people for special thanks.

Dr. Cook and the other faculty at OSU for transforming me from a programmer into a scientist.

Sherry Yang who is always ready and willing to help.

Tom Sturtevant, Dr. Dave Sandeberg, Rick Charon and John Bertani who allowed me to use their classes for my expirement.

Paula A. Hannan

This paper is dedicated to my family:

Alice and John Hannan

Paul and Yvonne Melkonian.

Table of Contents

1	Introduction	1
2	Control Flow Metrics	4
	2.1 McCabe's Cyclomatic Complexity	4
	2.2 Knot Count	6
	2.3 What Does Knot Count Measure	8
	2.4 Why Use Knot Count	12
	2.5 Metric Guidelines or Thresholds	21
3	Experiment	23
	3.1 Subjects	23
	3.2 Materials	23
	3.3 Task	29
	3.4 Results	34
4	Conclusion	48
	Bibliography	49

Appendices

A - Background Data
- CS212 Background Data
- CS317 Background Data
- CS312 Background Data
- CS431 Background Data

B - Materials
- Pascal Materials
  - Pascal Background Questionaire
  - Easy Program
    - Comprehension Quiz - Forward Reasoning Question First
    - Comprehension Quiz - Backward Reasoning Question First
  - Hard Program
    - Comprehension Quiz - Forward Reasoning Question First
    - Comprehension Quiz - Backward Reasoning Question First
- C Materials
  - C Background Questionaire
  - Easy Program
    - Comprehension Quiz - Forward Reasoning Question First
    - Comprehension Quiz - Backward Reasoning Question First
  - Hard Program
    - Comprehension Quiz - Forward Reasoning Question First
    - Comprehension Quiz - Backward Reasoning Question First

Index Of Tables

Table		Page
2.1	Sensitivity of Metrics to Program Structure	20
3.1	Metric Values for Pascal Programs	27
3.2	Metric Values for C Programs	29
3.3	Percent of Subjects Who Attempted All Questions	33
3.4	Pascal Subjects - Total Class Performance	36
3.5	Pascal Subjects - Average Performance by Student	36
3.6	Pascal Subjects - Performance on First vs. Second Program	39
3.7	C Subjects - Total Class Performance	40
3.8	C Subjects - Average Performance by Student	41
3.9	Control Flow Metrics	46
3.10	Size Metrics	47

Index of Figures

Figure		Page
2.1	Program with Flow Graph	4
2.2	V(G) and Compound Conditionals	6
2.3	V(G) and Simple Conditionals	6
2.4	Program with Jump Table	7
2.5	Knots with Iteration	8
2.6	Knots with If Then Else Statement	9
2.7	Knots with Case Statement	10
2.8	Knots with Goto Statement	10
2.9	Knots with Return Statement	11
2.10	Knots with Break Statement	12
2.11	Knots with Continue Statement	12
2.12	Program Segment with Multiple Exits from a Loop	13
2.13	Program Segment without Multiple Exits from a Loop	14
2.14	Program with Unnecessary Branching	15
2.15	Program with Reduced Unnecessary Branching	15
2.16	Program with Lower Independence Between Statements	16
2.17	Program with Reduced Independence Between Statements	17
2.18	Program Segment with Else Goto	18
2.19	Program Segment with Else Goto Statement Removed	18
2.20	Program Segment with Bushy Decision Tree	19
2.21	Program Segment with Bush Decision Tree Removed	19
3.1	Sample Comprehension Quiz Questions	25
3.2	Computing Overlap Rectangle	28
3.3	Percent Attempted on Both Programs	32
3.4	C Subjects - Percent Attempted Forward vs. Backward Reasoning Groups	34
3.5	Pascal Subjects - Performance Hard vs. Easy Program	36
3.6	Pascal Subjects - Percent Attempted Forward vs. Backward Reasoning Questions	37
3.7	Pascal Subjects - Performance Forward vs. Backward Reasoning Questions	38
3.8	Pascal Subjects - Effect of Program Ordering on Performance	39
3.9	C Subjects - Performance Hard vs. Easy Programs	40
3.10	C Subjects - Percent Attempted Forward vs. Backward Reasoning Questions	41
3.11	C Subjects - Performance Forward vs. Backward Reasoning Questions	43
3.12	C Subjects - Performance by Group	45
3.13	C Subjects - Effect of Program Ordering on Performance	46

1 INTRODUCTION

What makes software complex? Is it the size of the product, the number of decisions made or the amount of information processed? There is no commonly agreed upon model or measure. Software complexity metrics, which attempt to measure program quality, fall into two groups; those that measure the software process and those that analyze the software product. "Process metrics quantify attributes of the development process and the development environment" [Conte, pg. 19]. One popular example is Boehm's CoCoMo model where inputs include factors like system speed, programmer skill, and project difficulty; the output is an estimation of effort and time [Lewis, pg. 54]. Product metrics, on the other hand, are "measures of the software product" [Conte, pg. 20]. There are three general types of product metrics; size metrics, data structure metrics and control flow metrics. Size metrics include lines of code and function counts. Data structure metrics include variable counts, variable life spans and Henry and Kafura's information flow [Henry]. Control flow metrics include decision counts, cyclomatic complexity and knot count.

Logical structure or control flow is one important component of complexity. Software complexity metrics that measure logical structure concentrate on the decisions made in a module and the branching those decisions cause. Two popular control flow metrics are McCabe's cyclomatic complextiy and the knot count. McCabe's cyclomatic complexity is based on the control flow graph associated with the module and corresponds to the number of decisions in a program [McCabe]. McCabe's measure is widely accepted because it is easy to compute and agrees with our intuition concerning logical complexity. The knot count, proposed by Woodward, Hennell and Hedley [Woodward], measures the number of overlapping transfers of control (e.g. knots). The number of knots measures complexity, particularly complexity introduced by poor programming practices such as unstructured code. The number of knots is easy to compute and is language independent.

The ability to classify a program as to its difficulty to test, understand or maintain is one expected benefit from the study of software complexity metrics. To classify a module using a complexity metric, however, we must set guidelines or threshold values, where a module exceeding the threshold is flagged as a possible problem. Most current metric thresholds are based on intuition, experience and ease of implementation. McCabe, for example, suggests the number of decisions in a module be less then 10 [McCabe, pg. 314] without proof or confirming empirical support. The validation of metrics, including the empirical confirmation of threshold values, is an especially pressing need [Conte, pg. 360].

This study concentrates on threshold values for the two most popular control flow metrics: McCabe's cyclomatic complexity and the knot count. We describe the results of an experimental study to empirically determine a threshold value for knot count for student programmers. The experiment was designed to measure the interaction between difficulty, as measured by knot count, and comprehension quiz scores. This experiment had two goals:

1. Show that there are threshold values for the knot count metric.
2. Discover knot count threshold values for students in Pascal and C.

This research was motivated by the need to establish threshold values for complexity metrics. First we show that the knot count is a useful control flow metric because of its ability to gauge the structuredness of code; then we describe the results of an experimental study to determine the threshold value for the knot count. Our work suggests a threshold value of 20 when evaluating upper division student programmers using C and 7 for beginning and intermediate Pascal students.

The second chapter describes control flow metrics and the issues involved in setting standards for the knot count. The third chapter describes the subjects, materials and procedures used in experiment and discusses the results. Finally chapter four reviews this experiment's contribution to software metrics and discusses the possibilities for future work.

2 CONTROL FLOW METRICS

Control flow metrics attempt to measure the contribution of logical structure to program complexity. This chapter defines the two most popular control flow metrics: McCabe's cyclomatic complexity and the knot count. It shows how the knot count measures the structured and unstructuredness of code. This chapter concludes with a discussion of the current research concerning threshold values for complexity metrics.

2.1 MC CABE'S CYCLOMATIC COMPLEXITY

Cyclomatic complexity is based on the module's flow graph. A flow graph represents basic blocks of statements with nodes, and the flow of control between blocks with edges. Here is a program with its associated flow graph:

```
1   procedure ignoreVowels(s:string; length:integer);
2   var
3       vowels : set of char;
4       i : integer;
5   begin
6       vowels:=['a','e','e','o','u']
7       for i:=1 to length do
8           if not (s[i] in vowels) then
9             write(string[i]);
10  end;
```

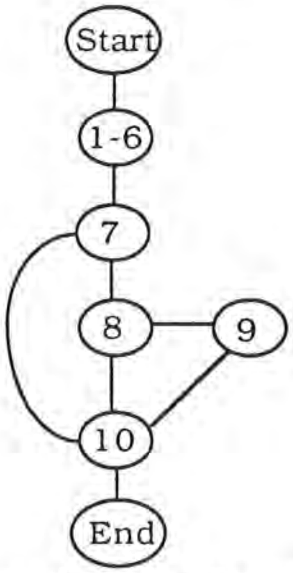


Figure 2.1 - Program with Flow Graph

The cyclomatic number V(G) of a graph G with n vertices, e edges and p connected components is

$V(G) = e - n + 2p.$

V(G) for the example program is three (8-7+2). V(G) was designed to "measure and control the number of paths through a program" [McCabe], which in turn indicates a minimum number of test cases. The number of paths in a program is related to the number of circuits in the flow graph; so that the cyclomatic complexity is the number of edges that must be removed in order to transform it to its "skeleton" - the graph without circuits or loops [Conte, pg. 66]. Computing the cyclomatic complexity by constructing the flow graph would be a time consuming task, however V(G) is only dependent on the number of edges and nodes, not how they are connected, so the computation of V(G) simplifies to the number of decisions + one. Counting the number of decisions however does not distinguish compound conditionals from simple conditionals. Figure 2.2 shows a program segment with a compound conditional where V(G) is two. Figure 2.3 shows the equivelent program segment without compound conditionals. The number of decisions has increased to two so that V(G) increases to three. Our intuition would suggest (and one study has confirmed [Cook86]) that these two program segments have similar complexity.

```
A    ..............
B    if sunny and warm then
C         wearshorts:=true
D    ..............
```

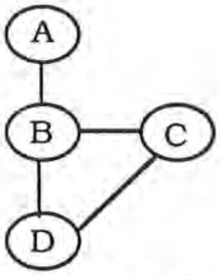


Figure 2.2 - V(G) and Compound Conditionals

```
A    ..............
B    if sunny  then
C         if warm then
D              wearshorts:=true
E    ..............
```

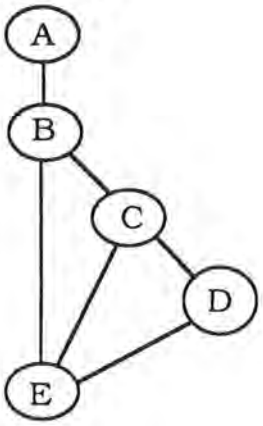


Figure 2.3 - V(G) and Simple Conditionals

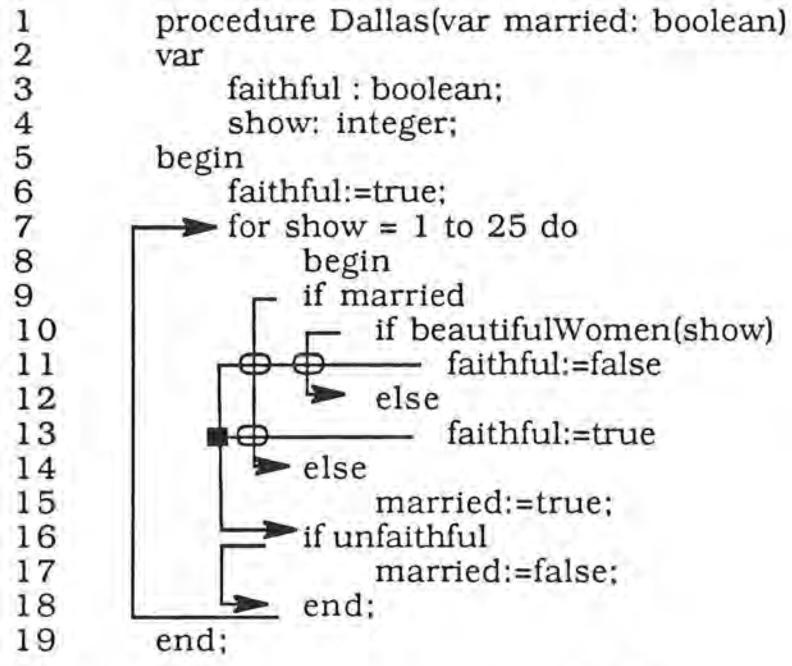
The most common solution to this problem is counting a compound conditional as one + the number of logical operators. Hence V(G) is one + the number of simple predicates.

2.2 KNOT COUNT

Informally a knot occurs when two separate control flow paths in a module cross; where a control flow path is an arc from a control flow operator to the destination of the transfer. We can define a flow of program control from statement a to statement b mathematically as an ordered pair of statement numbers (a,b). A knot occurs when two jumps (a,b) and (p,q) exist such that

1) min(a,b) < min(p,q) < max(a,b) and max(p,q) > max(a,b)
or
2) min(a,b) < max(p,q) < max(a,b) and min(p,q) < min(a,b)

The knot count is easily computed by creating a list of the ordered pairs and then comparing the pairs for overlap. As an example, lets compute the knot count for the following program:



```
1       procedure Dallas(var married: boolean)
2       var
3           faithful : boolean;
4           show: integer;
5       begin
6           faithful:=true;
7           for show = 1 to 25 do
8               begin
9               if married
10                  if beautifulWomen(show)
11                      faithful:=false
12                  else
13                      faithful:=true
14              else
15                  married:=true;
16              if unfaithful
17                  married:=false;
18              end;
19      end;
```

Jump Table

From	To	Crosses
9	14	(11,16), (13,16)
10	12	(11,16)
11	16	
13	16	
16	18	
18	8	

Figure 2.4 - Program with Jump Table

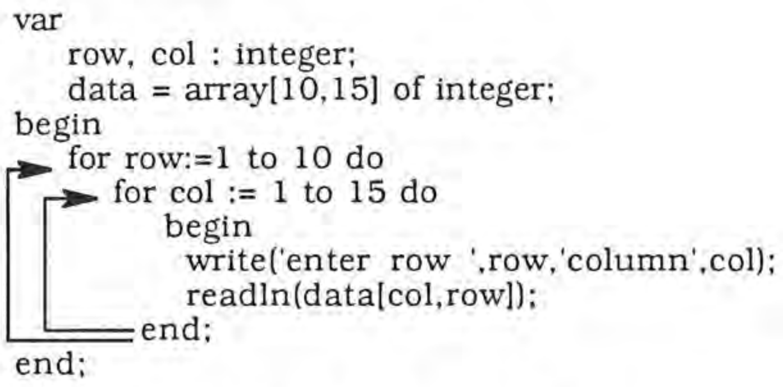
Note that there is no arc when the flow of control is to the next contiguous statement.

2.3 WHAT DOES KNOT COUNT MEASURE?

In this section we consider the knot count for the various structured and unstructured constructs in order to substantiate the claim that the knot count measures unstructuredness.

