AN ABSTRACT OF THE THESIS OF

James R. Ullrich for the degree of Master of Science in Computer Science presented on March 12, 1986.

Title: An Experimental Investigation Of The Cloze Procedure As A Measure of Program Understanding

Abstract approved: _____

Curtis Cook

Previous research in program understanding has been hampered by the lack of an easy tool to measure the degree of that understanding. The *cloze* procedure, suggested by Cook, Bregar, and Foote (1984) was an attempt to find a simple, reliable, and valid measure of program understanding. In this procedure, tokens are systematically removed from a computer program; it is the task of the subject to fill in the missing tokens with their correct values. The current study was an extension of this procedure; instead of filling in the blanks for the missing tokens the subjects were asked to match each blank with a token from a complete list of all missing tokens. The presentation of the matching cloze procedure was controlled by a computer; thereby making it possible to record a great deal of detail about the experimental session.

The matching cloze procedure was given to beginning, intermediate, and advanced Computer Science students using two different levels of program difficulty. The results of Adelson (1981, 1984) and Chase and Simon (1973) would predict an interaction between these independent variables. In fact, the task seemed too easy and few statistical differences were found between the groups. Interestingly, reaction time data strongly indicated that some of the tokens were not treated independently of one another.

An Experimental Investigation Of The Cloze Procedure As A Measure Of Program Understanding

by

James R. Ullrich, Ph.D.

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed March 12, 1986

Commencement June 1986

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented March 12, 1986

Table of Contents

List of Tables

An Experimental Investigation Of The Cloze Procedure As A Measure Of Program Understanding

Introduction

Program understanding is crucial to the two most time consuming and costly phases of the software life cycle: program testing and maintenance. Numerous controlled experiments have attempted to assess the effect of various program and programmer qualities on understanding. The dependent variables used to define program understanding in these experiments have included measures such as the comprehension quiz, the time to locate a bug or perform a modification, the accuracy of reproducing a functionally equivalent program without notes, as well as subjective reports. In each of the cases the operational definition of program understanding was not derived from any theoretical basis or validation procedure. The lack of justification and validation raises serious questions about meaningfulness and usefulness of the results of these experiments.

The goal of the present research was to find a theoretically sound measure of program understanding which is reliable, valid, and easily utilized in experimental research. A potential candidate for this measure the 'cloze' score. In the cloze procedure subjects are presented a program listing with some of the tokens (operands, operators, reserved words, etc.) replaced with blanks and are required to fill in the blanks. The cloze score is the proportion of blanks filled in correctly.

In a preliminary experiment (Cook, Bregar, and Foote, 1984), the cloze procedure was compared with the most commonly used and accepted measure of program understanding--the comprehension quiz. Insofar as the cloze

procedure has been used in reading comprehension research it appears on theoretical grounds to tap an important cognitive component of understanding in an analogous research area. Cook, Bregar, and Foote gave computer science students of varying levels of programming experience either a cloze version of a Pascal sorting program or a comprehensive quiz over the program. Results for the cloze procedure closely approximated those of the comprehension quizzes for both programs and for each level of experience. The cloze procedure shows promise over the comprehension quiz in that it is not as subject to experimenter bias, it is not as dependent upon the skill of the author, and it is not as dependent upon the skill of the person doing the scoring as the comprehension quiz. The reliability, validity, and objectivity of the cloze procedure are discussed in this paper.

Cook, Bregar, and Foote (1984) used a paper and pencil version of the cloze procedure in which subjects filled in blanks with what they thought were the missing tokens. Their task permitted the measurement of accuracy but did not permit either the measurement of the time between nor the order of the responses. In the present study, a computer is used to present the stimuli and record the responses. With this apparatus, it is possible to investigate not only the correctness of the match but also the interresponse times and the order in which the subjects completed the task. The recording of this level of detail provides for an extremely rich set of dependent variables available for *a posterori* "data snooping" with attendant hypothesis formation.

In the present study, program complexity and subject experience are factorially varied to determine if the cloze procedure is experimentally sensitive and if it is related to theoretical predictions. Almost any theory of programing understanding would predict that a more complex program would be more

difficult to understand. Similarly, it is difficult to imagine a situation in which the experienced subjects do not perform better than novices (Adelson, 1984 provides a counter example). In fact, an interaction would be expected between subject experience and program complexity. This interaction may be demonstrated by noting that on a very easy task, both beginners and experts would do very well, whereas on a very difficult task, only the experts would do well.

The presence of an interaction between experimental and subject variables is well known in the psychological literature. For example, Chase and Simon (1973) have found an interaction between experience and stimuli in the ability of novice and master chess players to recall chess configurations. This interaction was interpreted as reflecting the ability of chess players to perceptually organize configurations of chess pieces. In the present context, the understanding of a program almost certainly involves knowledge of the syntax and semantics of the program. Experts almost certainly will have more of both types of knowledge whereas novices may well only have knowledge of the syntactic structure. Behaviorally, novices may well "bounce" around the program filling in those tokens which only require syntactic knowledge of the program. Experts may be able to progress through the program sequentially using both types of knowledge. Similarity, Chase and Simon were able to identify different chunking strategies in the perception of chess configurations between novice and master chess players.

These differences between the cognitive representations of programs formed by experts and novices was explored extensively by Adelson (1981). She found that in the free recall of small computer programs novices tended to organize around syntactic categories and that experts tended to organize around

semantic categories. In another study Adelson (1984) novices were able to answer concrete questions about programs better than experts and that experts were able to answer abstract questions about these same programs better than novices. In these two studies as well as the Chase and Simon (1973) study, experts were found to favor abstract representations of complex problems and novices were found to favor concrete representations. These differences are manifest in interactions between the subject variable (expertise) and task difficulty in each of these experiments.

In the present experiment, subjects varying in experience are presented with a matching version of the cloze task with programs varying in difficulty. The subjects are asked to fill in the missing tokens from a list of the actual missing tokens; the elapsed time and the order of the responses are recorded. The findings of previous research suggests that there should be a significant interaction between programmer expertise and program difficulty. On easy programs, both experts and novices should be able to proceed though the program in a straight forward manner; on the difficult programs, novices are expected to match "bounce" around the programs to a much greater degree.

Previous Studies of Program Understanding

Definitions of reliability, validity, and sensitivity

It is appropriate to formally define what is meant by the terms reliability, validity, and sensitivity. A reliable variable is one in which repeated observations will yield the same value. Correctness of an answer in the cloze procedure is a reliable variable insofar as repeated observations by the same or different observers will yield the same numerical result; "goodness" of code may be an unreliable variable in that different people may have different criteria of "goodness." A dependent variable without the property of reliability is, of course, of little value.

A dependent variable is sensitive if it is reliable and if experimental manipulations produce variations in its value. Some variables such as time to complete a task may be too sensitive in that the effects of an experimental variable may be masked by individual differences in programmer ability or other factors. Other variables may not be sensitive enough because manipulations which are theoretically and practically important may not produce differences between groups. For example, measuring the length of a program by the weight of paper might be an example of a dependent variable which is too insensitive, even though it is highly reliable. The sensitivity and reliability of measurements are obviously dependent upon experimental controls.

Validity is a more difficult construct to define. First, a variable must have face or construct validity insofar as it must appear to tap that which is of theoretical interest. Time to fix an error in a program may be an appropriate measure of some constructs, but it probably does not capture the essence of the low- and high-levels of program understanding. A valid measure of a

construct must reflect variations in that construct. Thus, a measure of program understanding must surely be higher for more skilled programmers than for novices.

Previous Experimental Manipulations

Numerous studies have attempted to assess the influence of the various aspects of a program and the programming process on understanding by using a variety of dependent variables. A partial list of these manipulations includes modularization (Woodfield and Dunsmore, 1981), comments (Woodfield and Dunsmore, 1981; Sheppard, Borst and Curtis, 1978), indenting (Weissman, 1974; Shneiderman and McKay, 1976; Curtis, Sheppard, and Milliman, 1979), structured coding (Weissman, 1974; Love, 1977), mnemonic variable names (Curtis, Sheppard, Milliman, Borst and Love, 1979; Weissman, 1974), program length (Curtis, Sheppard, Milliman, Borst and Love, 1979), flowcharts (Shneiderman, Mayer, McKay and Heller, 1977), documentation (Sheppard, Kruesi and Curtis, 1979), control flow (Weissman, 1974; and Shneiderman, 1982) , and data flow (Weissman, 1974). The following paragraphs contain brief critiques of a number of these studies and their experimental measures.

The comprehension quiz or "question-answer" technique was used by Shneiderman (1982), Woodfield and Dunsmore (1981), Sheppard, Kruesi and Curtis (1981) and by Weissman (1974). The use of the comprehension quiz is highly suspect in that it is difficult to use objectively. It is very dependent upon the skill of the author, it is difficult to compare between experiments, and is subject to experimenter bias.

Time measures, such as the time to perform a modification (Curtis, Sheppard and Milliman, 1979) and time to locate a bug (Curtis, Sheppard and

Milliman, 1979; Sheppard, Curtis and Milliman, 1979) have also been utilized in investigations of program complexity. Brooks (1980) has criticized time measures in noting that irrelevant behavior may form a significant and highly variable portion of the total time. For example, the amount of time spent on irrelevant sections of code may dominate the measure.

Halstead's measure of programming effort, E (Gordon, 1979) and McCabe's $\nu(G)$ (Curtis, 1986) have received considerable attention in the research literature. These are metrics defined upon a program; not behavioral measures of the understanding of a program by a programmer. As such they cannot be used *a priori* as a measure of program understanding.

Reproduction of functionally equivalent program without notes was used as a measure by Shneiderman (1976), Shneiderman (1977), and Curtis, Sheppard, Milliman, Borst and Love (1979). This measure is limited to very small programs. Even though a programmer may be extremely familiar with, say, a typical compiler, he or she will be unable to reproduce it exactly. Furthermore, the procedure is difficult to score and may be subject to experimenter bias.

Speed of hand execution of program was used as a dependent measure by Weissman (1974). There is no easy way to insure that hand execution of a program involves knowledge of the overall structure or organization of a program (Brooks, 1980).

The many different measures and the lack of formal procedures for validation makes any comparison or interpretation of the experimental results difficult and suspect. Program understanding must include both low-level (each statement of program) and high-level (module or overall program) comprehension. It is not clear that time to perform a modification, time to locate a bug, and speed of hand execution require high-level program

comprehension. Therefore, these dependent variables lack necessary construct or face validity. Some studies (Curtis, Sheppard, Milliman, Borst and Love, 1979; Curtis, Sheppard and Milliman, 1979) have shown varying correlations between Halstead's E measure and understanding measured by time to locate a bug. The interpretation of these studies is difficult because the experimenters performed the statistically indefensible operation of correlating an independent variable and a dependent variable. Specifically, these authors calculated correlation coefficients between the time to locate a bug (dependent variable) and Halstead's E for a number of programs (independent variable). It is well known that the magnitude of correlation coefficients can be manipulated by the restriction or expansion of the range of the independent variable.

