CONDITIONAL STATEMENTS AND PROGRAM CODING: AN EXPERIMENTAL EVALUATION[+]

Iris Vessey

Ron Weber*

University of Queensland
Australia

October 1982

Draft Paper Only-Comments Welcome

Center for Research on Information Systems
Computer Applications and Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #45

GBA #82-74 (CR)

+Acknowledgement: We are indebted to Bert Holland of NYU for his assistance with the doubly multivariate statistical procedures and to Kieran Mathieson for his comments on an earlier version of this paper.

*Visiting Professor, Computer Applications and Informations Systems Area, Graduate School of Business Administration, New York University, 1982-83.

CONDITIONAL STATEMENTS AND PROGRAM CODING: AN EXPERIMENTAL EVALUATION

1. Introduction

The conditional statement occupies a central place in the theory and practice of programming. In terms of theory, the form of the conditional has been the focus of much of the debate in the structured programming literature. Should the GOTO statement be prohibited so that the conditional is implemented in an IF-THEN-ELSE form rather than an IF-GOTO form (see [7, 13])? In terms of practice, several studies have shown the high frequency with which conditional statements are used in program code. Elshoff [5], for example, studied 120 commercial PL/I programs and found that conditionals constituted 17.8 per cent of statement types used, ranked second behind assignment statements (see also [12, 19]). Interestingly, only 17 per cent of IF statements used the ELSE clause; the correlation coefficient between IF statements and GOTO statements was .9.

Even with fourth generation nonprocedural languages, conditional statements still appear to be a primary construct. After extensive experience with Focus, Read and Harman [18, p. 109] conclude: "The 4GLs are currently being called 'nonprocedural languages', but this term is too limited because in large, real world systems a liberal sprinkling of procedural code (e.g. IF statements) is essential." Both Focus and Nomad, for example, provide several forms for implementing conditional statements [10, 16]. Indeed, it is a moot point whether nonprocedural languages will ever replace procedural languages (and conditionals). Welty and Stemple [24] found that users performed better when formulating complex queries in procedural code than they did using nonprocedural code. They hypothesized that procedural code allowed the users to think in terms of a concrete model that facilitated their coding the queries. Thus, at least for the forseeable future, the importance of conditional statements seems assured.

In spite of their importance, however, the empirical research undertaken on conditional statements is meagre. Given that most programming languages allow conditionals to be implemented in several ways, one would expect that programmer performance using the different forms would be a significant concern. Of the few studies undertaken, perhaps the research conducted by Sime, Green, and Guest [9, 22, 23] is the pioneering and seminal work. Specifically, they investigated programmer performance when conditionals were implemented as a nested structure (IF-THEN-ELSE) instead of a branch-to-label structure (IF-GOTO). Across several experiments they found evidence in favor of the nested structure.

For some researchers, the form that the conditional should take now seems somewhat a closed matter. Shneiderman [21, p. 81] concludes: ". . . controlled experiments and a variety of informal field studies indicate that the choice of control structures does make a significant difference in programmer performance. Evidence supports the anecdote that the number of bugs in a program is proportional to the square of the number of GOTOs, . . ." Others, however, are sceptical. Sheil [20, p.107] concludes: "Unless the (uncited) 'informal studies' which are alluded to (by Shneiderman) are very compelling, the evidence suggests only that deliberately chaotic control structure degrades performance."

While intuitively we believe in the superiority of the IF-THEN-ELSE form of the conditional over the IF-GOTO form, like Sheil we do not believe that the matter is settled. Upon close examination of the research conducted by Sime, Green, and Guest, we argue there are some important confoundings that make their results equivocal. We do not mean these comments to be a criticism of Sime, Green, and Guest since their studies were exploratory. Nevertheless, in this research we attempt to control for these confoundings and examine once again the nested conditional versus the branch-to-label conditional.

The remainder of the paper proceeds as follows. Section 2 reviews the prior research conducted by Sime, Green, and Guest and summarizes their findings. Section 3 identifies further research questions that arise from their work, thereby providing the rationale for the current research. Section 4 describes the languages investigated in the current research. Section 5 presents the theory and hypotheses underlying our research. Section 6 describes the research methodology used. Sections 7 and 8 present the data analysis and discuss the results obtained. Section 9 identifies some limitations of the research. Finally, section 10 presents our conclusions.

2. Prior Research

In their first attempt to investigate the effects of different implementations of the conditional contruct, Sime, Green, and Guest [22] studied two languages: JUMP, a language incorporating a branch-to-label conditional; and NEST, a language incorporating a nested conditional (see Tables 3 and 4). To enable them to study a limited but controlled set of effects, both were artificial languages with severely restricted syntax and semantics. Subjects, who were inexperienced programmers, first learned one language and wrote solutions for five problems; then, six weeks later, they learned the other language and wrote solutions for the same problems. The subjects worked interactively. When they were satisfied their program was complete, the system checked their code for syntactic errors. When these errors were corrected, the program was executed. Semantic errors were identified in a novel way. The problems involved constructing programs to follow a cooking recipe to prepare food for a "mechanical hare." The hungry hare gave an auditory signal if its food was not prepared correctly.

Table 1 summarizes the reported experimental results. A significantly greater number of subjects in the JUMP group failed to complete the experimental tasks in the allotted time. Furthermore, the JUMP group made significantly more semantic errors and took significantly more time to complete a problem than the NEST group. In terms of the transfer condition, subjects who first learned NEST and then JUMP had a significantly greater number of problems that took longer to complete on the second trial than the group that first learned JUMP and then NEST. Both groups showed improvement, but it seems as if it is more difficult to program in JUMP after learning NEST whereas it is easier to program in NEST after learning JUMP.

[Insert Table 1 about here]

In their second experiment, Sime, Green, and Guest [23] investigated two more micro-languages that incorporated a nested conditional: NEST-BE, which used the words "begin" and "end" to designate the scope of a conditional; and NEST-INE, which designated the scope of a conditional by repeating the predicate with a "NOT" term (see Table 3). The syntax of these languages resolved ambiguity that arose with the NEST language when multiple actions were associated with a condition.

Three groups of subjects having no programming experience learned the JUMP, NEST-BE, and NEST-INE languages. Again, they undertook five problems to feed the hungry hare. In this experiment, however, a transfer condition was not investigated.

Table 1 summarizes the reported results. The difference between the proportions of each group successfully completing all problems in the allotted time was not statistically significant, although JUMP outperformed NEST-INE, which in turn outperformed NEST-BE. Significantly more <u>semantic</u> errors per problem were made by the JUMP group, and NEST-INE outperformed NEST-BE. Significantly more <u>syntactic</u> errors were made by the NEST-BE group, and NEST-INE outperformed JUMP. When both syntactic and semantic errors were considered together, the NEST-BE group solved significantly fewer problems on the first attempt, and JUMP outperformed NEST-INE. In terms of error lifetimes, however, NEST-INE programs could be debugged significantly faster, and JUMP outperformed NEST-BE.

Two important conclusions derive from these results. First, the superiority of the nested conditional and the branch-to-label conditional is equivocal. Açcording to some performance criteria, the JUMP language outperforms the NEST-BE language. It seems, therefore, that the choice of language used to implement the nested conditional is an important issue. Second, the results provided Sime, Green, and Guest with important insights as to why at least some implementations of the nested conditional may be superior to the branch-to-label conditional. They identified two fundamental programming tasks: taxonomizing -- identifying the conditions under which certain actions must be performed; and tracing -- describing the order in which instructions are executed. Both are important activities during the program design, coding, and debugging tasks. They argue that the two nested languages, with their automatic indenting, redundantly code (spatially) much sequence information, thereby facilitating the tracing task. The JUMP language, they argue, cannot be indented successfully, and so a task requiring clear sequence infomation, such as drafting a program, will be easier in the nested languages than in JUMP. In terms of the taxonomizing task, NEST-INE facilitates identifying what conditions produce an action through redundantly coding the predicates. JUMP and NEST-BE, however, require more extensive search of the sequence information to determine, for example, which NOT condition is associated with a particular action.

These results motivated Green [9], in two carefully controlled experiments, to examine whether experienced programmers indeed achieved different levels of performance, depending on the language used, across tracing and taxonomizing tasks. In the first experiment he presented subjects with a program written in either JUMP, NEST-BE, or NEST-INE, gave them a set of truth-values for the program predicates, and asked them to indicate the action that would be taken as a result of these truth-values. He hypothesized that the time taken to identify the action using JUMP would be slightly longer than the time taken using NEST-BE or NEST-INE because subjects had to identify labels in the JUMP program; however, there should be no differences between the nested forms. For the tracing task, the results confirmed his expectations: JUMP took slightly longer (statistically significant difference) than the nested forms; and although NEST-INE took slightly longer than NEST-BE, the difference was not statistically significant (see Table 1).

In the second experiment, Green again presented subjects with a program written in either the JUMP, NEST-BE, or NEST-INE languages. This time, however, he gave subjects the action to be taken and asked them to provide the truth-values for the predicates that would invoke that action. Since this was a taxonomizing task, he hypothesized that subjects would take longest with JUMP, and NEST-BE would take longer than NEST-INE. Again, the results confirmed his expectations: JUMP took longer than NEST-BE, and NEST-BE took longer than NEST-INE. The differences were statistically significant (see Table 1).

3. Rationale for the Current Research

While intuitively we agree with the results obtained by Sime, Green, and Guest, there are two areas where we take exception. First, the experimental task that Sime, Green, and Guest [22, 23] gave their subjects comprised three programming tasks: design, coding, and debugging. The subjects were naive or inexperienced programmers. It seems they were not given formal training in how to undertake each of the three tasks; thus, their results may reflect somewhat idiosyncratic approaches to the tasks. Perhaps with formal training in each task, the differences among the languages may dissipate. We develop this issue in the first section below. Second, while the nested languages were indented, JUMP was not indented. Green [9, p.105] argues that "GOTO conditionals cannot, in general, be indented." We disagree. In the second section below, we argue that Sime, Green, and Guest have adopted a language syntax that is too restrictive. Consequently, coding their language in indented form indeed is difficult, but indenting can be accomplished easily if the syntax is relaxed slightly. This modified language is a closer approximation of real-world programming languages and allows a "fairer" comparison with the nested languages.

3.1 Programming Task Confounding

In the two experiments conducted by Sime, Green, and Guest [22,23], subjects learned the micro-languages primarily from written instructions. They do not appear to have been trained in a language-free context with respect to good design, coding, and debugging practices. Like Floyd [6] we argue there are "paradigms" in programming -- good practices that exist independently of programming languages. Languages should be evaluated in terms of how well they allow these paradigms to be implemented as code. Perhaps for the simple experimental tasks used by Sime, Green, and Guest, the inexperienced subjects intuitively identified the good design, coding, and debugging practices that should have been used to solve the problems assigned. We suspect that this is not the case, however, at least for the JUMP language, if not for the nested languages.

Consider, first, the program design task. For the simplistic experimental problems given to subjects, there are two primary activities. First, the conditions under which an action is performed must be determined. This is the taxonomizing task identified by Sime, Green, and Guest. Furthermore, to check the accuracy and completeness of the taxonomy developed, tracing might also be performed. Second, the order in which conditional tests will be carried out in the program must be determined. When Sime, Green, and Guest presented their instructions to subjects, the taxonomizing task had already been completed. For example, a typical instruction was: "Boil: All things which are juicy, not hard and not tall." Subjects still had to perform the second task, however. As we show below, there is a formal way to determine the sequence in which tests should be performed. If subjects are not shown this method, part of the effects obtained by Sime, Green, and Guest may reflect that, for naive subjects, the languages they evaluated facilitated differentially choosing the sequence of tasks to be performed. Had the subjects been shown how to choose the sequence of tasks, part of the effects obtained may have disappeared.