Structured Constructs

iteration - Woodward shows that properly nested looping constructs such as for and while are implemented without any knots [Woodward, pg. 48]. For example:

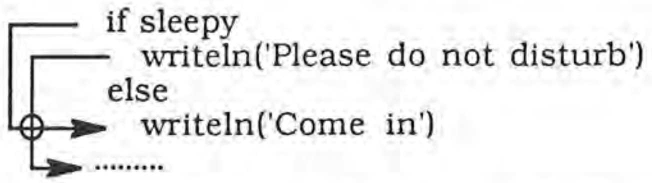


```
var
   row, col : integer;
   data = array[10,15] of integer;
begin
   for row:=1 to 10 do
      for col := 1 to 15 do
         begin
           write('enter row ',row,'column',col);
           readln(data[col,row]);
         end;
end;
```

Figure 2.5 - Knots with Iteration

The fact that knot count does not measure the complexity of iteration could be cited as a disadvantage; but studies show that alternation, which increases the knot count, not iteration, is the major source of errors in a program [Lewis, pg. 403].

if then else - Control flows from the if statement to the else block and from the last statement in the if block to the statement after the else block creates a knot.



```
if sleepy
  writeln('Please do not disturb')
else
  writeln('Come in')
.........
```

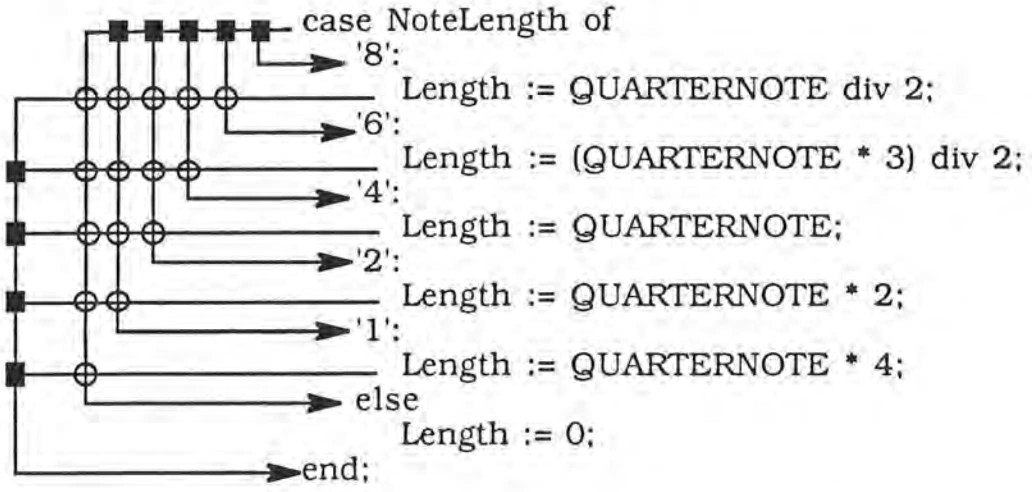
Figure 2.6 - Knots with If Then Else Statement

Note that an if then statement (no else) does not create a knot.

case - The case construct is a more structured alternative to the computed goto used in Fortran. Woodward shows that a computed goto statement creates n(n-1)/2 knots, where n is the number of labels. The Fortran computed goto has the format

GO TO (L_1,L_2,.....,L_m) EXPRESSION

A table of possible integer values of the expression along with the corresponding labels or their statement numbers can also represent a computed goto. The case statement can be represented as a similar table, where the values the expression can assume are not necessarily integer. These tables can be thought of as the list of jumps used to compute the knot count. Therefore an n-way case produces a knot count of n(n-1)/2. The following example [Folts, pg. 42] has six labels and 15 knots.



```
case NoteLength of
 '8':
    Length := QUARTERNOTE div 2;
 '6':
    Length := (QUARTERNOTE * 3) div 2;
 '4':
    Length := QUARTERNOTE;
 '2':
    Length := QUARTERNOTE * 2;
 '1':
    Length := QUARTERNOTE * 4;
 else
    Length := 0;
end;
```

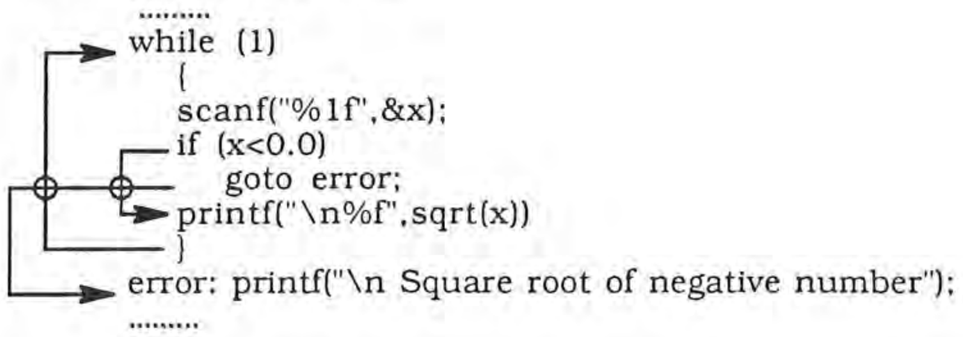
Figure 2.7 - Knots with Case Statement

A logically equivalent if then else block produces n knots.

Unstructured constructs

A language such as C provides more opportunities to create knots. Besides the if then else, and the switch, C has several unstructured constructs (goto, return, break and continue) which can create knots.

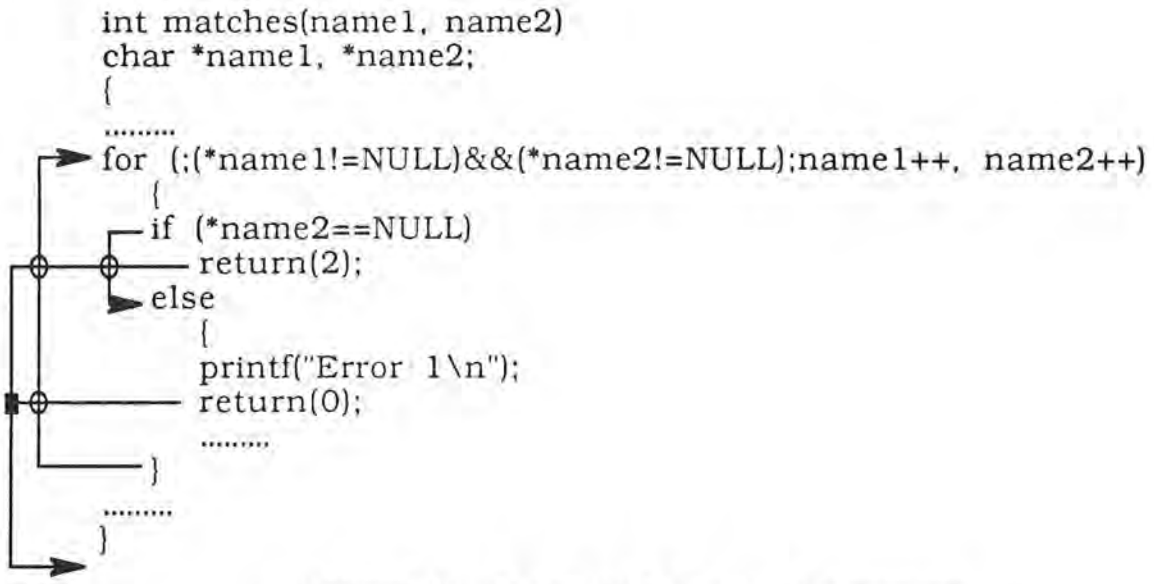
goto - Unstructured jumping is an obvious source of knots. This sample has two knots:



```
.........
while (1)
    {
    scanf("%1f",&x);
    if (x<0.0)
        goto error;
    printf("\n%f",sqrt(x))
    }
error: printf("\n Square root of negative number");
.........
```

Figure 2.8 - Knots with Goto Statement

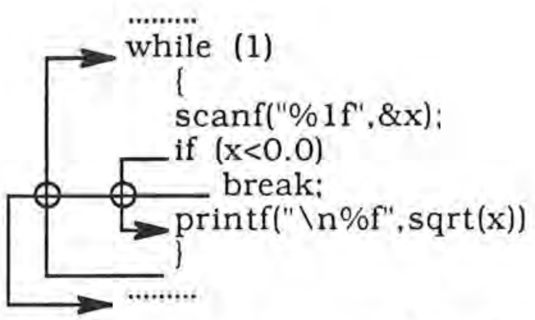
return - The return statement terminates the execution of a function and returns control to the calling environment. Returns are frequently used to exit a routine early, given certain conditions. The return can be thought of as a goto where the label is the function's ending brace. This example, from a communication utility, [IRC] has three knots created by returns.



```
int matches(name1, name2)
char *name1, *name2;
{
.........
for (;(*name1!=NULL)&&(*name2!=NULL);name1++, name2++)
    {
    if (*name2==NULL)
        return(2);
    else
        {
        printf("Error 1\n");
        return(0);
        .........
    }
.........
}
```

Figure 2.9 - Knots with Return Statement

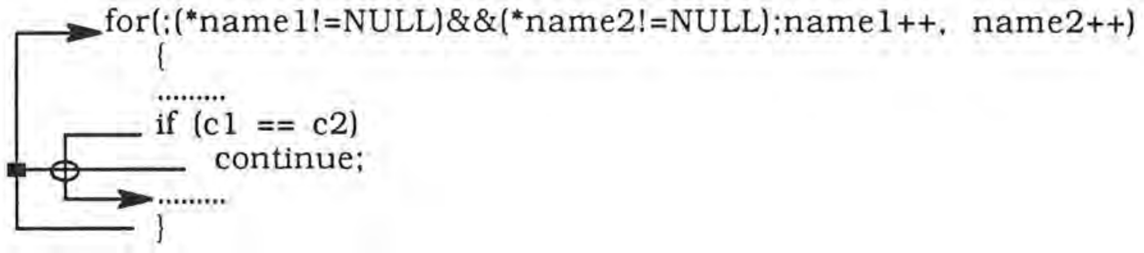
break - The break statement terminates the execution of the innermost enclosing loop or switch statement. The break can be thought of as a goto where the label directly follows the innermost enclosing loop or switch statement. The break statement will cause at least one knot since the flow of control must cross the flow of control for the enclosing loop or switch statement. The break statement can create knots with any additional flow of control lines near it. This example, from *A Book on C,* [Kelly, pg. 164] has two knots.



```
.........
while (1)
   {
   scanf("%lf",&x);
   if (x<0.0)
       break;
   printf("\n%f",sqrt(x))
   }
.........
```

Figure 2.10 - Knots with Break Statement

continue - The continue statement causes the current iteration of a loop to stop and the next iteration of the loop to begin. The continue statement only causes knots if it is enclosed in an if or switch statement. The use of continue without an if or switch statement is unlikely since any code that follows the continue would never be executed. This example, from a communication utility, [IRC] has one knot created by a continue statement.



```
for(;(*name1!=NULL)&&(*name2!=NULL);name1++, name2++)
   {
   .........
   if (c1 == c2)
       continue;
   .........
   }
```

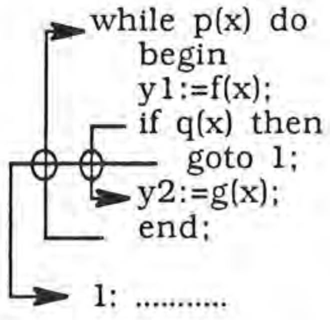
Figure 2.11 - Knots with Continue Statement

2.4 WHY USE KNOT COUNT?

Evangelist lists several widely accepted programming style rules and tests the sensitivity of several metrics to these structuring rules [Evangelist]. He divides these rules into several different categories such as "increase modularity", "add functionality", "simplify the logic", "replace complex branching with complex expressions", "clarify the nature of computation" and "improve readability". We will concentrate on his rules related to control flow complexity: "logical simplification" and "clarify the nature of computation". We will show that in many cases the knot count decreases with the application of the rules.

Logical Simplification

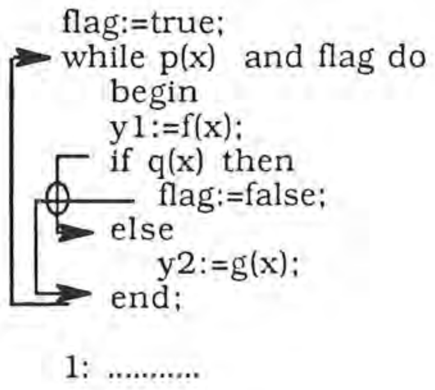
Avoid Multiple Exits from Loops - Evangelist notes that V(G) increases if you modify the code to avoid multiple exits. Jumping out of a loop is frequently listed as an example of unstructured programming [McCabe, pg. 315] and unstructured programming is measured by knot count. An example with two knots looks like [Evangelist, pg. 537]:



```
while p(x) do
  begin
  y1:=f(x);
  if q(x) then
     goto 1;
  y2:=g(x);
  end;

1: ...........
```

Figure 2.12 - Program Segment with Multiple Exits from a Loop

While the alternative which uses a flag has one knot and looks like:



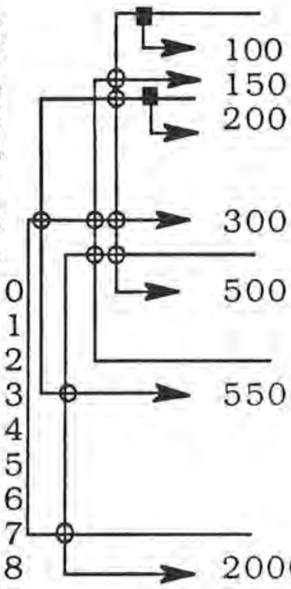
```
flag:=true;
while p(x)  and flag do
    begin
    y1:=f(x);
    if q(x) then
        flag:=false;
    else
        y2:=g(x);
    end;

1: ..........
```

Figure 2.13 - Program Segment without Multiple Exits from a Loop

Note that V(G) is 3 in the first case and increases to 4 in the alternative. Thus McCabe says the second example is more complex while intuition and the knot count say otherwise.

Avoid Unnecessary Branching - V(G) does not change when you follow this precept [Evangelist, pg. 537]. Knot count on the other hand captures the complexity of unstructured branching.