Originally Shneiderman (1977) required a subject to reconstruct a program, verbatim, after studying it for a period of time. This was later relaxed by requiring the subject to reproduce a functionally equivalent program. He assumed a strong relation between the ability to memorize program statements and the ability to understand their intended function. His measure is only feasible for toy or small programs because in large programs understanding may be confounded with subjects long term memory skills.

Of these measures of program understanding, the comprehension quiz seems to be the most generally used and accepted. The comprehension quiz includes fill-in-the-blank, multiple-choice, as well as open ended type questions. Many examinations assessing classroom performance are of this type; indeed it has almost become a *defacto* standard in colleges and universities. A well constructed comprehension quiz can have many advantages: it can measure low-level and high-level understanding, it is not limited by program length, it is flexible, and it has been shown to be sensitive

to experimental manipulations. it does however have a number of limitations; these include, but are not limited to:

1. Accurate, reliable, and valid questions are difficult to write. Much success depends upon the skill of the person or persons writing the questions. The questions must cover all aspects of a program, the content of one question must not provide the answer to another.

2. It is difficult to operationalize. How does one compare the questions written by two different people for the same material? How can one compare the results about different experimental material? This essential problems is getting and scoring comparable questions of unequal quality...comparing "apples with oranges."

3. The accuracy and reliability are severely limited by the ability of the person or persons doing the scoring (if the questions are not multiple choice or true/false).

4. Questions about the reliability and validity of each set of questions must be answered anew for each new experiment. It is difficult to "standardize" a series of questions between experiments.

5. And, importantly, the questions may be subject to experimenter bias. It is well known in the psychological literature that the experimental milieu and the subtle phrasing of questions can and frequently do have an effect upon the demand characteristics of an experimental situation. These, in turn, can influence the outcome of the experimentation.

In all of the above studies no attempt was made to validate the measure of understanding by comparing different behavioral measures or by a theoretical justification.

Since measurement and experimentation are complementary processes, the results of an experiment are limited by the measures used. One study (Shepard, Curtis, Milliman and Love, 1979), used the ability to reconstruct a functionally equivalent program without notes as the measure of program understanding, and found that meaningful variable names were not significant. This is surprising because it is contrary to one of the basic tenets of

programming that has been reaffirmed in practice over many years and almost certainly reflects the power of the measures and statistical tests employed by these authors. The (lack of) results in this study underscores the need to validate and demonstrate the sensitivity and reliability of a proposed measure of program understanding before it is used as a standard measure.

Cloze Procedure

The word "cloze" refers to the human tendency to complete a familiar but not quite finished pattern. In a cloze procedure certain parts of the text are replaced by blanks and the subject is required to fill in the blanks. The cloze score is the proportion of the number of blanks filled in correctly. The cloze procedure was originally developed to measure comprehension in English readability studies (Taylor, 1953). Shneiderman (1980, page 27) mentioned the cloze procedure as a measure of program composition tasks as well as comprehension, but cited no studies of program comprehension which employed it.

The cloze procedure almost certainly taps both the syntactical and the semantic understanding of a program. Many tokens which could be eliminated from a program are those which are determined by the syntax of a language. For example, if the left bracket, '[', were eliminated, the syntax of the language would demand a match for the corresponding right bracket, ']' remaining. On the other hand, there are semantic constraints within a program. If the deleted token is the increment to a variable in an assignment statement, the semantics of a program might dictate that the increment be unity instead of some other constant. Thus, the cloze procedure could be sensitive to knowledge of both the syntax and the semantics of a program.

In adapting the cloze procedure to study programs, subjects are presented a program listing with some of the tokens (operands, operators, reserved words, etc.) replaced with blanks and are required to fill in the blanks. Norcio (1979) has proposed using the cloze procedure to measure program understanding. But he used a different approach than that described here. Instead of tokens,

he replaced entire statements with blank lines and required subjects to supply the correct statements for the blank lines. Soloway, Ehrlich and Bonar (1982) have also used this version of the cloze procedure in experiments comparing the programming plans for novice to expert programmers.

In a preliminary experiment Cook, Bregar, and Foote (1984) compared the cloze procedure to a comprehension quiz. The participants in the experiment were computer science students in a sophomore Pascal programming course (CS 212), a junior data structures course (CS 318), and a senior operating systems course (CS 415). CS 212 is a prerequisite for CS 318, and CS 318 is a prerequisite for CS 415. The courses defined three levels of programming experience.

Each subject was randomly given one of two versions of a Pascal program and either a cloze version of the program or a comprehension quiz over it. Two versions of a Shell sort were used for the Pascal program. Very few errors (5%) were made by any of the students in filling in the blanks when the tokens were reserved words, parentheses or brackets; these tokens were called "giveaways". The presence of giveaways suggested counting only the usual operators and operands in future cloze procedure experiments. These authors compared the three main effects (class, test method and program version) using an analysis of variance. The results indicated that the class and program version were significant, but the test method was not significant. The more experienced groups of programs had fewer errors on the cloze procedure and the comprehension quiz. Thus, these results strongly suggest that for each program version and each class the cloze scores give a close relative approximation to the comprehension quiz scores. This study is important because it demonstrated that the cloze procedure can measure some of the

things measured by a comprehension quiz and that it is sensitive to experimental manipulations.

The cloze procedure is very easy to construct and score in that it is only necessary to identify the individual tokens and recognize if answer provided by the subject is the missing token. Cook, Bregar, and Foote eliminated every fifth token from their Pascal programs. Indeed demonstration programs have been written to identify each token in a COBOL or a Pascal program. In general the cloze procedure has none of the limitations of the other measures of program understanding.

The Present Study

The goal of this research is to find a measure of program understanding which is reliable, valid, and easily utilized. Many different and varied measures of program understanding have been used in experiments designed to assess the impact of various program characteristics and programming techniques on software quality. Frequently these techniques have been based upon personal programming experience and intuition and have not been systematically compared with other measures. It is proposed to investigate the viability of the cloze procedure as a reliable and valid measure of program understanding by using the technique in a variety of experimental conditions which are widely believed to affect program understanding.

To reduce variability and facilitate the determination of the correctness of a response, a "matching" procedure was used instead of the "fill in the blank" procedure used by Cook, Bregar, and Foote. With this modification, the subjects were presented with a sorted list of the deleted tokens as well as the programs with the tokens deleted; their task was to match the blanks in the program with the tokens. There was a one to one mapping of blanks and tokens which produce a correct program.

The matching cloze procedure has a great deal of face and construct validity in that its completion involves both semantic and syntactic knowledge of the program; and it is easy to use and easy to score. Other questions of the validity of this procedure remain--does it correlate highly with other measures and is it sensitive to experimental manipulations which are widely believed to affect program understanding. The matching cloze procedure provides a rich set of observations for protocol analysis in the spirit of Newall and Simon

(1972), for analysis of interresponse times, for analysis of particular tokens, etc.

This study was designed as a factorial one in which program complexity and subject experience would be varied. Pascal programs, varying in complexity was presented to three different strata of students ("Novices" , "Intermediate", and "Experienced"). Program complexity was the "within subjects" factor; subject experience was the "between subjects" factor (Kirk, 1968; Winer, 1971). Each subject was scorred for performance on the cloze procedure.

Methods

Design

This study utilized a split plot factorial design in which there were three between subjects factors and a single within subject factor with a sample size of two. Kirk (1968) classifies this design as a *SPF 3 2 2 . 53* with an n of 2. The three between subjects factors were level of experience, type of apparatus, and order of presentation. The single within subject factor had fifty-three levels, the first twenty-one of which corresponded to missing tokens from an easier program and the last thirty-two of which corresponded to missing tokens from a longer, more difficult program. Under the conditions of this design, there was a total of twelve different groups of subjects for each of the possible combinations of level of experience (3), type of computer (2), and order of presentation (2).

Subjects

The subjects were volunteers from classes at Oregon State University in the summer of 1985. They were recruited from CS212 (Techniques for Computer Programming), CS317 (Data Structures and Programming) CS411 (Assemblers and Compilers). Membership in one of these classes was the operational definition of level of experience; it was assumed that the students from CS212 were less experienced than those from CS317 who were in turn less experienced then those from CS411. This definition seems reasonable in that CS212 is a prerequisite for CS317 which in turn is a prerequisite for CS411. Each subject was a volunteer and was paid ten dollars for his or her participation in this experiment.

Program materials

Two programs were selected for stimulus materials; the easier (MILL178) was the same as selected by Cook, Bregar, and Foote (1984). This shell sort procedure was originally obtained from Miller (1981, p. 178). The second program (GROG202) was a concordance to count the occurrences of words within a text file; it was obtained from Grogono (1983, p. 202). It was identical to that used in the Soloway, Bonar, and Ehrlich (1983) study of indentation.

As in the Cook, Bregar, and Foote (1984) article, a token was defined to be

> ...a variable indentifier name, a constant, an operator (arithmetic or logical), a single parenthesis, or a single bracket. A single colon (not part of an assignment operator), a semicolon and a comma were not counted as tokens.

Every fifth token was deleted from the entire MILL178 program segment, and every fifth token was deleted from the center of the GROG202 program. A total of twenty-one and thirty-two tokens were deleted from the MILL178 and GROG202 programs, respectively. A copy of each of the programs is presented in Appendixes III and IV; the deleted tokens are presented in a bold typeface.

The MILL178 program is easily judged to be easier to comprehend than the GROG202 program; it is shorter, 50 lines in contrast to 123 lines. In addition the GROG202 program involves the more advanced topics of sets, packed words, buffers, and character manipulations which the MILL178 program did not.

Apparatus

The matching cloze procedure was presented to the subjects on either an IBM XT or an IBM AT personal computer; each of which was equipped with an AMDEX 310A monochrome video monitor.

The screen of each of the monitors was divided into three windows for presentation of the program with the missing tokens, for presentation of the tokens, and for presentation of information reflecting which window was active. The task of the subject was to match a blank in the left hand window with one of the missing tokens in the right hand window and insert that missing token into the blank. In a paper and pencil version of this task, this would correspond to the subject writing in the answer into a blank and scratching out answer just used. Figure 1 contains an example of a typical screen display.

The window for the presentation of the programs was presented in the upper left portion of the screen and was twenty six columns in width by twenty two lines in length. The programs were presented in this window, with the missing tokens indicated by five underscore characters. When the subject filled in one of these blanks the indicated token was illustrated in reverse video. The subject could choose the slot in which to insert the token by manipulating cursor control keys.

In the right hand section of the screen there was a window nine characters wide by twenty two lines deep which contained a lexical sorting of the tokens, one per line. The subject could choose which token to insert by manipulating cursor control keys. After a token was used, it was displayed in a lower intensity to enable to subject to see which token had already been selected.

