Development of an environment for testing programming ability using the cloze technique

Kiyoka Takahashi

The University of Montana

Follow this and additional works at: https://scholarworks.umt.edu/etd

Let us know how access to this document benefits you.

Recommended Citation

Takahashi, Kiyoka, "Development of an environment for testing programming ability using the cloze technique" (1987). Graduate Student Theses, Dissertations, & Professional Papers. 5541.
https://scholarworks.umt.edu/etd/5541
COPYRIGHT ACT OF 1976

This is an unpublished manuscript in which copyright subsists. Any further reprinting of its contents must be approved by the author.

MANSFIELD LIBRARY
UNIVERSITY OF MONTANA
DATE: 1987
DEVELOPMENT OF AN ENVIRONMENT FOR TESTING PROGRAMMING ABILITY USING THE CLOZE TECHNIQUE

By
Kiyoka Takahashi
B.S., University of Dubuque, 1984

Presented in partial fulfillment of the requirements for the degree of Master of Science University of Montana

1987

Approved by

Chairman, Board of Examiners

Dean, Graduate School

Date
The Cloze technique is an accepted measure of natural language understandability, and is in the process of being accepted as a good measure of computer software understandability.

This paper first states the importance of the easily understandable software. Then it discusses the Cloze technique as a measure of both natural language text understandability and computer programming understandability. Finally, the paper explains the automated Cloze system developed by the author and a demonstration experiment conducted to test the Cloze system.
Table of Contents

Abstract.........................................................ii
Table of Contents..................................................iii
List of Tables and Figures........................................iv

1. Introduction..................................................1
2. Related Work and Literature.................................11
3. Software Development Phase................................19
4. Cloze Experiment..............................................23

Appendix A. Requirements and Specifications ..................35
Appendix B. Design Documents....................................69
Appendix C. Program Listings......................................94

Bibliography.....................................................106
List of Tables and Figures

Figure 1 ......................................................... 6
Figure 2 ......................................................... 6
Figure 3 ......................................................... 8
Figure 4 ......................................................... 10
Figure 5 ......................................................... 12
Figure 6 ......................................................... 14
Figure 7 ......................................................... 33
Figure 8 ......................................................... 33
Figure 9 ......................................................... 34
Figure 10 ......................................................... 34

Table 1 ............................................................. 24
Table 2 ............................................................. 26
Table 3 ............................................................. 30
Table 4 ............................................................. 30
Chapter One
Introduction

1.1 Background

Consider the life cycle of a computer system. After requirements and specifications have been written, the system is designed, the programs are coded, and the system is tested. Each time the programs do not meet the requirements, the errors have to be located and corrected. But the correction is impossible without familiar knowledge of the program(s) being tested. Then we are in the last phase of the software life cycle -- the maintenance phase. The longer the system is in service, more it needs revision. The revision becomes necessary due to changing requirements, because the systems have to be moved to another machine, or perhaps an adjustment will have to be made to support a new environment. As with the testing phase, these modifications are not possible without proper understanding of the system. The point is made that in a system's life cycle, the understandability of the program is crucial.

Numerous studies have shown that the most time consuming and costly phases of the software life cycle are testing and maintenance. This is due to the qualities of programs being written. Even if the programs meet the requirements,
they may be difficult to understand by the people assigned to perform testing and maintenance or even by the same person(s) who originally wrote the programs. If people can produce easily understandable programs, this data processing nightmare could soon be facilitated.

Despite the fact that many computer science teachers have been trying to find ways to teach students to write clear code and to teach them how to understand programs, these educators still lack a good measure of program understandability. The methods available seem to have more disadvantages than advantages. Once they find the reliable and accurate measure, 'better' programs could be pointed out as well as 'bad' ones. The measure helps educators and instructors to show their students a way to produce clear and easily understandable codes.

1.2 Introduction to Problem

Researchers have attempted to determine the programmer quality and program understandability. Thus far the most widely accepted measure of program understanding is the comprehension quiz, sometimes called a 'question-answer' quiz. In the comprehension quiz, the examiner asks multiple-choice or short-answer questions concerning a program. The biggest drawback of the comprehension quiz is the skill and effort required to produce a good quiz which tests both low- and high-level understanding. Furthermore
the subjectivity of the grader could influence the scores.

Less accepted techniques to measure program understanding include the time taken to locate a bug in a program, the time taken to make modification to a program, the accuracy of reproducing functionally equivalent program, and the subjective report. These techniques all have weaknesses (e.g. many unintended factors can easily influence the outcome) and lack reliability.

A recently noticed method of measuring program understandability is 'cloze' score. Some researchers have demonstrated that 'cloze' score closely correlates to the score of the comprehension quiz [Cook 1984, Hall 1986, Ullrich 1986, Herrington 1987]. The cloze score is said to be superior to all the measures listed above because of its ease of creating and grading and because its reliability. In using the cloze procedure, first, the subject is given a program with blanks, and asked to fill in those blanks with correct words or symbols. The number of correct answers, the cloze score, indicates how well the program is understood by the subject.

1.3 Proposed Research

Objectives

The objective of the proposed project was to develop and test an environment for experimentation and testing using the cloze procedure. The results of the proposed project
was used to prove an environment for teasing subjects' ability to understand programs in the languages Modula-2 and Pascal. The final phase of the project was to conduct a small demonstration experiment using the cloze procedure. This experiment was performed by taking two graduate students in computer science and testing them on about ten Pascal programs of varying length and difficulty. The hypotheses to be tested were (1) do students make less errors in easier programs compared to more difficult ones, (2) do students make less errors in shorter programs as compared to longer ones, and (3) do students perform better (have fewer errors) as they are tested on more programs.

Methodology

The language Modula-2 was used to develop the set of programs to accomplish the proposed task. This project was written using the Logitech Modula-2 compiler version 3.0. Formal software development methods such as data flow diagrams and structured design were used. A finite state automaton was used to produce a lexical scanner procedure; this scanner takes raw Modula-2 and Pascal programs and converts them into a form suitable for input to a testing program. The testing program presents the program with missing blanks on the screen, fills in the blanks as typed by the subject, and records the responses. The testing program is able to run on an IBM AT and its compatible.
The proposed cloze program works on two different screen types:

1) IBM Monochrome,
2) IBM EGA,

The cloze program utilized a Logitech Mouse in addition to the keyboard for the IBM personal computer.

Description of the proposed project

The proposed project consists of five major programs.

The first program -- *PreCloze.mod*, reads in a semantically correct program written in Modula-2 or Pascal and identifies those tokens which to be deleted from the text of the program during testing. The source program should have named <name>.pas or <name>.mod and output file was named <name>.clz. The selected tokens are marked by "@" in front and ")" at the end of the token. The tokens are selected by using either of the following strategies chosen by the experimenter:

1) Selection of every nth token from within a selected region,

The experimenter is able to declare as to whether reserved words and/or punctuation characters are candidates for removal. This PreCloze program does not remove or change any comments and does not remove any white space.
For example, using the program in figure 1 and the first strategy, every fifth token is selected to be extracted. The tokens selected are: "number", ":", ":="", "100", ":>"", "sum"", ":;"", and "1". Figure 2 contains the output of the proposed procedure using figure 1 as input.

program total;
{program to find the sum of 1 to 100}
var
    number : integer;{counter from 100 to 0}
    sum : integer;{the running sum}
begin
    sum := 0;
    number := 100;
    while ( number > 0 ) do
    begin
        sum := sum + number;
        number := number - 1
    end; {of the while loop}
end.

Figure 1. Source input program (total.pas).

program total;
{program to find the sum of 1 to 100}
var
    @(number) : integer;{counter from 100 to 0}
    sum @(=) integer;{the running sum}
begin
    sum @(=) 0;
    number := @(100);
    while ( number @(>) 0 ) do
    begin
        @(sum) := sum + number@(=);
        number := number - @(1)
    end; { of the while loop }
end.

Figure 2. Output of PreCloze using the program in figure 1 as input and selecting every fifth tokens.
The second program of the project -- Cloze.mod, reads the file <name>.clz prepared by PreCloze.mod and displays the cloze procedure on the screen. A subject uses the cursor control keys to control the cursor and screen and fills answers in the blanks. The screen is updated as the subject made responses. The program produces two files from the responses made by the subject -- the response file and playback file.

The response file contains the following:

1) information on the subject -- name, group number, class number,
2) information on the experiment -- name of the program this experiment was conducted (cloze.mod by Kiyoka Takahashi), program name, time and date,
3) experimental data -- sequence of actual responses, time taken for each response since start of the experiment, correctness of the response, where it was inserted or deleted, if inserted, the correct token,
4) statistical information -- elapsed time, number and percentage of the correct response(s), number and percentage of the incorrect response(s).

The playback file contains the following:

1) information on the subject -- name, group number, class number,
2) Actual key strokes or mouse commands given by the
subject.

The file names are coded to reflect the subject's name, his or her class and group number -- the first letter indicates a program name, next digit is a group number, next 3 digits are class number and last 4 letters are the first 4 letters of his or her last name. The response file has ".RSP", playback file has ".PB" extensions.

For example, if program name is Total.pas, subject's name is Pat Rogers, class number is CS 131, and group number is 2, then response file has "T2131ROGE.RSP" and playback file has "T2131ROGE.PB" as their file names.

This program may be called from a command file to run as many <name>.clz file as the experimenter desires. One suggested experiment is to first run an interview session to collect demographic information and an instructional session before running <name>.clz files prepared by PreCloze.mod. The sequence of the suggested experiment is shown in Figure 3.

```
Interview
Instructions
Test(example)
Test(program1)
Test(program2)
...
```

Figure 3. Sequence of the suggested experiment.
The third program — Playback.mod, uses playback file generated by Cloze.mod and <name>.clz generated by PreCloze.mod. The function of this program is to playback the testing session of the experimental session. The experimenter is able to playback step by step to observe what actually was displayed on the subjects screen.

The fourth program — Paper.mod, uses the response file generated by Cloze.mod and <name>.clz, and produce a paper and pencil version of the cloze procedure (see Figure 4 for an example of a paper and pencil version of the cloze procedure). The program produces a print out of the program with underlines replacing the missing tokens, and original program. The programs with underlines is to be used to conduct cloze test and the original program contains key answers. This could be used in a classroom situation.