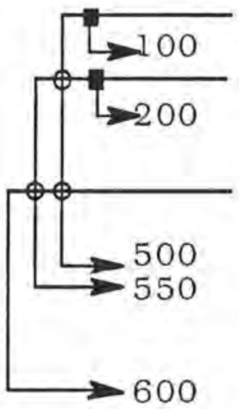
Woodward gives the following example [Woodward, pg. 48]:



```
            CALL TPR
            IF (ZR) 500,500,100
     100    CALL TED
     150    IF (Z3) 200,200,550
     200    ZG = ZG + 1
            ZC = 0
            CALL TCO
     300    CALL TRA
            GOTO 2000
     500    CONTINUE
            Z3 = 1
            GOTO 150
     550    CONTINUE
            CALL TEC
            ZB = ZB + 1
            ZC = ZC + 1
            GOTO 300
    2000    RETURN
            END
```

Figure 2.14 - Program with Unnecessary Branching

The improved version looks like:



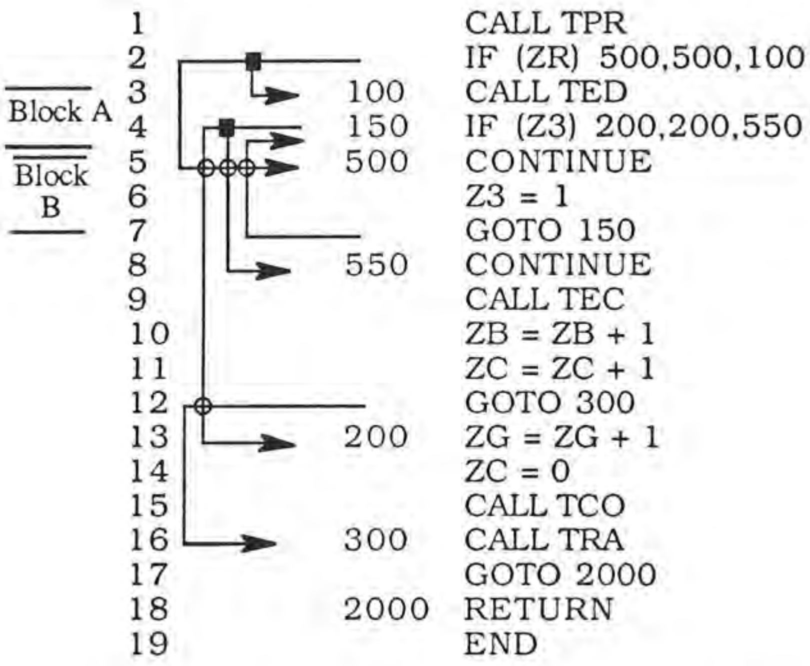
```
            CALL TPR
            IF (ZR) 500, 500, 100
     100    CALL TED
            IF (Z3) 200,200,550
     200    ZG = ZG + 1
            ZC = 0
            CALL TCO
            GOTO 600
     500    Z3 = 1
     550    CALL TEC
            ZB = ZB + 1
            ZC = ZC + 1
     600    CALL TRA
            RETURN
            END
```

Figure 2.15 - Program with Reduced Unnecessary Branching

Both examples have two conditionals so V(G) = 3. Yet the knots have been reduced from nine to three.

Reduce Interdependence Between Statements (or keep related parts together) -Code movement decreases or increases the knot count while V(G) stays constant. Program A, the original version given by Woodward, has nine knots. Simply by exchanging statements 5-9 with 10-17 the knot count is reduced to four.

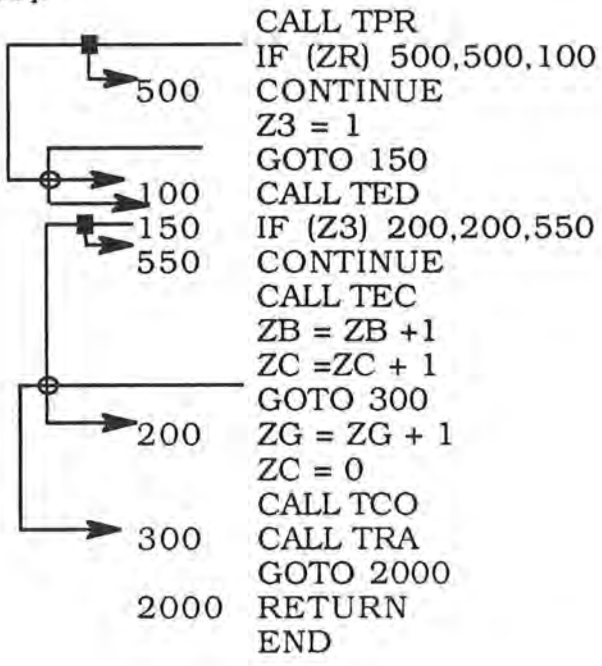
The improved version given looks like [Cook, pg. 117]:



```
 1          CALL TPR
 2          IF (ZR) 500,500,100
 3    100   CALL TED
 4    150   IF (Z3) 200,200,550
 5    500   CONTINUE
 6          Z3 = 1
 7          GOTO 150
 8    550   CONTINUE
 9          CALL TEC
10          ZB = ZB + 1
11          ZC = ZC + 1
12          GOTO 300
13    200   ZG = ZG + 1
14          ZC = 0
15          CALL TCO
16    300   CALL TRA
17          GOTO 2000
18   2000   RETURN
19          END
```

Figure 2.16 - Program with Lower Independence Between Statements

To reduce the knot count further exchange Block A and Block B [Cook, pg. 119].

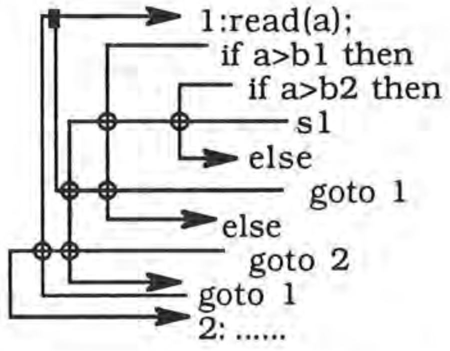


```
          CALL TPR
          IF (ZR) 500,500,100
    500   CONTINUE
          Z3 = 1
          GOTO 150
    100   CALL TED
    150   IF (Z3) 200,200,550
    550   CONTINUE
          CALL TEC
          ZB = ZB +1
          ZC =ZC + 1
          GOTO 300
    200   ZG = ZG + 1
          ZC = 0
          CALL TCO
    300   CALL TRA
          GOTO 2000
   2000   RETURN
          END
```

Figure 2.17 - Program with Reduced Independence Between Statements

Clarify the Nature of Computation

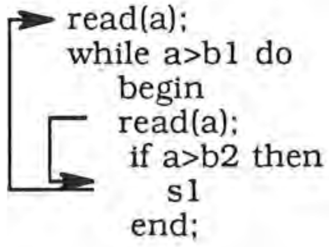
Avoid Else Goto - Evangelist notes that V(G) does not change when avoiding the else goto structure. Avoiding else goto reduces the knot count, since it is directly related to the structuredness of the program segment. The unstructured version below has 6 knots [Lewis, pg. 494]:



```
1:read(a);
  if a>b1 then
    if a>b2 then
      s1
    else
      goto 1
  else
    goto 2
goto 1
2: ......
```

Figure 2.18 - Program Segment with Else Goto

After restructuring the program segment has no knots.



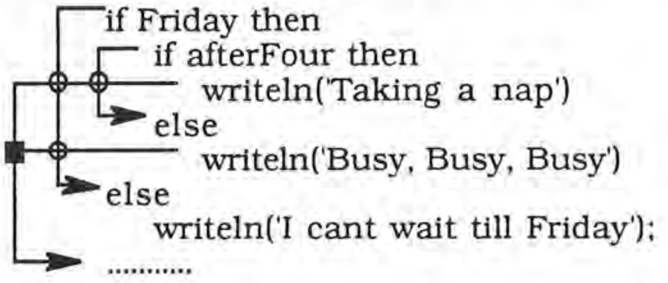
```
read(a);
while a>b1 do
   begin
   read(a);
    if a>b2 then
       s1
    end;
```

Figure 2.19 - Program segment with Else Goto Removed.

Use While Statements - The previous example also illustrates another programming style rule: replace branching structures with while loops. V(G) stays the same in the second version, yet the knot count is reduced to zero. Replacing branching structures with while loops can only reduce the knot count since while statements do not create knots.

Use If Elseif Elseif - Kernighan and Plauger in their book *The Elements of Programming Style* recommend avoiding the use of an if inside a then. (They call this structure a bushy decision tree.) Instead, they suggest using an if elseif elseif... structure. The knot count effectively measures the difference between a bushy decision tree and the if elseif structure. The control flow of an if nested inside a then crosses the control flow of the outer if. The control flow of an if inside an else statement control falls through to the next statement without crossing any other control flows.

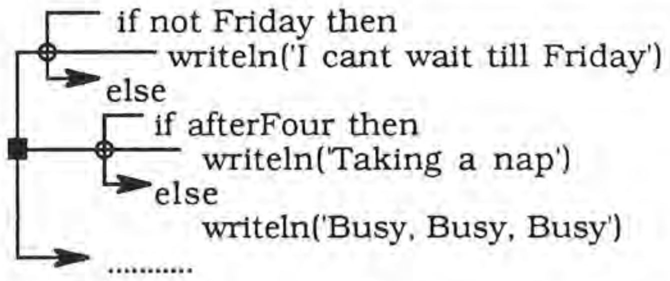
This bushy decision tree has three knots.



```
if Friday then
    if afterFour then
        writeln('Taking a nap')
    else
        writeln('Busy, Busy, Busy')
else
    writeln('I cant wait till Friday');
...........
```

Figure 2.20 - Program Segment with Bushy Decision Tree.

But the equivalent if elseif structure has 2 knots.



```
if not Friday then
    writeln('I cant wait till Friday')
else
    if afterFour then
        writeln('Taking a nap')
    else
        writeln('Busy, Busy, Busy')
...........
```

Figure 2.20 - Program Segment with Bushy Decision Tree Removed.

Kernighan and Plauger give two program segments to show the difference in these structures. The original program segment has a knot count of nine while the improved version has a knot count of five. V(G) does not measure the bushyness of the decision tree since it only counts decisions.

Use Structured Programming - Many of the previous style rules replace unstructured versions with structured ones. A structured program is one that is reducible to a single entry single exit construct

[Lewis, pg. 365]. To reduce a program replace the structured programming constructs (which are all single entry single exit) with a single node or meta construct. A single entry single exit construct does not have multiple exits and therefore can not have flow of control lines crossing. Any knots calculated for a structured program are considered unessential knots, so that a structured program can also be defined as one with zero essential knots. McCabe's measure does not adequately capture the reduction in complexity due to structured programming practices. In summary

Table 2.1 - Sensitivity of Metric to Program Structure

RULE	GROUP[1]	MCCABE[2]	KNOTS
Reduce interdependence	3	increases or unchanged	decreases or unchanged
Avoid Unnecessary Branches	3	unchanged	decreases
Avoid multiple exits	3	increases	decreases
Use if ... elseif ...	5	unchanged	decreases
Avoid .... else goto ...	5	unchanged	decreases

[1] The codes for rule group given by Evangelist are: 3, logical simplification; and 5, clarify the nature of computation.

[2] Using the extended version of McCabe's, where each condition is counted.

This section has included several example code segments where modifications that involve only rearranging the source code statements did not change V(G), but significantly changed the knot count. Several papers have suggested extensions to V(G) such as coupling it with other metrics [Myers, Hansen]. Many of these extensions to V(G) are justified by using a code sample showing an improved metric rating which corresponds to intuition or a style rule. It is easy to find examples which increase or decrease a metric but is important to analyze the probable response of a metric in a generic sense rather then on an example by example basis [Evangelist]. In this section we have argued that, in general, knot count decreases as code becomes more structured. Evangelist suggests that we need metrics specifically defined to measure structured programming practices. In the case of control flow structuring rules the knot count seems to be a good candidate for such a metric.

2.5 Metric Guidelines or Thresholds

There is a general consensus that control flow complexity should be minimized but there is little agreement as to what the guidelines or threshold values should be. McCabe recommends that programmers limit the complexity of modules based on V(G) instead of physical size. He feels that if a modules cyclomatic complexity exceeds 10 then it should be broken into smaller units. This decomposition does not decrease the total complexity of the program however since the number of disjoint components (p) increases.

The threshold of 10 for McCabe's V(G) is thoroughly entrenched in the folklore of computer science. In his paper McCabe discusses the threshold for V(G) only as an interesting observation, but he does not prove or substantiate it. Most professionals realize intuitive complexity is frequently quite different from actual complexity, yet the threshold of 10 for McCabe's is frequently taken as fact rather than opinion. This threshold has been accepted despite the fact that many studies show cyclomatic complexity is little better than lines of code; [Schneidewind, Evangelist] despite the fact that V(G) responds poorly to accepted programming standards; and despite the fact that there have been no confirming studies. There is evidence that relatively high values of V(G) identify error prone modules [Kafura], but there is little evidence to support the threshold of ten.

Some studies have included knot count as one of the metrics, [Gibson, Baker], but none address the knot count threshold specifically. In particular, no studies exist which attempt to set guidelines for the knot count. Threshold values for the knot count would be especially useful for educators since they are valuable in evaluating student programs. In order to evaluate students programs, however, the threshold values must be chosen with care. In the next chapter we describe our experiment that attempted to establish a threshold value for the knot count.

3 Experiment

This section presents a study of control flow complexity metrics using C and Pascal students. The goal of the experiment was to find a general threshold value for knot count for students.

3.1 Subjects

The experiment was conducted in four programming classes at Oregon State University; one beginning, two intermediate and one advanced. The 86 Pascal subjects were from two classes: CS212, a second class in programming, and CS317, a data structures class. CS212 is a prerequisite for CS317. Both classes use Pascal for class assignments. The 82 C subjects were from two classes; CS312, a class where students first learn C and UNIX, and CS431 a senior level applications programming class. CS312 is a prerequisite for CS431. Students in CS431 are given the choice of doing class work in Pascal or C.

3.2 Materials

The materials were composed of three parts: a background survey with directions and two programs packages.

Background survey

The background survey was given to determine the expertise of subjects. Moher and Schneider state "For researchers using student subjects the message is clear: experimental designs must take into account both experiential and aptitudinal differences in subject pools" [Moher]. Moher and Schneider recommend using grade point averages and number of computer science classes. Since Oregon State is on the quarter system (three quarters per year), we felt that it would be difficult for subjects to accurately remember and count every computer science class. We were also concerned that the subject's interpretation of a computer science class would be different. For instance must a computer science class be offered by the Computer Science Department? Are classes in mathematics, engineering, or statistics which require programming or the use of program packages to be interpreted as computer science courses? To overcome these problems we asked the subjects to give the number of classes in which they used the experimental language (either C or Pascal). Also included in the background survey was class level, grade point averages and the number of programming languages used in classes or professionally. To minimize reporting errors GPAs were translated into letter grades where:

3.7 - 4.0 = A
2.7 - 3.6 = B
2.0 - 2.6 = C

The subjects completed the background survey before the experimental tasks were performed. The results are summarized in Appendix A.

A chi-square test was run on all data and confirmed that there was little significant difference in background between the cells.

Program Package

Each program package consisted of two pages stapled together; the first a comprehension quiz and the second a program listing. Each subject received two program packages, one with a low knot count (easy program) and one with a high knot (hard program).