On the screen this task was quite simple; the subjects viewed a computer program in the left screen with blanks for missing tokens; the missing tokens were sorted and presented in the right screen. The subject manipulated cursor keys to position the cursor on a blank indicating a missing token in the left screen and on a token in the right screen; then she or he pushed the "Ins"

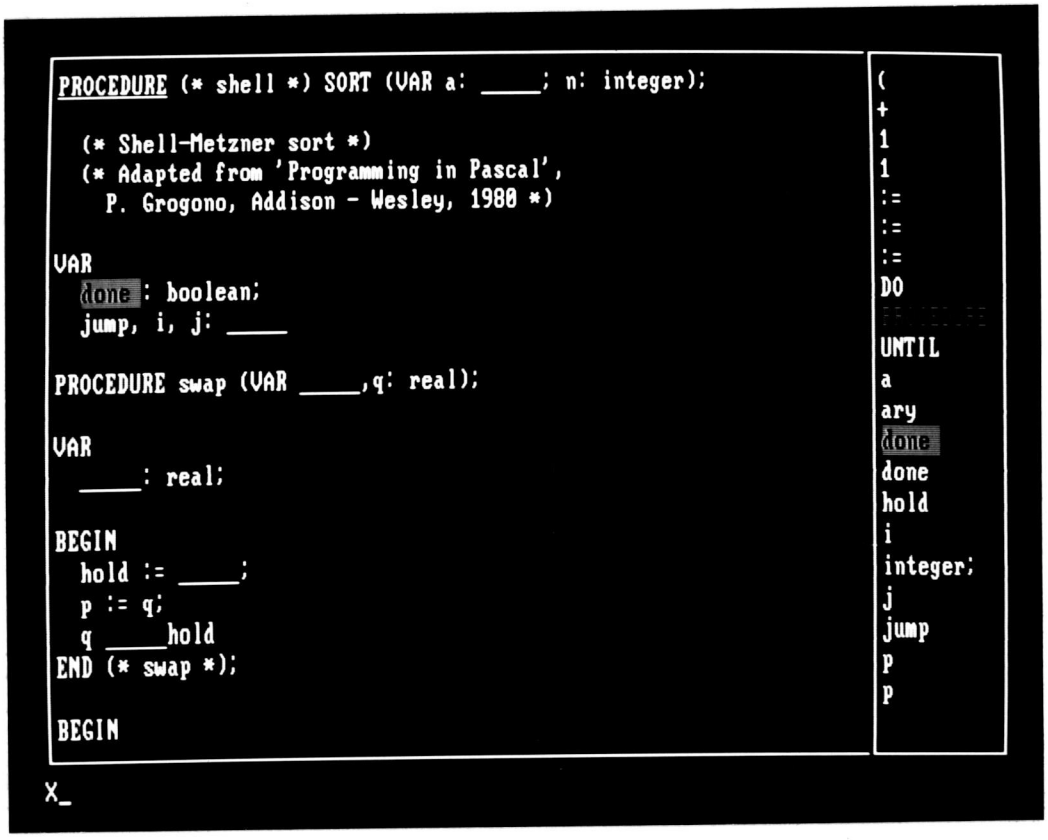

Figure 1: Typical Screen Display

(insert) key to move the token from the right window to the left window. The cloze program then replaced the blank in the left window with the indicated token in reverse video and the intensity of the display of the chosen token was decreased to show it has been used. The subject could then move the cursor to the next blank and next token to be selected and repeat the procedure.

Insofar as the size of the windows was limited, it was not possible to display an entire program to the subjects. Accordingly, the controlling program provided for the scrolling of programs in the left window and scrolling of tokens in the right window whenever appropriate keys were pressed.

The task of learning to use the cloze program and completing the task was judged to be a difficult one by the experimenter. To reduce potential confounding of the variable of practice on the task of manipulating the cursor controls, inserting the tokens, etc. with that of the topic of interest--the Pascal programs, the subjects were given two training tasks. In the first task, the subjects were presented with instructions on how to complete the task via the cloze procedure itself. The instructions in English were presented in the left screen with occasional key words replaced by blanks; the missing key words could be found in the left screen. With this technique, the subjects read the instructions and learned to operate the apparatus in a single step; by actively executing the instructions all subjects seemed to have little difficulty learning the mechanics of the computerized cloze procedure. The specific directions are provided in Appendix I with the missing words presented in a boldface typeface.

In the second task, the subjects were asked to insert missing tokens into a short greatest common divisor algorithm; in this fashion they were asked to complete a small task that was quite similar to that used in the actual test. The greatest common divisor program is presented in Appendix II, again with the

missing tokens presented in boldface. The experimenter was available to answer occasional questions during the two training tasks. The procedure of training the subjects with the directions about how to operate the apparatus and with a small Pascal program was judged to be an effective one; the only difficulty experienced was two or three subjects for whom English was not their native language had occasional difficulty with the English instructions.

Figure 1 presents an example of the appearance of the screen in a middle of a hypothetical experimental session. In this example, one can see that the token *PROCEDURE* has been inserted into the first line of the program in the left hand window, the underscoring indicates that this was one of the tokens which had been deleted. In addition, the intensity of that token has been reduced on the ninth line of the right hand window, also indicating that this token has been utilized. The hypothetical subject has just inserted the token *done* into the eight line of the left hand screen; since the cursor has not been moved, the token is presented in inverse video. As before, the intensity of the token just inserted is reduced on line thirteen of the right hand window. The current location of the cursor is always indicated by reverse video, and missing tokens in the left window by underscores.

The use of the computerized version of the matching task made it possible to record a great deal of detailed information about an experimental session. In particular, every keystroke and the time to the nearest hundredths of a second were recorded. With this level of detail, it is possible to reproduce a virtually identical description of what transpired on the monitor of the personal computers. In practice, only the keystrokes which inserted tokens and their corresponding times were analyzed.

A complete copy of the cloze program is presented in Appendix V; it is

written in TURBO Pascal, Version 3.0.

Results

Reaction time data

By using computers to present the cloze matching task to the subjects and to record the time and key of each keystroke of the subjects it is possible to tabulate a huge number of dependent variables for analysis. The data were considerably simplified by examining only the first insertion of a token into a blank slot; those keystrokes which lead to manipulations of the two windows, deletion of tokens from slots and their subsequent corrections were discarded. Additional analyses which are not included in this thesis indicated that this simplification did not affect the results of the study. With this data reduction, it was possible to calculate the time since the beginning of the experiment, t; the time since the previous insertion or deletion, Δt, and the sequential number of the insertion, *trial*. The *trial* numbers were incremented for each insertion and deletion made by the subject. Each of these variables were subjected to a SPF 3 2 2 . 53 and a SPF 3 2 2 . 2 analysis of variance (Kirk, 1968) in which there were three levels of experience E, two levels of computer C (IBM AT vs. IBM XT), two levels of order O (MILL178 first vs. GROG202-first). The single within-subjects variable was program P (MILL178 vs. GROG202) or the fifty-three specific tokens T within the two programs. There were twenty-one tokens in the MILL178 program and thirty-two tokens in the GROG202 program. In one sense, the SPF 3 2 2 .2 ANOVAs were redundant, the results of these analyses could have been derived from the other ANOVAs by appropriate definitions of contrasts among means; performing a second ANOVA was considerably easier insofar as the appropriate statistical program was readily available. Both analyses had a sample size of two. The Ullrich-Pitz ANOVA routine (Ullrich and

Pitz, 1977) was used for all analyses of variance.

When the first analysis of variance on t, the elapsed time since the beginning of the experiment was performed, only tokens (T), and the triple interaction between computer, order, and tokens (COT) were statistically significant; $F(52,624) = 45.379$, $p < 0.000000$ and $F(52,624) = 1.446$, $p < 0.02448$, respectively. When an ANOVA on the mean elapsed time across all tokens within a given program was performed (the SPF 3 2 2 .2 design), only the factor reflecting the two computer programs (P) was statistically significant, $F(1,12) = 136.208$, $p < 0.00000$. The mean elapsed time for all insertions on the MILL178 program was 497.3 seconds and 1187.3 seconds for the GROG202 program. No other main effects or interactions were statistically significant at the 0.05 level in either analysis.

Realizing that the variability of t for tokens early in a program was likely considerably less than the variability of t for tokens late in a program, and since this heterogeneity of variances might affect the interpretation of the ANOVAs, the analyses were repeated with a logarithmic transformation. The results were very similar to the ANOVA without the transformation; T, $F(52,624) = 77.204$ $p < 0.00000$, and COT, $F(52,624) = 1.443$, $p < 0.02559$ were statistically significant. When the elapsed times were collapsed into a single score for each program, only the program factor was significant, $F(1,12) = 201.375$ $p < 0.00000$.

When ANOVAs were performed on the time since the last insertion or deletion, Δt, T and P were significant, $F(52,624) = 6.386$ $p < 0.0000$ and $F(2,12) = 33.769$ $p < 0.00020$, respectively. In addition, the OP interaction was significant, $F(1,12) = 5.016$ $p < 0.04286$ when the tokens were collapsed for each of the programs. The mean Δt for the MILL178 program were 45.5796 and 35.1367, when that program was presented first and second respectively. The mean Δt

for the GROG202 programs were 56.7852 and 60.3962 when that program was given to the subjects second and first, respectively. It is apparent that the mean reaction time on an individual token is longer when that token is embedded in a more complex program (GROG202) than when that token is in an easier program (MILL178).

To further analyze the time since the previous insertion or deletion, Δt, with respect to the type of tokens; the tokens were dichotomized into groups reflecting whether that token was syntatically or semantically required. Those tokens which were classified as required by the syntax of Pascal were primarily key words whereas those classified as required by the semantics of the program were primarily variable names. The tokens labeled with asterisks in Table 3 are those which were classified as syntatically required, the remainder are classified as semantically required. The first token, "PROCEDURE" in each program was ignored for this analysis. Averages were computed for each dichotomy and each program, collapsing the 53 scores down to four for each subject. When SPF 3 2 2 . 4 ANOVA was performed on this data only the repeated measure variable reflecting the token classifications for the two programs was significant, $F(3,36) = 9.689$, $p < 0.001$. For the MILL178 program the average Δt scores were 46.68 and 44.46 seconds for the syntatically required and semantically required tokens, respectively. These are not significantly different from each other using Tukey's HSD test (Kirk, 1968). For the more difficult GROG202 program, the average Δt scores were 50.52 and 66.10 seconds for the syntatically required and semantically required token, respectively. Tukey's HSD test indicated that this difference was significant at the 0.01 level.

The analyses of variance on the integer variable *trial* produced significant

interactions for EC, EO, T, and COT when analyzed for each token and EC, EO, and P when collapsed across tokens for each program. The F values, degrees of freedom, and p values are presented in Table 1; Table 2 presents the means for each of the significant interactions for the second analysis. When ANOVAs were performed on these data with logarithmic transformations, the same pattern of results emerged. Examination of the means in Table 3 fails to yield a conclusion which can be easily summarized verbally; at best the interpretation is quite difficult.

SPF 3 2 2 . 53

Source	df	F	p
EC	2,12	6.366	0.01299
ECO	2,12	5.560	0.01926
T	52,624	42.921	0.00000
COT	52,624	1.406	0.03520

SPF 3 2 2 . 2

Source	df	F	p
EC	2,12	6.091	0.01480
EO	2,12	6.011	0.01539
P	1,12	248.525	0.00000

Table 1: Statistically significant effects on the variable *trial*

CS212	CS317	CS411	
14.3484	14.4006	15.1230	IBM AT
14.9775	15.4159	14.1801	IBM PC
14.8263	14.4747	15.1449	IBM AT
14.4996	15.4418	14.1518	IBM PC

Table 2: Mean trial number of an insertion

The Δt scores for each individual token in each program is presented in Table 3. An informal observation of these scores is quite interesting; consider the 16th and 17th missing tokens in the MILL178 program, i and j. These two

tokens control the direction of the sort, whether it is ascending or descending; they occur as part of that a semantic entity. The i token had a mean Δt score which was double that of any other token in that program, the score for the j token was the lowest. It is clear that subjects spent time on the the pair of tokens, and answering the two in rapid succession. A similar, but less pronounced phenomenon occurred in the GROG202 program; consider the 7th and 8th missing tokens REPEAT and UNTIL. The REPEAT had one of the longest times, the UNTIL one of the shortest. Again, the explanation is that subjects spent time on the pair, and answering in rapid succession. It is clear that programs cannot be considered to be a linear sequence of tokens, each independent of its neighbors.