The fifth program — Analysis.mod, reads response file and prepares data sets for a statistical analysis for the following information:

1) delta-times,
2) order statistics -- nth insertion for slot number m.
program total;
{ program to find the sum of the integers }
{ 1, 2, ..., 100 }
var
    _________ : integer;  { counter from 100 to 0 }
    sum _________ integer;  { the running sum }
begin
    sum _________ 0;
    number := _________;
    while ( number _________ 0 ) do
        begin
            _________ := sum + number _________
            number := number - _________
        end;  { of the while loop }
end.

Figure 4. Example of output of Paper.mod, total.q. This is produced using total.clz of figure 2.
Chapter Two

Related Work and Literature

2.1 The Original Cloze Procedure

W. L. Taylor introduced cloze technique in his 1953 paper titled "Cloze Procedure." Ever since, the technique has been studied and experiments have been conducted by linguists all over the world and it is accepted as a reliable tool to measure readability of natural language text. The cloze scores are found to be reliable in estimating reading comprehension, estimating language proficiency and teaching effectiveness. Furthermore the procedure is said to be superior in validity to any of the other such methods that have been proposed and studied carefully. Also the results of the experiments done on bilingual students and foreign students learning English have showed that the cloze scores are highly correlated to the results of the standard comprehension tests. The process of conducting this procedure is a simple one. First, a sample of prose is chosen. Then some words or letters of the text are selected and replaced by blanks. The text with blanks are given to a student and the student will guess the correct answer and fill in the blanks. See Figure 5 for examples of cloze experiment applied to English text.
Figure 5. Examples of cloze experiment [Oller 1979]

For the first example, the underscores indicate deleted letters, the answers are: two, three, four and five. As can be seen in second example, the selected letters in the text could be replaced by something other than blanks. In this case letter 'x' is used to indicate unknown letters. The mutilated words in the second example are sighted near the edge of town and shot by a local farmer. The entire words are deleted from the last example and replaced by underlines. The missing words are who, retardation, to and in.

After a prose is chosen, the experimenter must select the words which will be deleted. The selection word could be done by following one of three method listed below:

(1) The fixed ratio method involves deletion of every nth (usually 5 to 10) word(s) of the text. So if n equal to 5 and 10 words are deleted, then the text contains roughly 50 words.

(2) The variable ratio method involves deletion of words decided by a rational selection procedure. For instance, delete only content words, or delete only
function words.

(3) The fixed ratio method with some constraints. For example, select every nth function word(s), select every nth word(s) but skip determiners (e.g. a, an, the).

For most purposes, the delation of every nth words (1) shows somewhat greater reliability and validity than deliberate selection of words (2) or combination of both methods (3).

The cloze score, the output of cloze procedure, is the number of blanks filled correctly. If the cloze test was conducted using a text with 50 blanks and a student made 10 correct responses and 40 incorrect responses, the student’s score for this cloze test is 10. The number of correct responses are occasionally divided by the number of the blanks to get its percentage. Thus, 10 correct answers out of 50 blanks yields the score of 20%. Scoring could be done in two ways; score as correct only the blanks filled with ‘exact’ words which has been deleted (cloze exact), or count blanks filled with words which are grammatically and contextually acceptable (cloze acceptable).

The name ‘cloze’, which has been mistaken for ‘close’, was invented by Taylor who found similarity between the technique he newly created and the process of closure celebrated by Gestalt psychologists. Consider the examples
shown in figure 6 following visual closure, the motion of completing imperfect visual patterns diagram:

*   *   
    
  .   .   .
  
  *   *   

Figure 6. Examples of visual closure [Oller 1979]

Lines could be drawn to complete a square and letter 'A' from above stars and dots. Compare this example with previous cloze examples, completing the text by filling in blanks is a very similar activity as completing imperfect visual patterns. Hence, Taylor used 'cloze', which is pronounced same as 'close' in 'close the window', to represent the activities of filling in blanks to complete the text.

Although the cloze procedure is found to be valid on most occasions, scores could be influenced by other factors other than the reader's language ability. One such factor could be familiarity of the subject which the text is written on. That is, students majoring in anthropology score higher on a text on anthropology than students majoring in music. This is so because completing the text is impossible without understanding the meaning of the text.
2.2 Cloze Procedure and Program Understanding

Shneiderman first proposed the use of the cloze score as a measure of program comprehension [Shneiderman 1980], but he did not conduct experiments. Today the cloze is found to be the acceptable measure of the software understandability. Cook, Bregar and Foote claim that the cloze procedure is better than any other software understandability testing technique [Cook 1984]. Researchers agree that the advantages of the cloze procedure over other measures (see 1.2) are: the cloze procedure (1) is easy to construct, (2) is easy to score, (3) is easy to interpret, (4) can be automated. However, since the technique is relatively new to the software engineering community, more intense research is needed before the community agrees on the proper use of the software cloze test.

An initial investigation was made by Cook, Bregar and Foote [Cook 1984]. In that experiment, subjects were divided into two groups. Each member of one group was given a Pascal program listing and a comprehension quiz over the program (the comprehension quiz used was carefully constructed to test both low- and high-level program comprehension). Another group was given descriptions of the a cloze procedure and cloze version of the same Pascal programs. These authors discovered that very few errors were made in filling blanks when the tokens were reserved
words, parentheses or brackets. These were called them "giveaways", and excluded when counting the cloze score to make the comparisons, so the cloze score represents the semantic understanding not the syntactic understanding. After this careful study of the data, they found a close correlation between the cloze score and the comprehension quiz (the comprehension quiz is the most widely accepted measure of program understanding (see 1.2)).

An interesting experiment was conducted by Ullrich [Ullrich 1986]. The cloze test, in this case, was given to the subjects along with deleted tokens in alphabetical order and the cloze procedure was implemented on a computer. The experimenter found this particular version of the cloze test too easy to be an effective tool. He also pointed out the possibility of studying the reaction time of the subjects to each of the blanks.

Further research by Hall and Zweben [Hall 1986] using larger programs conflicted with Cook's result that the cloze score and comprehension quiz score are closely correlated. They demonstrated that the cloze score was not always in accord with the comprehension quiz as Cook, Bregar, and Foote first thought. Hall and Zweben demonstrated that the validity of the cloze score is dependent on the nature of the deleted items in a particular cloze test. As Cook, Bregar, and Foote had done in their experiment, Hall and Zweben also found that some
items were more often answered incorrectly than others. They went on to categorize the items being deleted into (1) program-independent items which were more often answered correctly and (2) program-dependent items which were more often answered incorrectly. The program-independent items are clarified as tokens which can be reproduced without the understanding of the program ('items that could be completed using rules, knowledge, or heuristic that were independent of the semantics of the corresponding programs ...' [Hall 1986]). The understanding of the particular program is necessary to fill in the program-dependent items correctly. The ratio of those two types of deletions to one another had a direct and predictable effect on the outcome of the test.

Thomas and Zweben [Thomas 1986] stated if two programs were to be tested to determine the relative difficulty to each other, the proportion of program-dependent items to the entire program must be similar. Unfortunately classification of program-dependent and program-independent items requires further research. Thomas and Zweben also pointed out that kinds of items being deleted (program-independent or program-independent) can have a significant effect on the results of a cloze test used for measuring software comprehensibility. They agree with Hall that the program-independent has a little or no use to the measure of program comprehension; therefore only program-
dependent items should be removed to improve the validity of cloze score.

Herrington and Zweben [Herrington 1987] conducted an experiment to investigate methods of scoring cloze tests. They compared two methods of scoring: verbatim and synonymic. Since those methods were first used by linguists (they are called 'cloze-exact' and 'cloze-acceptable' by linguists), and both methods are used when cloze score in prose is discussed. She concluded that '...due to the relative infrequency of deletions for which synonyms are possible in software cloze tests, and due to the apparent objectivity with which synonymic scoring can normally be done in the software domain, that synonymic scoring is the method of [choice for] scoring software cloze tests.' [Herrington 1987]. In the same experiment, the subjects were found to use low-level reasoning to complete cloze items first.
3.1 Requirements and Specification

The requirements and specifications were developed using the techniques for structured analysis [DeMarco 1978]. The requirements and specifications for this project are presented in Appendix A and briefly described below.

The data dictionary (Figure 7 of the requirements and specifications) documents each data flow in the data flow diagram. Conventions used in the data dictionary are:

- $=$ left hand side of '=' could be decomposed to the right hand side of '=

- $a + b$ there are one data item 'a' and one data item 'b'

- $[a | b]$ there is either one data item 'a' or one data item 'b'

- $\{a\}$ there are one or more of data item 'a'

- $(a)$ there is one data item 'a' or no data,

*comment* anything between * and * is a comment.

The context diagram (Figure 1 of the requirements and specifications) shows the domain of the study. The bubble labeled "Cloze experiment" in the middle of the diagram represent a context of the system. Whereas the boxes, which lay outside the bubble, are the net originator and receiver.
of data (for example, 'Program' is a net originator, 'Person being tested' and 'Experimenter' are net originators and receivers of data). The named vectors are net inputs and outputs of the domain. Finally, the context diagram is a "departitioned" version of the level 0 diagram (Figure 2 of the requirements and specifications).

The data flow diagrams (Figure 2 to 6 of the requirements and specifications) analyze and decompose the domain of the study. They show logical flow of data in the system rather than flow of control. Conventions used in the data flow diagrams are:

A **bubble** represents transformations of data,

A **named vector** represents data flow,

A pair of **parallel lines** enclosing a name represent files,

Diagrams have **multi levels** to improve their readability. By further describing each bubble with another diagram, the number of bubbles on one page can be kept to a minimum. When a bubble is further described, the bubble is called a parent bubble, and the diagram which describe the bubble is called a child diagram. Every bubble with no child diagram has **mini spec** which verbally explains the bubble. The level of diagram can be identified by the number of decimal point(s) in the bubbles which make up the diagram, and the diagram's parent bubble or a
bubble's child diagram can be identified by the number in the bubble(s).

A brief description of Figure 3 is as follows:

This diagram is a child diagram of bubble 1 (Figure 2). No bubble in this diagram has a child diagram. A user inputs selecting_mode, which according to data dictionary is composed of either Manual or Auto, then bubble 1.1 determines the select mode using the information (selecting_mode) received from the user. If the mode is Manual then do bubble 1.3 which reads program_name, which is used to get correct_program, and select_tokens from the user and produces program_with_selected_tokens. If the mode is Auto then do bubble 1.2 which reads program_name, which is used to get correct_program, and select_info from the user and produces program_with_selected_token. All of the data in the data flow diagrams are described in data dictionary.