Comprehension Quiz

The six comprehension questions alternated forward and backward reasoning questions. Forward reasoning questions are those which ask what the program will output from a given input; backward reasoning questions ask what input will produce a given program output. Figure 3.1 shows some sample questions; questions one and three are forward reasoning and questions two and four are backward reasoning.

1. How many syllables will this function calculate for the word briar.

2. Give any combination of four letters that will return a syllable count of 0. The result need not be a valid word.

3. How many syllables will this function calculate for the word ciao?

4. Given the word chick, give a single vowel (a,e,i,o,u) that can be added after the i so that the syllable count remains unchanged at 1. The result need not be a valid word.

Figure 3.1 - Sample Comprehension Quiz Questions

In one class, CS431, one question for the easy program was thrown out due to an error discovered during administration. While this error was corrected before the experiment was administered to the second C class (CS312), CS431 was graded out of a total of five;

three forward and two backward reasoning questions.

Program Modules

Since we were interested in studying the knot count, care was taken to chose program modules which had widely varying knot counts but fairly consistent cyclomatic complexities. In addition the modules were selected from two different problem domains and did not assume any domain specific knowledge.

All modules were implemented using the same style guidelines. Typographical style, which is an important factor of comprehension [Oman], was the same for all modules. Meaningful variable names were chosen. No comments were used since it is difficult to control the affect of comments on subject performance.

Pascal

The Pascal module with a low knot count, procedure **value**, is a simplified version of a program in [Jones, pg. 35-36] and was translated to Pascal from Modula II. Procedure **value** converts strings containing scientific notation into three integers; whole, fraction and power. The module converts the digits before a '.' into the integer whole, the digits after the '.' but before an 'E' (or 'e') into the integer fraction and any digits after the E into the integer power. For example the string '123.45E6' would return whole = 123, fraction = 45, and power = 6. The original program accepts a wider range of inputs (for example '+' or '-') than the version used in this expirement. Procedure **value** uses two string functions; StringToInt and Copy, which were explained at the top of the comprehension quiz.

The high knot count module, (**syllableCount**), estimates the number of syllables in a word by counting the number of vowels. This module Special rules are applied when contiguous vowels are encountered. For the **syllableCount** module the special rules only occur when the first vowel of the pair is an "a" or an "i". As an example consider the word "special". The **syllableCount** module gives a correct count of two; one for the combination ia and one for the e. (spe-cial) This method of counting syllables is not totally accurate (a perfect system would require a dictionary search) but it does provide a reasonable estimation. (The orignal algorithm used a large number of rules and achieved 99.5 percent accuracy [Fang].) The syllable count module is a simplified version of the program used in [Cook86, pg. 340]. That version implements several additional rules, which were deleted for this experiment.

In choosing our modules we attempted to hold McCabe's V(G) and lines of code reasonably constant and only vary the knot count. Metric values for all modules and are shown below:

Table 3.1 - Metric Values for Pascal Programs

Procedure	K	$V(G)_1$	$V(G)_2$	LOC	V	E
value	8	10	11	66	1430	57711
syllableCount	22	13	13	57	905	26560

K= Knot Count V=Halstead's Volume E=Halstead's Effort LOC=Lines of Code
$V(G)_1$=Original Cyclomatic Complexity $V(G)_2$=Extended Cyclomatic Complexity

C

The low knot count module, **findOverlap**, calculates the area of intersection of two rectangles. This module was developed specifically for this experiment. The overlap module uses the upper right corner and lower left corner coordinates for two rectangles as input and finds the rectangle created by the intersection. If there is an overlap, it computes the area of the overlap rectangle otherwise it returns 0.

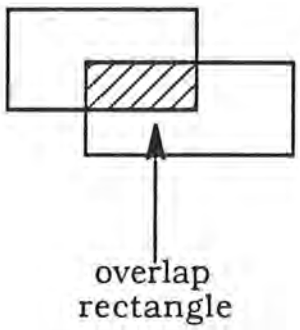


Figure 3.2 - Computing Overlap Rectangle

The module with a high knot count is a version of the syllable counting routine used in the Pascal experiment. The major difference between the C and Pascal versions of the **syllableCount** routine is caused by the lack of sets in C. Two sets are used in the Pascal version; the set 'vowels' and the set 'lstc'. The C function 'int isVowel(c)' (which returns 1 (true) if the character is a vowel (a,e,i,o,u) and 0 (false) otherwise) is used in place of the set 'vowels'. The set 'lstc' is simplified to the character constant 'c'. Thus the Pascal version of the routine has the statement

```
if not(previous in lstc)
```

which is replaced by

```
if (previous!='c')
```

in the C version. This means that the syllable count for some words will be overestimated by the C function but not by the Pascal module. For example the Pascal module will return a syllable count of 1 for the word liar while the C function will return a syllable count of 2. The C version also handles strings slightly differently to avoid bound errors. The C function had the following statements:

```
if (i==(length-1))
  second = ' '
else
  second = word[i+1];
```

Which creates an extra knot.

Metric values for both functions are shown below:

Table 3.2 - Metric Values for C Programs

Procedure	K	V(G)1	V(G)2	LOC	V	E
findOverlap	9	8	9	45	805	24645
syllableCount	23	14	14	57	873	35082

K= Knot Count V=Halstead's Volume E=Halstead's Effort LOC=Lines of Code
V(G)1=Original Cyclomatic Complexity V(G)2=Extended Cyclomatic Complexity

3.3 Task

The materials were handed out randomly to all subjects. The subjects were told that they could not use any reference materials. First the subjects were told to read and fill out the background survey carefully. When the background survey was finished the subjects were told to answer the comprehension questions as accurately and quickly as possible and that they might not finish in the 10 minutes provided. They were told to work on the first program and not start on the second program until directed to do so. When the time was up the subjects were asked to turn to the second program and not to return to the first program.

Experimental Design

Our experimental design had to consider three factors: first the design had to consider subject variability (studies have shown a wide variation in programer ability [Brooks, pg. 209]); second we felt subjects would not perform optimally for longer than 30 minutes; any longer would fatigue the subjects and effect their performance; and third we needed to allow enough time to complete each program.

We considered three alternative experimental designs:

1. Have each subject only work on one program, either the hard or the easy program for 20 minutes.
2. Use a within subjects design where each subject worked on both programs for 15-20 minutes. (In a within subjects design each subject is exposed to all levels of the experimental variable; in this case each subject would be given two programs one with a low knot count and one with a high knot count.)
3. Use a within subjects design where each subject worked ten minutes on each program.

Administration added 5-10 minutes to the total time needed. Even though design option one only required thirty minutes it was rejected because it did not address the problem of subject variability. We chose a within subjects methodology because it has been the most effective way to minimize subject variability [Brooks, pg. 209]. Design option two, was rejected because it would require 45 minutes of concentrated work by the subjects. We concluded that design option three was the best choice - the subjects could perform at their best for at most thirty minutes, and the subject variability issue was resolved.

Each module had two versions of the question sets. One with a forward reasoning question first and one with a backward reasoning question first. This gave a total of 4 versions (2 (modules) X 2 (questions)) of the materials. Each class was divided randomly into 4 cells; each having one of the four possible orders. The cells were:

cell 0 -	easy module, forward reasoning question first hard module backward reasoning first
cell 1 -	hard module, forward reasoning question first easy module backward reasoning first
cell 2 -	easy module backward reasoning first hard module forward reasoning first.
cell 3 -	hard module backward reasoning first easy module forward reasoning first.

Dependent measure

The dependent measure was the percentage of questions attempted that were answered correctly. So that

$$\text{performance} = \frac{\text{number correct}}{\text{number attempted}}$$

The dependent measure, which is referred to throughout this paper as performance, was only computed for subjects who attempted both programs, since it is undefined when the number attempted is 0.

The dependent measure was chosen for two reasons: accuracy was stressed in the instructions and the short time allotted had a significant impact on the number of questions that the subjects could complete. Accuracy was stressed in two ways during the instructions;

first the subjects were told to work as accurately as possible, and second they were told that they might not finish in the time allotted. We believe that telling them that they might not finish reinforced the emphasis on accuracy. The short amount of time allowed impacted the number of questions subjects could reasonably answer. No subject group was able to complete more then 65% of the questions in the time allotted. Figure 3.3 shows the average percent of questions attempted for the easy and hard programs for each of the four classes. Note that CS 431 was the only classes that attempted more questions for the easy program than the hard one. The lower number of questions attempted on the easy program suggests that in most cases the subjects spent more time trying to answer the questions for the easy program.

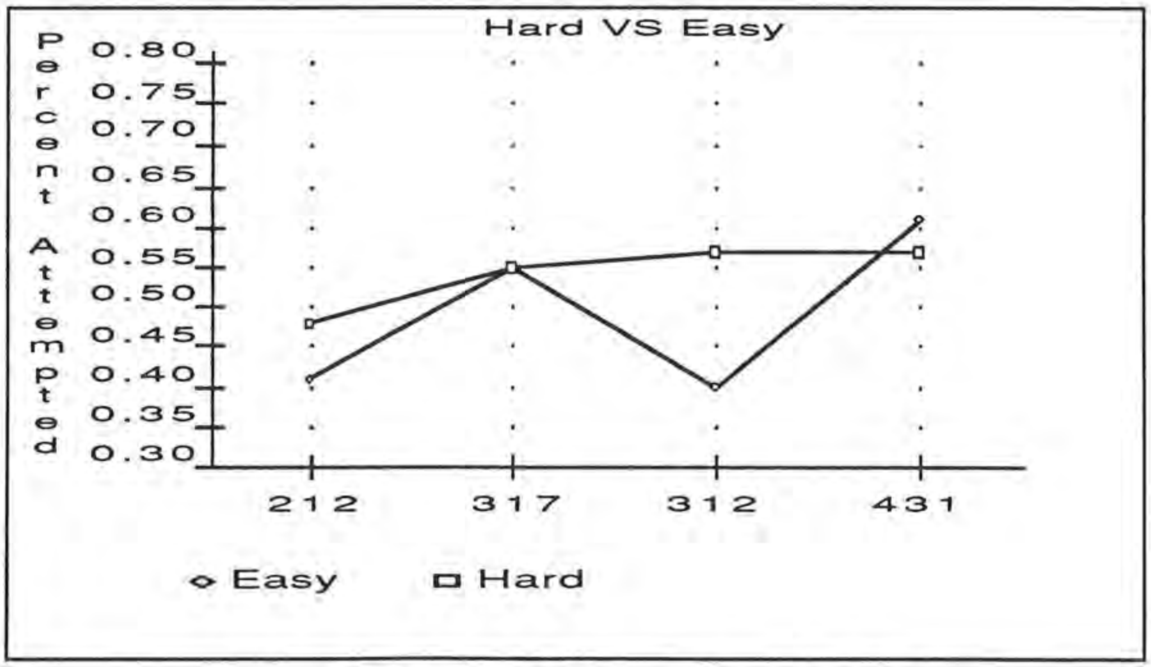


Figure 3.3 - Percent Attempted on Both Programs

To complete all six questions a subject would have to average less then two minutes per question. Table 3.3 shows the percent of the students who answered all questions, which ranged from under 5% to a little over 30%. In retrospect a comprehension quiz with four questions would probably have been better, although it would not have exercised all paths in the program.

Table 3.3 - Percent of Subjects Who Attempted All Questions

Class	Easy Program	Hard Program
212	.17	.22
317	.26	.26
312	.04	.32
431	.08	.22

Another example illustrates why measuring a subjects performance based on the number attempted is important. Consider the number of questions attempted for C classes. Figure 3.4 shows that the subjects in the two cells which had forward reasoning questions first attempted more forward reasoning questions, and similarly those subjects in cells which had backward reasoning questions first attempted more backward reasoning questions. This suggests that the subjects attempted to answer the questions in order.

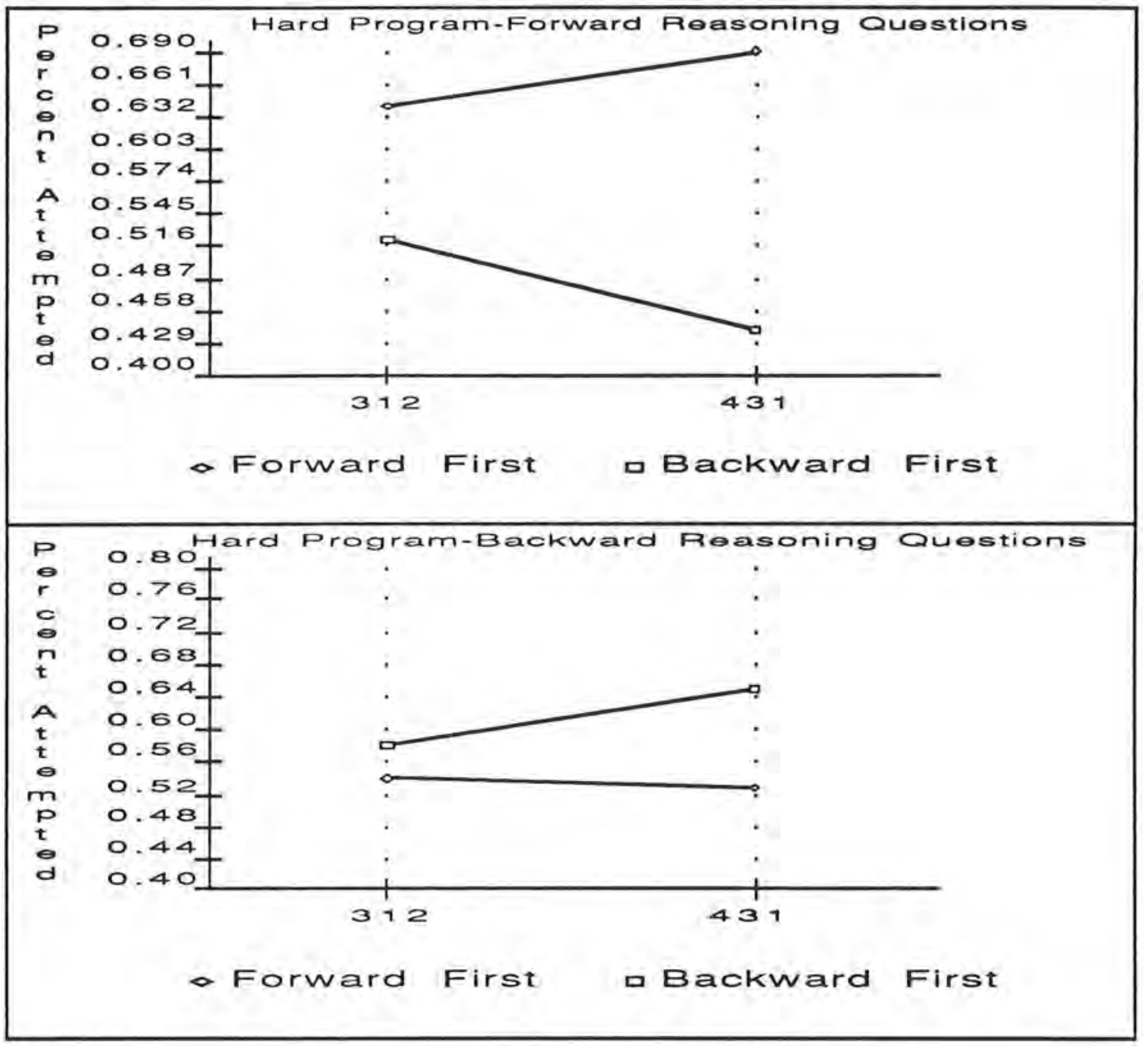


Figure 3.4 - C Subjects - Percent Attempted Forward vs. Backward Reasoning Groups

3.4 Results

This section discusses the results of the experiment. First we discuss the results of the Pascal experiment, then the results of the C experiment, and finally we draw some conclusions from the results. The measure of subject performance is the number of questions answered correctly divided by the number of questions attempted.