MILL178 Program

Number	Token	Time	Errors	Errors (CBF)
1	PROCEDURE	33.1737	0	1
2	ary	57.2483	7	14
3	done	47.9175	1	0
4	integer;	40.6179	1	0
5	p	47.3508	1	2
6	hold	29.8292	1	2
7	p	26.8858	2	1
8*	:=	28.5050	0	0
9*	:=	50.9171	0	0
10	1	67.7600	3	13
11	jump	36.6967	1	12
12*	:=	47.0079	0	0
13	1	28.6050	0	2
14*	DO	22.6117	0	1
15	+	39.7058	1	5
16	j	100.075	9	3
17	i	12.7088	9	3
18*	(	37.8267	0	0
19	a	17.6729	1	0
20	done	26.5104	0	0
21*	UNTIL	47.8950	0	1

			37	

Table 3: Time since previous insertion or deletion, Δt

Table 3, continued

GROG202 Program

Number	Token	Time	Errors
1	PROCEDURE	139.103	0
2	wordtype	148.407	2
3	' '	69.8638	2
4	charindex	96.7296	8
5	0	61.8888	1
6*	BEGIN	52.6471	1
7*	REPEAT	91.2887	0
8*	UNTIL	14.2879	3
9*	IN	56.6758	3
10	eof	63.4846	2
11	0	35.7421	3
12*	DO	45.7033	0
13	maxwordlen	61.4229	4
14	charcount	27.5404	0
15	charcount	59.1342	1
16*	IF	67.9446	2
17	blank	68.1688	4
18	ch	42.9792	2
19*	:=	31.9529	1
20	maxwrdlen	62.7604	1
21*	]	44.4792	0
22	buffer	51.6579	2
23*	END	53.1600	0
24	wordtype	82.0804	0
25*	VAR	56.3100	0
26*	]	41.1388	1
27*	..	24.2133	0
28	packword	55.3404	4
29	charpos	44.5421	5
30*	DO	28.6408	0
31	charpos	13.8738	2
32	letters	81.7400	1

	Total		55

Table 3: Time since previous insertion or deletion, Δt

Number of errors

When an examination of the number of errors made by the subjects was made it was discovered that only 92 errors were made on the first insertion by all subjects under all conditions of this experiment. Table 3 presents the number of errors broken down by the specific tokens for each of the two programs. Table 4 presents the total number of errors for each of the experimental groups in the study.

Group	Errors
CS212	39
CS317	32
CS411	33
IBM AT	44
IBM PC	48
Easy-Hard	53
Hard-Easy	39

Table 4: Number of errors for each group

The number of errors made by individual subjects varied from a minimum of zero to a maximum of nine. No formal statistical tests on these data were made because any tests would be suspect because of the low error rate. The only differences in error rate among the various experimental conditions seems to exist were between the two programs and the order in which the programs were presented. Using error rate as a criterion, the MILL178 program does seem to be easier than the GROG202 program; and there seem to be more errors when the easier program is presented first.

The MILL178 program was nearly identical to one of those used by Cook, Bregar, and Foote (1984); the comments in the Pascal program had been deleted

and the name of a procedure was changed (from *SWAP* to *Miller*), and some variable names had been changed to something which was less meaningful. While there was a total of 37 errors made on this program by the twenty-four subjects in this study, 60 errors were made by Cook, Bregar, and Foote's thirteen subjects. These errors are also reported in Table 3 under the heading CBF errors. It is readily apparent that the matching cloze procedure with comments is considerably easier than the traditional cloze procedure without comments. Nevertheless, the two groups of subjects tended to make errors on the same tokens; the Pearson product moment correlation coefficient on the number of errors made across the twenty-one tokens was 0.42, $p < 0.05$.

Order of token insertions

A major tenent of the present investigation was that more experienced programmers would progress through the programs in a straight-forward or linear fashion and that less experienced programmers would move through the programs in a more scattered manner. To test this hypothesis, scatter plots of each insertion and deletion were made for each subject and for each program; the abscissa was the trial number of that response, the ordinate was the ordinal number of the missing token within the particular program. Extensive visual inspection of these scatter plots failed to reveal any consistent pattern. As a further check, Pearson product moment correlations were computed for each scatter plot; this statistic is a crude measure of the degree to which the scatter plot followed a straight line. Those subjects who filled in the first token on first trial, the second token on the second trial, etc. would have a correlation near 1.0; those subjects who bounced around the program in their answers would have correlations much closer to zero.

There were 48 correlation coefficients computed, one for each subject under each experimental condition; these coefficients were then subjected to a *SPF 3 2 2 .2* ANOVA. The only statistically significant effects were the CO interaction ($F(1,12) = 7.258$ $p < 0.01868$) and the CP interaction ($F(1,12) = 6.459$ $p < 0.02468$). Inspection of the average correlation coefficients indicated that the IBM-AT, MILL178-first combination and the IBM-XT, GROG202-first combinations were more highly correlated then their counterparts. Similarly, the IBM AT, MILL178 program and the IBM XT GROG202 program combinations were more highly correlated then their counterparts. It is difficult to relate these findings to the major experimental hypotheses.

Discussion

Three results seem to follow from this experimentation: 1) the GROG202 program was more difficult than the MILL178 program, 2) the subjects made very few errors on the task, and 3) the reaction time on some tokens is much longer than that on other tokens. On the more difficult program, those tokens which were required by the syntax of Pascal were inserted faster than those tokens for which semantic knowledge was necessary; there was no difference in the speed of insertion of these two types of tokens for the easier program. The first result is hardly surprising insofar as the programs were selected so as to vary in difficulty.

The fact that the overall difficulty of the task, as judged by the number of errors, was not great was somewhat surprising; particularly when pretesting of students at Lewis and Clark College had indicated a much higher error rate. Cook, Bregar, and Foote reported a much higher error rate on the MILL178 program than was found in the current experimentation. There are a number of factors which could account for the low error rate; there is no reason to favor one explanation over another without additional data.

1. The subjects in the present experimentation were likely more motivated than those used by Cook, Bregar, and Foote. The present subjects were paid volunteers whereas the Cook, Bregar, and Foote subjects were unpaid volunteers who participated during part of a regularly scheduled class. In addition, the computerized nature of this task may have been more impressive to the computer science students who served as subjects. The experimenter's subjective impression of these subjects was that he had never seen subjects that were as motivated as were those in the present experiment.

2. Inclusion of comments and mnemonic names for variables in the programs used in present study undoubtedly made the task easier; Cooke, Bregar and Foote's version of the MILL178 program had comments removed and some variable names changed.

3. The 1-1 matching version of the cloze task is simply much easier than the corresponding fill-in-the-blank version. The subjects must merely recognize, perhaps through a process of elimination, which token goes in a given slot in contrast to generating the needed token.

4. It is possible that the subjects in this experiment were considerably more sophisticated than those in Cook, Bregar, and Foote; casual observation suggests that current Oregon State University students in Computer Science classes have a greater familiarity with Pascal than those students only a few years earlier.

5. Finally, it may be that the computerized version of the matching task is not a sensitive, reliable, and valid tool which can be used in the investigation of program understanding. The evidence gathered in the current experimentation suggests that this may be the case. This interpretation is supported by the fact that experimental manipulations which had an extremely high *a priori* chance of producing differences failed to do so.

The occasional and difficult to understand higher order interactions which were found the analysis of variance are interpreted as random variations; there were a large number of F tests performed in the analysis of the current data and about five percent would be expected to show statistical significance. A number of other statistical tests were performed on that data which were not discussed; these failed to show a consistent and statistically significant pattern. The conclusion seems inescapable; the matching cloze procedure is too easy to be a sensitive tool. Cook, Bregar, and Foote (1984) and Soloway, Bonar, and Ehrlich (1983) used similar programs as the current study (MILL178 and GROG202, respectively) and these authors found differences between groups.

It is disappointing not to find that the matching version of the cloze procedure was not a more viable tool for the investigation of program understanding. This is particularly true insofar as the mechanics of setting up, scoring, and analyzing an experiment using this procedure are extremely straightforward. Once the program and the starting point are selected, the

selection of tokens, presentation, scoring, and analysis can be largely automatic. The conventional cloze procedure requires considerably greater effort in scoring. Future research is suggested utilizing techniques to make the task more difficult. These techniques could include introducing more tokens than answers, allowing multiple uses of a given token, removing comments, and making variable names less meaningful.

The third conclusion that there were substantial differences in the reaction times of some of the tokens is particularly interesting. Chase and Simon (1973) used reaction times to identify differences in the perceptual organization of expert and novice chess players. Casual observation of subjects in the present investigation did reveal there there were indeed differences between subjects. Further analysis of a larger number of programs for a smaller group of subjects which are more diverse might substantial differences akin to those reported by Chase and Simon (1973) and by Adelson (1981). Systematic analysis of these reaction times should prove to be fertile ground for further research into the cognitive aspects of program understanding.

Bibliography

Adelson, B. (1984). When novices surpass experts: the difficulty of a task may increase with expertise. *Journal of Experimental Psychology* 10, 483-495.

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & Cognition* 9, 422-433.

Berztiss, A. T. (1971). *Data structures theory and practice.* Academic Press, New York.

Brooks, R. E. (1980). Studying programmer behavior experimentally: the problem of proper methodology, *Comm. ACM*, 23, 207-213.

Chase, W. G., and Simon, H. A. (1973). Perception in Chess., *Cognitive Psychology* 5, 55-81.

Cook, C., Bregar, W, and Foote, D. (1984). A preliminary investigation of the use of the cloze procedure as a measure of program understanding. *Information Processing & Management*, 20, 199-208.

Curtis, B. (1986) Conceptual issues in software metrics. Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, 154-157.

Curtis B., Sheppard, S. B., and Milliman, P. (1979). Third time charm: stronger replication of the ability of software complexity metrics to predict programmer performance, *Proceedings Fourth International Conference on Software Engineering*, Munich, Germany, 356-360.

Curtis B., Sheppard, S. B., Milliman, P., Borst, M. A., and Love, T. (1979). Measuring the psychological complexity of software maintenance tasks with Halstead and McCabe metrics, *IEEE Trans. Software Eng.*, Vol SE-5, 96-104.

Embley, D. W. (1978). Empirical and formal language design applied to a unified control construct for interactive computing, *International J. Man-Machine Studies*, 10, 197-216.

Gordon, R. (1979) Measuring improvements in program clarity, *IEEE Trans. Software Eng.*, Vol SE-5, 79-90.