3.2 Design

The structured design discussed in "Structured Design" [Yourdon 1979] was used to develop the design document (see Appendix B for the design document). The "structured design is the process of deciding which components interconnected in which way will solve some well-specified problem" [Yourdon 1979]. In other words, structured design is the collection of methods which enable the designers to
develop a "good" system design. First, structure charts are drawn, and then a description of each box is written. Structure charts are the tools discussed by Yourdon and are used widely in this design document.

The structured charts document the system by identifying modules and the interconnections between modules. Then the descriptions for modules are written in combination of English and Pascal-like code. Conventions used in structured charts are:

- **The named boxes** represent modules,
- **The unnamed vector** form box A to box B indicates module A calls module B,
- **The named circle with arrow** indicate a data item passed from one box to another. The data is passed in the direction of the arrow.

3.3 Implementation

The structured programming was used during entire programming. Structured programming is "a set of guidelines and techniques for writing programs as a nested set of single-entry, single-exit blocks of code, using a restricted number of constructs" [Yourdon 1979]. The system is developed under Logitech Modula-2 version 3.0 compiler and its tools.
Chapter Four
Cloze Experiment

4.1 Introduction

A demonstration cloze experiment is conducted as a part of the testing phase of the project. The intention of this demonstration experiment is to test the project in an environment which is close to what will be used by experimenters in the future. Another reason is to gain the knowledge of the cloze procedure.

4.2 Method of Experiment
Subjects

The subjects were two Computer Science graduate students at the University of Montana. They both had three and half years of programming experience and had completed required undergraduate courses at the University. Although one is the first year graduate student and other is the fourth year graduate student, both subjects are assumed to have extensive experience in the programming in the Pascal language.
Table 1. Programs, their source and their description.

<table>
<thead>
<tr>
<th>Program</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPOLY</td>
<td>[Sedgewick 1984]</td>
<td>Polynomial addition</td>
</tr>
<tr>
<td>CONCOR</td>
<td>[Grogono 1983]</td>
<td>Counts occurrences of each word in a text.</td>
</tr>
<tr>
<td>FIXEDLEN</td>
<td>[Schneider 1982]</td>
<td>Reads a file of variable length (1-80) of characters and produce a file with the same text with the fixed length (80).</td>
</tr>
<tr>
<td>JOSEPHUS</td>
<td>[Garland 1987]</td>
<td>Consider N people arranged in a circle. Proceeding clockwise, every Mth person leaves until the only one remains.</td>
</tr>
<tr>
<td>POS</td>
<td>[Garland 1986]</td>
<td>Given string s and s', find the first occurrence of s' as a substring of s.</td>
</tr>
</tbody>
</table>

Program materials

Six programs were selected for stimulus materials. The programs and their sources and descriptions are listed in Table 1, and programs and tokens selected are listed on appendix B. Twenty five tokens were deleted from each of the programs except from two programs. Deleted tokens were every fifth tokens, and those included punctuations and reserved words. Two programs, a shell sort and a concordance (to count the occurrences of words within a text file) were the same program used by Ullrich [Ullrich,
In those programs, tokens were the same as chosen by the original experimenter selected (21 tokens for shell sort and 32 for concordance program). All other programs were taken verbatim from the computer science text books and the comments were left as they were found in the text. Variable names, and programming styles followed the original program as much as possible, blank lines were added to the program wherever seemed appropriate, and the indentation of the program was not exactly the same as the original program. Short descriptions of the programs were added to the programs whose original comments are not sufficient to describe what the programs do. The author considers that the comments, indentation, and descriptive variable names to be a part of the program understanding and testing subjects without them would be unfair. Before the programs were used in the experiment all were tested for correctness. Tokens were considered to be the lexical tokens (tokens the lexical analyzer recognizes). Difficulty and length of the programs were defined to be as follows. The length is the number of total tokens in the program excluding comments and blank lines. A ranking of the difficulty was constructed based on the number of boolean expressions, which represent relational or logical expressions (see Table 2). If more than one program had same number of boolean expressions, program with more tokens was considered to be more difficult than the shorter
ones. This situation for three programs is given in Table 2.

<table>
<thead>
<tr>
<th>Program</th>
<th>Length</th>
<th># of and number of boolean expression for each program and the ranking of difficulty.</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td># of booleans</td>
<td>ranking of difficulty</td>
</tr>
<tr>
<td>ADDPOLY.PAS</td>
<td>274</td>
<td>4</td>
</tr>
<tr>
<td>CONCOR.PAS</td>
<td>368</td>
<td>16</td>
</tr>
<tr>
<td>POS.PAS</td>
<td>207</td>
<td>6</td>
</tr>
<tr>
<td>FIXEDLEN.PAS</td>
<td>230</td>
<td>4</td>
</tr>
<tr>
<td>JOSEPHUS.PAS</td>
<td>194</td>
<td>3</td>
</tr>
<tr>
<td>SHELL.PAS</td>
<td>212</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2

Apparatus

The demonstration cloze experiment was presented to the subject on either an IBM AT personal computer or Zenith Z-100 PC series (IBM PC compatible). The screen of each of the monitors was divided into two parts for presentation of the program with the missing tokens (program window), and for insertion of the tokens (insert window). The task of subject was to move a cursor to blanks (in program window) and to type in answers in the insert window. In a paper and pencil version of this task, this would correspond to the subject writing in the answer into a blank. Refer to 'requirements and specifications' document for examples of the program and insert window.
The program window occupied most of the screen. It was 78 characters wide by 19 lines high. The program with missing tokens was presented in this window, and the missing tokens were replaced by 10 underline characters. Only 19 lines of the program was displayed at any time. The subject was required to press screen control keys ('Home', 'End', 'PgUp', 'PgDn', 'up arrow' or 'down arrow') to move the screen and to press cursor control keys ('right arrow', and 'left arrow') to move the cursor.

The insert window was located at the bottom of the screen. It was 80 characters wide and 2 lines high. To start insertion of the token, an insert key had to be pressed. The characters were then typed in and the token insertion was completed when the subject pressed one of the cursor control keys, screen control keys, insert, delete keys, or enter key.

After the insertion was made, the blank (underline characters) in the program window where the cursor was at was replaced with the string. The cursor was displayed in blinking reverse video. The blank which is replaced by typed token is displayed in reverse video, where the blank which has not been filled is displayed in normal video.

Since both of the students were subjects of Ullrich's experiment [Ullrich 1986], they were assumed to have some knowledge of the cloze experiment and were only given oral and written instruction of the procedure (not given
instruction session in English). A simple Pascal program to solve the greatest common divisor problem (see Appendix B for the program) was given to the subjects to gain familiarity with the apparatus and the cloze procedure. This program was a short version of the programs to be tested later on the experiment. The experimenter was available to answer occasional questions during the entire experiment.

Every valid keystroke and the time the keystroke was made since the beginning of the session was recorded, which was not possible with paper and pencil version of the experiment. These information will enable the entire session to be played back later.

4.3 Results

Order of the test used for the experiment is listed below (see appendix B for the list of programs with selected tokens marked).

1) concor.pas
2) shell.pas
3) fixedlen.pas
4) josephus.pas
5) addpoly.pas
6) pos.pas
7) concor.pas'
8) shell.pas'
Concor.pas and shell.pas were given to the subjects twice, once at the beginning and then toward the end (they are marked concor.pas' and shell.pas') of the experiment. The improvement of the cloze score over time on the same programs was observed. Raw data produced by the experiment is listed on Table 3 and 4. The correctness of the final response (last correct) was used for analysis.

The percentages of the last correct response made by each subject were graphed by the order tested, by the order of difficulty, by the number of the total tokens in the program, and by the number of tokens deleted from the program (Figure 7 to 10).
<table>
<thead>
<tr>
<th></th>
<th>correct/wrong</th>
<th>% correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>concor.pas</td>
<td>29/3</td>
<td>90.62</td>
</tr>
<tr>
<td>shell.pas</td>
<td>16/5</td>
<td>76.19</td>
</tr>
<tr>
<td>fixedlen.pas</td>
<td>25/0</td>
<td>100</td>
</tr>
<tr>
<td>josephus.pas</td>
<td>25/0</td>
<td>100</td>
</tr>
<tr>
<td>addpoly.pas</td>
<td>24/1</td>
<td>96.0</td>
</tr>
<tr>
<td>pos.pas</td>
<td>24/1</td>
<td>96.0</td>
</tr>
<tr>
<td>concor.pas</td>
<td>29/3</td>
<td>90.62</td>
</tr>
<tr>
<td>shell.pas'</td>
<td>18/3</td>
<td>85.71</td>
</tr>
</tbody>
</table>

Table 3  Result of the experiment for subject A shown in the order tested.

<table>
<thead>
<tr>
<th></th>
<th>correct/wrong</th>
<th>% correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>concor.pas</td>
<td>30/2</td>
<td>93.75</td>
</tr>
<tr>
<td>shell.pas</td>
<td>19/2</td>
<td>90.48</td>
</tr>
<tr>
<td>fixedlen.pas</td>
<td>24/1</td>
<td>96.0</td>
</tr>
<tr>
<td>josephus.pas</td>
<td>23/2</td>
<td>92.0</td>
</tr>
<tr>
<td>addpoly.pas</td>
<td>24/1</td>
<td>96.0</td>
</tr>
<tr>
<td>pos.pas</td>
<td>25/0</td>
<td>100.0</td>
</tr>
<tr>
<td>concor.pas'</td>
<td>30/2</td>
<td>93.75</td>
</tr>
<tr>
<td>shell.pas'</td>
<td>19/2</td>
<td>90.48</td>
</tr>
</tbody>
</table>

Table 4  Result of the experiment for subject B shown in the order tested.
4.4 Discussion

The first hypotheses was do students make less errors in easier programs as compared to more difficult ones. This hypotheses was tested by comparing the percentage of errors on the two easiest programs (josephus and fixedlen) to the two hardest programs (concor and pos). Subject A scored 100% on the two easiest and 90.62% and 96% on the two hardest programs respectively, Subject B scored 96% and 92% on the two easiest and 93.75% and 100% on the two hardest programs (see Figure 10). The data from the shell program, rated the second easiest, was ignored in this analysis because subjects' verbal responses indicated that they found it very difficult. Ignoring this data from this program, the data are at best weakly supportive of hypothesis that there were fewer errors in the easy programs.

The program which attracts the most attention is the shell program which is not considered in the process to get the above result. Although the shell program was rated as the second easiest, the subjects' performance on the program was poorer. In addition, the subjects' verbal responses indicated that the shell program was the most difficult. Their verbal responses and their poor performance suggest that the methods used by the author to determine the difficulty of the program were not accurate.
The second hypotheses was do students make less errors in shorter programs as compared to longer ones. Subject A scored 100% and 96% on the two shortest (josephus and pos), and 90.62% and 96% on the two longest (concor and addpoly). Subject B scored 92% and 100% on the two shortest and 93.75% and 96% on the two longest programs (see Figure 9). From the data, the hypothesis is not supported.

