AN ALGORITHM DEVELOPMENT PROGRAM USING WARNIER-STYLE BRACES

by

JOSEPH KENT CAMPBELL

B.S., Kansas State University, 1971
M.S., Kansas State University, 1976

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

Approved by:

David A. Gustafson

Major Professor

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

page

Figure

ACKNOWLEDGEMENTS

I would like to thank my Major Professor and friend, Dr. David Gustafson for the consistent encouragement and help he has given me during this long process. Special thanks also to my other committee members, Dr. Virg Wallentine, Dr. Paul Fisher, and Dr. Elizabeth Unger for their support and confidence.

This report would never have been completed without the backing of my wife, Shirley. I hope I appreciate her as much as she deserves.

CHAPTER ONE - INTRODUCTION

Background

The concept of top-down design is an abstract idea which is difficult for some computer science students to grasp. The difficulty of visualizing top-down development is especially noticeable in students just starting in computer science. This is due in part to the method by which lower level problems are presented. Often in beginning courses the problem definition phase is done independently of the student, either by an author or the teacher. Students usually do not have developmental questions as part of the programming assignment. Rather the first problems require little more than coding a solution. Thus the student begins work in computer science encouraged to approach problems in a wholistic manner rather than using true top-down methodology [Schn 82,Hass 82b].

Hassell [Hass 82a] points out that thinking in a top-down manner may be foreign to some students. If one has never worked on a program that is too large to comprehend totally (i.e. wholistic) then it is difficult to see the real need for breaking the a problem into modules. Meyers [Meye 78] reminds us that in the past, introductory instruction had in fact often worked in reverse, starting from instruction writing and then working back to earlier stages of design and requirements.

The advent of interactive programming has also contributed to poor programming habits. "Interactive programming is particularly prone to those problems that result from the lack of a systematic disciplined, thoughtful approach to software development" [Well 81]. Visualizing

problems as a series of small steps to be executed seems to be a simple, straight-forward idea on the face of it, but to put this into practice requires a great deal of patience and thought.

Beginning students are naturally apprehensive about getting the task done and are anxious to view the end result. The effort required to break the problem down may seem like a waste of time and effort, especially if the problem is stated in such a way that the solution is more or less obvious. If students were better able to visualize the top-down design then the concept should be easier to understand [Hass 82b]. It is important that students understand the concept since it is the one of the important principles of programming and software engineering. It has been difficult, however, to incorporate structured design principles into initial courses on programming. It is important that the design methodology does not appear to be more of a burden than an aid, because at the beginning stages the methodology may not be essential [Well 81].

Survey of Pertinent Literature

In this literature survey this author will discuss the evolution of top-down design from established problem solving techniques, examine the research pertinent to learning programming, and discuss the implications of this research to program development and software engineering. Also reviewed will be some of the software tools which support top-down design and construction. The study of how humans go about problem solving has taken on renewed vigor in the past fifteen or so years[Scan], primarily due to the rapid development of computer

science and the skills necessary to put problems in a form which computers can use. There is general agreement that the problem to be solved must be understood as a first step [Scan 77,Hunt 62,Jack 75,Thar 80].

After the problem is understood, the person must make judgements as to relevant knowledge[Jack 75]. This second step means that the person must place the necessary facts of the problem into their proper relationship. The next step is to construct some course of action, which depends heavily upon the success of the first two steps. Generally a problem, especially one which can lead to a computer solution, will have many possible solutions. Thus this step involves discriminating between the choices and making the best possible selection of a solution. It has been shown that this selection process correlates quite well with the experience level of the individual [McKe 81].

The final step in this problem solving technique is to implement the decision. Jackson, Scandura, and Brooks point out that although we can show decision making and problem solving as somewhat discrete steps, the semantics of these parts may be impossible to determine [Jack 75,Scan 77]. For instance, how does one determine when true understanding of a problem is reached? The need for understanding the mental processes involved in problem solving has always been realized by researchers, but the need to apply research principles to the art of programming came into clearer focus in the mid 1970's as a new discipline called software engineering began to take definite form.

As software engineering developed, the techniques of problem solving were used as the foundation of methodologies. Some of these techniques were founded on facts but the application to computer problem solving also included the experiences of software designers and intuition [Mohe 82], [Tria 79]. Some ideas about problem solving were extended to become principles of software engineering without the necessary basic research to prove the connection [Mohe 82,Curt 83].

Some of the psychological research about problem solving and short term memory found its way into the computer science literature. In particular, one theory is that humans can conceptualize about seven items, plus or minus two, at a time [Mill 56]. The magic number seven should not be used to argue that a program should consist of no more than seven modules or that a leader's span of control is seven people. Issues such as these are affected by processes unrelated to those under which the magic number seven was validated [Curt 83]. Other ideas, such as viewing problem solving as a series of discrete steps were applied to the new science. While these ideas were valid in their original context, the application to the new science was not based on established research [Mohe 82]. Quite naturally, as one might expect, personal observations as precepts sparked a great deal of controversy, witness the debate about GOTO's. Initially, there were a few attempts to verify the fundamentals of software engineering and the mental processes involved in programming in a laboratory setting [Mohe 82].