Consider, next, the coding task. Here we identify still another type of activity to be performed; namely, the sequencing activity whereby a program design must be converted into the linear (physically contiguous) sequence of program code. It is similar to tracing in that a programmer proceeds from conditions to actions. Nevertheless, it differs insofar as tracing follows instructions that are logically contiguous whereas sequencing determines the physical order of instructions.

Given a formal program design, sequencing is a straightforward activity. Consider Table 2, which shows a decision table representation of Problem 5 in Sime, Green, and Guest [23]. The decision table constitutes a formal representation of the program design. Each rule shows the necessary taxon information. Moreover, providing the decision table has been sorted according to the usual rules, it prescribes the order in which conditions should be tested -- namely, the order of the rows in the condition stub (see, e.g., [3]). Sequencing then proceeds by progressively partitioning the table into "yes" and "no" branches, representing these branches via a decision tree, and coding the tree in a left-to-right fashion (preorder traversal) . Table 2 shows how a decision tree has been used to partition the decision table. Table 3 shows our coding of Table 2 in the NEST, NEST-BE, and NEST-INE languages. Note that the sequence of code in each solution reflects a preorder trasversal of the decision tree drawn to partition the decision table. Note, also, that our solutions correspond to those given by Sime, Green, and Guest [23].

[Insert Tables 2 and 3 about here]

The decision table, decision tree, and preorder traversal techniques are language-independent formal program design and coding techniques (paradigms). Once a programmer knows these techniques, sequencing is a somewhat trivial task, providing that the language syntax used to implement the design permits preorder traversal of the tree. It is here that the deficiencies in the JUMP language proposed by Sime, Green, and Guest become apparent.

Consider Table 4, which shows the JUMP solution for the decision table shown in Table 2. The correspondence between the table (tree) and JUMP solution is not clear-cut. Why? The syntax of JUMP does not permit a preorder traversal of the tree. This can be rectified easily, however, by a slight modification to the syntax. If the language allows an unconditional GOTO, Table 4 shows the solution. Note that the unconditional GOTO has not been used in an unconstrained fashion. It has been used to reflect the NOT branch of the decision tree. Furthermore, the order of labels is determined easily by labeling according to a preorder transversal of the tree (see Table 2). Again, providing the language accommodates the NOT branch of the tree, the program design and sequencing tasks are language independent. Insofar as the widely-used programming languages that provide a branch-to-label conditional also permit an unconditional transfer of control, Sime, Green, and Guest's comparison of JUMP with the nested languages is somewhat "unfair". JUMP programmers are forced to undertake a more complex sequencing task, and the experimental results do not bear on the relative strengths and weaknesses of nested versus branch-to-label implementations of conditionals in real-world programming languages.

[Insert Table 4 about here]

As a consequence of the above arguments, we also question the results obtained by Sime, Green, and Guest [23] with respect to the debugging capabilities of the languages. Recall that Sime, Green, and Guest [23] found errors in NEST-INE easiest to correct. Furthermore, Green [9] found the tracing and taxonomizing tasks (fundamental to debugging) to be easiest in the nested languages. There are two problems with these results. First, in terms of Sime, Green, and Guest, if for naive programmers the languages facilitated differentially the program design task, the errors existing at the end of coding may not have been equally serious. Thus, subjects may have undertaken debugging starting from different bases. In addition, to the extent that the program designs were not equally clear to the subjects, the debugging results are confounded. Second, in terms of Green's research, the JUMP results may reflect the absence of a clear NOT condition expressed via an unconditional transfer of control. We do not question the internal validity of Green's results, only the external validity in terms of languages that provide branch-to-label conditionals.

In the current research we attempt to disentangle various effects. Since program design can be carried out independently of the language used, this factor is controlled. Furthermore, we believe Green's research provides compelling evidence that nested languages, especially NEST-INE, will outperform JUMP for debugging tasks. Whether nested languages will outperform JUMP for the sequencing task, however, is less clear-cut. We see this as a strategic hypothesis to test. Does use of structured programming control constructs facilitate only coding, only debugging, or both activities?

3.2 Indentation

Sime, Green, and Guest [23] argue that indentation is beneficial because it redundantly codes information that facilitates the tracing task. Where there is no test, the next action is on the next line with no change in indentation. Where there is a test, the action satisfying the condition is on the next line, which is indented more, and the NOT condition is the next action on the same level of indenting as the test. Green [9] argues that indentation also facilitates the taxonomizing task in that relevant or irrelevant sets of conditions can be identified quickly, simply by associating levels of indentation. Clearly, therefore, Sime, Green, and Guest regard the ability to indent a language as being an important factor affecting a programmer's ability to carry out the tracing or taxonomizing tasks.

In terms of the experiments conducted by Sime, Green, and Guest, only the nested languages have been indented. Indeed, Sime, Green, and Guest [23, p.115] argue: "JUMP has no indentation, and in fact it cannot be successfullly indented unless the language is _considerably restricted_" (our emphasis). Conversely, we argue that a slight _relaxation_ in the syntax of JUMP allows it to be indented at least somewhat meaningfully. Again, the relaxation involves using an unconditional GOTO to represent the NOT branch in the decision tree -- the same adaptation proposed in the previous section. Table 5 shows the indented form of the modified JUMP syntax applied to the decision table shown in Table 2.

[Insert Table 5 about here]

Whether or not the advantages of the nested languages over JUMP with respect to the tracing and taxonomizing tasks would disappear, given the indented modified form of JUMP, is a research issue. We suspect they would decrease but not disappear. Even with the indented modified form of JUMP, it is still impossible to choose a set of predicate truth-values that enables a programmer to trace through the program in a sequence comprising physically contiguous instructions. A physical "jump" is unavoidable. Thus, we suspect the indented JUMP is still more difficult to understand. Nevertheless, as we show later, further modifications to the JUMP syntax enable even more meaningful indentation to be effected. Of course the research problem is to compare the "best" JUMP syntax with the "best" nested syntax.

In the current research we do not evaluate the effects of the indented form of JUMP on performance in the tracing and taxonomizing tasks. As discussed in the previous section, the primary focus of the research is the sequencing task. Nevertheless, since Sime, Green, and Guest have argued that indentation is a major factor affecting programmer performance, we attempt to control this factor by evaluating both the indented and unindented forms of the languages.

4. Languages Investigated

In light of our arguments in the previous section, in this research we did not investigate the JUMP language proposed by Sime, Green, and Guest [22]. Instead, we examined a modified version of JUMP, JUMP-M, which permitted an unconditional GOTO so the NOT condition in a decision tree could be implemented easily. The three nested languages, NEST, NEST-BE, and NEST-INE, were essentially the same, although there were some minor modifications made that we will explain below.

Table 7 shows the indented language implementations (of the decision table shown in Table 6) that were investigated in this research. To distinguish the indented forms from the unindented forms, the languages have been labeled JUMP-M-I, NEST-I, NEST-BE-I, and NEST-INE-I. The unindented forms of the languages are the same, except that each new line starts flush with a common left-hand margin.

[Insert Tables 6 and 7 about here]

Consider, first, JUMP-M. It differs in four respects from the original JUMP. First, note how an unconditional GOTO has been used for each NOT branch in the decision tree; for example, GOTO 14 for the NOT branch of "IF juicy." Second, the unconditional GOTO has been used, also, to branch to the end of the logic, which may be the end of the program or the start of a new set of logic. Again, we argue this modification makes the language more representative of existing JUMP-style languages. Third, the labels used are numeric rather than alphanumeric (such as L14), consistent with languages such as BASIC and FORTRAN. Fourth, unlike JUMP, JUMP-M can be indented easily. Note how a nested structure now exists in the language, although it still does not seem as clear-cut as the structure existing in the other three languages. Nevertheless, by using an unconditional GOTO to branch to the end of logic (e.g. GOTO 17), this indented form of JUMP appears to provide a neater (and more realistic) nested structure than the first indented version of JUMP shown in Table 5.

Consider, next, the nested languages. NEST-BE has been modified by deleting "THEN" from "IF-THEN" in the original syntax. NEST-INE has been modified by inserting "THEN" after an IF-test in the original syntax. There are two reasons for these modifications. First, a performance measure used in the current research is time taken to perform the sequencing task. Presumably time taken will be affected by the number of words that have to be written in the program solution. With the above modifications, the NEST-BE and NEST-INE solutions for Table 6 each contain 67 words; thus, timing differences between these two languages can be attributed to other factors. Second, given that an action or another test is signaled by a "BEGIN" in NEST-BE, including a "THEN" is redundant. In the original syntax of NEST-INE, however, there is no key-word (such as THEN) to signal the forthcoming action or further test.

Hopefully the above modifications to the nested languages do not affect too much the extent to which our results can be generalized to the syntax used by Sime, Green, and Guest. We believe the modifications to be minor. The modifications to the JUMP syntax, however, are more serious. Clearly it is an impossible research task to evaluate the performance consequences of all possible syntactic variations that might be made to a single programming language. If researchers are to evaluate different languages, they will save themselves substantial work if they can approach the "optimum" syntax for a particular language as soon as possible.

Finally, there is one aspect of the sequencing task that has not been mentioned so far but which Table 6 illustrates; namely, the procedure to be followed when an action is common to several rules in the decision table. The procedure is simple. As Table 6 shows, the programmer circles the common action and the common set of condition entries that invoke the action. As soon as the condition is tested and the relevant entries hold, the action(s) is written in the sequence of code (see Table 7). Thus, the action is taken irrespective of the entries for all lower level tests within the branch of the decision tree.

5. Theory and Hypotheses

This section describes the theory that underlies our research and the hypotheses derived from the theory that we test empirically. The theory is in a primitive state; it represents a first pass, although it is based on the earlier work of Sime, Green, and Guest. We attempt to predict a priori how the four languages, JUMP-M, NEST, NEST-BE, and NEST-INE, will affect programmer performance during the sequencing (and hence the coding) task. We attempt also to predict the effects of using the indented versus unindented forms of the four languages. The scope of the theory is restricted. We assume programmers start with a common program design and they know how to perform the sequencing task. The theory covers performance up to the point where the first version of the code has been written. Performance is assessed in terms of five criteria: the time taken to write the first version of the code; the number of different types of syntax errors made; the total number of syntax errors made; the number of semantic errors made; and the number of error-free programs per subject.

5.1 Time Taken to Perform the Sequencing Task

In this research the time taken to perform the sequencing task is the elapsed time between subjects being presented with a decision table and their completing a coded solution for the decision table. It comprises the time to construct the decision tree and to convert the tree to code. For reasons that will be explained later, subjects did not check their solution; thus, the time taken ends after the first pass at writing a coded solution.

5.1.1 Unindented Language Form

We postulate that three factors primarily affect the time taken to perform the sequencing task. The first factor is the number of words that must be written for each solution. Given the syntax of the four languages, JUMP always exceeds NEST-BE and NEST-INE, which have equal word counts, and these two languages always exceed NEST.

The second factor is whether or not the language syntax permits users to proceed straight down a branch or whether it forces them to jump to an alternate branch before proceeding. To illustrate this point, consider the JUMP-M solution in Table 7. The first instruction tests the truth-value of "juicy". If "true" exists, the left branch of the decision tree is taken. The next instruction, however, is not for the left branch. It is for the right branch -- the "false" truth-value. Thus, the programmer must flip branches before proceeding down the left branch. We postulate this mental operation slows the sequencing task. None of the nested languages suffer this problem.