The third hypotheses was to test the effect of practice. To test this hypothesis, the shell and concordance programs were presented twice. The percentage correct of the shell program were 76.19% and 85.71% for subject A, 90.48% and 90.48% for subject B the first and second time, respectively. The data for the concordance program were 90.62% and for subject A and 93.75% for subject B both times they were tested (see Figure 7). Comparison of these numbers indicates that performance did improve slightly with practice on one program and not the other.

These results are, at best, only suggestive because of the small number of subjects. More data is needed to conclusively test the questions posted by the hypotheses.
Figure 7 Graphed in the order tested

Figure 8 Graphed in the number of selected tetons
Figure 9 Graphed in the total number of tokens

Figure 10 Graphed in the order of difficulty
APPENDIX A.

Requirements and Specifications of the Cloze System
### Table of Pseudo keys

<table>
<thead>
<tr>
<th>Keys for IBM AT and vt100 terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ERASE_LETTER</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>DOWN</strong></td>
</tr>
<tr>
<td><strong>END</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>HOME</strong></td>
</tr>
<tr>
<td><strong>INSERT</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>LEFT</strong></td>
</tr>
<tr>
<td><strong>PGDN</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>PGUP</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>RIGHT</strong></td>
</tr>
<tr>
<td><strong>UP</strong></td>
</tr>
</tbody>
</table>

Table 1
Mini_spec 1.1 Determine selecting mode
Get a mode of selecting tokens.

Step 1) Ask how to select tokens -- manual select or auto select? Default is auto select.

Auto select
-- give a number n, starting point, ending point, whether to select reserved_word and punctuation or not, then the program will select every nth token beginning at the starting point and finishing at the ending point. If reserved_word were not to be selected then the program will skip reserved words as well as punctuation.

Manual select
-- tokens are selected by the user. The user views the program on screen and moves cursor around and selects tokens.

Mini_spec 1.2 Auto select
If auto select mode is chosen in 1.1 then program selects tokens automatically.

Step 1) Get a name for correct_program. 
(See example 1 for an example of correct_program.)
Step 2) Ask user the following
include or skip reserved_word as tokens
include or skip punctuation as tokens
starting point of the selection
ending point of the selection
integer n which is in the range from 1 to 10.
Step 3) Do lexical analysis and decompose correct_program into an array of tokens.
Step 4) Select every n_th token
** select reserved_word and punctuation only if they are candidates for removal.
Step 5) Put "@(" in front and ")" at the end of the selected tokens and save program with ".clz" extension.
(See example 2 for an example of program_with_selected_tokens.)
Example 1
Example of correct_program: total.pas
This is a valid Pascal program. A valid Modula-2 program could be a correct_program. At least one blank line needed after the program.

** line# are not part of the file.

```
line#       | program total;
1           | { program to find the sum of integers }
2           | { 1,2 ..., 100}
3           | var
4           | number : integer; {counter from 100 to 0}
5           | sum : integer; {the running sum}
6           | begin
7           | sum := 0;
8           | number:= 100;
9           | while ( number > 0 ) do
10          | begin
11          | sum := sum + number;
12          | number := number - 1
13          | end; { of the while loop }
14          | end.
15
16
```
Example 2
Example of program with selected tokens: total.pas
This file is produced by bubble 1.2 (Auto Select) using total.pas (correct program) of example 1.

Every fifth token is selected to be extracted from total.pas. They are: (1) "number", (2) ":", (3) ":="", (4) "100", (5) ">", (6) "sum", (7) ";", and (8) "1".

Program total;
{ program to find the number of integers }
{ 1, 2, ..., 100 }
var
  number : integer; { counter from 100 to 0 }
  sum : integer; { the running sum }
begin
  sum := 0;
  number := 100
  while ( number > 0 ) do begin
    sum := sum + number
    number := number - 1
  end; { of the while loop }
end.

8 tokens
16 lines
Mini_spec 1.3 Manual select
If manual select is chosen in 1.1 then do the following.

Step 1) Get a name for correct_program.
(See example 1 for an example of correct_program.)
Step 2) Do lexical analysis and decompose correct_program into an array of tokens.
Step 3) Display a correct_program on the screen.
Step 4) Let the user select tokens

Repeat the following
User can use UP and DOWN keys (See Table 1 for the pseudo keys) to move a cursor up a line or down a line, PGUP, PGDN moves a cursor to a previous and next page of program, HOME display first page of program and puts a cursor at the very first token, and END displays the last page of program and puts a cursor at the very last token. Also user can use RIGHT and LEFT keys to move to right next or left next token. Press 'enter' key to select a nonselected token or deselect a already selected token. Selected tokens are displayed in reverse video and cursor is displayed as an underline.

Until user types 'E' or 'e'.
Step 5) Put "@(" in front and ")" at the end of the selected tokens and save the program on the disk with ".clz" extension.
Mini_spec 2.1 Get data

Repeat step 1 to step 5
Step 1) Get user_input
  case key pressed

INSERT -- get an insert response
  1) Transfer control to an insert window. Insert window is located at the bottom of program window. (See example 3 for an example of screen.)

  2) Repeat the following
  Record any letter or symbol typed except tailing space in the insert window. These letters and symbols will be a typed_token. The string of ERASE_LETTER can be used to erase a symbol or a letter in the insert window. Until any cursor_movement or 'enter' key is pressed.
  3) Get time token insertion ended (time_response_made).
  4) Get correctness (C for correct, W for wrong).
  5) Let response_data = I (for insert) + correctness + time_response_made + current line# + token# + typed_token + correct_token.
  6) Return control to a program window.
  7) Update screen.

DELETE -- get a delete response
  1) Get the time DELETE key is pressed (time_response_make).
  2) Let response_data = D (for delete) + current line# + token#.
  3) Update screen.

ONE OF CURSOR_movement -- get a cursor movement response
  1) Update screen.

'E' FOR END -- get an end response
  1) Get time the E key is pressed (time_response_made).
  2) Let response_data = E(for end) + time_response_made + 0(line#) + 0(token#)
  3) Update screen.

end case

Until the user press "E" or "e" to end
Example 3
Example of screen:
Since total.pas is too short to show screen handling, a longer program is chosen for an example.

The program shown below is sumlines.clz produced by bubble 1.2 (Auto Select) using sumlines.mod (not shown). Every fifth tokens are selected. Punctuation are not candidate for removal.

MODULE SumLines;

(* Sum each line in the input file *)
(* from p 88 of Gleaves book *)

FROM InOut @(IMPORT)
  OpenInput, CloseInput,
  EOL, termCH, @(Done), WriteLn,
  ReadInt, WriteInt, WriteString;

@(VAR) i, sum: INTEGER;
BEGIN
  @(OpenInput) ("TEXT");
  IF NOT DONE @(THEN)
    WriteString("File not opened");
    HALT;
  END;
  @(sum) := 0;
  ReadInt @(i);
  WHILE Done DO
    @(INC) (sum, i);
    IF termCH = @(EOL) THEN
      WriteInt(sum, 7);
      @(WriteLn);
      sum := 0;
      END;
      ReadInt(@(i));
    END;

    CloseInput;
END SumLines.

%%
10 tokens
33 lines
The screen before any key is pressed. The program window displays 19 lines of program and is surrounded by a rectangle. A underline below the rectangle is an insert window. The program window displays first 19 lines of program and cursor is at the first blank (first missing token) which is located at line 6 (see previous page for line #). Cursor will be displayed in reverse video.

MODULE SumLines;

(* Sum each line in the input file *)
(* from p 88 of Gleaves book *)

FROM InOut □□□□□□□□□□□□□□□
  Openlnput, CloseInput,
  EOL, termCH, __________, WriteLn,
  ReadInt, WriteInt, WriteString;

__________ i, sum: INTEGER;

BEGIN _______ ("TEXT");
  IF NOT DONE __________
    WriteString("File not opened");
  HALT;
END;
__________ := 0;
While the user is inserting a token "IMPORT" (i.e. after pressing INS and typed "IMPORT").

```
MODULE SumLines;
(* Sum each line in the input file *)
(* from p 88 of Gleaves book *)

FROM InOut Openlnput, CloseInput,
    EOL, termCH, WriteLn,
    ReadInt, WriteInt, WriteString;

_________ i, sum: INTEGER;

BEGIN
    ("TEXT");
    IF NOT DONE __________
        WriteString("File not opened");
        HALT;
        END;
    __________ := 0;

IMPORT
```
(3) After user press one of RIGHT, LEFT, UP, DOWN, PGDN, PGUP, HOME, END, or Enter or click of mouse which terminates insert.

Line 6 is redrawn with "IMPORT" replacing the first blank. Since cursor is still at the first token, "IMPORT" is shown in flashing reverse video.

MODULE SumLines;

(* Sum each line in the input file *)
(* from p 88 of Gleaves book *)

FROM InOut IMPORT
   OpenInput, CloseInput,
   EOL, termCH, ________, WriteLn,
   ReadInt, WriteInt, WriteString;

_________ i, sum: INTEGER;
BEGIN
   _________ ("TEXT");
   IF NOT DONE _________
      WriteString("File not opened");
      HALT;
      END;
   _________ := 0;
(4) After pressing END.

Last 19 lines of program is displayed. Cursor is still at line 6.

** Cursor never moves unless RIGHT, LEFT, or mouse command is given.

```plaintext
WriteString("File not opened");
HALT;
END;

:= 0;
ReadInt (i);
WHILE Done DO
   _________(sum, i);
   IF termCH = _________ THEN
      WriteInt(sum, 7);
   _________;
   sum := 0;
   END;
   ReadInt(__________);
END;

CloseInput;
END SumLines.
```
(5) After pressing UP three times.

Cursor is still at line 6.

BEGIN
  String\("\text"\);
  IF NOT DONE
    WriteString("File not opened");
    HALT;
  END;

  \$ := 0;
  ReadInt (i);
  WHILE Done DO
    \$ := 0;
    \$ := \$ + i;
    IF termCH = \$ THEN
      WriteInt(sum, 7);
      \$ := 0;
    END;
    ReadInt (\$);
  END;
END;
(6) After PGUP or HOME (both result in the same screen)

"IMPORT" will be displayed in reverse video on screen.