Grogono, P. (1983). *Programming in Pascal*, (2nd Ed.), Addison-Wesley, Reading, Massachusetts.

Halstead, M. (1977). *Elements of software science*, Operating and Programming Systems Series, Elsevier Computer Science Library, New York.

Kirk, R. (1968). Experimental Design: Procedures for the Behavioral Sciences. Brooks/Cole, Belmont, California.

Klare, G. R. (1974-1975). *Assessing readability, Reading Research Quarterly*, 10, 63-102.

Klare, G. R. (1976). A second look at the validity of readability formulas, *J. Reading Behavior*, 8, 129-152.

Love, T. (1977). An experimental investigation of the effects of program structure on program understanding, *ACM SIGPLAN Notices*, 10, 105-113.

Miller, A. R. (1981). *Pascal programs for scientists and engineers*, Sybex Publishing Company, Berkeley, California.

Moher, T. and Schneider, G. M. (1981). Methods for controlled experimentation in software engineering, *Proceedings Fifth International Conference on Software Engineering*, San Diego, California, 224-233.

Norcio, A. F. (1979). The cloze procedure: a methodology for analyzing computer program understanding, *ACM Computer Science Conference*, Dayton, Ohio.

Norcio, A. F. (1979). Factors affecting the comprehension of computer programs, National Computer Conference, New York.

Sheppard, S. B., Borst, M. A., and Curtis, B. (1978). Predicting programmer ability to understand and modify software, *Proceedings of Symposium on Human Factors and Computer Science*, Washington, D. C., 115-135.

Shepard, S. B., Curtis, B. Milliman, P. and Love, T. (1979) Modern coding practices and programmer performance, *Computer*, 138-146.

Sheppard, S. B., Kruesi E., and Curtis, B. (1981). The effects of symbology and spatial arrangement on the comprehension of software specifications, *Proceedings Fifth International Conference on Software Engineering*, San Diego, California, 207-214.

Shneiderman, B. (1976). Exploratory experiments in program behavior, *International J. Computer and Information Sciences*, 5, 123-143.

Shneiderman B. (1977). Measuring computer program quality and comprehension, *International J. Man-Machine Studies*, 9 465-478.

Shneiderman, B. (1980). *Software psychology : Human factors in computer and information systems*, Winthrop Publishers Inc., Cambridge, Massachusetts.

Shneiderman, B. (1982). Control flow and data structures documentation: two experiments, *Comm. ACM*, 25, 56-63.

Shneiderman, B., Mayer, R. McKay D., and Heller, P. (1977). Experimental investigation of the utility of detailed flowcharts in programming,*Comm. ACM*, 20 373-381.

Shneiderman, B. and McKay, D. (1977). Experimental investigation of computer program debugging and modification, *Proceedings of Sixth International Congress of the International Ergonomics Association.*

Soloway, E., Ehrlich, K., and Bonar, J. (1982). Tapping into Tacit programming knowledge, *Proceedings Human Factors in Computer Systems Conference,* Gaithersburg, Maryland, 52–57.

Soloway, E. Bonar, J., and Ehrlich, K. (1983). Cognitive Strategies and Looping Constructs: An Empirical Study. 26, 853–867.

Taylor, W. L. (1953). Cloze procedure: a new tool for measuring readability, *Journalism Quarterly,* 30, 415–433.

Ullrich, James R. and Pitz, Gordon F. (1977). A general purpose analysis of variance routine. *Behavior Research Methods and Instrumentation* 9, 301.

Weissman, L. (1977). Psychological complexity of computer programs: an experimental methodology, ACM SIGPLAN Notices, 10 , 105–113.

Winer, B.J. (1971). Statistical Principles in Experimental Design, McGraw–Hill, New York.

Woodfield, S. N. and Dunsmore, H.E. (1981). The effect of modularization and comments on program comprehension, Proceedings of Fifth International Conference on Software Engineering, San Diego, California, 215–223.

APPENDICES

Appendix I

Directions Given to Subjects

Missing tokens are in **boldface** type

Thank you for participating in this
study. The task we are about to
give you is a computerized matching one.
You can see that some of the words
of this text in the left screen have
been **deleted** and replaced with blank lines or
slots. The actual deleted words appear in
alphabetical order in the right screen.

Your **task** is to correctly match the
deleted word in the right screen with
the appropriate slot in the left **screen**.

This is essentially a computerized version
of the standard matching task in which
you **match** one item on the right
screen with one slot on the left screen.
There is an exact match between items and slots
in both number.

To move the cursor around, you may use
the LEFT and **RIGHT** arrow keys (4 and 6)
on. To move the screen up or down, you
may use the **UP** or DOWN arrows (2 or 8
on the keypad). Additionally, you may
use the **HOME** key to go to the top,
the **END** key to go to the end,
and the PAGE-UP and PAGE-DOWN keys to move
the text up or down one half page.
Try the keys.

To change screens, enter the "C" key
(for "C"hange). When the C key is pressed,
the cursor switches between screens.
The currently active screen is indidated by
an **X** at the bottom of the screen.

To INSERT one of the words on the right
screen into a slot in the left screen
press the **INSERT** key after you have
matched the blank slot in the left screen
with the blank slot in the right screen.

The matching is done by moving the cursor left and right with **LEFT-ARROW** and RIGHT-ARROW keys. The LEFT-ARROW key moves the cursor to the **left** and **up**, the RIGHT-ARROW key moves the cursor to the right and down. To insert a word into a blank slot, the cursor in the desired position by **pushing** the right or left arrow keys for each window. It is likely that you will have to do several moves and screen changes to indicate the answer in the blank slot. You actually do the insert, you press the INSERT key at the top of the numeric keypad. When you do the insert, you will notice that the **intensity** of the word changes.

If you change your mind about an answer you can erase or **delete** a word by matching the answer with the blank and pressing the DELETE key at the top of the keypad. You must have an exact match in order to delete an answer.

You finish the session by hitting the E key for **EXIT**.

Appendix II

Greatest Common Divisor Program

Missing tokens are in **boldface** type

```
PROGRAM gcd (input,output);

VAR
  r,m,n: integer;

BEGIN
  readln(m,n);
  REPEAT
    r := m MOD n;
    m := n;
    n := r;
  UNTIL n = 0;
  writeln(m);
END.
```

Appendix III

Program MILL178

Missing tokens are in **boldface** type

```
PROCEDURE (* shell *) SORT (VAR a: ary; n: integer);

  (* Shell-Metzner sort *)
  (* Adapted from 'Programming in Pascal',
    P. Grogono, Addison - Wesley, 1980 *)

VAR
  done : boolean;
  jump, i, j: integer;

PROCEDURE swap (VAR p,q: real);

VAR
  hold: real;

BEGIN
  hold := p;
  p := q;
  q := hold
END (* swap *);

BEGIN
  jump := n;
  WHILE jump > 1 DO
    BEGIN
      jump := jump DIV 2;
      REPEAT
        done := true;
        FOR j := 1 TO n - jump DO
          BEGIN
            i := j + jump;
            IF a[j] > a[i] THEN
              BEGIN
                swap(a[j],a[i]);
                done := false
              END (* IF *)
          END (* FOR *)
      UNTIL done
    END (* while *)
END (* sort *);
{ Count occurences of each word in a text. }
```