The third factor is the cognitive complexity of the grammatical structural unit used to represent the "true" and "false" branches of the decision tree. Here we postulate NEST-BE and NEST-INE to be more complex than JUMP-M and NEST. The complexity arises because in both languages each branch of the tree must be terminated with an END. As the number of levels of nesting increases, remembering the END is needed to terminate each branch requires extra mental operations and slows the sequencing task. For example, in the NEST-BE solution to Table 6, there is a tendency to forget the second END needed to terminate the leftmost branch of the decision tree after the "chop" action is taken and terminated by END. NEST-INE suffers a similar problem, although it is not as serious since the redundant predicate associated with each END acts as a memory cue. In JUMP-M and NEST, the programmer does not have to purposefully terminate the branch. In JUMP-M the branch terminates when a new IF or GOTO is encountered; in NEST it terminates when a new IF or OTHERWISE is encountered.

We postulate that the complexity of the language structural unit is the primary factor affecting time taken in the sequencing task since it imposes the most onerous mental operations. Thus, in terms of time taken, NEST-BE will exceed NEST-INE which, in turn, will exceed JUMP-M and NEST. Furthermore, JUMP-M will exceed NEST because it requires more words to be coded, but more importantly because it does not allow coding to occur by proceeding straight down a branch. This analysis leads to the following hypothesis:

H1: $\mu_{BT} > \mu_{IT} > \mu_{JT} > \mu_{NT}$

where: μ_{BT} = average time taken for NEST-BE

μ_{IT} = average time taken for NEST-INE

μ_{JT} = average time taken for JUMP-M

μ_{NT} = average time taken for NEST

5.1.2 <u>Indented Language Form</u>

Indenting is an extra operation that must be performed in the sequencing task. When is indentation useful? We postulate that indentation is useful when the spatial pattern provided by indentation signals to the programmer that some sequencing operation is needed which otherwise would be obscure. Recall that in the previous section we argued the need for an END was sometimes an obscure aspect of coding in NEST-BE and NEST-INE. Thus, we predict that indentation will facilitate the sequencing task in these two languages. For JUMP-M and NEST, however, we predict that indentation will inhibit rather than facilitate the sequencing task. The problems associated with using the END termination do not exist in these languages; thus, use of indentation constitutes an overhead. This analysis leads to the following hypotheses:

H2 (a): $\mu_{BT} > \tilde{\mu}_{BT}$

H2 (b): $\mu_{IT} > \tilde{\mu}_{IT}$

H2 (c): $\tilde{\mu}_{JT} > \mu_{JT}$

H2 (d): $\tilde{\mu}_{NT} > \mu_{NT}$

where the terms have the usual meaning and the tilde signifies the mean for the <u>indented</u> language form.

5.2 <u>Number of Types of Syntax Errors</u>

The number of types of syntax errors is measured simply by a count of the different forms of syntax error that exist in the code produced. Two factors affect this variable: (a) the number of possible error types that can be made, given the syntax of the language; and (b) the probability of each type of syntax error being made. It is important to recognize that this measure is an expected value. One language may have more syntactic rules that can be broken than another language, but if the probability of each rule in the former language being violated is small, fewer types of syntax error may be made using it compared with the latter language.

We do not attempt a rigorous analysis of why certain types of syntax errors are more prevalent than others. This would require an exhaustive listing of all types of syntax error that could be made with a language and development of a psychological theory that would predict the error proneness of each syntactic rule. While linguistics research has made substantial progress in this area in terms of natural language, developments in terms of artificial languages have been slow in forthcoming. Moreover, there are few empirical studies of programming languages that provide the data to test a theory or the basis upon which a theory might be constructed (see, e.g., [12,14]). Thus, the analysis below is founded upon our intuition and experience.

5.2.1 Unindented Language Form

We postulate that the number of types of syntactic error made when using a language is primarily a function of the complexity of the syntax of the basic structural unit in the language. NEST-BE has the most complex structure:

```
IF __
BEGIN __ END
ELSE
BEGIN __ END
```

Compared with the other languages, there is more scope (degrees of freedom) for forgetting a BEGIN, END, or ELSE, particularly when several levels of nesting exist or the NOT branch of a decision tree is being coded.

NEST has the simplest structure:

```
IF __ THEN
OTHERWISE __
```

Indeed, no syntactic aspects of this structure seem especially error-prone.

For both JUMP-M and NEST-INE, only one type of syntax error seems likely. The structure of JUMP-M is:

```
        IF __ GOTO __
        GOTO __
    __ (true branch)
    __ (false branch)
```

We predict that the most likely error will be forgetting a label on the true or false branch statement.

The structure of NEST-INE is:

```
        IF __ THEN __
        NOT __
        END __
```

We predict that the most likely error will be forgetting an END, again, especially when several levels of nesting exist or the NOT branch in a decision tree is being coded.

This analysis leads to the following hypothesis:

H3: $\mu_{BS} > \mu_{IS} = \mu_{JS} > \mu_{NS}$

where: μ_{BS} = average number of types of syntax error for NEST-BE

μ_{IS} = average number of types of syntax error for NEST-INE

μ_{JS} = average number of types of syntax error for JUMP-M

μ_{NS} = average number of types of syntax error for NEST

5.2.2 Indented Language Form

Indentation increases the scope for a syntax error in that indentation might be forgotten, the wrong levels of indentation may be used or matched, etc. We postulate that indentation will reduce the number of types of syntax error made, only if it counteracts the tendency to make an error type in the unindented form of the language.

Given that we have argued few aspects, if any, of the NEST syntax are error-prone, we predict that indentation will increase the number of error types made. On the other hand we predict that indentation will reduce the number of types of syntax error made in NEST-BE and NEST-INE, in that there is less likelihood of a BEGIN, ELSE, or END being forgotten. The redundant structure provided by indentation acts as a cue to remind the programmer of the key-words needed.

Predicting the effects of indentation on JUMP-M is more difficult. We have argued previously that forgetting a line label is the most likely type of error. Will indentation help the programmer to remember a label is needed? We do not see how. Indeed, indentation seems problematic in any language where a straightforward linear flow of logic cannot be established via each sequential instruction. To illustrate this point, consider the JUMP-M-I syntax in Table 7. The first level of nesting, which starts at line label 1, applies not to instruction 2 but to instruction 1. Similarly, the level of nesting that starts at line label 14 applies to instruction 2. In other words, a level of nesting does not immediately follow the motivating instruction; other instructions intrude in the meantime. We predict that this interference makes correct indentation somewhat difficult in JUMP-M-I.

This analysis leads to the following hypotheses:

H4 (a): $\mu_{BS} > \tilde{\mu}_{BS}$

H4 (b): $\mu_{IS} > \tilde{\mu}_{IS}$

H4 (c): $\tilde{\mu}_{JS} > \mu_{JS}$

H4 (d): $\tilde{\mu}_{NS} > \mu_{NS}$

where the terms and the tilde have the usual meaning.

5.3 Number of Syntax Errors

The number of syntax errors is measured by counting the total number of syntax errors made in the code produced, irrespective of error type. Presumably this measure is correlated with the number of types of syntax error made. Nevertheless, the number of _types_ of syntax error made may be small in a language but a particular type may be especially error prone so the _number_ of syntax errors made is high.

Again, in the absence of a refined psychological theory of programming languages, our analysis below is in a primitive state. It primarily reflects our intuition and experience.

5.3.1. Unindented Language Form

Given the simple structure of the four micro-languages investigated, our predictions with respect to the _number_ of syntax errors made are the same as our predictions for the number of _types_ of syntax errors made. Again, we postulate that NEST-BE has the most complex structure, and, as a consequence, the greatest number of syntax errors will be made with this language. The simple structure of NEST leads to the prediction that few syntax errors will be made. NEST-INE and JUMP-M are the intermediate cases; we predict some errors will be made in terms of a missing END or a missing label respectively.

This analysis leads to the following hypothesis:

H5: $\mu_{BN} > \mu_{IN} = \mu_{JN} > \mu_{NN}$

where: μ_{BN} = average number of syntax errors for NEST-BE

μ_{IN} = average number of syntax errors for NEST-INE

μ_{JN} = average number of syntax errors for JUMP-M

μ_{NN} = average number of syntax errors for NEST

5.3.2. Indented Language Form

Similarly, our predictions for the indented form of the language are the same as those for the number of types of syntax error measure: indentation will facilitate use of NEST-BE and NEST-INE but inhibit use of JUMP-M and NEST. Hence, we propose:

H6(a) : $\mu_{BN} > \tilde{\mu}_{BN}$

H6(b) : $\mu_{IN} > \tilde{\mu}_{IN}$

H6(c) : $\tilde{\mu}_{JN} > \mu_{JN}$

H6(d) : $\tilde{\mu}_{NN} > \mu_{NN}$

where the terms and the tilde have the usual meaning.

5.4 Number of Semantic Errors

Given that programmers are presented with an accurate and complete program design (decision table), semantic errors are those errors made during the construction of the decision tree or the first pass of the coding task that would cause incorrect logic to be executed.

5.4.1 Unindented Language Form

We postulate that two factors primarily affect the number of semantic errors made using the micro-languages. The first factor is whether or not the language syntax permits users to proceed straight down a branch in the decision tree or whether it forces them to jump to an alternate branch before proceeding. This factor has been discussed already. In essence we argue that "jumping branches" is a cognitively complex process that is both syntactically and semantically error prone.

The second factor is the error proneness of the language with respect to syntax. To the extent that users have to concentrate more to prevent syntax errors, they are more likely to make semantic errors.

With respect to the first factor, JUMP-M is at a disadvantage relative to the nested languages. With respect to the second factor, we have argued already that NEST will outperform NEST-INE and JUMP-M, which will in turn outperform NEST-BE. We postulate that the first factor has the major influence on the number of semantic errors made. This analysis leads to the following hypothesis:

H7 : $\mu_{JM} > \mu_{BM} > \mu_{IM} > \mu_{NM}$

where: μ_{JM} = average number of semantic errors for JUMP-M

μ_{BM} = average number of semantic errors for NEST-BE

μ_{IM} = average number of semantic errors for NEST-INE

μ_{NM} = average number of semantic errors for NEST

5.4.2 Indented Language Form

Since indentation does not affect the first factor that we predict will affect the number of semantic errors, we postulate that any effects will occur through the second factor. Since we have argued previously that indentation facilitates coding in NEST-BE and NEST-INE but inhibits coding in JUMP-M and NEST, this leads to the following hypothesis:

H8(a) : $\tilde{\mu}_{JM} > \mu_{JM}$

H8(b) : $\mu_{BM} > \tilde{\mu}_{BM}$

H8(c) : $\mu_{IM} > \tilde{\mu}_{IM}$

H8(d) : $\tilde{\mu}_{NM} > \mu_{NM}$

where the terms and the tilde have the usual meaning.

5.5 Number of Error-Free Programs

This variable measures the number of times that code is free of syntactic or semantic errors after the first pass of the sequencing task across several programming problems. Clearly it is correlated with the variables that measure the number of syntax errors and the number of semantic errors that are likely to occur. Nevertheless, the relationship is not perfect; the probability of at least one syntactic or semantic error occurring may be somewhat unrelated to the total number of syntactic or semantic errors made.