MODULE SumLines;

(* Sum each line in the input file *)
(* from p 88 of Gleaves book *)

FROM InOut IMPORT
OpenInput, CloseInput,
EOL, termCH, __________, WriteLn,
ReadInt, WriteInt, WriteString;

__________ i, sum: INTEGER;

BEGIN
__________ ("TEXT");
IF NOT DONE __________
WriteString("File not opened");
HALT;
END;
__________ := 0;
(7) After DEL, "IMPORT" is deleted.

The first blank is displayed in reverse video.

```pascal
MODULE SumLines;

(* Sum each line in the input file *)
(* from p 88 of Gleaves book *)

FROM InOut □□□□□□□□□□□
Openlnput, Closelnput,
EOL, termCH, __________, WriteLn,
ReadInt, WriteInt, WriteString;

__________ i, sum: INTEGER;
BEGIN
    __________ ("TEXT");
    IF NOT DONE __________
    WriteString("File not opened");
    HALT;
END;
__________ := 0;
```
(8) After putting mouse in the third blank (line 11) and clicked.

Cursor is displayed in reverse video.

```
MODULE SumLines;
(* Sum each line in the input file *)
(* from p 88 of Gleaves book *)

FROM InOut _________
    OpenInput, CloseInput,
    EOL, termCH, _________, WriteLn,
    ReadInt, WriteInt, WriteString;

□□□□□□□□□□ i, sum: INTEGER;

BEGIN _________ ("TEXT");
    IF NOT DONE _________
    WriteString("File not opened");
    HALT;
    END;
    _________ := 0;
```
Mini_spec 2.2 Update screen

Step 1) Get old_screen and user_input.
Step 2) Update screen The program window can display up to 19 lines. (See the example 3 for an example of screen.)

case user_input of
    cursor_movement --
    case cursor_movement of
        up -- Display a line previous to the line currently at the top of screen, and replace every line on screen by its previous line. So the last line will disappear from the screen.
        down -- Display a line next to the line currently at the end of screen, and replace every line on screen by its next line. So the top line will disappear from screen.
        left -- Move the cursor to the left next blank to current cursor position.
        right-- Move the cursor to the right next blank to current cursor position.
        home -- Display first page of program without moving the cursor. i.e. display first 19 lines of program.
        end -- Display last page of program without moving the cursor. i.e. display last 19 lines of program.
        pgup -- Display last half of the previous page and first half of the current page without moving the cursor. i.e. display previous 11 lines followed by first 11 lines of the current page. If there are less than 11 lines between the current top line to the beginning of program, then display the first 19 lines.
        pgdn -- Display first half of the next page and last half of the current page without moving the cursor. i.e. display last 11 lines of the current page followed by next 11 lines. If there are less than 11 lines between the current last line to the end of program, then display the last 19 lines.
    end case cursor_movement
    I(insert) -- Insert a token at the cursor’s position and redraw the line.
    D(delete) -- Delete a token at the cursor’s
position and redraw the line.

E(end) -- Clear screen.

end case user input

(See example 3 for an example of screen handling.)

Mini_spec 2.3 Get statistical data

When user quits the process by typing 'E' or 'e' do the following:
Get number of correct first try and its percentage,
number of correct last try and its percentage, number of
incorrect first try and its percentage, number of
incorrect last try and its percentage.
(See example 4 for an example of response_file.)

Mini_spec 2.4 Produce file name

Step 1) Get user_data from the user.
Step 2) Get program_name from
program_with_selected_tokens.
Step 3) Produce a file_name whose first letter
indicates a program_name, second number is a
group#, next 3 are class#, and the last 4
letters are first 4 letters of subject's last
name in capital letters. If subject has a last
name which is shorter than 4 letters, use
blanks to fill the spaces.

Example:  Name: Pat Rogers
          Group#: 2
          Class#: cs131
          Program name: Total.pas

          file_name is T2131ROGE

Step 4) Save user_data on response_file.

Mini_spec 2.5 Display program

Step 1) Get program_name.
Step 2) Display the first page (19 lines) of
program_with_selected_tokens. Replace
selected_tokens with underlines, and place
cursor at the first underlined blank (first
selected_token). Screen should look like the
one in example 3.1. Control is at the program
window.
Example 4
Example of response_file: T2131ROGE.RSP
This file is produced by bubble 2.1, 2.3, 2.4 using total.clz (program_with_selected_tokens) of example 2.

Column are defined as follows:
1st column -- I(insert), D(delete), or E(end)
2nd column -- correctness -- C(correct) or W(wrong)
3rd column -- time since start of experiment
4th column -- line#
5th column -- token#
6th column -- typed_token (token typed by the subject)
7th column -- correct_token

ROGERS, PAT
131
2
7-3-1987 10:32am
TOTAL.CLZ
%%
IC 00:08.1 5 1 number number
IC 00:20.3 6 2 : :
IC 00:40.0 8 3 := :=
IC 00:49.5 12 7 ; ;
IC 01:02.0 12 6 sum sum
IW 01:10.0 13 8 2 1
D 01:14.0 13 8
IC 01:20.3 13 8 1 1
IW 02:00.0 9 4 99 100
IW 02:30.0 10 5 >= >
E 03:10.0
%%
First correct 5 62.5%
Last Correct 6 75%
First Wrong 3 37.5%
Last Wrong 2 25

Correctness of responses (NOT PART OF THE RESPONSE FILE)
If token was inserted only once, the response is consider to be the first try as well as the last try.

<table>
<thead>
<tr>
<th>token number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st try</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>W</td>
<td>W</td>
<td>C</td>
<td>C</td>
<td>W</td>
<td>5/8 62.5%</td>
</tr>
<tr>
<td>2nd try</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>W</td>
<td>W</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>6/8 75%</td>
</tr>
</tbody>
</table>
Example 5
Example of playback file: T2131ROGE.PB
This file is produced by bubble 2.1 (Decide Correctness) and 2.4 (Produce File Name) using total.clz (program_with_selected_tokens) of example 2. And this file will be used by bubble 5.2 (Simulate Step by Step).

DESCRIPTION
(not part of the file)

ROGERS, PAT
131
2
7-3-1987 10:32am
TOTAL.CLZ
%-%
I number
right
I: ; down
right
right
right
right
I: ; left
I sum
right
right
right
right
I 2
D
I 1
up
up
left
left
left
left
I 99
right
I >=
E
%-

<-- end of user data
<-- user inserted "number". Since cursor had not been moved, this insert was made at the first blank.

<-- user requested to move screen down a line (scroll down one line).
<-- cursor was moved to right next blank.

<-- cursor was moved to left next blank.

<-- whatever was in the current cursor position (in this case second blank) was deleted
<-- user request -- move screen up a line (scroll up one line).

<-- E for the END
<-- this marks the end, other information may be added below in the future
Mini_spec 3.1 Get delta_times

(See example 6 for an example of delta_times.)
Step 1) Get first inserts of each token.
Step 2) For each first insert do the following:
        get delta_times by subtracting time previous
        insertion or deletion is made from time the
        token is typed.
Step 3) Get average delta_times. This information will
        be part of statistical_analysis file.

Mini_spec 3.2 Get order statistics

(See example 6 for an example of order statistics.)
Calculate r which is derived by using the following
formula:

\[ r = \frac{nxy - \bar{x}\bar{y}}{\sqrt{\left( n\bar{x} - (\bar{x})^2 \right) \left( n\bar{y} - (\bar{y})^2 \right) } } \]

** -1 ≤ r ≤ 1
This information will be part of statistical_analysis file.
(See example 7 for an example of SPSSx file.)
Example 6
Example of statistical analysis: T2131ROGE.ANA
This file is produced by bubble 3.4 (Combine Data) using T2131ROGE.RSP of example 4.

NAME: Rogers, Pat
CLASS: cs131
GROUP: 2
DATE: 7-3-1987
TIME: 10:32 am
PROGRAM NAME: TOTAL.PAS

** Delta times (in seconds) **

<table>
<thead>
<tr>
<th>token#</th>
<th>time token inserted</th>
<th>time of the last response</th>
<th>Delta times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.1</td>
<td>0.0</td>
<td>= 8.1</td>
</tr>
<tr>
<td>2</td>
<td>20.3</td>
<td>8.1</td>
<td>= 12.2</td>
</tr>
<tr>
<td>3</td>
<td>40.0</td>
<td>20.3</td>
<td>= 19.7</td>
</tr>
<tr>
<td>4</td>
<td>49.5</td>
<td>40.0</td>
<td>= 9.5</td>
</tr>
<tr>
<td>5</td>
<td>62.0</td>
<td>49.5</td>
<td>= 12.5</td>
</tr>
<tr>
<td>6</td>
<td>70.0</td>
<td>62.0</td>
<td>= 8.0</td>
</tr>
<tr>
<td>7</td>
<td>120.0</td>
<td>80.3</td>
<td>= 39.7</td>
</tr>
<tr>
<td>8</td>
<td>150.0</td>
<td>120.0</td>
<td>= 20.0</td>
</tr>
</tbody>
</table>

Average delta times $129.7/8 = 16.21$ sec

** Order statistics **

<table>
<thead>
<tr>
<th>X</th>
<th>Y (token #)</th>
<th>X*Y</th>
<th>n=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

$r = 0.00067$
Example 7

Example of SPSS file: T2131ROGE.SPX
This file is produced by bubble 3.4 (Combine Data) using T2131ROGE.RSP of example 4. And this file will be used to analyze data using SPSSx.

T2131ROGE
%%
Rogers, Pat
131
2
7-3-1987 10:32 am
TOTAL.PAS
%%
1 2 3 7 8 4 5
8.1 12.2 19.7 9.5 12.5 8.0 39.7 20.0
%%

DESCRIPTION
(not part of the file)

name of other files
name of subject
class number
group number
token # in order
inserted
delta times
Mini_spec 4.0 Print

Step 1) Get program_with_selected_tokens.
Step 2) Print a program with underline replacing selected tokens and a program with selected tokens underlined (cloze_print_out).
(See example 8 for an example of clone_print_out.)
Example 8

Example of close_print_out:
Those files are produced by bubble 4.0 (Print) using total.clz (program_with_selected_tokens). This will be on paper.

```
line#
1  program total;
2   {  program to find the sum of the integers }
3      ( 1,2, ..., 100 )
4  var
5   number: integer;{counter from 100 to 0}
6   sum: integer;{the running sum}
7  begin
8   sum := 0;
9   number := 100;
10  while ( number > 0 ) do
11     begin
12         sum := sum + number;
13         number := number - 1
14     end; { of the while loop }
15  end.
```
Mini_spec 5.1 Display on screen

Step 1) Get program_name from the user.
Step 2) Display the first page (19 lines) of program_with_selected_tokens. Replace selected_tokens with underlines, and place cursor at the first underlined blank (first selected_token). Screen should look like the one in example 3.1. The insert window at the bottom of program window is not used for playback process.