Pascal

This section describes the results of the experiment for the Pascal classes. First we discuss performance on the low verses the high knot count programs. We also consider some secondary results in this section including the subjects performance on forward and backward reasoning questions and the effect of program ordering of programs on the results.

The graph in Figure 3.5 illustrates the difference in performance between the hard and easy Pascal programs. The differences are interesting but not statistically significant. Both classes performed below our expectations. It is surprising that the CS 212 classes had a slightly higher performance score than CS 317. The CS 212 subjects attempted fewer questions, so possibly they were more careful.

The poor performance of both classes suggests that students in their second or third programming class should limit the knot count of their programs to 7. (The procedure **value**, the easy Pascal program had a knot count of 8.) Further research is needed to validate this limit.

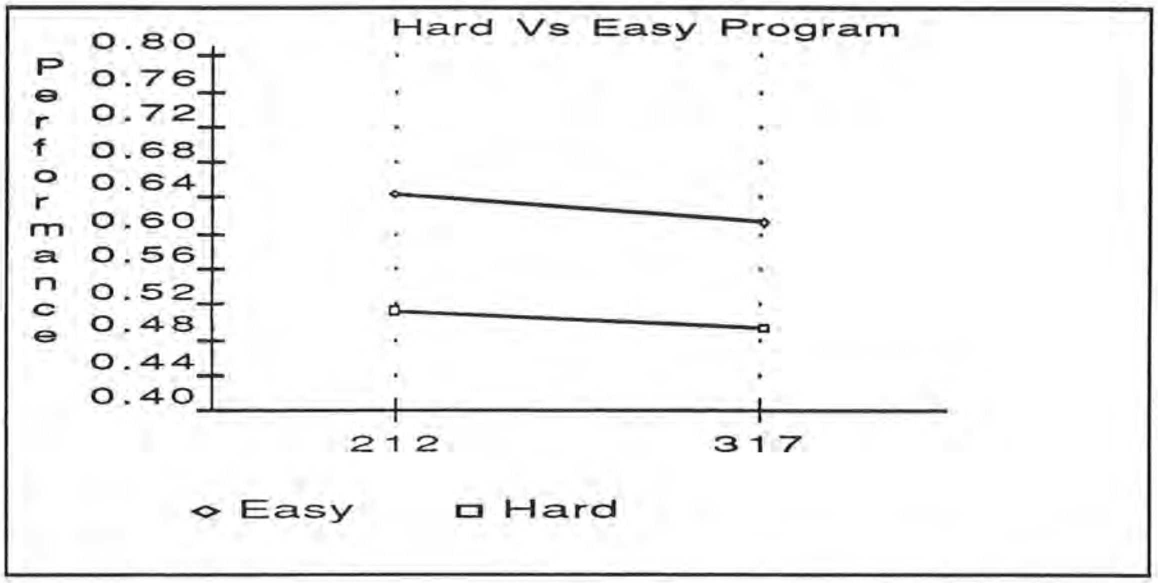


Figure 3.5 - Pascal Subjects - Performance Hard vs. Easy Program

Table 3.4 - Pascal Subjects - Total Class Performance

Class	Easy	Hard
212	0.64	0.51
317	0.61	0.49

1 Total Class Performance = $\frac{\text{Total Number Correct for Class}}{\text{Total Number Attempted for Class}}$

Table 3.5 - Pascal Subjects - Average Performance by Student

Class	Easy	Hard	t-value	ρ
212	.56 (.39)	.52 (.33)	.61	.54
317	.52 (.34)	.47(.38)	.44	.67

(Standard Deviation)

Secondary Results

The Pascal subjects attempted more forward reasoning questions than backward reasoning questions for both programs (See Figure 3.6).

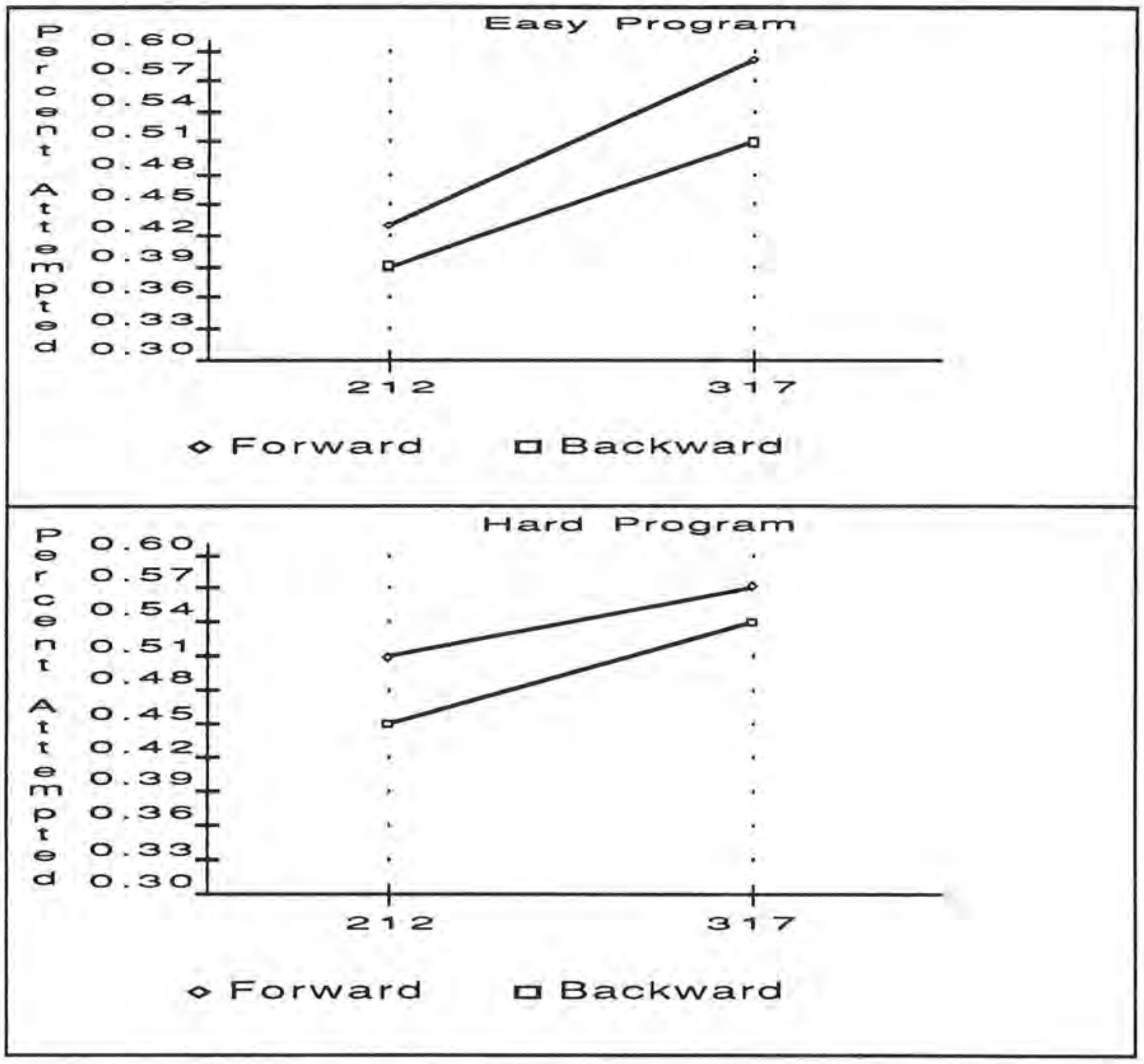
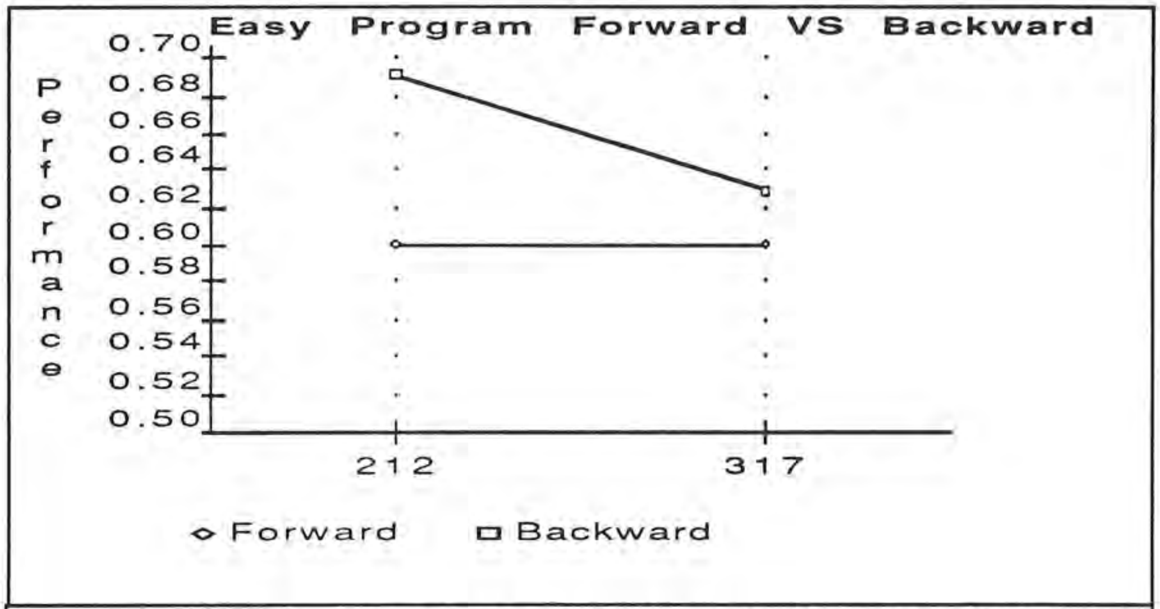


Figure 3.6 - Pascal Subjects - Percent Attempted Forward vs. Backward Reasoning Questions

From Figure 3.7 we see that in general the subjects performed better on backward reasoning questions. One possible explanation is that there was a certain amount of false confidence concerning forward reasoning questions. Another possible explanation is that backward reasoning questions require more time, which pays off in better understanding.



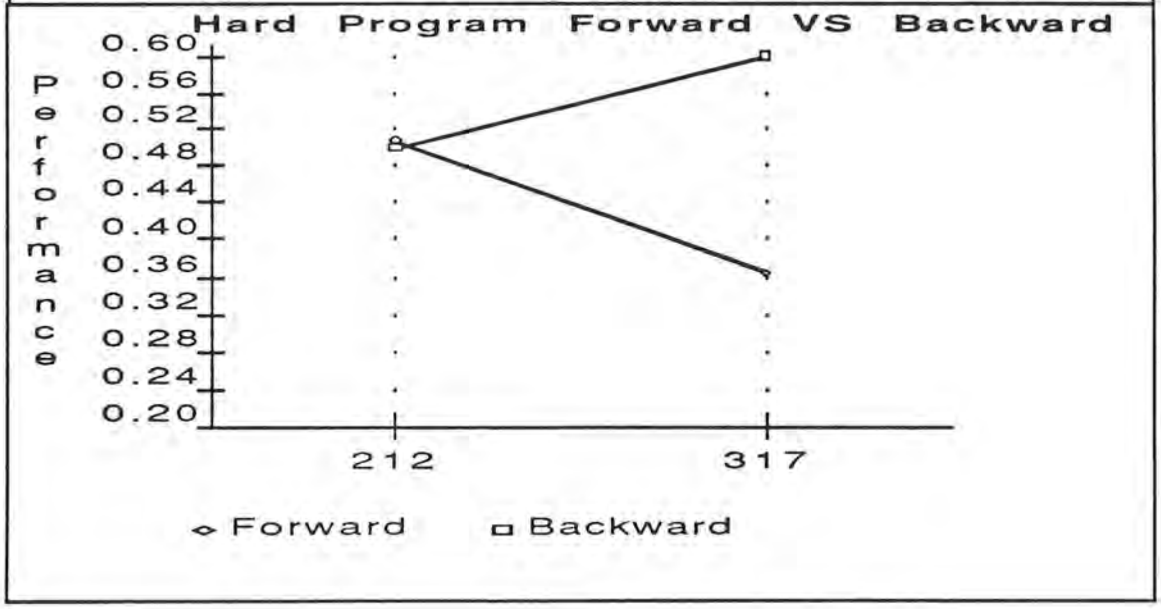


Figure 3.7 - Pascal Subjects - Performance Forward vs. Backward Reasoning Questions

Two subject groups had the hard program first and two had the easy program first. The two programs were alternated to lessen learning affects and fatigue. The following two graphs (Figure 3.8) show the difference in performance between those subjects who had the hard program first (hard first) and those subjects who had the easy program first (easy first).

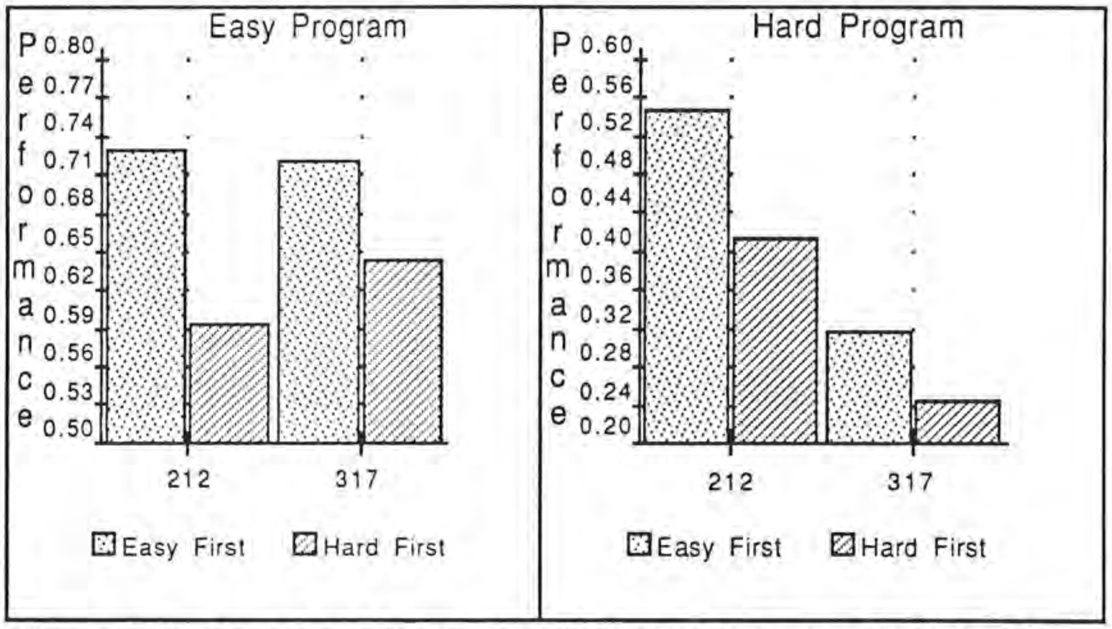


Figure 3.8 - Pascal Subjects - The Effect of Program Ordering On Performance

Table 3.6, which shows the average performance on the first program whether hard or easy, suggests that any unknown interaction between learning affects and fatique was mitigated by alternating the programs in the within subjects design.