5.5.1 Unindented Language Form

The analysis underlying the variables that measure the number of syntactic errors and the number of semantic errors is relevant here. Given the rote nature of the sequencing task, we postulate that syntactic errors are more likely to occur than semantic errors. In light of hypotheses 5 and 7, therefore, we propose the following:

H9: $\mu_{NF} > \mu_{IF} > \mu_{JF} > \mu_{BF}$

where: μ_{NF} = average number of error-free programs for NEST

μ_{IF} = average number of error-free programs for NEST-INE

μ_{JF} = average number of error-free programs for JUMP

μ_{BF} = average number of error-free programs for NEST-BE

5.5.2. Indented Language Form

Again, our analysis of the effects of indentation on the number of error-free programs follows from our analysis of the effects of indentation on the number of syntax errors and the number of semantic errors: indentation will facilitate use of NEST-BE and NEST-INE and inhibit use of JUMP-M and NEST. This analysis leads to the following hypotheses:

H10(a) : $\tilde{\mu}_{BF} > \mu_{BF}$

H10(b) : $\tilde{\mu}_{IF} > \mu_{IF}$

H10(c) : $\mu_{NF} > \tilde{\mu}_{NF}$

H10(d) : $\mu_{JF} > \tilde{\mu}_{JF}$

where the terms and the tilde have the usual meaning.

6. Research Methodology

To test the hypotheses described in the previous section, a laboratory experiment was conducted. The experimental design used was a mixed design involving two factors and a repeated measure. The two between-subjects factors were type of language used and indentation. The former was measured at four levels, each level representing one of the four micro-languages. The latter was measured at two levels: indented or unindented. The within-subjects variable was measured at three levels, each level representing one of three consecutive trials at the experimental task.

6.1 Subjects

Subjects in the experiment were volunteers from a first-year undergraduate introductory accounting class. The rules for human experimentation in our university prohibit us from having students act as experimental subjects as a compulsory part of their degree. Thus, researchers have no control over subject selection, and the usual threats to external validity apply (see [3]). Nevertheless, students who participate in experiments are motivated to undertake the experiment, so the limitations of compulsory participation tend not to exist.

There were two requirements for students participating in the experiment: first, they had to be available for three hours continuously; second, they had to have little or no programming experience. With respect to this latter requirement, we decided to run the experiment with inexperienced subjects because we did not want the results to be confounded by a subject's experience with a particular language. Training requirements for the micro-languages are not severely onerous, and so we opted for inexperienced subjects.

Eight time slots initially were made available during which volunteers could undertake the experiment. They were asked to indicate their first three preferences. An attempt was made to balance the groups in terms of numbers. The eight slots proved insufficient; consequently, a further five time slots were made available. An attempt was then made to balance the groups in terms of sex ratio and the ratio of full-time to part-time students. Unfortunately, these attempts to balance the groups proved unsuccessful. Because of the three-hour time requirement, part-time students could participate primarily only on evenings and weekends. Full-time students were loathe to participate at these times. The treatments were allocated randomly to the first eight time slots; then, depending on subject numbers in the respective groups, the remaining five time slots were allocated to the treatments. This resulted in JUMP-M-I being allocated three time slots and NEST-INE-I, NEST-BE, and NEST each being allocated two time slots. On our first attempt to solicit students, we obtained 59 volunteers out of a class size of 600. Two more attempts to obtain further volunteers from the class failed.

6.2 Materials

Three sets of materials were developed for the experiment. The first set comprised training exercises that were used to teach subjects how to write code using either the indented or the unindented form of a particular language. The materials for each exercise consisted of two sheets of paper stapled together. The front page of the first sheet contained a space for the subject's code number and the date of the experiment. The inside page of the first sheet contained a decision table -- the program design. The inside page of the second sheet was blank except for a single vertical line if the subject was to use the unindented form of the language, or seven vertical lines if the subject was to use the indented form. These lines provided the left-hand margins for code. Thus, the arrangement of the materials allowed the subject to view the decision table on the left-hand sheet while writing code on the the right-hand sheet. There were seven training exercises developed.

The second set of materials comprised three exercises for the experiment proper. The exercises were arranged in the same way as the training materials so subjects could view the decision table at the same time as they wrote the code. The decision tables for each exercise contained eight conditions, nine rules, six levels of nesting, and nine actions. Table 6 is one of the exercises used in the experiment proper.

The selection of a decision table to represent the program design is purposeful. If structured methodologies are used to implement a system, the output of the system design phase is either a decision table, decision-tree, or structured English as a representation of the logic to be coded (see, e.g., [4]). We chose decision tables because we believe they provide the clearest represention of the logic to be coded. Furthermore, as discussed earlier, they are amenable to the sequencing task via the technique of creating a decision tree to partition the table. The number of conditions in the decision table also was limited to eight since proponents of the structured methodologies advocate choosing "chunks" of logic that can be represented on a single page -- an acknowledgement of the limitations of short-term memory (see, e.g., [4]).

The third set of materials comprised a short debriefing questionnaire that asked subjects to indicate whether they were full-time or part-time students, whether they had computer experience and, if so, the nature of that experience.

6.3 Administration

The experiment was administered in two stages. The first stage was a pilot study in which each of the eight treatments (4 languages X 2 levels of indenting) was tested. No modifications were required to the experimental materials. However, some practice was needed before the experiment could be administered smoothly. Furthermore, in light of questions asked by the pilot subjects and their results, training was modified to highlight areas that were error-prone or ambiguous.

The second stage was the primary study. It, in turn, consisted of two phases. The first phase was a training session involving seven practice exercises. The exercises progressively increased in difficulty from decision tables involving three conditions and code involving three levels of nesting to decision tables involving eight conditions and code involving five levels of nesting. The training session started with the experimenter showing subjects how to partition the decision table and to construct the decision tree. The subjects were told that time was to be a performance variable and, as they gained facility with the tasks, they should attempt to perform them more quickly. They then were given the first three exercises. Next they were shown how to perform the sequencing task in the particular language that they had been assigned. They then coded the first three exercises on which they had worked already. The fourth exercise involved them undertaking the complete experimental task. To ensure consistency of style, they were told to write rather than print the code. At the start of the sixth exercise, subjects were told they would be timed. Their objective was to be complete coding accuracy in minimum time. To motivate them to comply with this objective, they were informed that their results would be discarded unless complete accuracy was achieved. In addition, they were told that for the final three exercises they were not to check back over their code once it was written. Thus, they should strive to write the code correctly on the first pass.

After the sixth exercise, subjects were given a short rest break. After the break, the second phase of the primary study commenced. Subjects were given a warm-up exercise administered as though it were a final experimental trial. Then they were given the final three trials, which represented the experiment proper. Between each trial, subjects were given a short rest break while their answers were collected.

Several aspects of the methodology need further explanation. First, seven practice exercises were given because our pilot study indicated that subjects needed about four exercises before they had achieved some proficiency. By about the fifth or sixth exercise, learning usually had ceased. Nevertheless, as a check for a learning effect (or fatigue), we conducted three experimental trials. Second, after each training exercise, subjects could ask questions. During the early exercises, they were also given some time to examine their answers and assimilate the knowledge required to perform the task. The experimenter used an overhead projector with colored pens to show the answers to exercises. Third, subjects were seated well apart in the room where the experiment was conducted. They were told to signal their completion of an exercise by raising a finger, and then to sit quietly so they would not disrupt other participants. The experimenter used a seating chart to record times from a stop watch. If more than six subjects participated in an experimental session, a research assistant recorded times for half the group. Fourth, by telling subjects that complete accuracy was needed for their results to be used, an attempt was made to have subjects manifest their performance through time. As Green [9] points out, time and accuracy are problematic performance measures since subjects can trade off one measure against the other. Fifth, subjects were told not to check their results since the extent of checking carried out by a subject would confound the effects of the type of language used on the performance measures. A subject's tolerance for ambiguity or propensity for risk taking, for example, may affect the amount of checking he/she undertakes.

As a final issue, the pilot study and primary study were administered by only one of us. There is evidence that indicates different experimenters can produce different effects and some form of control, such as the one we used, is needed (see [2]). Furthermore, it required some practice before the experiment could be administered proficiently. Since practice opportunities were limited, it seemed important that only one of us should conduct the experiment. The experiment was administered 22 times: 9 pilot administrations (NEST-BE was repeated) and 13 primary administrations.

7. Data Analysis

Data was analyzed in two steps. First, a doubly multivariate analysis of covariance with repeated measures model was fitted to the data for time taken, the number of types of syntax errors made, the number of syntax errors made, and the number of semantic errors made (see [1]). Recall that each subject performed the experimental task three times, and on each occasion they were measured on the above four criteria. Since the variables are correlated, a multivariate analysis is necessary. The covariate used was a subject's tertiary education score (TE). This score is calculated on the basis of their high school performance and is the primary admission criterion for entry to the university and its respective departments. We thought that this variable might be a surrogate measure of programming ability for naive and novice programmers.

The second step in the analysis was to fit an analysis of variance model to the data for the number of error-free programs that a subject prepared. Though this is a separate statistical model, the dependent measure is correlated with the other four dependent measures; thus, the results described below should be treated with caution.

7.1 The Multivariate Model

The multivariate model allows all main effects and interactions to be investigated. There were three main effects -- language (L), indentation (I), and trial (T). Consequently, there were three first-order interactions -- LI, LT, and IT -- and one second-order interaction LIT.

Table 8 shows the means and standard deviations for the performance measures (averaged over the three trials) for each language type-indentation combination. Since all measures were moderately right-skewed, they were transformed to correct for this non-normality. Time taken was transformed using the formula $\log_{10} X$ and the other three variables were transformed using the formula $\log_{10} (X+1)$ (see [11]). Appendix A contains the means and standard deviations for the untransformed variables.

[Insert Table 8 about here]

At the .05 significance level, only three effects were significant using the approximation to the F-test -- L, I, and LI. Thus, there is no evidence of a learning effect or a fatigue effect across the three trials of the experiment or an interaction between the within- and between-subjects factors. Since the LI interaction is significant, the main effects must be treated cautiously and the cell means must be examined individually.

While the overall tests of significance indicate a difference exists between the vectors of means for the performance measures, they do not indicate which of the dependent measures produce the significant effects. To determine which dependent measures produced the significant multivariate effect for the interaction term LI, we used a procedure decribed by Messmer and Homans [15]. Basically the procedure involves a series of step-down tests whereby a single criterion variable is first introduced into a univariate analysis of variance model and the F-statistic calculated. This criterion variable then becomes a covariate in a univariate analysis of covariance model and a second criterion variable is tested for significance. This criterion variable then joins the first as a covariate and a third criterion variable is tested for significance. Progressively all except the last criterion variable enter as covariates in the model.

To use the approach it is necessary for the researcher to have some _a priori_ ordering of importance of the criterion variables. In terms of the experiment conducted, we postulated that any treatment differences would be manifested primarily in terms of the time needed to complete the task. The relative importance, however, of the number of types of syntax error, the number of syntax errors, and the number of semantic errors was not clear-cut.

Table 9 shows four different orders of step-down tests used and the significance levels attained for the LI interaction. Since TE score was not significant as a covariate in the overall MANCOVA model, it was removed from the step-down models to increase the degrees of freedom available for the tests.

Given that the individual tests of significance for each criterion variable are related, the Bonferroni inequality can be used to calculate the individual statement levels of significance that must be attained [17]. Using a .05 family level of significance, the individual statement levels of significance are given by:

$$.05 = 1 - \prod_{i=1}^{4} \alpha_i$$

$$\therefore \quad \alpha_i = .013$$

Using $\alpha = .01$, Table 9 shows that only time taken reaches significance. Two other performance measures approach significance: number of types of syntax errors for the first step-down order and number of syntax errors for the second step-down order. Thus, in terms of the level of significance attained, there is strong support for the LI interaction having an effect on time taken and weak support for an effect on the number of types of syntax error and the number of syntax errors.