Mini_spec 5.2 Simulate step by step

Reproduce inputs made by subject step by step. An interval between each input is some uniform value which is long enough for the observer to recognize each input.

Step 1) Get program_with_selected_tokens.
Step 2) Get playback_file.
Step 3) Simulate the subjects inputs:
if there is a pause_signal (Ctrl a), then stop until next pause_signal (Ctrl a) is received.
Fig. 1 Context diagram of the system
Correct_program

Select Tokens 1

Selection_of_tokens

Playback 5

Playback file

Playback file info

Programs with Selected tokens

Do Cloze 2

Print 4

Statistical File

Screen

Response

Cloze_print_out

Analyze 3

Statistical_analysis

Fig 2 Level 0 diagram
Fig. 3 Level 1 Select diagram
Fig. 4 Level 1 Do Cloze diagram
Fig. 5 Level 1 Analyze diagram
Program name

Display on screen

Program with selected tokens

File name

Pause signal

Simulate step by step

Playback

Playback file

Fig. 6 Level 1 Playback diagram
Cloze_print_out = *print out
tokens with
missing_tok
+ *print out
tokens
underlined*

Correct_program = *correct Pasca

Cursor_movement = [ UP | DOWN |
| END | PGUP |

Delta_times = sequence of re

file_name = *name of playb

New_screen = screen

Old_screen = screen

Order_statistics = r which is a r

Playback = *simulation of

Playback_file = user_data
+ program_name
+ { user_input }

Program_name = *name of a cor

Program_with_selected_tokens
= number_of_toke
+ number_of_line
+ program_with_@( )

Response_data = [ I *insert* | I
+ ( [ C *correct*
+ time_response_mα
+ line_
+ token_
+ { typed_token

Response_file = user_data
+ program_name
+ { Response_dat
+ statistical_da

Fig. 7 Data dictions
Screen = screen_info
     *internal screen representation*
     + current_curser_position

Selected_tokens = {{ legal_Pascal_token
                     | legal_Modula-2_token }}

Selecting_mode = [ Manual | Auto ]

Select_info = starting_point
               + ending_point
               + ( no_punctuation )
               + ( no_reserved_word )
               + n *integer value between 1 to 10*

Selection_of_tokens = selecting_mode
                       + selected_tokens

Spssx_file = file_name
             + user_data
             + list of token#’s as inserted
             + delta_times of tokens in the order of token# above

Statistical_analysis = user_data
                       + program_name
                       + delta_times
                       + order_statistics
                       + spssx_file *delta_times readable by spssx*

Statistical_data = number_first_correct
                  + percentage_first_correct
                  + number_last_correct
                  + percentage_last_correct
                  + number_first_wrong
                  + percentage_first_wrong
                  + number_last_wrong
                  + percentage_last_wrong

User_data = subject_name
            + class_#
            + group_#
            + time_of_the_experiment
            + data_of_the_experiment

User_input = [ cursor_movement
                | I *Insert* + typed_token
                | D *delete*     | E *end* ]
APPENDIX B.

Design Document for the Cloze System
1. PreClose.mod

**DATA STRUCTURE**

**CONST**

- MAXTOKEN = 1000

**TYPE**

- TokentypeType = (PUNC, RW, COM, other)
- SelectType = (YESSELECT, NOSELECT)
- TokeneleType = RECORD
  - name: INTEGER (pointer to the NameArray)
  - namelen: INTEGER
  - chosen: BOOLEAN := FALSE
  - selecttype: SelectType
  - tokentype: TokentypeType
- END record

- TokenlistType = ARRAY [1..MAXTOKEN]OF TokeneleType
- SelectmodeType = CHAR {a for AUTO, m for MANUAL}
- PrognameType = ARRAY[1..20] OF CHAR

**VAR**

- TokenList: TokenlistType (global value)
- NameArray: ARRAY[1..10000] OF CHAR (global value)

**PROCEDURES**

---

**AutoSelect( )**

Global value created: TokenList

Read the program name.

Decompose the program into tokens.

Select tokens automatically.

Save the program with selected tokens marked under a new name.

**VAR**

- ProgramName: PrognameType

Read from terminal (ProgramName)

IF ProgramName ends with '.pas'

THEN PasLex(in ProgramName) {this creates TokenList}

ELSE ModLex(in ProgramName) {this creates TokenList}

DoAutoSelect( ) {this makes changes in TokenList}

SaveTokenList(in ProgramName) {this saves TokenList in a new file}

---

**DoAutoSelect( )**

Side effect: TokenList

Read information needed to select tokens automatically.

Select the tokens using the TokenList created by PasLex or ModLex and information read (Let TokenList’s choose := TRUE if the token is selected).

**TYPE**

- SelectinfoType = RECORD
  - startpt: linenum := 0
  - endpt: linenum := lastline
  - nopunc: BOOLEAN := TRUE
  - norw: BOOLEAN := TRUE
  - n: CARDINAL
- END record
VAR  SelectInfo: SelectInfoType

read from terminal (SelectInfo.startpt, SelectInfo.endpt, SelectInfo.nopunc, SelectInfo.norw, SelectInfo.n)

FOR each token in TokenList
IF norw (reserved word not to be selected)
    THEN selecttype := NOSELECT for all reserved words
IF nopunc (punctuation not to be selected)
    THEN selecttype := NOSELECT for all punctuation

Skip all lines until the startline is reached
(i.e., Increment index until TokenList[index].linenum >= SelectInfo.startpt)

REPEAT the following
    Select every (SelectInfo.n)th YESSELECT tokens (i.e., TokenList[i].choose := TRUE)
UNTIL (the last token is reached)
    or (passed the SelectInfo.endpt)

DoManualSelect( )  Side effect: TokenList
    Use TokenList created by PasLex or ModLex.
    Build a liked list structure (for screen representation) using TokenList.
    Display the program on the screen.
    Update the screen as the user presses keys (by calling other procedures).
    Select the token if the selection key (enter key) is pressed.

TYPE  KeystrokeType=(up,down,right,left,home,finn,quit, enter)

TokenpointerRecord = RECORD
    linehead: BOOLEAN
    ltoken, rtoken,
    nextline, prevline: TokenpointerType
    tokencount: INTEGER (index for TokenList)
    namelen: INTEGER (0 for linefeed)
END record

TokenpointerType = ^TokenpointerRecord

VAR  KeyStroke: KeystrokeType
    FirstLine, (first token of the first)
    LastLine, (first token of the last line)
    cursor: TokenpointerType

SetupList(out FirstLine, LastLine)
Display the first page
Cursor := the first token
LOOP
    get(KeyStroke)
    CASE KeyStroke OF
up: UpALine() (update screen)
down: DownALine() (update screen)
right: MoveRight() (update screen)
left: MoveLeft() (update screen)
home: GoHome() (update screen)
finn: GoEnd() (update screen)
quit: exit (exit from this procedure)
enter: ChooseThis() (let choose := TRUE)

END case
END loop

GetMode(out SelectMode)
read from terminal (SelectMode)

ModLex(in ProgramName) Side effect: TokenList, NameArray
Open program with ProgramName
Read in and setup modfsm (fig.3).
Does lexical analysis using modfsm, and creates TokenList.
TokenList will be modified by
DoManualSelect or DoAutoSelect.

open(ProgramName)
read and setup(modfsm) {see fig.3 for Modula-2 finite state machine.}
Do lexical analysis and save tokens and tokentypes in TokenList (global value)
IF TokenList[current].tokentype = COM
THEN TokenList[current].selecttype := NOSELECT
ELSE TokenList.selecttype := YESSELECT

ManualSelect() Global value created: TokenList
Decompose the program into tokens.
Select tokens manually.
Save the program with selected tokens marked under a new name.

input(ProgramName)
IF ProgramName ends with '.pas'
THEN PasLex(in ProgramName)
ELSE ModLex(in ProgramName)
DoManualSelect( )
SaveTokenList(in ProgramName)

PasLex(in ProgramName) Side effect:TokenList, NameArray
Open program named "ProgramName".
Read in and setup pasfsm (fig.2)
Does lexical analysis using pasfsm,
and creates TokenList.
TokenList will be modified by
DoManualSelect or DoAutoSelect.

open(ProgramName)
readin(pasfsm)  {see fig.2 for pascal finite state
machine}
Do lexical analysis and save tokens and tokentypes in
TokenList
IF TokenList[current].tokentype = COM
    THEN TokenList[current].selecttype := NOSELECT
    ELSE TokenList.selecttype := YESSELECT

PreCloze( )
VAR      SelectMode: SelectmodeType
GetMode(out SelectMode )
IF SelectMode = 'M'
    THEN ManualSelect()
    ELSE AutoSelect()

SetupList(out FIRSTLINE, LASTLINE: TokenPointerType)
Called by DoManualSelect.
Using the TokenList build a linked list.
(See Fig.8 for description of the linked list.)

SaveTokenList(in ProgramName: PrognameType)
VAR      ClzFileName: PrognameType
ClzFileName := ProgramName replacing '.pas' or '.mod'
             with '.clz'
onopen(ClzFileName)
Save TokenList in ClzFile -- mark the selected token
(choose := TRUE) with '@(' at the beginning and ')') at the
end of it.
Fig. 1 PreCloze.mod design
Fig. 2 Pasfsm (finite state machine for Pascal)
Fig 3 Moafsm (finite state machine for Module-2)
2 cloze.mod

DATA STRUCTURE

CONST  FIRSTLINE = 1
       UNDERLINES = ' __________'  {10 underlines}

TYPE  CorrectnessType = (CORRECT, WRONG, NA )
      PgmptrType = POINTER TO PgmNode
      PgmNode = RECORD
       name, (pointer to StrSpace)
       namelen, linenum: INTEGER
       ltoken, rtoken: PgmptrType
       CASE selected: BOOLEAN OF
       TRUE: firstcorrect,
          lastcorrect: CorrectnessType
       prevtoken, nexttoken: PgmptrType
          typed, (pointer to TypedSpace)
       typedlen: INTEGER
       FALSE: (none)
       END record

PrognameType = ARRAY[1..20] OF CHAR
Keytype = ( INSERT, DELETE, UP, DOWN, LEFT, RIGHT,
           HOME, FINN, PGUP, PGDN, FINISH, MOUSE )

VAR  StrSpace: ARRAY[1..10000]OF CHAR (global value)
     TypedSpace: ARRAY[1..10000]OF CHAR (global value)
     RSPfile, PBfile: file {global value}
     LastLine: INTEGER (mark the last line)
     PgmArray: ARRAY[1..MAXLINE] OF PgmptrType

PROCEDURES

BuildLinkedList(in ProgName: PrognameType)
   Side effect: StrSpace, LastLine, PgmArray, PgmNode
   ProgName is provided by StartCloze.
   Read in and setup simplefsm {see fig. 5}.
   Open the file whose name is ProgName.
   Lexical analyze the program and create PgmArray and. PgmNode.
   read(simplefsm) {read in finite state machine to recognize
      selected tokens (surrounded by '@(' and ')'), carriage return and others. See
      fig. 5 for simplefsm}
   open(ProgName)
   Do lexical analysis and save tokens in PgmNode, at the same
   time, create PgmArray, and save strings in StrSpace.
   Set LastLine to be the number of the lines on the
   program.  (See fig.9 for an example of the PgmArray and PgmNode)

Cloze()  Create PgmArray, PgmNode and StrSpace by calling StartCloze.
   Do cloze procedure by calling DoEachResponse--
modification to PgmArray, PgmNode, TypedSpace and RSPfile, PBfile.