Table 3.6 - Pascal Subjects - Performance on First vs. Second Program

	212	317
First Program	0.57	0.48
Second Program	0.57	0.48

C

This section discusses the results of the experiment for the C classes. First we discuss performance on the low verses the high knot count programs. As with the Pascal section, we also illustrate several different secondary results in this section.

The graph in Figure 3.9 illustrates the difference in performance between the hard and easy program. For both classes a t-test indicated a significant difference in performance for the easy and hard programs.

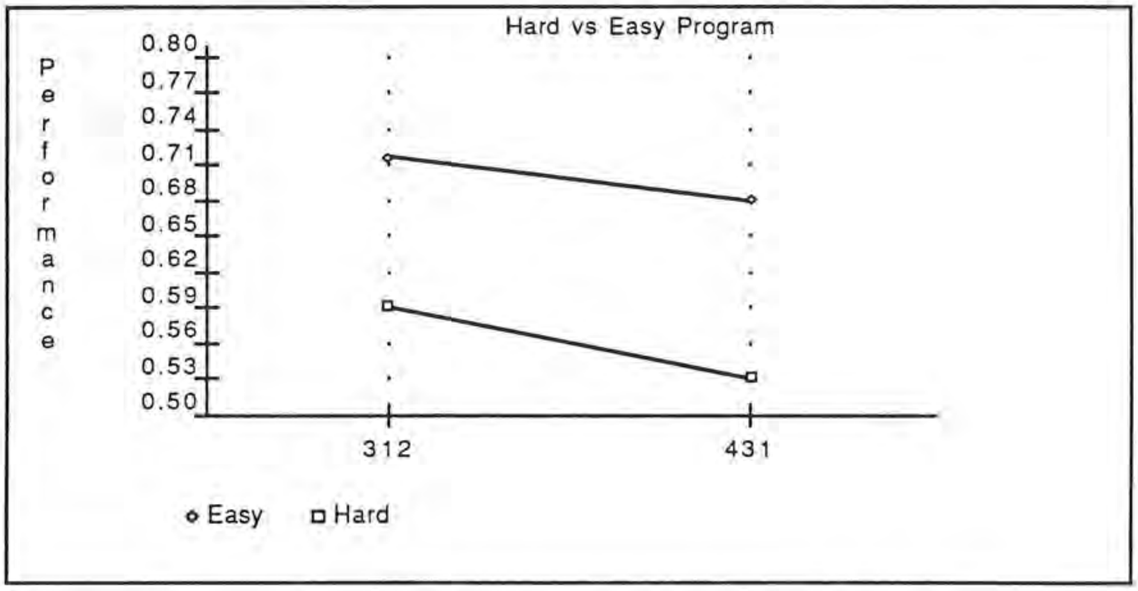


Figure 3.9 - C Subjects - Performance Hard vs. Easy Programs

Table 3.7 - C Subjects - Total Class Performance[1]

Class	Easy	Hard
312	0.71	0.59
431	0.68	0.53

1 Total Class performance = $\frac{\text{Total Number Correct for Class}}{\text{Total Number Attempted for Class}}$

Table 3.8 - C Subjects - Average Performance by Student

Class	Easy	Hard	t-value	ρ
312	.76 (.33)	.51 (.35)	3.29	.002
431	.68 (.33)	.52 (.33)	1.73	.094

(Standard Deviation)

The results suggest that students in upper division classes should limit the knot count of their C programs to 20.

Secondary Results

In general the C subjects attempted more forward reasoning questions than backward reasoning questions. See Figure 3.10.

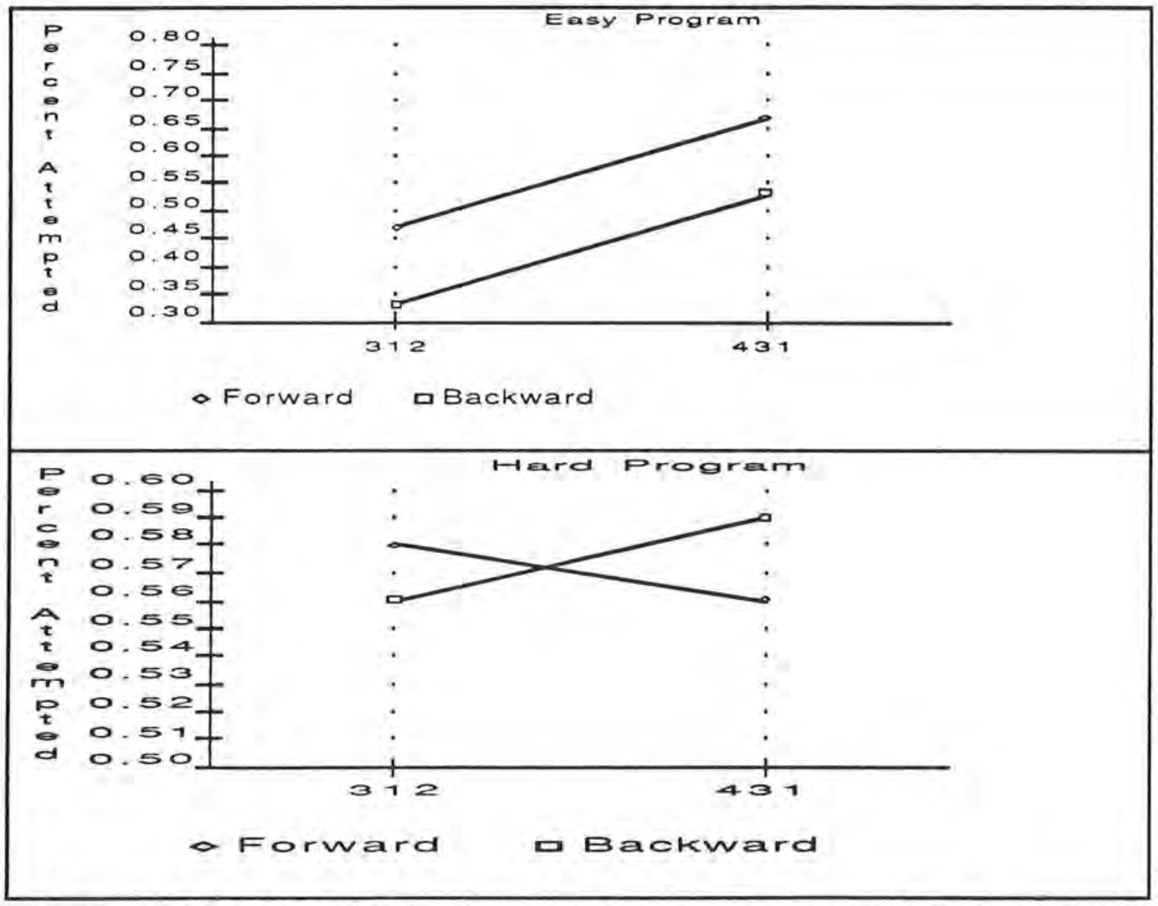


Figure 3.10 - C Subjects - Percent Attempted Forward vs. Backward Reasoning Questions

Figure 3.11 shows that on the hard program the subjects performed better on backward reasoning questions; on the easy program the subjects performed better on the forward questions. These results bring up several interesting questions. Is there an interaction between difficulty of the program and the ability to answer forward vs. backward reasoning questions or were one set of questions easier than the other? In this experiment when the students performed poorly on a program (Pascal subjects on both programs and C subjects on the hard program), they did better on the backward reasoning questions. When they performed reasonably well on a program (C students on the easy program) they did better on forward reasoning questions.

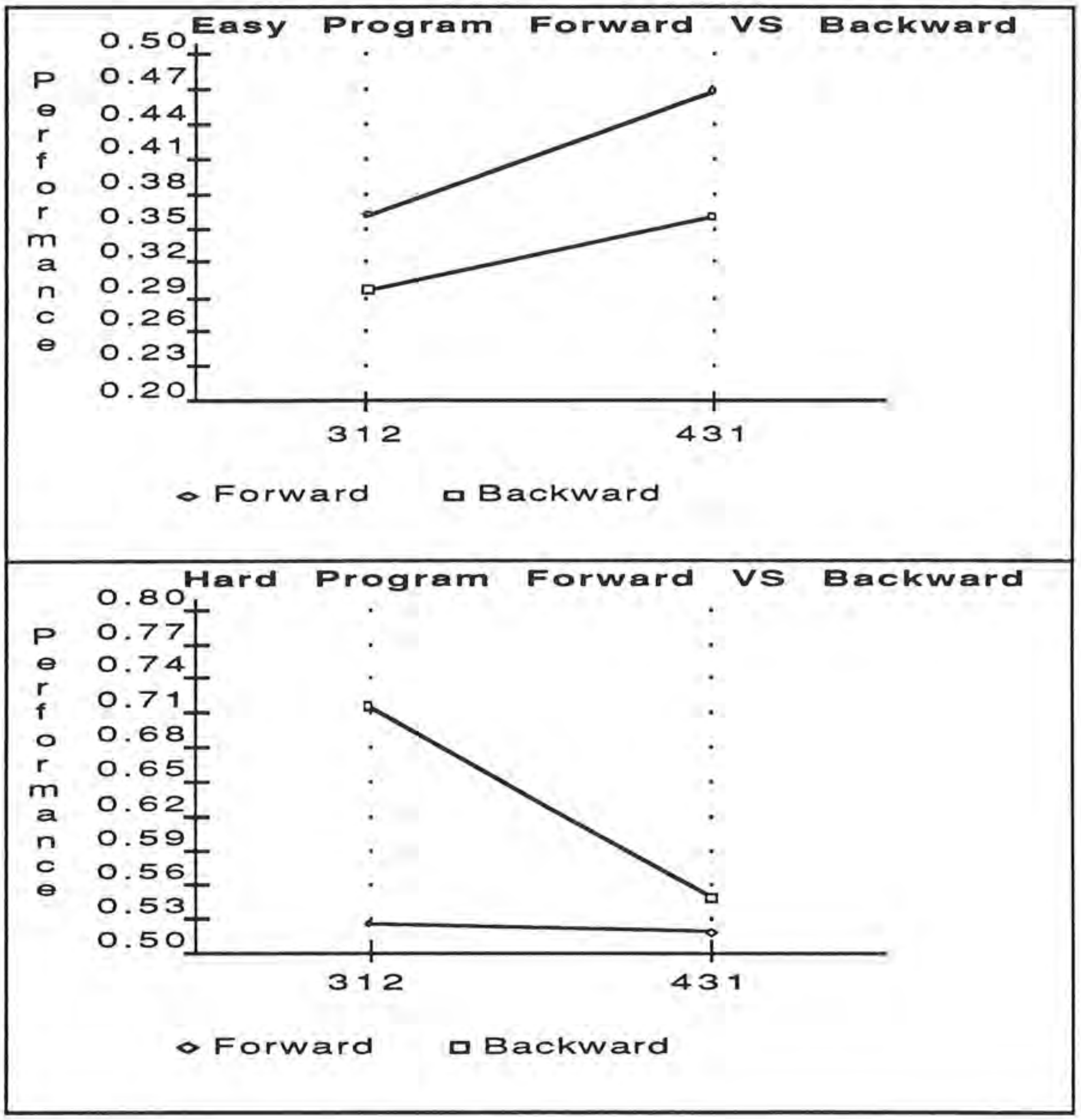


Figure 3.11 - C Subjects - Performance Forward vs. Backward Reasoning Questions

More interesting questions arise when you look at Figure 3.12 where the subjects were grouped by what type of question they had first. Two groups had a forward reasoning question first (forward first) and two groups had a backward reasoning questions first (backward first). What effect does the amount of time have on the ability to understand forward and backward reasoning questions correctly? If the amount of time allotted for each question was strictly controlled, would students perform equally as well on forward as on backward reasoning questions? It would appear that for the easy program starting out with a backward question first is an advantage. For the hard program the results are inconclusive. Perhaps subjects who attempt a backward reasoning question first spend more time trying to understand a program before answering, and this pays off in increased understanding. Further research is needed to investigate these questions.