Figure 1 shows the nature of the interaction for each of these three dependent measures. In terms of time taken, the interaction arises because indentation facilitates use of NEST-BE and NEST-INE but inhibits use of JUMP-M and NEST. Thus, the direction of the means supports the analysis underlying hypothesis 2. For the unindented form of the language, NEST outperformed JUMP-M, and JUMP-M outperformed NEST-BE and NEST-INE. Except that the means for NEST-BE and NEST-INE are equal, these results support the analysis underlying hypothesis 1.

[Insert Figure 1 about here]

In terms of the number of types of syntax error, the interaction term was significant because indentation facilitated use of NEST-BE but inhibited use of JUMP-M, NEST, and NEST-INE. Except for the NEST-INE result, the direction of the means supports the analysis underlying hypothesis 4. For the unindented form of the languages, from best to worst performance, the order is NEST, NEST-INE, JUMP-M, and NEST-BE. Again, apart from the NEST-INE result, the direction of the means supports the analysis underlying hypothesis 3.

In terms of the number of syntax errors made, indentation facilitates use of NEST-BE only and inhibits use of JUMP-M, NEST, and NEST-INE. Except for the NEST-INE result, the direction of the means supports the analysis underlying hypothesis 6. For the unindented form of the language, from best to worst performance, the order is

NEST, NEST-INE, JUMP-M, and NEST-BE. Again, apart from the NEST-INE result, the direction of the means supports the analysis underlying hypothesis 5.

Since the number of semantic errors was not significant for the LI interaction effect, there is no support for hypotheses 7 and 8. Nevertheless, it is interesting to note that in terms of NEST-BE and NEST-INE, the previous effect of indentation on the two syntax error measures was preserved: indentation facilitated use of NEST-BE but inhibited use of NEST-INE.

In summary, given the correlated nature of the dependent variables and the number of comparisons of means that must be made, the significant interaction term in the multivariate model has arisen primarily because of differences in the time taken for the various treatments. The number of types of syntax error and the number of syntax errors have weaker levels of significance and, as a consequence, may be due to chance. Moreover, it is primarily the effects of indentation on JUMP-M and NEST that have produced the significant interaction effect for time taken.

7.2 The Analysis of Variance Model

Since the TE score was not significant in the multivariate model, it was not included in the analysis of variance model used to determine the effects of language and indentation on the number of programs that a subject coded correctly over the three trials. Table 10 shows the means and standard deviations for the various language-indentation combinations. The main effects were insignificant, but the LI interaction was significant (F = 3.164; df = 3,51; p = .03) indicating a cross-over effect.

[Insert Table 10 about here]

For the unindented language form, from best to worst performance, the order is NEST-INE, NEST, JUMP-M, and NEST-BE. Except that NEST-INE outperformed NEST, the direction of the means supports the analysis underlying hypothesis 9.

When the indented forms of the languages are used, Figure 2 shows a familiar pattern. Indentation has facilitated use of NEST-BE but inhibited use of NEST and NEST-INE. Performance using JUMP-M improves slightly when indentation is used. Thus, there is partial support for the analysis underlying hypothesis 10.

[Insert Figure 2 about here]

Again, these results should be treated with caution. Because the dependent variables are correlated and a large number of pairwise comparisons of means have been undertaken, the stability of these results is suspect.

8. <u>Discussion of Results</u>

What, then, are the implications of the results for the way in which conditional statements should be implemented in programming languages? The following sections draw some conclusions based on our findings.

8.1 Relative Capabilities of the Languages Investigated

Based on their two studies, Sime, Green, and Guest [23 p. 114] concluded: "It is always easier to draft a semantically correct program in a nesting language, whether multiple actions are required or not; but when they are required, the likelihood of a syntactic mistake goes up -- particularly in NEST-BE." Recall, from Table 1, NEST had outperformed JUMP on all criteria, but the results for JUMP, NEST-BE, and NEST-INE were equivocal: more subjects had successfully completed each problem in JUMP, there were more error-free problems per subject in JUMP, and JUMP had outperformed NEST-BE in terms of the number of syntactic errors per problem and error-lifetimes. However, more semantic errors per problem had been made in JUMP. Note, not all the differences were statistically significant.

Our results are somewhat contradictory. Consider, first, the three nested languages. In terms of the indented form of the languages (the form used by Sime, Green, and Guest), NEST-BE outperformed NEST and NEST-INE in terms of all five dependent measures. Except for the number of error-free problems per subject, it also outperformed NEST. We find this to be a surprising result since we have argued NEST-BE is the most complex language. To some extent the results are affected by outliers, but Sime, Green, and Guest seem to have had this problem too -- they had non-normal distributions that led them to use non-parametric statistics. For whatever reason, subjects had more difficulty using NEST and NEST-INE. In terms of syntax errors, for example, subjects using NEST-INE forgot a THEN, inserted an extra THEN after the NOT condition, and missed an

END. In NEST, they forgot a THEN or an OTHERWISE, misplaced an OTHERWISE, and ironically one subject frequently misspelled the predicates, which we counted as a syntax error. In NEST-BE, the anticipated errors -- forgetting a BEGIN, END, ELSE -- were not prevalent.

When the unindented forms of these languages were used, however, the syntax difficulties experienced with NEST and NEST-INE were substantially reduced. Indeed, NEST consistently turned out to be the best language across the first four performance criteria. For the number of error-free problems performance measure, NEST-INE outperformed NEST, which in turn outperformed NEST-BE.

With respect to JUMP-M, in general it appears to perform neither substantially worse nor substantially better than the unindented form of the nested languages, except in terms of the time taken with the indented form of JUMP-M. We discuss this issue below.

There are several possible reasons why our results tend to differ from those obtained by Sime, Green, and Guest. First, in many cases we are comparing simple differences in means rather than evaluating statistically significant differences. Thus, the results may reflect sampling variability. Moreover, in all the research conducted so far, the sample sizes have been small; thus, the power of the statistical tests is low. Second, though we varied the syntax of the nested languages only in minor ways, perhaps this had an effect on the results. Third, and probably most important, our experimental task differed from the task used by Sime, Green, and Guest. Whereas we provided subjects with a common program design and showed them how to perform the coding task, Sime, Green, and Guest had subjects undertake some program design and did not provide them with a coding paradigm. As we have argued earlier, differences among the languages may dissipate if subjects are first shown design and coding paradigms that are independent of the language used.

In summary, for the sequencing task we find little evidence in favor of nested languages over a branch-to-label language. Moreover, in light of our results, we question the stability of the prior research findings that show NEST-BE to be syntactically error-prone and branch-to-label languages to be semantically error-prone. The relative performance of these languages for the tracing and taxonomizing tasks, however, is another issue. Given Green's research, we suspect that the argument for nested languages must be couched in terms of their superiority for debugging and modification tasks and not design and coding tasks. Nevertheless, in light of our criticisms of the JUMP syntax, Green's experiments need to be replicated with JUMP-M.

8.2 Usefulness of Indentation

The data provides at least some support for our analysis that predicts indentation will be beneficial when the syntactic structure used to implement a conditional structure is complex. Indenting facilitated use of NEST-BE, which had the most complex syntactic structure. It seems, however, that the basic syntactic structure of NEST-INE is sufficiently simple that indentation as an overhead inhibits the sequencing task. The results for NEST were in the predicted direction.

Indentation in JUMP-M caused particular problems. Subjects had substantial difficulty determining the appropriate level of indentation for the second GOTO representing the NOT branch of the decision tree (see Table 7). Even with the modified JUMP syntax, it still seems as if branch-to-label languages are not especially amenable to indentation. The difficulties are manifested primarily in the time taken to perform the experimental task.

Earlier we criticized Sime, Green, and Guest for their failure to control indentation as a factor that may affect programmer performance. Our results support this view. Because of their simple structure, we hypothesized that NEST and NEST-INE would be the superior languages. These hypotheses were supported, only for the unindented forms of the languages. Thus, it now seems important when comparing nested languages and branch-to-label languages to specify whether the indented or unindented forms of the languages are being evaluated. Again, these results are confined to the sequencing task, but they motivate a re-examination of the relative performances of the languages for the tracing and taxonomizing tasks.

9. Limitations of the Research

Aside from the primitive nature of the theory that drives our research, there are several methodological limitations that undermine the generality of our results. We discuss these limitations in the sections below under the classification scheme proposed by Cook and Campbell [3].

9.1 Statistical Conclusion Validity

Statistical conclusion validity means that it is reasonable to draw conclusions about covariation between variables based on the statistical model used. For this research the major threat to statistical conclusion validity is the low power of the tests caused by the small sample sizes.

Table 11(a) shows the cell sizes for the different treatments. They are small. Unfortunately this problem is difficult to overcome because time requirements for the experiment are substantial. Consequently, it is difficult to get subjects to participate in the research. Sime, Green, and Guest experienced similar problems. To some extent, use of repeated measures designs helps overcome the effects of small sample sizes.

[Insert Table 11 about here]

9.2 Internal Validity

An experiment is internally valid if covariation between two variables can be attributed to the variable being manipulated. Unfortunately, our inability to carry out randomization procedures undermines the internal validity of the experiment. Thus, the experimental design is not a true design; it is a quasi-design [3].

Table 11 shows descriptive statistics for various characteristics of the subjects in each cell. The cells are imbalanced, especially with respect to the number of part-time students in each cell, and to a lesser extent, with respect to the number of females and the number of subjects having some computer experience. This imbalance potentially confounds the results; for example, it may be that some computer experience facilitates (or inhibits) task performance.

We cannot prove or disprove that a confounding has occurred. In our opinion, however, the effects of these different characteristics on task performance, if any, have been minimal. As a covariate, TE score turned out to be insignificant. When we examined the debriefing questionnaire, computer experience was either exposure to the use of computer reports or a brief introduction to BASIC at high school. To some extent the variables also counteract each other. For example, consider the unindented NEST-INE cell. It comprised only part-time students and the proportion with computer experience was higher than the other cells, thereby potentially biassing task performance positively. But it also had the lowest average TE score and the highest proportion of females. We postulated that TE score is positively related to task performance, and historically, females have outperformed males in terms of verbal ability but performed less well in terms of analytical ability. Again, it is unfortunate that the cells are imbalanced, but we argue that task characteristics rather than individual differences have been the primary determinants of performance.

9.3 External Validity

An experiment has external validity if its results can be generalized to the larger population of individuals in which one is interested and to other environmental conditions. Thus, there is an issue of population validity and an issue of ecological validity.

With respect to population validity, our research has strengths and weaknesses. We are concerned with the population of individuals who potentially will carry out some programming in their lives. Our subjects were volunteers who indicated they were motivated to participate in the experiment because of their developing interest in computers. Thus, they were members of the population in which we were interested. Nevertheless, we were unable to select randomly from the volunteer group, and clearly our volunteer group is not a random sample from the entire target population wanting to learn to program.

With respect to ecological validity, there are several issues. First, the experiment has been conducted with micro-languages. Whether the results would generalize to languages containing a full range of programming capabilities is another question. Second, programming involves using other logical constructs besides the conditional construct. The ways in which these constructs interact may be important. Third, we have investigated performance only for a limited range of tasks -- various cooking problems. Other types of tasks need to be investigated.

10. Conclusions

The primary objective of our research was to gain insight into the relative advantages and disadvantages of two ways of implementing conditional structures in programming languages: via nested implementations or via branch-to-label implementations. The research is relevant to the ongoing debate within the structured programming literature as to the merits of prohibiting the GOTO statement.

We have argued that the prior research carried out by Sime, Green, and Guest has two limitations: first, it has confounded design issues with coding and debugging issues, especially in terms of the way conditionals can be implemented in the syntax of JUMP; second, it has failed to control adequately for the effects of indentation. By modifying the syntax of JUMP to increase its "ecological validity" and to enable indentation to be controlled, and by confining the experimental task to coding, we have attempted to extend their work.