Get statistical info and save it on RSPfile.

input(ProgName)
StartCloze(in ProgName)
DoEachResponse( )
EndCloze( )

DeleteToken( )  Side effect: screen, RSPfile, PBfile, PgmNode
Delete token, which cursor is pointing, from the screen (replace the token with UNDERLINES).
Get time 'delete' key is pressed.
Save necessary information on RSPfile and PBfile.

Cursor.ptr^typed := UNDERLINES
Rewrite the line Cursor is pointing.
Write in RSPfile(in DELETE, time since the start of
experiment, Cursor.line, Cursor.ptr^tokennum)
Write in PBfile(in DELETE)

DoEachResponse( )  Side effect: TypedSpace, PgmNode, screen
Display the first page of PgmArray
(created by BuildLinkedList). Update the screen and save information
about the response made to RSPfile and PBfile by calling other procedures.

TYPE CursorType = RECORD
  linenum: INTEGER (index of PgmArray)
  ptr: PgmPtrType
END record

VAR PageHead, PageEnd: INTEGER
Cursor, OldCursor: CursorType
Key: KeyType

Display the first Page (19 lines) of PgmArray
Cursor := the first deleted token
LOOP
  IF PageHead <= Cursor.line <= PageEnd
     THEN BlinkCursor(in Cursor)
     OldCursor := Cursor
     GetKeyPressed(out Key )
  CASE Key OF
    INSERT: InsertToken()
    DELETE: DeleteToken()
    UP:    Up()
    DOWN:  Down()
LEFT: GoLeft()
RIGHT: GoRight()
HOME: Home()
FINN: End()
PGUP: PgUp()
PGDN: PgDn()
FINISH: EXIT

END case
IF PageHead <= OldCursor.line <= PageEnd
THEN UnblinkCursor(in OldCursor)
END loop

EndClose()
get number of first correct, first wrong, last correct, last wrong and their percentages, then save them in the RSPfile

GetKeyPressed(out Key: KeyType)
input(key)

GetUserData(out UserData: UserDataType)
read from terminal(UserData.name, UserData.class, UserData.group)

MakeFileName(in UserData: UserDataType, ProgName: PrognameType, out FileName: PrognameType)
{make name using rule below}
first letter = the first letter of ProgName
second letter = UserData.group
third to fifth letters = UserData.class
sixth to ninth letters = first four letters of UserData.name

InsertToken() Side effect: screen, RSPfile, PBfile, TypedSpace
Replace token which cursor is pointing with typed string.
Get time insertion is completed.
Save necessary information on RSPfile and PBfile.

VAR CorT: CorrectnessType
Token: ARRAY[1..80] OF CHAR

read(Token) from an insert window
Get time Token is typed
Save Token in TypedSpace and
Cursor.ptr^.typed := index of TypedSpace
IF Token is same as the correct token
    THEN CorW := CORRECT
    ELSE CorW := WRONG
IF Cursor.ptr^.firstcorrect = NA
    THEN Cursor.ptr^.firstcorrect := CorW
    ELSE Cursor.ptr^.lastcorrect := CorW
Rewrite the line Cursor is at with new token.
Write in RSPfile(in INSERT, CorW, Time, Cursor.line,
                 Cursor.ptr^.tokennum, Token,
                 Cursor.ptr^.name )
Write in PBfile(in INSERT, Token )

MoveScreen
{move screen consists of the following procedures}
Up()
Move screen up a line & write in PBfile(UP)

Down()
Move screen down a line & write in PBfile(DOWN)

PgUp()
Move screen up a page & write in PBfile(PGUP)

PgDn()
Move screen down a page & write in PBfile(PGDN)

GoLeft()
Move cursor to next left selected token.
Write in PBfile(LEFT).

GoRight()
Move cursor to next right selected token.
Write in PBfile(RIGHT)

SaveUserData(in UserData: UserdataType,
             ProgName: PrognameType)
    Global value: RSPfile, PBfile
    UserData is provided by GetUserData
    ProgName is provided by StartCloze
Write in RSPfile & PBfile UserData, time, and ProgName
**StartClose**(in ProgName: PrognameType)
   Side effect: RSPfile, PBfile
   Correct user information.
   Create RSPfile and PBfile names, open both files.
   Save user information on RSPfile and PBfile.
   Create PgmArray and PgmNode by calling BuildLinkedList.

    **TYPE**   **UserdataType** = RECORD
       name: ARRAY[1..20]OF CHAR
       class: ARRAY[1..3] OF CHAR
       group: CHAR {0..9, A..Z}
    END

    **VAR**   **FileName**: PrognameType
               **UserData**: UserdataType

    GetUserData(out UserData )
    MakeFileName(in UserData, ProgName, out FileName )
    RSPfile := FileName + '.RSP'
    PBfile := FileName + '.PB'
    open(RSPfile, PBfile )
    SaveUserData(in UserData, ProgName )
    BuildLinkedList(in ProgName )
Fig. 4 Cloze design
Fig 5 Simplefsm (finite state machine for Simple Lex)
3. Playback.mod

DATA STRUCTURE

CONST FIRSTLINE = 1
UNDERLINES = '________________' {10 underlines}

TYPE PgmPtrType = POINTER TO PgmNode
PgmNode = RECORD
  ltoken, rtoken: PgmPtrType
  linenum: INTEGER
  CASE selected: BOOLEAN OF
    TRUE: prevtoken,
    nexttoken: PgmPtrType
    typed: INTEGER {pointer to TypedSpace}
    typedlen: INTEGER
    FALSE: name: INTEGER {pointer to StrSpace}
    name1en: INTEGER
  END record

PrognameType = ARRAY[1..20] OF CHAR

VAR StrSpace: ARRAY[1..10000] OF CHAR {global value}
TypedSpace: ARRAY[1..10000] OF CHAR {global value}
PgmArray: ARRAY[1..100] OF CHAR {global value}
PBfile: file {global value}
LastLine: INTEGER {set by BuildLinkedList}

PROCEDURES

BuildLinkedList(in ProgName: PrognameType)
  Side effect: StrSpace, LastLine, PgmArray, PgmNode
  ProgName is provided by StartPB.
  Read in and setup simplefsm {see fig. 5}.
  Open the file whose name is ProgName.
  Lexical analyze the program and create PgmArray and PgmNode.
  read(simplefsm) {read in finite state machine to recognize
  selected tokens (surrounded by '@(' and '))'), carriage return and others. See
  fig. 5 for simplefsm}
  open(ProgName)
  Do lexical analysis and save tokens in PgmNode, at the same
  time, create PgmArray, and save strings in StrSpace.
  Set LastLine to be the number of the lines on the
  program.
  (See fig.9 for an example of the PgmArray and PgmNode)

DoPlayback()
  Side effect: screen
  Use PgmArray created by BuildLinkedList.
  Display the first page.
  Playback responses made by subject by calling
  other procedures.
  Parse when Ctl-a is pressed until Ctl-a is
  pressed again.
**TYPE**  

```
KeyType = ( INSERT, DELETE, UP, DOWN, LEFT, RIGHT,
            HOME, FINN, PGUP, PGDN, FINISH, MOUSE )
```

**CursorType** = RECORD  

```
    linenum: INTEGER {index of PgmArray}
    ptr: PgmptrType
END record
```

**VAR**  

```
Move : KeyType
Cursor, OldCursor: CursorType
```

Display the first page  
Cursor := the first deleted token  
LOOP  

```
    IF PageHead <= Cursor.line <= PageEnd
    THEN BlinkCursor(in Cursor)
    OldCursor := Cursor
    Delay 1/2 second
    IF Crl a is pressed
    THEN stop till Ctl-a is pressed
    GetNextMove(in PBfile, out Move)
    CASE Move OF
        INSERT: InsertToken() {update screen}
        DELETE: DeleteToken() {update screen}
        UP: Up() {move screen up a line}
        DOWN: Down() {move screen down a line}
        LEFT: GoLeft() {move cursor left}
        RIGHT: GoRight() {move cursor right}
        HOME: Home() {display the first page}
        FINN: End() {display the last page}
        PGUP: PgUp() {move screen up a page}
        PGDN: PgDn() {move screen down a page}
        FINISH: EXIT {exit this loop}
    END case
    IF PageHead <= OldCursor.line <= PageEnd
    THEN UnblinkCursor(in OldCursor )
END loop
```

**GetNextMove(out Move: KeyType)**  

```
(PBfile open by StartPB)
read from PBfile (Move)
```

**Playback()**  

```
Read name of the PBfile.
Open PBfile by calling StartPB.
Read in each response from PBfile and play them back on the screen by calling DoPlayback.
```

**VAR**  

```
PName: PrognameType
```

```
read from terminal(PName)
StartPB(in PName)
DoPlayback( )
```
StartPB(in PBfileName: PrognameType)
VAR ProgName: PrognameType

open(PBfileName)
read ProgName from PBfile
read from PBfile and print on the screen (username, class, group, date and program tested)
BuildLinkedList(ProgName)

UpdateScreen()
{move screen consists of the following procedures}
  Up() {move screen up a line}
  Down() {move screen down a line}
  PgUp() {move screen up half a page}
  PgDn() {move screen down half a page}
  GoLeft() {move cursor to the next left blank}
  GoRight() {move cursor to the next right blank}
  InsertToken() {replace cursor with token read from PBfile}
  DeleteToken() {replace cursor with UNDERLINES}
Playback design

Paper - design

Fig.6
4. Paper.mod

DATA STRUCTURE

TYPE PrognameType = ARRAY[1..20] OF CHAR

PROCEDURES

MakeFileNames(in ProgName: PrognameType,
out AnsFileName, QsFileName: PrognameType)
AnsFileName := ProgName replacing '.clz' by '.ans'
QsFileName := ProgName replacing '.clz' by '.q'

Paper( )Side effect: StrSpace, LastLine, PgmArray, PgmNode
ProgName is provided by StartPB.
Read in and setup simplefsm (see fig. 5).
Open the file whose name is ProgName.
Lexical analyze the program and create PgmArray and
PgmNode.