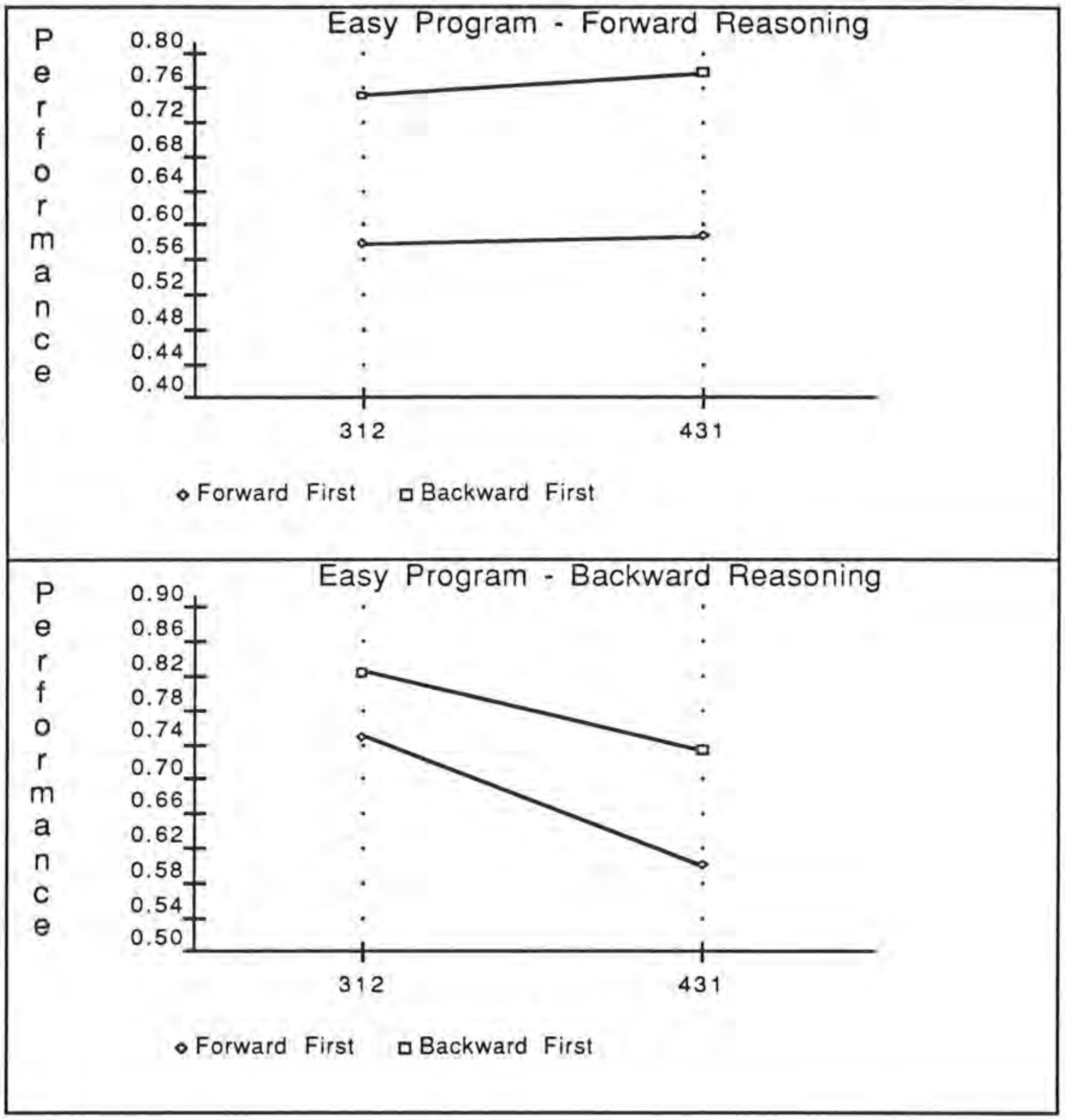


Figure 3.12 - C Subjects - Performance by Group

Two subject groups had the hard program first and two had the easy program first. The two programs were alternated to lessen learning affects and fatigue. The following two graphs (Figure 3.13) illustrate that alternating the programs was an important precaution. These graphs show the difference in performance between those subjects who had the hard program first (hard first) and those subjects who had the easy program first (easy first).

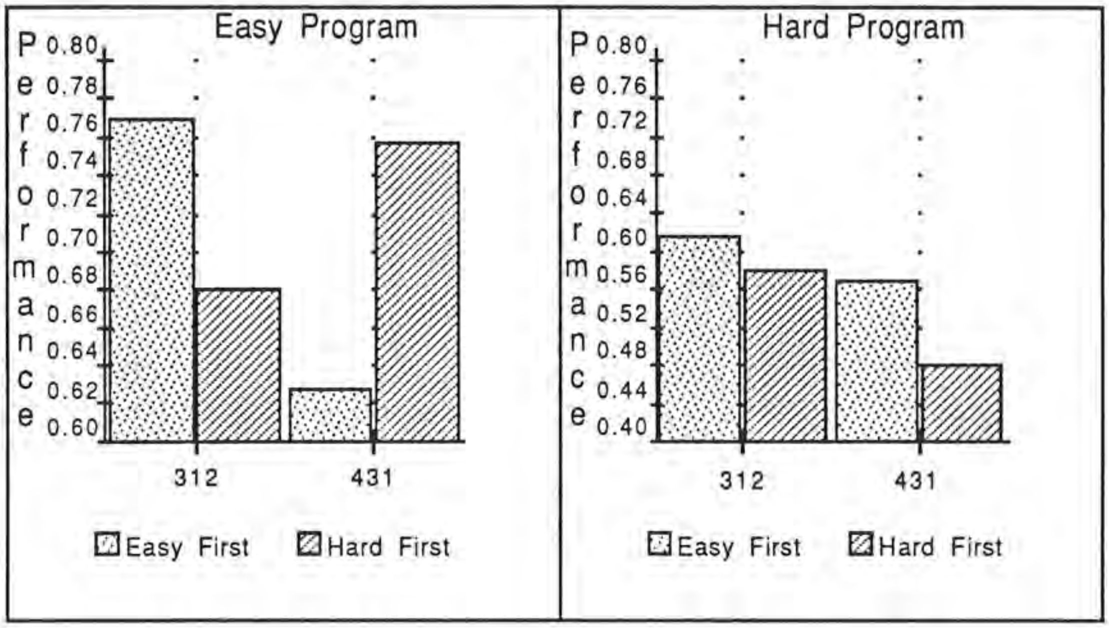


Figure 3.13 - C Subjects - Effect of Program Ordering On Performance

Discussion For Pascal and C

Table 3.9 gives the metric values for McCabe's and the knot count for all four programs:

Table 3.9 - Control Flow Metrics

Procedure	K	$V(G)_1$	$V(G)_2$
value	8	10	11
findOverlap	9	8	9
syllableCount (Pascal)	22	13	13
syllableCount (C)	23	14	14

There is a considerable difference in the knot counts for the easy and hard Pascal and C programs while the differences in V(G) are relatively small. Note that V(G) for the easy program was below

McCabe's threshold of 10 and V(G) for the hard program was above the threshold. For Pascal we found no significant difference in performance scores between the hard and easy programs. Also the performance of the subjects was quite poor. However, for the C program we did find a significant difference in performance between the easy and hard programs. This suggests that the threshold of 10 does not hold across languages and that metric threshold values for different programming languages should be set differently based on experience and/or language.

Table 3.10 gives the Lines of Code (LOC), Halstead's Volume (V) and Effort (E) metric values and the average class performance for each program. This table points out lack of correlation between these metrics and performance. Lines of Code, Halstead's Volume and Effort decrease from the easy to the hard Pascal program yet the performance also decreased. In addition the relatively small changes in these measures from the easy to the hard program does not seem to account for the decrease in performance.

Table 3.10 - Size Metrics

Procedure	LOC	V	E	Performance
value	66	1430	57711	0.63
findOverlap	45	805	24645	0.70
syllableCount (Pascal)	57	905	26560	0.50
syllableCount (C)	57	873	35082	0.56

4 Conclusion

This research was motivated by the need to set threshold values for control flow metrics. First we showed that the knot count is a useful control metric because of its ability to gauge the structuredness of code; then we described the results of an experimental study to determine the threshold value for the knot count. This work establishes a threshold value of 20 when evaluating upper division student programmers using C. Unfortunately we were unable to determine a threshold for beginning Pascal students, although we saw some evidence the threshold should be seven or less.

This research has uncovered several interesting questions:

- Should different threshold values for metrics be set differently based on experiential factors or programming language?
- Should a knot count threshold value of 7 be used for beginning and intermediate Pascal students?
- Should 10 be a knot count threshold for beginning C students and 20 for advanced students?
- What is the interaction among degree of comprehension, order of questions, time to answer questions and the ability to perform forward vs. backward reasoning tasks?

Bibliography

[Baker]	Baker, A. L., Zweben, S. H. "A Comparison of Measures of Control Flow Complexity". *IEEE Transactions on Software Engineering*, Vol. SE-6 (6), November 1980, pp. 506-512.
[Brooks]	Brooks, R. E. "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology". *Communications of the ACM*, Vol. 23 (4), April 1980, pp. 207-213.
[Conte]	Conte, S. D., Dunsmore, H. E., and Shen, V. Y. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1986.
[Cook]	Cook, C. R. "Graph Theoretic Program Complexity Measures". *Proceedings West Coast Conference on Combinatorics, Graph Theory, and Computing*, Humboldt State University. September 5-7, 1979, pp. 109-124.
[Cook86]	Cook, C. R., Harrison, W. "Are Deeply Nested Conditionals Less Readable". *The Journal of Systems and Software*, Vol. 6 (4), 1986, pp. 335-341.
[Evangelist]	Evangelist, W. M. "Program Complexity and Programming Style". *Proceedings Int'l Conference on Data Engineering*, Los Angeles, CA. April 24-27, 1984. IEEE Silver Springs, MD., pp. 534-541.
[Fang]	Fang, I. E. "By Computer: Flesch's Reading Easy Score and a Syllable Counter". *Behavioral Science*, 13, 1968, pp. 249-251.
[Folts]	Folts, J., Beekman, G., Johnson, M. *Oh! Turbo 5 Pascal!*. W. W. Norton & Company, New York, 1988.
[Gibson]	Gibson, V. R., Senn, J. A. "System Structure and Software Maintenance Performance". *Communications of the ACM*, Vol. 32 (3), March, 1989, pp. 347-357.
[Hansen]	Hansen, W. J. "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)". *ACM SIGPLAN Notices*, Vol. 13 (3), March, 1978, pp. 29-33.

[Henry]	Henry, S., Kafura, D. "Software Structure Metrics Based on Information Flow". *IEEE Transactions on Software Engineering*, Vol. SE-7 (5), September, 1981, pp. 510-518.
[Kafura]	Kafura, D., Canning, J. "A Validation of Software Metrics Using Many Metrics and Two Resources". *Proceedings: 8th International Conference on Software Engineering*, 1985, pp. 378-213.
[Kelly]	Kelly, A., Pohl, I. *A Book On C*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1984.
[IRC]	Internet Relay Chat V2.0, University of Oulu, Computing Center.
[Jones]	Jones, "Entering Data: A "Real" Problem". *Journal of Pascal, Ada, & Modula-2*, Vol. 7 (3), May-June, 1988, pp. 34-37.
[Kernighan]	Kernighan, B. W., Plauger, P. J. *Elements of Programming Style*, McGraw-Hill, New York, NY, 1974.
[Lewis]	Lewis, T. G. *CASE Computer-Aided Software Engineering*. InformaTex Press, Corvallis, OR, 1988.
[McCabe]	McCabe, T. J. "A Complexity Measure". *IEEE Transactions on Software Engineering*, Vol. Se-2 (4), December, 1976.
[Myers]	Myers, G. J. "An Extension to the Cyclomatic Measure of Program Complexity". *ACM SIGPLAN Notices*, Vol. 12 (10), October, 1977, pp. 61-64.
[Moher]	Moher, T., Schneider, M. "Methods for Improving Controlled Experimentation in Software Engineering". *Proceedings: 5th International Conference on Software Engineering*, 1981, pp. 224-233.
[Oman]	Oman, P. W. Cook, C. R. "Typographic Style is More than Cosmetic". Oregon State University Computer Science Dept., Tech Report 89-60-5, 1989.

[Schneidewind]	Schneidewind, N. F., Hoffmann, H. M. "An Experiment in Software Error Data Collection and Analysis". *IEEE Transactions on Software Engineering*, SE-5 (3), May, 1979, pp. 276-286.
[Walker]	Walker, J. T. *Using Statistics for Psychological Research: An Introduction*, Holt, Rinehard and Winston, New York, 1985.
[Woodward]	Woodward, M. R., Hennell, M. W., Hedley, D. "A Measure of Control Flow Complexity in Program Text". *IEEE Transactions on Software Engineering*, Vol. SE-6 (1), January, 1979, pp. 45-50.

Apendix A - Background Data

CS212 Background Data
CS317 Background Data
CS312 Background Data
CS431 Background Data

CS212 Background Data

Class

Cell	Freshman	Sophmore	Junior	Senior/Post-Bac
0	4	5	4	3
1	9	3	3	2
2	3	4	2	5
3	3	3	1	8
Total	19	15	10	18

Overall GPA

Cell	A	B	C
0	4	10	2
1	3	13	1
2	4	6	3
3	0	8	5
Total	11	37	11

Computer Science GPA

Cell	A	B	C
0	7	5	3
1	4	12	0
2	4	5	3
3	3	6	2
Total	18	28	8

Number of Classes Using Pascal

Cell	1	2	3 or More
0	1	13	2
1	3	8	6
2	1	10	3
3	0	14	1
Total	5	45	12

Number of Languages

Cell	1	2 or More
0	10	3
1	12	5
2	8	6
3	6	9
Total	36	23

CS317 Background Data

Class

Cell	Freshman/ Sophmore	Junior	Senior or Post-Bac
0	3	1	1
1	1	5	0
2	3	2	1
3	3	1	1
Total	10	9	3

Number of Languages

Cell	1	2	3 or more
0	1	3	2
1	3	2	1
2	3	0	3
3	3	1	1
Total	10	6	7

Overall GPA

Cell	A	B	C
0	3	9	0
1	0	11	1
2	1	9	1
3	0	9	0
Total	4	38	2

Number of Languages used Professionally

Cell	0	1 or more
0	5	1
1	6	0
2	4	2
3	5	0
Total	20	3

Computer Science GPA

Cell	A	B	C
0	0	3	0
1	1	4	1
2	1	2	1
3	0	3	1

Number of Classes Using Pascal

Cell	2	3	4 or more
0	0	5	1
1	1	4	1
2	1	3	2
3	0	3	2
Total	2	15	6

CS312 Background Data

Class

Cell	Soph.	Junior	Seniors or Grads
0	1	4	5
1	3	3	6
2	2	7	3
3	2	3	6
Total	8	17	20

Number of Languages

Cell	2 or 3	4 or 5	6 or 7
0	2	8	1
1	5	6	1
2	4	7	1
3	3	8	0
Total	14	29	3

Overall GPA

Cell	A	B	C
0	0	10	1
1	2	9	0
2	2	8	1
3	1	9	1
Total	5	36	3

Number of Languages Used Proffesionally

Cell	0	2 to 4
0	10	1
1	10	2
2	8	4
3	9	2
Total	37	9

Computer Science GPA

Cell	A	B	C
0	2	6	1
1	2	7	0
2	3	4	2
3	6	3	1
Total	13	20	4

Number of Classes Using C

Cell	1	2
0	11	0
1	11	1
2	12	0
3	9	2
Total	43	3

CS431 Background Data

Class

Cell	Junior	Senior Or Grad
0	1	7
1	0	9
2	0	10
3	0	11
Total	1	37

Overall GPA

Cell	A	B	C
0	1	7	0
1	1	8	0
2	2	4	4
3	2	5	0
Total	6	24	4

Computer Scienc GPA

Cell	A	B	C
0	2	5	0
1	2	7	0
2	3	5	2
3	2	5	0
Total	9	22	2

Number of Classes Using C

Cell	1-2	3	4-5
0	1	3	3
1	3	4	2
2	3	3	4
3	2	3	3
Total	9	13	12

Number of Languages

Cell	4-5	6-7
0	1	7
1	6	3
2	5	5
3	4	5
Total	16	17

Number of Languages Used Professionally

Cell	0	1-2	3-4
0	4	2	2
1	5	3	1
2	7	3	0
3	6	1	2
Total	5 7	38	

Using C Currently?