Our results are somewhat in conflict with those obtained by Sime, Green, and Guest. Given inexperienced programmers start with a common program design and given they are trained in how to perform the coding tasks, we find no clear-cut evidence in favor of the nested conditional over the branch-to-label conditional for the sequencing (coding) task, unless the indented or unindented form of each language is considered also. If it is accepted that JUMP-M is a more ecologically valid implementation of conditional constructs using the GOTO statement than JUMP, further research needs to be undertaken in terms of the tracing and taxonomizing tasks. We suspect that the superiority of the nested conditional will be established in these tasks.

Two other issues arise from our research. First, like Sime, Green, and Guest, our results show that the relative advantages and disadvantages of the nested conditional versus the branch-to-label conditional cannot be considered independently of the language in which the constructs are implemented. The syntax of the language appears to be a critical factor affecting the error proneness of the language. Unfortunately we know of no way to investigate the "pure" constructs independently of the languages in which they are implemented.

Finally, our results show the issue of indentation cannot be ignored. This finding runs contrary to the findings of prior research which, as yet, has been unable to establish a significant effect for indentation, in spite of the researcher's prior beliefs (see [21] for a summary of the research). Our results suggest that indentation may be beneficial when redundancy is needed to cope with a complex grammatical structure in a language.

Sime, Green, and Guest [22]

No. subjects who completed experiment	: NEST >> JUMP
No. semantic errors per problem	: JUMP >> NEST
Time taken per problem	: JUMP >> NEST
No. problems per subject taking longer	: JUMP >> NEST

Sime, Green, and Guest [23]

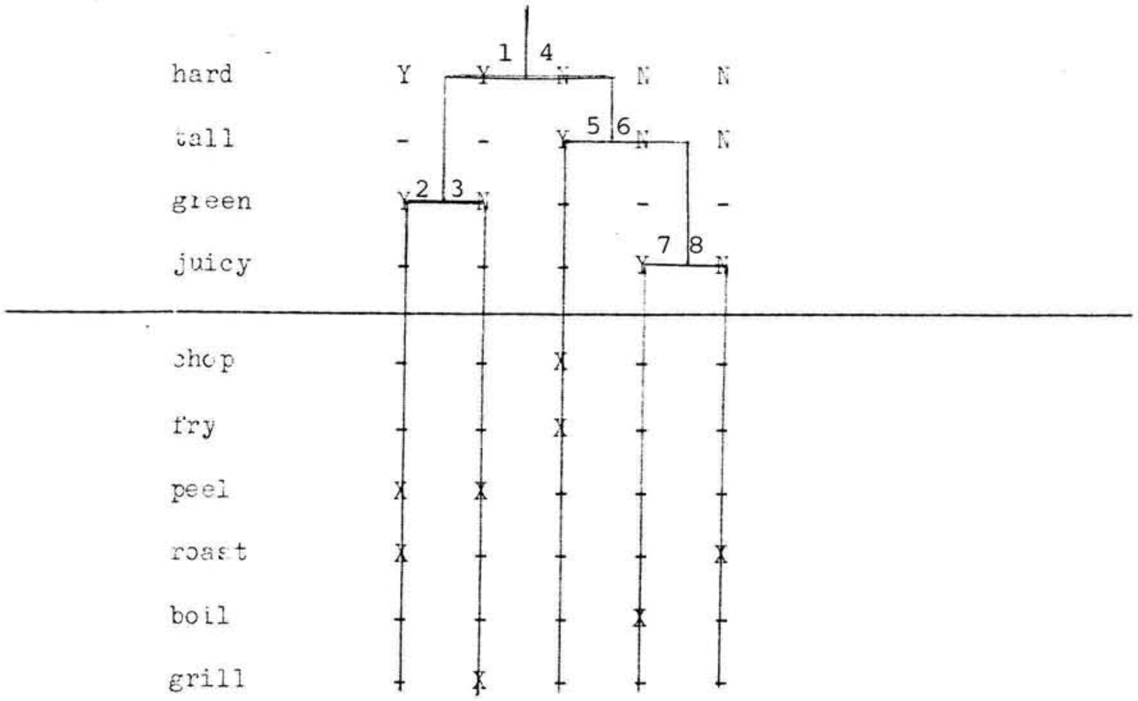
No. subjects successfully completing each problem	: JUMP > NEST-INE >NEST-BE
No. semantic errors per problem	: JUMP >> NEST-BE >NEST-INE
No. syntactic errors per problem	: NEST-BE >> JUMP > NEST-INE
Error-free problems per subject	: JUMP > NEST-INE >> NEST-BE
Error lifetimes	: NEST-INE >>JUMP > NEST-BE

Green [9]

Time for tracing task	: JUMP >> NEST-INE > NEST- BE
Time for taxonomizing task	: JUMP >>NEST-BE >> NEST-INE

Note: >> indicates a statistically significant difference
 > indicates a difference that is not statistically significant

Table 1: Summary of Research Results from Experiments by Sime, Green, and Guest



hard	Y	Y	N	N	N
tall	-	-	Y	N	N
green	Y	N	-	-	-
juicy	-	-	-	Y	N
chop	-	-	X	-	-
fry	-	-	X	-	-
peel	X	X	-	-	-
roast	X	-	-	-	X
boil	-	-	-	X	-
grill	-	X	-	-	-

Table 2: Decision Table Representation of Problem 5 in Sime et al. [23]

```
NEST
----
IF hard THEN peel
   IF green THEN roast
   OTHERWISE grill
OTHERWISE
   IF tall THEN chop fry
   OTHERWISE
      IF juicy THEN boil
      OTHERWISE roast
```

```
NEST-BE
-------
IF hard THEN
BEGIN peel
   IF green THEN
   BEGIN roast
   END
   ELSE
   BEGIN grill
   END
END
ELSE
BEGIN
   IF tall THEN
   BEGIN chop fry
   END
   ELSE
   BEGIN
      IF juicy THEN
      BEGIN boil
      END
      ELSE
      BEGIN roast
      END
   END
END
```

```
NEST-INE
--------
IF hard peel
   IF green roast
   NOT green grill
   END green
NOT hard
   IF tall chop fry
   NOT tall
      IF juicy boil
      NOT juicy roast
      END juicy
   END tall
END hard
```

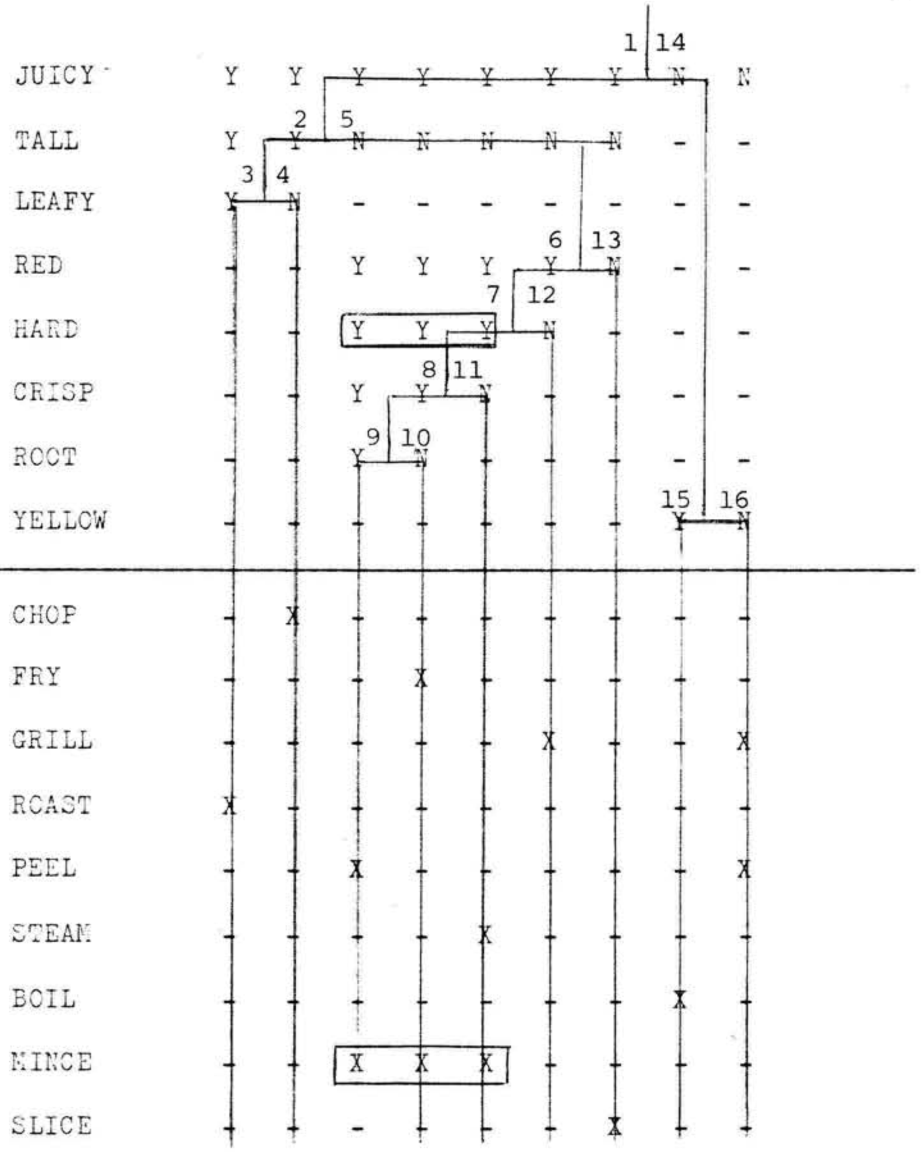
Table 3: Nested Conditional Solutions for Problem 5 in Sime, et al. [23]

```
        JUMP                          MODIFIED JUMP
        ----                          -------------
    IF hard GOTO L1                   IF hard GOTO L1
    IF tall GOTO L2                   GOTO L4
    IF juicy GOTO L3              L1  IF green GOTO L2
    roast stop                        GOTO L3
L1  IF green GOTO L4              L2  peel roast stop
    peel grill stop               L3  peel grill stop
L2  chop fry stop                 L4  IF tall GOTO L5
L3  boil stop                         GOTO L6
L4  peel roast stop               L5  chop fry stop
                                  L6  IF juicy GOTO L7
                                      GOTO L8
                                  L7  boil stop
                                  L8  roast stop
```

Table 4: Branch-to-Label Solutions for Problem 5 in Sime et al. [23]

```
        INDENTED MODIFIED JUMP
        ----------------------
                IF hard GOTO  L1
                GOTO  L4
L1                  IF green GOTO  L2
                    GOTO  L3
L2                  peel roast stop
L3                  peel grill stop
L4                  IF tall GOTO  L5
                    GOTO  L6
L5                  chop fry stop
L6                      IF juicy GOTO  L7
                        GOTO  L8
L7                      boil stop
L8                      roast stop
```

Table 5: Indented Branch-to-Label Solution
for Problem 5 in Sime et al. [23]



JUICY	Y	Y	Y	Y	Y	Y	Y	N	N
TALL	Y	Y	N	N	N	N	N	-	-
LEAFY	Y	N	-	-	-	-	-	-	-
RED	-	-	Y	Y	Y	Y	N	-	-
HARD	-	-	Y	Y	Y	N	-	-	-
CRISP	-	-	Y	Y	N	-	-	-	-
ROOT	-	-	Y	N	-	-	-	-	-
YELLOW	-	-	-	-	-	-	-	Y	N
CHOP	-	X	-	-	-	-	-	-	-
FRY	-	-	-	X	-	-	-	-	-
GRILL	-	-	-	-	-	X	-	-	X
ROAST	X	-	-	-	-	-	-	-	-
PEEL	-	-	X	-	-	-	-	-	X
STEAM	-	-	-	-	X	-	-	-	-
BOIL	-	-	-	-	-	-	-	X	-
MINCE	-	-	X	X	X	-	-	-	-
SLICE	-	-	-	-	-	-	X	-	-