VAR ProgName, QsFileName, AnsFileName: PrognameType
QsFile, AnsFile: file

PaperSetup() (read in finite state machine to recognize
selected tokens (surrounded by '§(' and
')'), carriage return and others. See
fig. 5 for simplefsm)

open(ProgName)
MakeFileNames(in ProgName, out QsFileName, AnsFileName)
open(QsFileName, AnsFileName)
Do lexical analysis and write out tokens in AnsFile and
QsFile. In QsFile, the deleted tokens are replaced by 10
underlines, AnsFile is same as an original '.pas' or
'.mod' program. (See specifications for the examples of
QsFile and AnsFile)

PaperSetup()
Setup finite state machine same as cloze and playback
5. DataAnalysis.mod

DATA_STRUCTURE

TYPE    PrognameType = ARRAY[1..20] OF CHAR

PROCEDURES

Analyze(  )
VAR    AnaFile, SpssFile: file

GetSubjectData(out AnaFile, SpssFile)
GetDeltaTimes(in AnaFile, SpssFile)
GetOrderStats(in AnaFile)

GetDeltaTimes(in AnaFile, SpssFile: file)
calculate delta times
output the delta times to AnaFile
output the order inserted to SpssFile
output the delta times to SpssFile

GetOrderStats(in AnaFile: file)
calculate order statistics
output the order statistics to AnaFile

GetSubjectData(out AnaFile, SpssFile: file)
VAR    RSPFileName,
       AnaFileName, SpssFileName: PrognameType

input(RSPFileName)
open(RSPFileName)
MakeFileNames(in RSPFileName,
               out AnaFileName, SpssFileName)
Open(A AnaFileName, SpssFileName) as AnaFile & SpssFile
Read from RSPFile(subject name, class, group, time,
date of the experiment, program tested on)
Write in AnaFile(subject name, class, group, time and
date of the experiment, program tested on)
Write in SpssFile(RSPFileName)
Write in SpssFile(subject name, class, group, time and
date of the experiment, program tested on)

MakeFileNames(in RSPFileName: PrognameType,
               out AnaFileName, SpssFileName: PrognameType)
AnaFileName := RSPFileName replacing '.rsp' by '.ana'
SpssFileName := RSPFileName replacing '.rsp' by '.spx'
Fig. 7 Analyze design

A - AnaFile
RSP - RSPFile
S - SpssFile
Example program to be used for data representation
Example of program_with_selected_tokens: total.pas

Program total;
{ program to find the number of integers }
{ 1,2,...,100 }
var
  @(number) : integer; { counter from 100 to 0 }
  sum @(:) integer; { the running sum }
begin
  sum @(:=) 0;
  number := @(100)
  while ( number @(>) 0 ) do
    begin
      @(sum) := sum + number@(;;)
      number := number - @(1)
    end; { of the while loop }
end.

%%
8 tokens
16 lines
Fig. 8 PreCloze.mod TokenizerType
structure example
Fig. 9 Close.mod, PlayBack.mod, Paper.mod
PgmArray & PgmKds structure example
APPENDIX C.

Program listings
program polyadd(input,output);
by jr ullrich Jan 15, 1987 )

( This program adds 2 polynomials:
  eq. ( 1 + 2X -3X**2 ) and ( 2 - X )
  gives ( 3 + X - 3 X**2 )

This program read N which is the number of coefficients
next reads pi's such that
  p(X) =p0 + p1X + p2X**2 +...+ pNX**(N-1) 1⩽ i ⩽ N
next reads qi's such that
  q(X) =q0 + q1X + q2X**2 +...+ qNX**(N-1) 1⩽ i ⩽ N
then adds pi + qi for all i in 1..N
at last writes out ri's such that
  r(X) = (p0 + q0) + (p1 + q1)X +...+ (pN + qN)X**(N-1) 1⩽ i ⩽ N )


type link = *node;
  node = record
    c : integer;
    next : link
  end;
var N : integer;
  z : link;

function readlist(N: integer): link;
( reads pi's such that
  p(X) =p0 + p1X + p2X**2 +...+ pNX**(N-1) 1<= i <= N
  then make a linked list representation of p(X) )
var i: integer; t: link;
begin
  t := z;
  for i := 0 to N-1 do
    begin
      t := new(t^.next);
      t := read(t^.c);
    end;
  t^.next := z;
  return t;
end;

procedure writelist(r:link);
( writes out ri's such that
  r(X) = (p0 + q0) + (p1 + q1)X +...+ (pN + qN)X**(N-1) 1<= i <= N )
begin
  while r <> z do
function add(p,q: link): link;
{ adds pi + qi for all i in 1..N
and produces r(X) such that
r(X) = (p0 + q0) + (p1 + q1)X +...+ (pN + qN)X**(N-1) 1<= i <=N } 

var @t: link;
begin 
t := z;
  repeat 
  @new(t^.next);
  t := t^.next;
  t^.c := p^.c + q^.c;
  p := p^.next;
  q := q^.next;
  until (p=z) and (q=z);
  t^.next := z;
  add := z^.next
  end;

begin 
readln(N);
new(z);
  writelist(add(readlist(N),readlist(N)))
end.

%%
54 lines
25 tokens selected
( Count occurrences of each word in a text. )

PROGRAM concordance (input,output);

CONST
  tablesiz = 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;

PROCEDURE readword(VAR packdword: @wordtype);

VAR
  buffer : ARRAY [S(charindex)] OF char;
  charcount : 0..maxwrdlen;
  ch : char;

BEGIN
  IF NOT eof
    THEN
      REPEAT
        read(ch)
      UNTIL eof OR (ch IN letters);
      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 : 1@..maxwrdlen;
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 { else, for, and with }

( concordance )

%%

114 lines
32 tokens selected
program fixedlength (textin:(), textout, output); 

('An Introduction of Programming and Problem Solving with ') 
('Pascal' by G. Schneider, S. Weingart & D. Perlman page 369 )

('This is a program that reads lines from a textfile of length') 
('1-80 characters and produces a second testfile containing') 
('fixed length 80 character lines')

const
maxline = 80; (fixed length line size) 
blank = ' '; (the padding character)

var
a(textin): text; (the input file with variable length lines) 
textout: () text; (the output file with fixed size files)
size: integer; (the current line size) 
i: integer; (for loop count)
ch: char; (current character being read)

a(begin)
reset(textin); 
rewrite(textout); 
while not eof(a(textin)) do
begin 
size := 0;
while (not eoln(textin)) and (size < maxline) do
begin
read((textin, ch); 
size := size + 1;
write(textout, ch); 
end; (while loop)
a(for i := (size + 1) to maxline a(do)
write(textout, 'blank');
readln(a(textin)); (skip to new line in the input file)
writeln(a(textout)); (put a line separator in the output file)
end; (of the while loop)
writeln('copy has been completed')
end. (of program fixedlength)
program Josephus(input, output);
{ from 'Instructor's guide to accompany Introduction to
    Computer Science with Applications in Pascal' by Garland p 190 }

( Program to solve the Josephus problem )
( N people are arranged in a circle. Proceeding clockwise, every)
( Mth person leaves until only one remains. Who is left? )

type nodePtr = 'node;
    node = record
        data: integer;
        link: nodePtr;
    end;

var i, m, n: integer;
    first, last: nodePtr;    { pointers of circular list }

begin
    writeln('Enter number of people in circle: ');
    readln(n);
    { build circular list with entries numbered 1 to n }
    new(first) @();
    last := first;
    @for i := 2 to @n do
    begin
        new(@last)^.link @();
        last := last^.@().link;
        last^.@().data := i
    end;
    last^.link @:= first;

    writeln('Enter number to count off by: ');
    readln(m);

    { print departures }
    writeln('Departures');
    while first <> @last do
    begin
        for i @:= 1 to m@1 do
        begin
            last := @first;
            first := first^.@().link
        end;
        @write(first^.data)@(': ');
        last^.link @:= first^.link @();
        dispose(first) @();
        first := last^.@().link
    end;
writeln();
writeln( first.data:1, ' is the last one left.' );
end.

XX
45 lines
25 tokens selected

@program gcd(input, output);

var
r, m, n: 3(integer);

begin
readln(m,a(n));
repeat
r := 3(m) mod n;
m := Q(n);
n := r;
until Q(n) = 0;
writeln(3(m));
end.
program postest(input, output);

form 'Introduction to Computer Science with Applications in Pascal'
by S. Garland page 315

const MaxChars = 20;

type varyingString = record
  length: integer;
  chars: packed array[1..MaxChars] of char
end;

var s1, s2: varyingString;

function pos(a, b: varyingString): integer;

  finds the first character of the first occurrence of b as a substring
  of a, returning its index if found and returning 0 if not

var
  match: boolean;
  start: 0..a(MaxChars);

procedure compare;

  sets match true if b matches the substring of a
  that begins at character position start

begin
  match := false;
  start := 0;
  while start + b.length <= a.length and not match do
  begin
    start := start + 1;
    compare
  end
end;

begin
  match := false;
  start := 0;
  while start + b.length <= a.length and not match do
  begin
    start := start + 1;
    compare
  end
end;
end;
if match then $\text{pos} := \text{start}$ else $\text{pos} := 0$;
end; \text{( of pos )}

begin
  s1.chars := 'abracadabra';
  s1.length := 11;

  s2.chars := 'ra';
  s2.length := 2;

  writeln( pos( s1, s2 ) );

end.

%%
43 lines
25 tokens selected
program x(input, output);
type ary = array[1..5] of real;
var data: ary;
  n: integer;
@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)
    end (until) done
  end (while)
end (sort);
begin
  SORT(data,5);
  for n := 1 to 5 do writeln(data[n]:5:2);
end.
Bibliography