Appendix IV

GORG202 Program

Missing tokens are in **boldface** type

```
PROGRAM concordance (input,output);
CONST
  tablesize = 100;
  maxwrdlen = 20;
TYPE
  charindex = 1..maxwrdlen;
  counttype = 1..maxint;
  tableindex = 1..tablesize;
  wordtype = PACKED ARRAY [charindex] OF char;
  entrytype =
  RECORD
    word : wordtype;
    count: counttype
  END;
  tabletype = ARRAY [tableindex] OF entrytype;
VAR
  table : tabletype;
  entry, nextentry : tableindex;
  tablefull : boolean;
  letters : SET OF char;
  { Read one word from the text. A word is a string of letters.
  Words are separated by characters other than letters. }
  PROCEDURE readword(VAR packdword: wordtype);
  CONST
    blank = ' ';
  VAR
    buffer : ARRAY [charindex] OF char;
    charcount : 0..maxwrdlen;
    ch : char;
    BEGIN
      IF NOT eof
        THEN
         REPEAT
          read(ch)
        UNTIL eof OR (ch IN letters);
      IF NOT eof
        THEN
          BEGIN
            charcount := 0;
```

```
        WHILE ch IN letters DO
          BEGIN
            IF charcount < maxwrdlen
              THEN
                BEGIN
                  charcount := charcount + 1;
                  buffer [charcount] := ch
                END;
              { then }
            IF eof
              THEN ch := blank
              ELSE read(ch)
          END;
        { while }
        FOR charcount := charcount + 1 TO maxwrdlen DO
        buffer [charcount] := blank;
        pack (buffer, 1, packdword)
      END { then }
 END;
{ readword }
{ Print a word. }
PROCEDURE printword (packdword : wordtype);
CONST
  blank = ' ';
VAR
  buffer : ARRAY [charindex]  OF char;
  charpos : 1r..nmaxwrdlen;
  BEGIN
    unpack (packdword,buffer,1);
    FOR charpos := 1 TO maxwrdlen DO
    write (buffer [charpos])
  END;
{ printword }


BEGIN { concordance }
  letters := ['a' .. 'z'];
  tablefull := false;
  nextentry := 1;
  WHILE NOT (eof OR tablefull) DO
    BEGIN
      readword (table [nextentry].word);
      IF NOT eof
        THEN
          BEGIN
```

```
            entry := 1;
            WHILE table [entry].word <>
              table [nextentry].word DO
            entry := entry + 1;
            IF entry < nextentry
              THEN table [entry].count :=
                table [entry].count + 1
              ELSE
              IF nextentry < tablesize
                THEN
                  BEGIN
                    nextentry := nextentry + 1;
                    table [entry].count := 1
                  END { then }
                ELSE tablefull := true
          END;
        { then }
    END;
  { while }
  IF tablefull
    THEN writeln ('The table is not large enough.')
    ELSE
    FOR entry := 1 TO nextentry - 1 DO
    WITH table [entry] DO
      BEGIN
        printword (word);
        writeln (count)
      END { else, for, and with }
END.
{ concordance }
```

```
{*************************************************************************}
{B+}                                {Input/output to con:                 }
{C+}                                {Control char interupt during  execution }
{I+}                                {All I/O checked for errors           }
{R+}                                {Range checking                       }
{V+}                                {Type checking on string passed as VAR }
{U+}                                {User may interrupt program anytime   }
{*************************************************************************}
{                          James R. Ullrich                               }
{*************************************************************************}
PROGRAM Cloze;
CONST
   {Screen Positioning Constants                                          }
   LeftUpperLeftX = 2;    LeftUpperLeftY = 2;
   LeftLowerRightX = 68;  LeftLowerRightY = 23;
   RightUpperLeftX = 70;  RightUpperLeftY = 2;
   RightLowerRightX = 79; RightLowerRightY = 23;
   MaxString = 70;
   BottomLine = 25;
   MaxRTIndex = 1000;
   MaxRSPIndex = 100;

   {Cursor movement keyboard constants:                                   }
   CursorHome = 71; CursorUp = 72;   CursorPgUp = 73;
   CursorLeft = 75;                  CursorRight = 77;
   CursorEnd = 79;  CursorDown = 80; CursorPgDwn = 81;

   {Other Keyboard Command Input Constants                                }
   SelectAnswer = 'S'; InsertAnswer = 'I';  EraseAnswer = 'E';
   ChangeScreen = 'C'; Quit = 'Q';


TYPE
    DisplayType     =   (N,NF_NC,NF_C,F_NC,F_C,          { <--left screen only }
                         NS_NC,NS_C,S_NC,S_C);          { <--right screen only}
        {Normal, Filled_in or not, Current or not, Selected or not}
```

```
    LeftRight       =   (Left,Right);
    UpDown          =   (Up,Down);
    WindowBound     =   ARRAY[1..4] of integer;
    StringType      =   STRING[70];
    DateStr         =   STRING[10];
    TimeString      =   STRING[11];
    Ptr             =   ^cell;
    PtrArray        =   ARRAY[Left..Right] of Ptr;
    RTRange   =   1 .. MaxRTIndex;
    RSPRange  =   1 .. MaxRSPIndex;
    Cell = RECORD
             Display:          DisplayType;        {display characteristics}
             RowId,ColId:      INTEGER;            {unique identifier}
             Data:             String[MaxString];  {the string  displayed}
             Lines:            Integer;            {number lines from top }
             Up,Down,Left,Right: Ptr;
             TokenLeft,TokenRight: Ptr;
             Match:            Ptr;
           END;
    RTData   =
           RECORD
             Char:             Char;
             Time:             TimeString;
           END;
    RSPData =
           RECORD
             Char:             Char;
             Correct:          Char;
             Time:             TimeString;
             LRowId,LColID,RRowID,RColID:   INTEGER;
           END;
CONST
    LeftWTable  : WindowBound  =
                      (LeftUpperLeftX,LeftUpperLeftY,
                       LeftLowerRightX,LeftLowerRightY);
    RightWTable : WindowBound =
```

```
                    (RightUpperLeftX,RightUpperLeftY,
                     RightLowerRightX,RightLowerRightY);

VAR
    {Data Structure Pointers                                              }
    UpperLeft,LowerRight,
        LowerLeft                   :    PtrArray;

    {Screen and Cursor pointers                                           }
    TopScreen,BottomScreen,
    LinePointer,CursorPoint         :    PtrArray;
    FirstToken                      :    PtrArray;
    LastToken                       :    PtrArray;

    CurrentScreen                   :    LeftRight;

    I,J,Temp                        :    INTEGER;
    Data                            :    StringType;

    RT: ARRAY[RTRange] of RTData;
    RSP: ARRAY[RSPRange] of RSPData;
    RTIndex                         :    RTRange;
    RSPIndex                        :    RSPRange;

    FV                              :    Text;
    RTFile                          :    Text;
    RSPFile                         :    Text;


{*************************************************************************}
{*************************************************************************}
Function Date: DateStr;
{Procedure to return the date; uses DOS call 2A hex                       }
{Returns a string of the form 1984/07/20                                  }
type
  regpack = record
```

```
            ax,bx,cx,dx,bp,si,ds,es,flags: integer;
          end;

var
  recpack:       regpack;                {record for MsDos call}
  month,day:     string[2];
  year:          string[4];
  dx,cx:         integer;

begin {Date}
  with recpack do
  begin
    ax := $2a shl 8;
  end;
  MsDos(recpack);                        { call function }
  with recpack do
  begin
    str(cx,year);                        {convert to string}
    str(dx mod 256,day);                     { " }
    str(dx shr 8,month);                     { " }
  end;
  date := year + '/' + month + '/' + day;
end;  {Date}

{*************************************************************************}
Function time: TimeString;
{Returns a string of the form 18:03:03.23,  HH:MM:SS.TH  T=tens H=Hund of sec }
type
  regpack = record
              ax,bx,cx,dx,bp,di,si,ds,es,flags: integer;
            end;

var
  recpack:          regpack;             {assign record}
  ah,al,ch,cl,dh:   byte;
  hour,min,sec,hund:     string[2];
```

```
begin   { Time }
  ah := $2c;                              {initialize correct registers}
  with recpack do
  begin
    ax := ah shl 8 + al;
  end;
  intr($21,recpack);                      {call interrupt}
  with recpack do
  begin
    str(cx shr 8,hour);                   {convert to string}
    str(cx mod 256,min);                        { " }
    str(dx shr 8,sec);                          { " }
    str(dx mod 256,hund);                       { " }
  end;
  time := hour+':'+min+':'+sec+'.'+hund;
end;   { Time }

{*****************************************************************************}
{*****************************************************************************}
Procedure InsertNewToken(LorR: LeftRight; VAR NewToken: PtrArray);
BEGIN
    IF FirstToken[LorR] = NIL THEN
    BEGIN
        FirstToken[LorR]                := NewToken[LorR];
        LastToken[LorR]                 := NewToken[LorR];
        FirstToken[LorR]^.Display       := SUCC(FirstToken[LorR]^.Display);
        FirstToken[LorR]^.TokenRight    := NIL;
        LastToken[LorR]^.TokenLeft      := NIL;

    END ELSE
    BEGIN
        LastToken[LorR]^.TokenRight     := NewToken[LorR];
        NewToken[LorR]^.TokenLeft       := LastToken[LorR];
        IF LorR = Left THEN
           NewToken[LorR]^.Display      := NF_NC
```

```
        ELSE
           NewToken[LorR]^.Display      := NS_NC;
        LastToken[LorR]                 := NewToken[LorR];
    END  { if };
END { InsertNewToken };



{*****************************************************************************}
PROCEDURE InsertDown(Dsply: DisplayType; R,C: Integer;
                     Dta: StringType; LorR: LeftRight);
{Procedure to insert a new node at the lower left of the graph              }
{This procedure is invoked when a new line of the input program is read     }

VAR Old:    Ptr;

BEGIN
    Old := LowerLeft[LorR];
    NEW(LowerLeft[LorR]);
    LowerRight[LorR] := LowerLeft[LorR];
    IF Old = NIL THEN
    BEGIN
        UpperLeft[LorR] := LowerLeft[LorR];
        LowerLeft[LorR]^.Lines := 1;
    END
    ELSE LowerLeft[LorR]^.Lines := old^.Lines + 1;
    WITH LowerLeft[LorR]^ DO
    BEGIN
         Display := Dsply;
         RowId      := R;
         ColId      := C;
         Data       := Dta;
         Up         := Old;
         Down       := NIL;
         Left       := NIL;
         Right      := NIL;
```

```
            TokenLeft  := NIL;
            TokenRight := NIL;
            Match      := NIL;
        END { with };
        WHILE Old <> NIL DO
        BEGIN
            Old^.down := LowerLeft[LorR];
            Old       := Old^.Right;
        END { while };
        IF (Dsply = NF_NC) OR (Dsply = NS_NC) THEN
        BEGIN
             InsertNewToken(LorR,LowerLeft);
        END;
END { InsertDown };


{*****************************************************************************}
PROCEDURE InsertRight(Dsply: DisplayType; R,C: Integer;
                      Dta: StringType; LorR: LeftRight);
{Procedure to insert a new node at the lower right of the graph              }
{This procedure is invoked when a new token string in input                  }

VAR Old:    Ptr;

BEGIN
    Old := LowerRight[LorR];
    NEW(LowerRight[LorR]);
    WITH LowerRight[LorR]^ DO
    BEGIN
         Display    := Dsply;
         RowId      := R;
         ColId      := C;
         Data       := Dta;
         Up         := Old^.UP;
         Down       := NIL;
         Left       := Old;
```

```
          Right      := NIL;
          TokenLeft  := NIL;
          TokenRight := NIL;
          Match      := NIL;
      END { with };
      Old^.Right := LowerRight[LorR];
      IF DSPLY = NF_NC THEN InsertNewToken(LorR,LowerRight);
END { InsertRight };


{*****************************************************************************}
PROCEDURE Bubble;
{Procedure to perform a bubble sort on the right hand data structure         }
{this changes the row and col id's,  and data                                }
{all pointers are left unchanged                                             }
VAR
    I,J,Temp:    Ptr;

BEGIN
    I := Upperleft[right];
    IF I^.TokenRight = NIL THEN ELSE