1
14
2
5
3
4
6
13
7
12
8
11
9
10
15
16

Table 6: Decision Table Representation of Experimental Problem

```
          JUMP-M-I
          --------
          IF juicy GOTO 1
          GOTO 14
 1           IF tall GOTO 2
             GOTO 5
 2              IF leafy GOTO 3
                GOTO 4
 3              roast
                GOTO 17
 4              chop
             GOTO 17
 5              IF red GOTO 6
                GOTO 13
 6                 IF hard GOTO 7
                   GOTO 12
 7                 mince
                       IF crisp GOTO 8
                       GOTO 11
 8                        IF root GOTO 9
                          GOTO 10
 9                        peel
                          GOTO 17
10                        fry
                       GOTO 17
11                     steam
                   GOTO 17
12                 grill
                GOTO 17
13              slice
          GOTO 17
14           IF yellow GOTO 15
             GOTO 16
15           boil
             GOTO 17
16           grill peel
17
```

```
   NEST-I
   ------
IF juicy THEN
   IF tall THEN
      IF leafy THEN
      roast
      OTHERWISE
      chop
   OTHERWISE
      IF red THEN
        IF hard THEN
        mince
           IF crisp THEN
              IF root THEN
              peel
              OTHERWISE
              fry
            OTHERWISE
            steam
         OTHERWISE
         grill
      OTHERWISE
      slice
OTHERWISE
    IF yellow THEN
    boil
    OTHERWISE
    grill peel
```

Table 7: Language Solutions to Decision
Table Shown in Table 6

```
NEST-BE-I
---------
IF juicy
BEGIN
   IF tall
   BEGIN
      IF leafy
      BEGIN roast
      END
      ELSE
      BEGIN chop
      END
   END
   ELSE
   BEGIN
      IF red
      BEGIN
         IF hard
         BEGIN mince
            IF crisp
            BEGIN
               IF root
               BEGIN peel
               END
               ELSE
               BEGIN fry
               END
            END
            ELSE
            BEGIN steam
            END
         END
         ELSE
         BEGIN grill
         END
      END
      ELSE
      BEGIN slice
      END
   END
 ELSE
 BEGIN
      IF yellow
      BEGIN boil
      END
      ELSE
      BEGIN grill peel
      END
 END
```

```
   NEST-INE-I
   ----------
 IF juicy THEN
    IF tall THEN
       IF leafy THEN
       roast
       NOT leafy
       chop
       END leafy
    NOT tall
       IF red THEN
         IF hard THEN
         mince
            IF crisp THEN
               IF root THEN
               peel
               NOT root
               fry
               END root
            NOT crisp
            steam
            END crisp
         NOT hard
         grill
         END hard
      NOT red
      slice
      END red
   END tall
NOT juicy
   IF yellow THEN
   boil
   NOT yellow
   grill peel
   END yellow
END juicy
```

Table 7 (cont'd): Language Solutions to Decision Table Shown in Table 6

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	2.61 (.082)	2.38 (.071)	2.35 (.055)	2.36 (.125)
Unindented	2.39 (.096)	2.21 (.045)	2.41 (.098)	2.41 (.112)

(a) Time Taken

Indented	.186 (.178)	.177 (.197)	.070 (.143)	.231 (.277)
Unindented	.152 (.167)	.036 (.113)	.180 (.213)	.057 (.121)

(b) Number of Types of Syntax Error

Indented	.227 (.243)	.244 (.317)	.077 (.159)	.377 (.428)
Unindented	.152 (.167)	.052 (.187)	.192 (.238)	.066 (.143)

(c) Number of Syntax Errors

Indented	.113 (.248)	.231 (.330)	.100 (.029)	.234 (.274)
Unindented	.150 (.262)	.144 (.239)	.137 (.198)	.057 (.154)

(d) Number of Semantic Errors

Table 8: Means and Standard Deviations for Dependent Measures (Logarithmic Transformation)

Performance Measure	1st Order Order	1st Order p	2nd Order Order	2nd Order p	3rd Order Order	3rd Order p	4th Order Order	4th Order p
Time Taken	1	.00	1	.00	1	.00	1	.00
No. of Types of Syntax Error	2	.02	3	.69	3	.10	4	.76
No. of Syntax Errors	3	.88	2	.03	4	.89	3	.13
No. of Semantic Errors	4	.37	4	.37	2	.08	2	.08

Table 9: Step-Down Tests for Performance Measures

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	1.29 (.95)	1.67 (.98)	1.38 (.74)	.67 (1.03)
Unindented	1.14 (1.35)	1.8 (1.03)	.88 (.35)	2.14 (.69)

Table 10: Means and Standard Deviations for Number of Error-Free Problems per Subject

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	7	6	8	6
Unindented	7	10	8	7

(a) Cell Sizes

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	945.0 (33.54)	923.7 (37.37)	963.1 (34.16)	927.3 (38.93)
Unindented	947.9 (20.47)	947.2 (42.48)	943.9 (35.86)	903.6 (29.81)

(b) TE Score - Mean and Standard Deviation

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	.43	.17	.50	.17
Unindented	.43	.50	.50	.57

(c) Percentage of Females

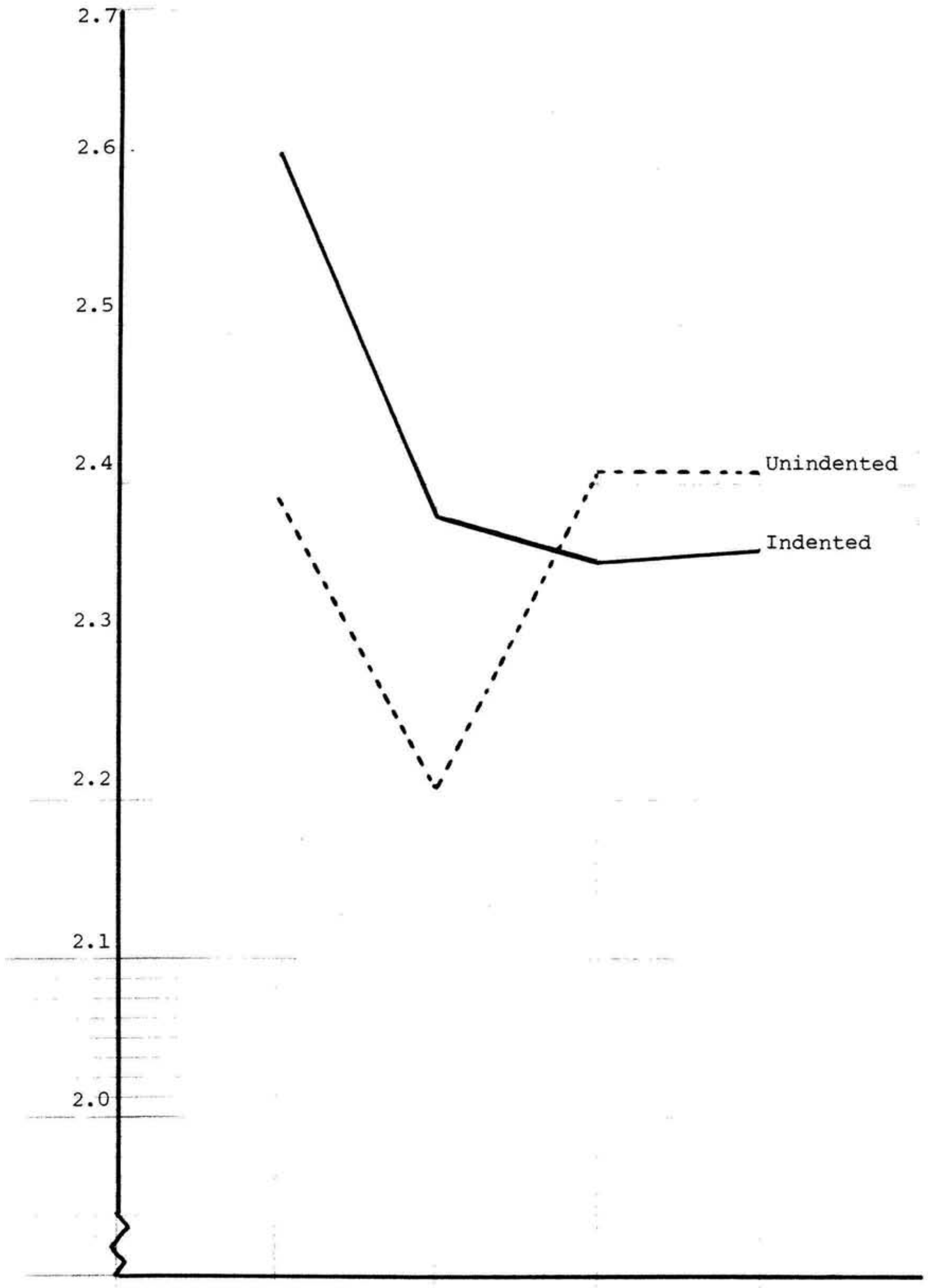
	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	0	0	0	0
Unindented	0	.40	.63	1

(d) Percentage of Part-Time Students

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	.14	.17	.50	.50
Unindented	.29	.50	.50	.86

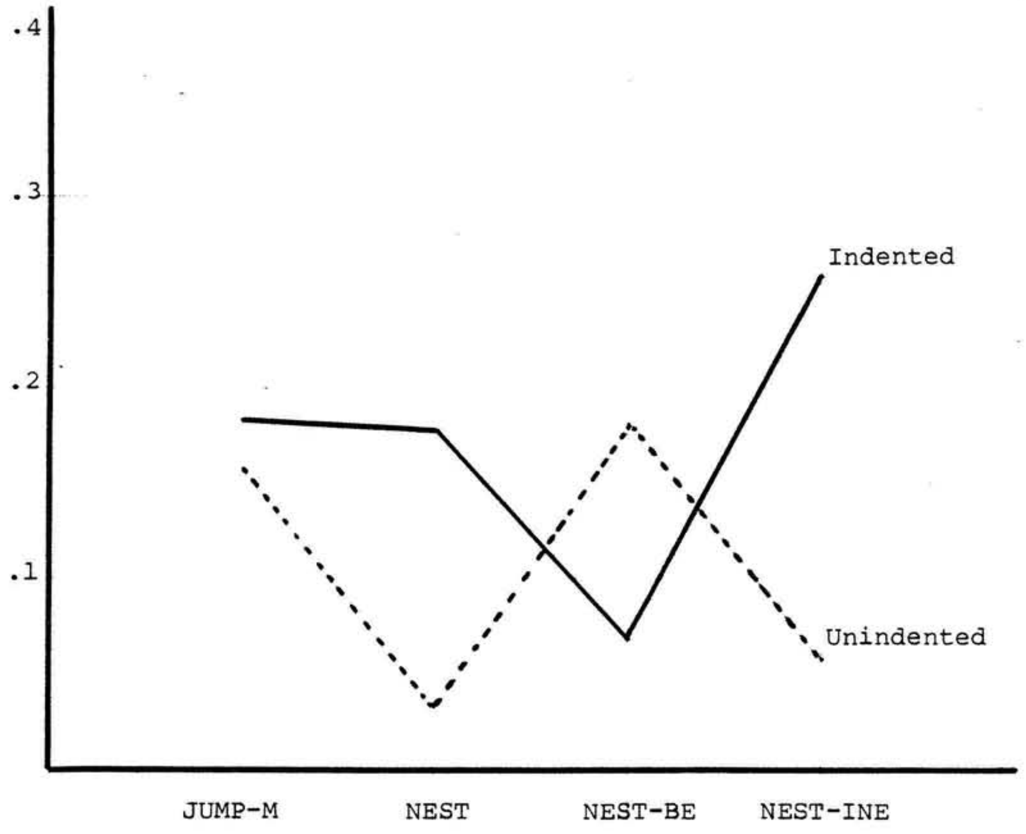
(e) Percentage of Students with Some Computer Experience