Cell	No	Yes
0	1	7
1	3	6
2	2	8
3	3	6
Total	9	27

Apendix B - Materials

- Pascal Materials
  - Pascal Background Questionaire
  - Easy Program
    - Comprehension Quiz - Forward Reasoning Question First
    - Comprehension Quiz - Backward Reasoning Question First
  - Hard Program
    - Comprehension Quiz - Forward Reasoning Question First
    - Comprehension Quiz - Backward Reasoning Question First
- C Materials
  - C Background Questionaire
  - Easy Program
    - Comprehension Quiz - Forward Reasoning Question First
    - Comprehension Quiz - Backward Reasoning Question First
  - Hard Program
    - Comprehension Quiz - Forward Reasoning Question First
    - Comprehension Quiz - Backward Reasoning Question First

You have been selected to participate in an experiment related to programming comprehension. Though your participation is optional, we request you to participate and try to answer all questions to the best of your ability.

This experiment will not effect your course grade in any way. **Thank You.**

Please fill out the following questionaire. Be as accurate as possible.

Class level FR SO JR SR POST-BAC GRAD

Overall GPA _______ (4.0=A) Computer Science GPA _______ (4.0=A)

How many classes, including this one, have you programmed in PASCAL? _______

The following matrix refers to programming experience. Mark all appropriate items with a √.

Language	Used In Class	Used Professionally (e.g. for pay)	Using Currently
PASCAL			
C			
COBOL			
FORTRAN			
ASSEMBLY			
LISP			

Please read the following instructions carefully.

- Please do not write your name on any of the materials.
- If you have questions at any time, please raise your hand.
- Do not refer to any text books, manuals or notes during the experiment.
- There are two sections to the experiment, each taking a maximum of ten minutes.
- If you finish early, please sit quietly and await further instructions.
- The scoring in this experiment is based on accuracy, so be careful.

Do not proceed to the next portion of the experiment until you are told to do so.

```
procedure value(s:string; sLast:integer; var whole, fraction, power : integer);
var
  plus: boolean;
  i, first, last, sign : integer;
  digits : set of char;
  temp : string;
begin
  digits := ['0','1','2','3','4','5','6','7','8','9'];
  i:=1;

  while s[i] in digits do
     i:=i+1;
  last:=i;
  temp:=copy(s,1,last-1);
  whole:=StringToInt(temp);
  if s[i] = '.' then
     begin
     i:=i+1;
     if s[i] in digits then
        begin
        first:=i;
        while s[i] in digits do
           i:=i+1;
        last:=i-1;
        temp:=copy(s,first,last-first+1);
        fraction:=StringToInt(temp);
        end
     else
        fraction:=0;
     end
   else
     fraction:=0;

   if i<=sLast then
     begin
     if (s[i] = 'e') or (s[i] = 'E') then
        begin
        i:=i+1;
        if s[i] = '-' then
           begin
           sign:=-1;
           i:=i+1;
           end
        else
           begin
           if s[i]='+' then
              begin
              sign:=1;
              i:=i+1
              end
           else
              sign:=1;
           end;
        first:=i;
        while s[i] in digits do
           i:=i+1;
        last:=i-1;
        temp:=copy(s,first,last-first+1);
        power:=sign * StringToInt(temp);
      end
      else
        power:=1
      end
    else
      power:=1;
  end;
```

String functions:
StringToInt(s:string) : int; - Returns the integer value of string s. If s contains non-numeric characters then StringToInt returns 0 and prints an error message.

```
Example:     t:=StringToInt('456')    then t=456
             t:=StringToInt(' ')      then t=0
```

copy(s:string; start, length: integer) : string; - Returns a substring of the string s, starting from position start and containing length characters.

```
Example:     t:=copy('1234',2,3) then t:='234'
             t:=copy('abcd',3,2) then t:='cd'
```

The following questions refer to the procedure value on the next page:

1. Given s='7.55' and sLast=4, what would the output of procedure value be?

   whole ___________ fraction ___________ power___________

2. Given s='546.7E78' what would sLast need to be so that power = 1?

   sLast ___________

3. Given s='099e-5' and sLast=6, what would the output of the procedure value be?

   whole ___________ fraction ___________ power___________

4. Given s='1230E+78' and sLast=8, what non-numeric character could you replace the 3 without creating an error message in StringToInt.

   s ___________

5. Given s='7.534E789', sLast=9, what would the output of the procedure value be?

   whole ___________ fraction ___________ power___________

6. What changes to s='987.45 E 7' need to be made in order for power to be = 7?

   s ___________

Do not proceed to the next section until directed.

String functions:

StringToInt(s:string) : int; - Returns the integer value of string s. If s contains non-numeric characters then StringToInt returns 0 and prints an error message.

```
Example:     t:=StringToInt('456')    then t=456
             t:=StringToInt(' ')      then t=0
```

copy(s:string; start, length: integer) : string; - Returns a substring of the string s, starting from position start and containing length characters.

```
Example:     t:=copy('1234',2,3) then t:='234'
             t:=copy('abcd',3,2) then t:='cd'
```

The following questions refer to the procedure value on the next page:

1. Given s='546.7E78' what would sLast need to be so that power = 1?

   sLast ____________

2. Given s='099e-5' and sLast=6, what would the output of the procedure value be?

   whole ____________ fraction ____________ power____________

3. Given s='1230E+78' and sLast=8, what non-numeric character could you replace the 3 without creating an error message in StringToInt.

   s ____________

4. Given s='7.534E789', sLast=9, what would the output of the procedure value be?

   whole ____________ fraction ____________ power____________

5. What changes to s='987.45 E 7' need to be made in order for power to be = 7?

   s ____________

6. Given s='7.55' and sLast=4, what would the output of procedure value be?

   whole ____________ fraction ____________ power____________

Do not proceed to the next section until directed.

```
function syllableCount(word:string;length:integer):integer;
var
  first, second, previous :char;
  scount,i:integer;
  vowels, lstc:set of char;

begin
scount:=0;
vowels:=['a','e','i','o','u','y'];
lstc:=['l','s','t','c'];

for i:=1 to length do
   begin
   first:=word[i];
   second:=word[i+1];
   if i>1 then
      previous:=word[i-1]
   else
      previous:=' ';
   if first in vowels then
      begin
      if second in vowels then
         begin
         if first='a' then
            begin
            if(second<>'e') then
               scount:=scount+1
            else
               if i<> 1 then
                  scount:=scount + 1;
            end
         else
            if first='i' then
               begin
               if second='e' then
                  scount:=scount+1
               else
                  if second='a' then
                     begin
                     if not(previous in lstc)then
                        scount:=scount+1;
                     end
                  else
                     if second='o' then
                        scount:=scount+1;
               end
            else
               scount:=scount+1;
         end
      else
         scount:=scount+1;
      end;
   end;

syllableCount:=scount;

end;
```

The following questions refer to the function syllableCount on the next page.

1. How many syllables will this function calculate for the word briar?

2. Given the word aerie, give a single letter that can replace the first e without changing the syllable count? The result need not be a valid word.

3. How many syllables will this function calculate for the word ciao?

4. Give any combination of four letters that will return a syllable count of 0 The result need not be a valid word.

5. How many syllables will this function calculate for the word solarium?

6. Given the word chick, give a single vowel (a, e, i, o, u) that can be added after the i so that the syllable count remains unchanged at 1? The result need not be a valid word.

The following questions refer to the function syllableCount on the next page.

1. Given the word chick, give a single vowel (a, e, i, o, u) that can be added after the i so that the syllable count remains unchanged at 1? The result need not be a valid word.

2. How many syllables will this function calculate for the word briar?

3. Given the word aerie, give a single letter that can replace the first e without changing the syllable count? The result need not be a valid word.

4. How many syllables will this function calculate for the word ciao?

5. Give any combination of four letters that will return a syllable count of 0 The result need not be a valid word.

6. How many syllables will this function calculate for the word solarium?

You have been selected to participate in an experiment related to programming comprehension. Though your participation is optional, we request you to participate and try to answer all questions to the best of your ability.

This experiment will not effect your course grade in any way. **Thank You.**

Please fill out the following questionaire. Be as accurate as possible.

Class level FR SO JR SR POST-BAC GRAD

Overall GPA _______ (4.0=A) Computer Science GPA _______ (4.0=A)

How many classes, including this one, have you programmed in C? _______

The following matrix refers to programming experience. Mark all appropriate items with a √.

Language	Used In Class	Used Professionally (e.g. for pay)	Using Currently
PASCAL			
C			
COBOL			
FORTRAN			
ASSEMBLY			
LISP			
BASIC			

Please read the following instructions carefully.

- Please do not write your name on any of the materials.
- If you have questions at any time, please raise your hand.
- Do not refer to any text books, manuals or notes during the experiment.
- There are two sections to the experiment, each taking a maximum of ten minutes.
- If you finish early, please sit quietly and await further instructions.
- The scoring in this experiment is based on accuracy, so be careful.

Do not proceed to the next portion of the experiment until you are told to do so.

```
#include <stdio.h>

typedef struct {
                int   top, left, bottom, right;
                } rectangle;


int findOverlap(r1,r2)
rectangle r1,r2;
{
rectangle overlap;
int overlapArea;

if (r1.top > r2.top)
   {
   overlap.top=r1.top;
   if (r1.left > r2.left)
      overlap.left=r1.left;
   else
      overlap.left=r2.left;
   }
else
   {
   overlap.top=r2.top;
   if (r1.left > r2.left)
      overlap.left = r1.left;
   else
      overlap.left = r2.left;
   }
if (r1.bottom > r2.bottom)
   {
   overlap.bottom=r2.bottom;
   if (r1.right > r2.right)
      overlap.right=r2.right;
   else
      overlap.right=r1.right;
   }
else
   {
   overlap.bottom=r1.bottom;
   if (r1.right > r2.right)
      overlap.right = r2.right;
   else
      overlap.right = r1.right;
   }
if ((overlap.top>overlap.bottom) || (overlap.left>overlap.right))
   overlapArea=0;
else
   overlapArea=(overlap.right-overlap.left) * (overlap.bottom-overlap.top);

return(overlapArea);
}
```

The following questions refer to the function findOverlap on the next page.

1. Given:

```
r1.top=20    r1.left=10    r1.bottom=70    r1.right=60
r2.top=20    r2.left=20    r2.bottom=60    r2.right=60
```

What is OverlapArea? ________

2. Given:

```
r1.top=380      r1.left=20    r1.bottom=415      r1.right=60
overlap.top=420               overlap.left=20
overlap.bottom=415            overlap.right=60
```

Give an example of r2.top for a second rectangle that would produce this overlap rectangle.

r2.top ________

3. Given:

```
r1.top=150   r1.left=5     r1.bottom=200   r1.right=60
r2.top=120   r2.left=15    r2.bottom=170   r2.right=70
```

What is OverlapArea? ________

4. Given:

```
r1.top=10    r1.left=220   r1.bottom=90          r1.right=260
r2.top=30    r2.left=210   r2.bottom=110         r2.right=290
```

Give one way can you change r2.right and not change the overlapArea at 2400 ?

r2.right ________

5. Given:

```
r1.top=115   r1.left=225   r1.bottom=190   r1.right=260
r2.top=135   r2.left=265   r2.bottom=210   r2.right=280
```

What is OverlapArea? ________

6. Given:

```
r1.top=15    r1.left=110   r1.bottom=50    r1.right=160
r2.top=5     r2.left=100   r2.bottom=70
```

Give one way to complete r2 so that overlapArea = 1750 ?

r2.right ________

Do not proceed to the next section until directed.

The following questions refer to the function findOverlap on the next page.

1. Given:
   r1.top=380 r1.left=20 r1.bottom=415 r1.right=60
   overlap.top=420 overlap.left=20
   overlap.bottom=415 overlap.right=60

Give an example of r2.top for a second rectangle that would produce this overlap rectangle.

r2.top _________

2. Given:
   r1.top=150 r1.left=5 r1.bottom=200 r1.right=60
   r2.top=120 r2.left=15 r2.bottom=170 r2.right=70

What is OverlapArea? _________

3. Given:
   r1.top=10 r1.left=220 r1.bottom=90 r1.right=260
   r2.top=30 r2.left=210 r2.bottom=110 r2.right=290

Give one way can you change r2.right and not change the overlapArea at 2400 ?

r2.right _________

4. Given:
   r1.top=115 r1.left=225 r1.bottom=190 r1.right=260
   r2.top=135 r2.left=265 r2.bottom=210 r2.right=280

What is OverlapArea? _________

5. Given:
   r1.top=15 r1.left=110 r1.bottom=50 r1.right=160
   r2.top=5 r2.left=100 r2.bottom=70

Give one way to complete r2 so that overlapArea = 1750 ?

r2.right _________

6. Given:
   r1.top=20 r1.left=10 r1.bottom=70 r1.right=60
   r2.top=20 r2.left=20 r2.bottom=60 r2.right=60

What is OverlapArea? _________

Do not proceed to the next section until directed.

```
#include <stdio.h>
#include <ctype.h>

int isVowel(c)
char c;
{
/* isVowel returns 1 (true)  if c is a vowel (a,e,i,o,u) and
           returns 0 (false) if c is not a vowel */
switch (c)
    {
     case 'a': case 'e': case 'i': case 'o': case 'u': return(1);
     default    : return(0);
     }
}


int syllableCount(word,length)
char *word;
int length;
{
char first, second;
int  scount,i;

scount=0;

for (i=0;i<length;i++)
   {
   first=word[i];
   if (i==(length-1))
      second=' ';
   else
      second=word[i+1];
   if (i!=0)
      previous=word[i-1];
   else
      previous=' ';
   if (isVowel(first))
      {
      if (isVowel(second))
         {
         if (first=='a')
            {
            if(second!='e')
               scount=scount+1;
            else
              if (i==1)
                 scount=scount + 1;
            }
         else
            if (first=='i')
               {
               if (second=='e')
                  scount=scount+1;
               else
                  if (second=='a')
                     {
                     if (previous!='c')
                        scount=scount+1;
                     }
                  else
                     if (second == 'o')
                        scount=scount + 1;
               }
            else
               scount=scount+1;
            }
      else
         scount=scount+1;
      }
   }

return(scount);
}
```

The following questions refer to the function syllableCount on the next page.

1. How many syllables will this function calculate for the word briar?

2. Given the word aerie, give a single letter that can replace the first e without changing the syllable count? The result need not be a valid word.

3. How many syllables will this function calculate for the word ciao?

4. Give any combination of four letters that will return a syllable count of 0 The result need not be a valid word.

5. How many syllables will this function calculate for the word solarium?

6. Given the word chick, give a single vowel (a, e, i, o, u) that can be added after the i so that the syllable count remains unchanged at 1? The result need not be a valid word.

The following questions refer to the function syllableCount on the next page.

1. Given the word chick, give a single vowel (a, e, i, o, u) that can be added after the i so that the syllable count remains unchanged at 1? The result need not be a valid word.

2. How many syllables will this function calculate for the word briar?

3. Given the word aerie, give a single letter that can replace the first e without changing the syllable count? The result need not be a valid word.

4. How many syllables will this function calculate for the word ciao?

5. Give any combination of four letters that will return a syllable count of 0 The result need not be a valid word.

6. How many syllables will this function calculate for the word solarium?