        WHILE I^.TokenRight^.TokenRight <> NIL DO
        BEGIN
            J := I^.TokenRight;
            WHILE J <> NIL  DO
            BEGIN
                IF I^.Data > J^.Data THEN
                BEGIN
                    {interchange}
                    Temp^.RowId := I^.RowId;
                    Temp^.ColId := I^.ColId;
                    Temp^.Data  := I^.Data;
                    I^.RowId    := J^.RowId;
                    I^.ColId    := J^.ColId;
                    I^.Data     := J^.Data;
                    J^.RowId    := Temp^.RowId;
```

```
                    J^.ColId    := Temp^.ColId;
                    J^.Data     := Temp^.Data;
                END;
                J := J^.Tokenright;
            END { while...J };
            I := I^.TokenRight;
        END { while...I };
END { Bubble };


{*****************************************************************************}
PROCEDURE OutputDataStructure(LorR: LeftRight);
VAR
    LRTemp,LLTemp                   :    Ptr;

BEGIN
    {loop to march down the data structure}
    LLTemp := UpperLeft[LorR];
    WHILE LLTemp <> NIL DO
    BEGIN

        {loop to march across the data structure}
        LRTemp := LLTemp;
        WHILE LRTemp <> NIL DO
        BEGIN
            WITH LRTemp^ DO
            BEGIN
                WRITELN ('Row Identification: ', RowId);
                WRITELN ('Col Identification: ', ColId);
{ i/o won't work WRITELN ('Display Char:       ', Display);}
                WRITELN ('Data:               $', Data,'$');

         WRITELN;
            END { with };
            LRTemp := LRTemp^.Right;
        END { while workig across the data structure };
```

```
        LLTEMP := LLTemp^.Down;
    END { while working down the data structure };
END { Outputing the data structure };



{*************************************************************************}
PROCEDURE ReadData;
{Procedure to read the data source (modified pascal source) and place it in   }
{the global structure 'prog'                                                  }
{The manditory format for the data source is:                                 }
{    #                          To signify start of string                    }
{    RowId,ColID                Unique row and column identifies for the      }
{                               particular string                             }
{    Type                       N or R for normal or blank                    }
{    String                     The actual string itself                      }

{Note the string may be a single token or a series of tokens                  }
{An example, the sequence                                                     }
{         #3 1 N for i := #3 2 B Start #3 3 N to Term #3 4 B do #3 5 N ;      }
{would represent the following screen display                                 }
{      for i := _____ to Term _____ ;                                         }
{where the blank fields contain the hidden tokens Start and do                }

CONST
    Pound: CHAR = '#';

VAR C:      CHAR;                          {a character read on input}
    IdRow,IdCol: INTEGER;
    Display: DisplayType;
    CurrStr: STRING[MaxString];                              {Token(s) string}
    First:   ARRAY[Left..Right] of BOOLEAN;                  {First token String}
                                                             {in a line of program}
BEGIN
    First[Left] := True;
```

```
First[right] := True;
REPEAT                                                  {until eof}
BEGIN
    READ(FV,C);                                         {first # in a line}
    REPEAT                                              {read/process a line }
    BEGIN
        CurrStr := '';
        READ(FV,IdRow);read(FV,IdCol);                  {read row and column id's}
        read(fv,c);  {throw this one asway}
        READ(FV,C);                                     {'N'ormal, 'R'everse }
        CASE UpCase(C) of
            'N': Display := N;
            'R': Display := NF_NC;
            ELSE
                 WRITELN ('Input Display of Wrong Type', C);
                 Delay(3000);
                 OutputDataStructure(Left);
        END { case };
        READ(FV,C);
        WHILE (C <> Pound) AND (NOT EOLN(FV)) DO  {form current string}
        BEGIN
             CurrStr := CurrStr + c;
             READ(fv,C);
        END { WHILE };
        IF EOLN(FV) THEN CurrStr := CurrStr + C;  {get the last character}

        {Here we build the left data structure }
        IF First[Left] THEN
           InsertDown(Display,IdRow,IdCol,CurrStr,Left)
        ELSE
           InsertRight(Display,IdRow,IdCol,CurrStr,Left);
        First[Left] := false;
        IF  Display = NF_NC  THEN { build  right data structure }
            InsertDown(NS_NC,IdRow,IdCol,CurrStr,Right);
    END;
    UNTIL EOLN(FV);
```

```
        READLN(FV);                                  {Swallow cr/lf}
        First[Left] := True;                              {Next token string first in line}
    END; {repeat reading lineS }
    UNTIL EOF(FV);
    CLOSE(FV);
END  { ReadData };

{*****************************************************************************}
{*****************************************************************************}
Procedure WriteString(Element: Ptr);
{Procedure to write out a string with diffrential display characteristics     }
BEGIN
    WITH Element^ DO BEGIN
    CASE Display OF
        N       :  BEGIN
                       TextColor(15); TextBackGround(00);
                       WRITE(Data)
                   END;
        NF_NC   :  BEGIN
                       TextColor(09); TextBackGround(31);
                       WRITE('     ')
                   END;
        NF_C    :  BEGIN
                       TextColor(08); TextBackGround(31);
                       WRITE('     ')
                   END;
        F_C     :  BEGIN
                       TextColor(08); TextBackGround(31);
                       WRITE(Match^.Data)
                   END;
        F_NC    :  BEGIN
                       TextColor(09); TextBackGround(31);
                       Write(Match^.Data)
                   END;
        NS_NC   :  BEGIN
                       TextColor(15); TextBackGround(00);
```

```
                    Write(Data)
                END;
        NS_C    : BEGIN
                    TextColor(08); TextBackGround(31);
                    Write(Data)
                END;
        S_NC    : BEGIN
                    TextColor(07); TextBackGround(31);
                    Write(Data)
                END;
        S_C     : BEGIN
                    TextColor(08); TextBackGround(31);
                    Write(Data);
                END;
    END { case }
    END { with };
    TextColor(15);  TextBackground(00);
END;


{*************************************************************************}
Procedure WriteLine(LorR: LeftRight;CurrentPtr: Ptr);
{procedure to write a line from beginning to end with the current display     }
{types in force...this is called after the cursor is moved                   }
VAR
    ScreenPosition: Integer;
    Index:          Integer;

BEGIN
    {First find the beginning of the line }
    WHILE CurrentPtr^.Left <> NIL DO CurrentPtr := CurrentPtr^.Left;

    {Then we write out the new line }
    IF LorR = Left
    THEN
        Window (LeftUpperLeftX,LeftUpperLeftY,
```

```
                LeftLowerRightX,LeftLowerRightY)
    ELSE
        Window(RightUpperLeftX,RightUpperLeftY,
               RightLowerRightX,RightLowerRightY);
    ScreenPosition := CurrentPtr^.Lines - TopScreen[LorR]^.Lines + 1;
    IF (ScreenPosition >= 1) AND (ScreenPosition <= 22) THEN
    BEGIN
        GoToXY(1, ScreenPosition);
{   code to clear line
        IF CurrentScreen = Left THEN
            FOR INDEX := 1 TO LeftLowerRightX - LeftUpperLeftX +1
                DO Write(' ')
       ELSE
           FOR INDEX := 1 TO RightLowerRightX - RightUpperLeftX + 1
               DO Write (' ');
}       WHILE CurrentPtr <> NIL DO
        BEGIN
            WriteString(CurrentPtr);
            CurrentPtr := CurrentPtr^.Right
        END { while };
    END { if };
{%%%%%    GoToXY(1,1);}
END { WriteLine };



{*****************************************************************************}
Procedure MoveScreen(LorR:  LeftRight; UorD: UpDown);
{Procedure to move the left or right screens either up or down                }
{Up means the screen moves up and cursor is at bottom left                    }
{Down means the screen moves down and the cursor is at the top left           }
BEGIN
    IF LorR = Left
    THEN
        Window (LeftUpperLeftX,LeftUpperLeftY,LeftLowerRightX,LeftLowerRightY)
    ELSE
```

```
        Window(RightUpperLeftX,RightUpperLeftY,RightLowerRightX,RightLowerRightY);
    GoToXY(1,1);
    IF UorD = Up
    THEN BEGIN
        DelLine;
        GoToXY(1,LeftLowerRightY - LeftUpperLeftY + 1);
            {same values for RightLowerRightY and RightUpperLeftY}
            {ie go to the bottom of the current window}
    END ELSE
        InsLine;

END { MoveScreen };


{*****************************************************************************}
Procedure WindowUpDown(LorR: LeftRight; UorD: UpDown);
VAR
    LRTemp  :       Ptr;

BEGIN
    IF (UorD = UP) and (TopScreen[LorR]^.up <> Nil)
    THEN BEGIN
        TopScreen[LorR]  := TopScreen[LorR]^.Up;
        If BottomScreen[LorR]^.Up <> Nil THEN
            BottomScreen[LorR] := BottomScreen[LorR]^.Up;
        MoveScreen(LorR,Down);
        LRTemp := TopScreen[LorR];
        WHILE LRTemp <> NIL DO
        BEGIN
            WriteString(LRTemp);
            LRTemp := LRTemp^.Right;
            { working across the data structue};
        END { while };
    END
    ELSE IF (UorD = Down) and (BottomScreen[LorR]^.Down <> Nil)
        THEN BEGIN
```

```
            BottomScreen[LorR]  := BottomScreen[LorR]^.Down;
            IF TopScreen[LorR]^.Down <> Nil THEN
                 TopScreen[LorR] := TopScreen[LorR]^.Down;
            MoveScreen(LorR,Up);
            LRTemp := BottomScreen[LorR];
        WHILE LRTemp <> NIL DO
        BEGIN
            WriteString(LRTemp);
            LRTemp := LRTemp^.Right;
            { working across the data structue};
        END { while };
    END;

{%%%%    GoToXY(1,1);}
END { WindowUpDown };


{*************************************************************************}
Procedure CursorLeftRight (Window,LorR: LeftRight);
{procedure to move the Cursor Left or Right  within a window              }
{jumping from one token slot to another                                   }
BEGIN
  IF (LorR = Left) and (CursorPoint[Window]^.TokenLeft <> NIL)
    THEN BEGIN
        CASE CursorPoint[Window]^.Display OF
            NF_C:   CursorPoint[Window]^.Display := NF_NC;
             F_C:   CursorPoint[Window]^.Display := F_NC;
            NS_C:   CursorPoint[Window]^.Display := NS_NC;
             S_C:   CursorPoint[Window]^.Display := S_NC;
        END { Case };
        WriteLine(Window,CursorPoint[Window]);
        CursorPoint[Window] := CursorPoint[Window]^.TokenLeft
       END;
         IF (LorR = Right) and (CursorPoint[Window]^.TokenRight <> NIL)
         THEN BEGIN
            CASE CursorPoint[Window]^.Display OF
```

```
                NF_C:   CursorPoint[Window]^.Display := NF_NC;
                F_C:    CursorPoint[Window]^.Display := F_NC;
                NS_C:   CursorPoint[Window]^.Display := NS_NC;
                S_C:    CursorPoint[Window]^.Display := S_NC;
            END { Case };
            WriteLine(Window,CursorPoint[Window]);
            CursorPoint[Window] := CursorPoint[Window]^.TokenRight
        END;
    CASE CursorPoint[Window]^.Display OF
       NF_NC:   CursorPoint[Window]^.Display := NF_C;
        F_NC:   CursorPoint[Window]^.Display := F_C;
       NS_NC:   CursorPoint[Window]^.Display := NS_C;
        S_NC:   CursorPoint[Window]^.Display := S_C;
    END { Case };
    WriteLine(Window, CursorPoint[Window]);
END { CursorLeftRight };
{*************************************************************************}
Procedure ClearRTMemory;
BEGIN
    FOR i := 1 to RTIndex - 1 DO
    BEGIN
         Writeln(RTFile,RT[i].char:5,RT[i].time:10);
    END;
    RTIndex := 1;
    RT[RTIndex].char := 'Z';
    RT[RTIndex].time := Time;
    RTIndex := RTIndex + 1;
END;

{*************************************************************************}
Procedure ClearRSPMemory;
BEGIN
    FOR i := 1 to RSPIndex - 1 DO
    BEGIN
        WITH RSP[i] DO
        BEGIN
```

```
            write(RSPFile,Char, Correct, Time, LRowId:5, LColId:3);