Table 11: Characteristics of Students Participating in Experiment

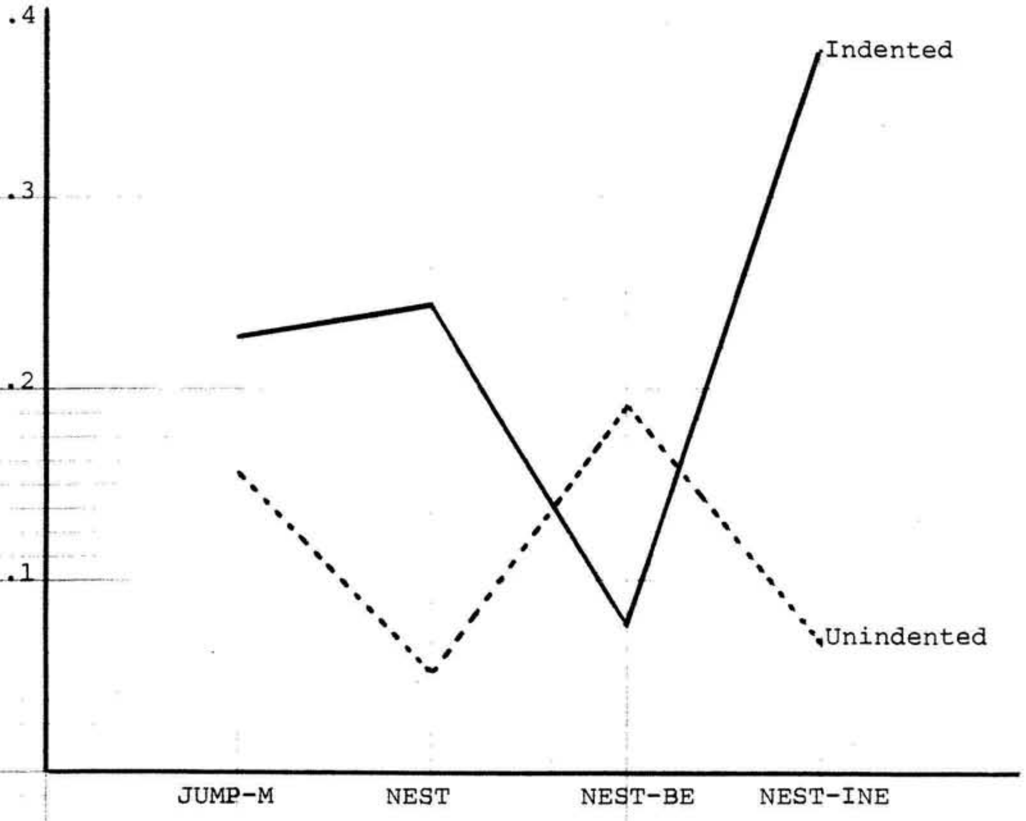


(a) Time Taken

Figure 1: Language by Indentation Interaction Ef



(b) Number of Types of Syntax Error



(c) Number of Syntax Errors

Figure 1 (cont'd): Language by Indentation Inter

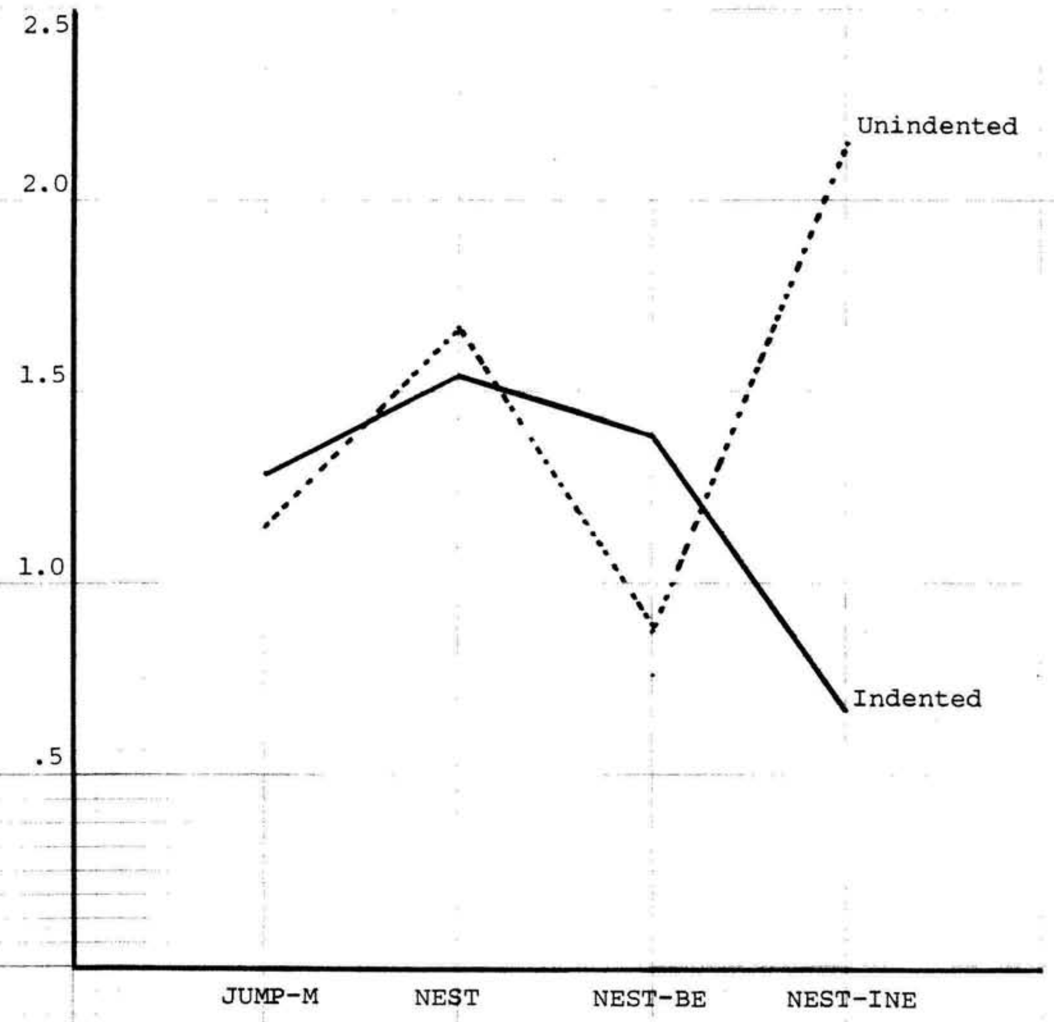


Figure 2: Number of Error-Free Problems

Appendix A: Means and Standard Deviations for Dependent Measures

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	418.6 (79.17)	244.4 (41.36)	223.1 (29.61)	237.8 (76.28)
Unindented	249.0 (57.59)	162.6 (16.89)	260.6 (55.33)	265.6 (81.89)

(a) Time Taken

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	.67 (.73)	.67 (.84)	.25 (.53)	1.11 (1.61)
Unindented	.52 (.60)	.13 (.43)	.71 (.91)	.19 (.40)

(b) Number of Types of Syntax Error

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	1.00 (1.45)	1.39 (2.34)	.29 (.62)	2.72 (3.38)
Unindented	.52 (.60)	.33 (1.47)	.83 (1.24)	.24 (.54)

(c) Number of Syntax Errors

	JUMP-M	NEST	NEST-BE	NEST-INE
Indented	.67 (1.98)	1.33 (2.17)	.38 (.71)	1.11 (1.61)
Unindented	.81 (1.89)	.70 (1.58)	.54 (.93)	.24 (.70)

(d) Number of Semantic Errors

REFERENCES

1. Bock, R.D. Multivariate statistical methods in behavioral research, McGraw-Hill, New York, 1975.

2. Christensen, L.B. Experimental methodology, 2nd ed., Allyn and Bacon, Boston, 1980.

3. Cook, T.D. and Campbell, D.T. Quasi-experimentation: design and analysis issues for field settings, Rand McNally, Chicago, 1979.

4. DeMarco, T. Structured analysis and system specification, Prentice-Hall, Englewood Cliffs, NJ, 1979.

5. Elshoff, J.L. A numerical profile of commercial PL/I programs. Soft. -- Prac. and Exper., 6, 4(Oct.-Dec. 1976), 505-525.

6. Floyd, R.W. The paradigms of programming. Comm. ACM, 22, 8 (Aug. 1979), 455-460.

7. Gilb, T. Structured programming. Computing, 16, Oct. 16, pp. 12-13; Oct. 23, p. 13.

8. Gildersleeve, T.R. Decision tables and their practical application in data processing, Prentice-Hall, Englewood Cliffs, NJ, 1970.

9. Green, T.R.G. Conditional program statements and their comprehensibility to professional programmers. J. Occup. Psychol., 50 (1977), 93-109.

10. Information Builders, Inc. 1982 Focus Users Manual, Info. Builders, New York, 1982.

11. Kirk, R.E. Experimental design: Procedures for the behavioral sciences, Brooks, Belmont, Calif., 1968.

12. Knuth, D.E. An empirical study of FORTRAN programs. Soft. -- Prac. and Exper., 1, 2, (April - June 1971), 105-133.

13. Knuth, D.E. Structured programming with go to statements. Comput. Sur., 6, 4, (Dec. 1974), 261-301.

14. Litecky, C.R., and Davis, G.B. A study of errors, error-proneness and error diagnosis in COBOL. Comm. ACM, 19, 1, (Jan. 1976), 33-37.

15. Messmer, D.J., and Homans, R.E. Methods for analyzing experiments wih multiple criteria. Dec. Sc., 11, 1(Jan. 1980), 42-57.

16. National CSS. Nomad II Reference Manual, National CSS, Wilton, Ct., 1981.

17. Neter, J., and Wasserman, W. Applied linear statistical models, Irwin, Homewood, Ill., 1974.

18. Read, N.S., and Harmon, D.L. Assuring MIS success. Datamation, 27, 2(Feb. 1981), 109-120.

19. Saal, H.J., and Weiss, Z. An empirical study of APL programs. Comput. Lang., 2, 3(1977), 47-59.

20. Sheil, B.A. The psychological study of programming. Comput. Sur., 13, 1(Mar. 1981), 101-120.

21. Shneiderman, B. Software psychology: Human factors in computer and information systems, Winthrop, Cambridge, Mass., 1980.

22. Sime, M.E., Green, T.R.G., and Guest, D.J. Psychological evaluation of two conditional structures used in computer languages. Int. J. Man-Mach. Stud., 5, (1973), 123-143.

23. Sime, M.E., Green, T.R.G., and Guest, D.J. Scope marking in computer conditionals -- A psychological evaluation. Int. J. Man-Mach. Stud., 9 (1977), 107-118.

24. Welty, C., and Stemple, D.W. Human factors comparison of a procedural and a nonprocedural query language. ACM Trans. Database Systems, 6, 4(Dec. 1981), 626-649.