            write(RSPFile,RRowID:5, RColId:3);
        END  { with };
        writeln(RSPFile);
    END;
    RSPIndex := 1;
    WITH RSP[RSPIndex] DO
        BEGIN
            Char := 'Z';    { P for Pause while writing to disk }
            Correct := 'Z';
            Time := RT[RTIndex].Time;
            LRowID := 0;
            LColID := 0;
            RRowID := 0;
            RColID := 0;
        END { with };
    RSPIndex := RSPIndex + 1;
END;

{*****************************************************************************}
FUNCTION ConsoleInput: Char;
{This function receives input from the console keyboard                       }
{The input character and its time stamp is written in memory                  }
{The character is edited and passed back to the calling program               }

VAR  C          :   CHAR;

BEGIN
     READ(KBD,C);
     IF C = char(27) THEN READ(KBD,C);     {use extended char set if necessary}
     RTIndex := RTIndex + 1;
     IF RTIndex <= MaxRTIndex THEN ELSE ClearRTMemory;
     RT[RTIndex].char := C;
     RT[RTIndex].time := Time;
     CASE C of
         'e','E'    : C := UpCase(C)  { Exit };
```

```
        'c','C'    : C := UpCase(C)  { Change Screen };
        'R'        : { Insert--82 };
        'S'        : { Delete--83 };
        'G'        : { Home--71 };
        'H'        : { Cursor Up--72 };
        'I'        : { Page Up--73 };
        'K'        : { Cursor Left--75 };
        'M'        : { Cursor Right--77 };
        'O'        : { End--79 };
        'P'        : { Cursor Down--80 };
        'Q'        : { Page Down--81 };
        ELSE
            Sound(Round(3000));Delay(200);NoSound;
            C := ConsoleInput; {recurse until good character };
        END;
     ConsoleInput := C;
END { ConsoleInput };

{*****************************************************************************}
PROCEDURE Delete;
{Procedure which 'undos' a previous insert                                    }
BEGIN
    IF (CursorPoint[Left]^.Display  = F_C)              AND
       (CursorPoint[Right]^.Display = S_C)              AND
       (CursorPoint[Left]^.Match = CursorPoint[Right])  AND
       (CursorPoint[Right]^.Match = CursorPoint[Left])  THEN
    BEGIN
        { ok to switch }
        CursorPoint[Left]^.Match      := NIL;
        CursorPoint[Right]^.Match     := NIL;
        CursorPoint[Left]^.Display    := NF_C;
        CursorPoint[Right]^.Display   := NS_C;
        WriteLine(Left,CursorPoint[Left]);
        write ('     ');
        WriteLine(Right,CursorPoint[Right]);
        { check to see if same and record output here }
```

```
            RSPIndex := RSPIndex + 1;
            IF RSPIndex <= MaxRSPIndex THEN ELSE
                { out of memory--write data to disk and clear array }
                ClearRSPMemory;
            WITH RSP[RSPIndex] DO
                BEGIN
                    Char := 'D';
                    Correct := 'D';
                    Time := RT[RTIndex].Time;
                    LRowID := CursorPoint[Left]^.Rowid;
                    LColID := CursorPoint[Left]^.Colid;
                    RRowID := CursorPoint[Right]^.RowId;
                    RColID := CursorPoint[Right]^.Colid;
                END { with }

        END
        ELSE BEGIN
            { one or more tokens already assigned }
            Sound(Round(3000));Delay(200);NoSound;
        END;
    END;


{*************************************************************************}
PROCEDURE Insert;
{Procedure which takes a token from the right screen and inserts it into the  }
{left screen after doing some inital error checking                           }
BEGIN
    IF (CursorPoint[Left]^.Display  = NF_C) AND
        (CursorPoint[Right]^.Display = NS_C) THEN
    BEGIN
        { ok to do the insert }
        CursorPoint[Left]^.Match      := CursorPoint[Right];
        CursorPoint[Right]^.Match     := CursorPoint[Left];
        CursorPoint[Left]^.Display    := F_C;
        CursorPoint[Right]^.Display   := S_C;
```

```
        WriteLine(Left,CursorPoint[Left]);
        write ('     ');
        WriteLine(Right,CursorPoint[Right]);
        { record output  }
        RSPIndex := RSPIndex + 1;
        IF RSPIndex <= MaxRSPIndex THEN ELSE
            { out of memory--write data to disk and clear array }
            ClearRSPMemory;
        WITH RSP[RSPIndex] DO
            BEGIN
                Char := 'I';
                IF CursorPoint[Left]^.Data = CursorPoint[Right]^.Data
                THEN Correct := 'C' ELSE Correct := 'U';
                Time := RT[RTIndex].Time;
                LRowID := CursorPoint[Left]^.Rowid;
                LColID := CursorPoint[Left]^.Colid;
                RRowID := CursorPoint[Right]^.RowId;
                RColID := CursorPoint[Right]^.Colid;
            END { with };
    END
    ELSE BEGIN
        { one or more tokens already assigned }
        Sound(Round(3000));Delay(200);NoSound;
    END;
END;


{*****************************************************************************}
Procedure Manager;
CONST
     HalfMaxLines = 12;          { Half the maximum lines on one screen }
VAR
    NoQuit:     Boolean;
    Index:      Integer;
BEGIN
    NoQuit := True;
```

```
GoToXY(1,25);  Write ('X');

While NoQuit DO BEGIN
 CASE ConsoleInput of
     'E'         : { Exit }
                   BEGIN
                        RSPIndex := RSPIndex  + 1;
                        NoQuit := False;
                   END;
     'C'         : { Change Screen }
                   If CurrentScreen = Left THEN
                      CurrentScreen :=Right ELSE CurrentScreen := Left;
     'R'         : { Insert--82 } Insert;
     'S'         : { Delete--83 } Delete;
     'G'         : { Home--71 }
                   WHILE
                       TopScreen[CurrentScreen] <> UpperLeft[CurrentScreen]
                   DO WindowUpDown(CurrentScreen,Up);
     'H'         : { Cursor Up--72 }     WindowUpDown(CurrentScreen,Up);
     'I'         : { Page Up--73 }
                   FOR INDEX := 1 TO HalfMaxLines DO
                      WindowUpDown(CurrentScreen,Up);
     'K'         : { Cursor Left--75 }   CursorLeftRight(CurrentScreen,Left);
     'M'         : { Cursor Right--77 }  CursorLeftRight(CurrentScreen,Right);
     'O'         : { End--79 }
                   WHILE
                       BottomScreen[CurrentScreen]<>LowerLeft[CurrentScreen]
                   DO WindowUpDown(CurrentScreen,Down);
     'P'         : { Cursor Down--80 }   WindowUpDown(CurrentScreen,Down);
     'Q'         : { Page Down--81 }
                   FOR INDEX := 1 TO HalfMaxLines DO
                      WindowUpDown(CurrentScreen,Down);
    ELSE
        {impossible to get here is ConsoleInput working properly}
        Sound(Round(750));Delay(1500);NoSound;
    END  { case };
```

```
        Window(1,1,80,25); TextColor(15);TextBackGround(0);
        IF CurrentScreen = Left THEN
        BEGIN
            GoToXY(RightUpperLeftX,25);
            Write (' ');
            GoToXY(1,25);
            Write('X');
        END
        ELSE BEGIN
            GoToXY(1,25);
            Write (' ');
            GoToXY(RightUpperLeftX,25);
            Write ('X');
        END { if };
    END { while };
END { Manager };


{*****************************************************************************}
{*****************************************************************************}
Procedure Frame(WTab : WindowBound);
{Procedure to draw a frame around a window                                    }
{Frame is drawn in cells one unit outside of the window boarder               }

  var
     UpperLeftX, UpperLeftY, LowerRightX, LowerRightY: Integer;
     i: Integer;
  begin
     UpperLeftX  := WTab[1]-1;  UpperLeftY  := WTab[2]-1;
     LowerRightX := WTab[3]+1;  LowerRightY := WTab[4]+1;
     GotoXY(UpperLeftX, UpperLeftY);  Write(chr(218));
     for i:=UpperLeftX+1 to LowerRightX-1 do Write(chr(196));
     Write(chr(191));
     for i:=UpperLeftY+1 to LowerRightY-1 do
     begin
        GotoXY(UpperLeftX , i);  Write(chr(179));
```

```
        GotoXY(LowerRightX, i);  Write(chr(179));
      end;
      GotoXY(UpperLeftX, LowerRightY);
      Write(chr(192));
      for i:=UpperLeftX+1 to LowerRightX-1 do Write(chr(196));
      Write(chr(217));
   end  { Frame };

{*************************************************************************}
PROCEDURE InitialScreen(LorR: LeftRight);
{Procedure to initialize the windows with text from data structure        }
{Leaves global pointers:  TopScreen and BottomScreen assigned             }

VAR
    MaxLines,NLines :   Integer;
    LRTemp,LLTemp   :   Ptr;

BEGIN
    IF LorR = Left THEN
        MaxLines := LeftLowerRightY  - LeftUpperLeftY  + 1
    ELSE
        MaxLines := RightLowerRightY - RightUpperLeftY + 1;

    {loop to march down the data structure}
    LLTemp := UpperLeft[LorR]; TopScreen[LorR] := UpperLeft[LorR];
    NLines := 1;
    WHILE (LLTemp <> NIL) AND (NLines <= MaxLines) DO
    BEGIN
        {loop to march across the data structure}
        LRTemp := LLTemp;  BottomScreen[LorR] := LLTemp;
        MoveScreen(LorR,Up);
        REPEAT
            WriteString(LRTemp);
            LRTemp := LRTemp^.Right;
            { working across the data structue};
        UNTIL LRTemp = NIL;
```

```
        LLTEMP := LLTemp^.Down;      {go down one more line}
        NLines := NLines + 1;        {and count it}
    END { while working down the data structure };
    WHILE NLINES <= MaxLines DO
    BEGIN
         Writeln;
         NLines := Nlines + 1;
     END { while }
END { Initial Screen  };


{*****************************************************************************}
PROCEDURE DemographicsOpenFiles;
VAR
    FileName:   STRING[14];
    OK:     BOOLEAN;
BEGIN
    REPEAT
        BEGIN
            clrscr;
            write('Enter name of source program: ');readln(FileName);
            assign(FV,FileName);
            {$I-} reset(FV) {$I+};
            OK := (IOResult = 0);
            IF NOT OK THEN writeln('Cannot find file named ',FileName);
        END
    UNTIL OK;

    write('Enter name of output file w/o filetype: '); readln(FileName);
    assign(RTfile,FileName+'.RT'); rewrite(RTfile);
    assign(RSPfile,FileName+'.RSP');rewrite(RSPfile);
END { DemographicsOpenFiles };


{*****************************************************************************}
```

```
PROCEDURE Initialize;
{It does just what it says                                                    }
VAR
    LorR    :       LeftRight;
BEGIN
  FOR LorR := Left to Right DO  {Initialize Pointers to Nil}
    BEGIN
      UpperLeft[LorR]     := NIL;     LowerRight[LorR]    := NIL;
      LowerLeft[LorR]     := NIL;
      TopScreen[LorR]     := NIL;     BottomScreen[LorR]  := NIL;
      LinePointer[LorR]   := NIL;     CursorPoint[LorR]   := NIL;
      FirstToken[LorR]    := NIL;     LastToken[LorR]     := NIL;
    END;

  DemographicsOpenFiles;              {Collect Demographics & Open Files}
  ReadData;                           {Read Data and build structure }
  Bubble;                             {Sort right hand window}

  ClrScr;                             {Clear Screen}
  Window(1,1,80,25);                  {open main window}
  Frame(LeftWTable);                  {draw left frame }
  Frame(RightWTable);                 {draw right frame}


  CurrentScreen       := Left;        {define screens}
  CursorPoint[Left]   := FirstToken[Left];
  CursorPoint[Right]  := FirstToken[Right];

  InitialScreen(Left);                { draw the left screen }
  InitialScreen(Right);               { draw the right screen }
  Window(1,1,80,25); TextColor(15);Textbackground(0);

  RTIndex := 1;
  RT[RTIndex].char := 'B';            { mark the beginning }
  RT[RTIndex].time := Time;
  RSPIndex := 1;
```

```
  WITH RSP[RSPIndex] DO
    BEGIN
       char := 'B';
       correct :=' ';
       time := RT[RTIndex].time;   { same starting times }
       LRowID := 0; LColID := 0; RRowID := 0; RColID :=0;
    END { with };
END { Initialize};



{*****************************************************************************}
{*****************************************************************************}
{*****************************************************************************}
BEGIN  { Main Program }
    Initialize;
    Manager;
    Window(1,1,80,25);
    ClrScr;
    ClearRTMemory;  ClearRTMemory;     close(RTFile);
           {we clear memory twice to save the RT for E response}
    ClearRSPMemory; CLearRSPMemory;    close(RSPFile);
END { main program }.
```