Starting in the late 1970's, a great deal of research has been carried out to verify the wisdom of programming techniques in general, and

software engineering principles in particular [Mohe 82]. Much of these research efforts have been concerned with exactly how persons learn programming and the best way to develop curriculum to teach programming concepts. Brooks [Broo 80] points out that if computer science researchers are to maintain creditability with behavioral researchers in other areas, we must pay close attention to methodological issues. Thus we must be careful about such issues as subject selection, examination environment, and performance requirements to insure the data results are as valid as possible.

Many factors influence the ways in which programmers approach the task of thinking about problems and writing programs. This makes the selection of subjects an extremely difficult task [Broo 80]. Moher [Moh 82] makes the point that in several reports, the variability due to subject differences often outweighed variability due to the independent variables. The problem of paucity of good basic research is further complicated by the realization that the results may only apply to a narrow range of application and may or may not be valid as extensions to programmers in general[Broo 80].

Despite these limitations, some interesting research has been reported, especially in the areas of cognitive style and what sorts of software tools are beneficial in the development process. Cognitive style is a measure of how one approaches learning and problem solving [Hass 82a]. It is a collection of preferred ways of viewing one's surroundings, thinking about relationships, and solving problems. Hassell [Hass 82a] reports on findings that show subjects are strongly influenced by their frame of reference in other situations. Subjects

who are strongly influenced by the clues and structures of their surrounding fields are called "field dependent" and at the other extreme are subjects who seem to function by some internal standard and can largely ignore extraneous clues in their surroundings. For instance, instructors at Tulane noted that field dependent persons, who traditionally do poorly in first programming courses, were doing better than expected in a course using a highly structured design tool called Structure Charts [Hass 82a]. An ensuing study led to the conclusion that the use of highly structured techniques provides the field dependent student with a structure to impose on the problem [Hass 82a].

There is no correlation between cognitive styles and intelligence, but differing cognitive styles have implications in the teaching of programming. Field independent persons can often impose their own organization to a problem, whereas field dependent persons find such things as Warnier-Orr diagrams and structure charts very helpful in organizing their ideas. However, graphical aids have been shown to help both groups [Hass 82a]. The traditional flowchart is not generally considered as a good graphical aid because it forces program design to early and is difficult to comprehend and even more difficult to modify, especially for extremely large programs [Hass 82b]. However a graphical design tool utilizing a tree like diagram has been shown to provide good results when used with beginning computer science students [Hass 82a]. An enhanced version of a Warnier-Orr diagram producing program, while not researched fully, has shown great promise, not only as a design tool but for documentation as well [Conr 84].

Steele [Stee 83] experimentally investigated the cognitive processes used during the design phase of the development cycle. This study determined that a person having the skills required to specify steps leading to a solution does not imply the reverse. That is, when presented with a solution a person may not have the skills to recognize the problem which led to that solution. At present, much of the material developed to teach programming treat design as a single cognitive skill. In his review of human factors research, Curtis [Curt 83] highlighted three principles which are supported by various studies carries out over the past ten years. First, comprehension is aided by a structured control flow. Second, languages should have the capacity to handle a rich variety of data structures. Third, it has been shown that task comprehension is enhanced when specifications make clear the information concerning data structures and organization.

Wexelblat [Wexe 81], in his informal survey of programmers, found that programmers are strongly influenced in some cases by their first programming language and stronger yet by how they were taught. He makes the point that it is extremely important to learn programming as a total concept and not just translating problems into code.

For the most part, research has shown the original ideas about top-down design and structure programming are valid [Curt 83]. It is important to remember that what we call software engineering is inextricably intertwined with how we learn to program. One must remember, however, that education in software engineering is fundamentally different from education in computer science [Free 76].

The two do share a large area of concern, but software engineering curriculum should also include teaching in management science, design methodology, and interpersonal communications. As Steele [Stee 1983] stated: "Through the cyclical scrutiny of both learning and teaching, the greatest inroads are made to the development of an effective curriculum."

A number of software tools have been developed over the past five to ten years [Houg 80] which emphasize, among other things, top-down development and the automation of the programming process. These programs tend to fit into two broad categories. Some of the programs are intended as documentation aids and the other group is mostly concerned with code generation. Occasionally there is some crossover when programs intended for code development lend themselves as documentation instruments. The reverse is seldom true [Tria 79]. The objective of the majority of these software aids is to produce good documentation of the completed design. It must be remembered "they offer minimal help in the design task itself"[Tria 79].

Tools help provide an interactive programming environment which allows the user to not be concerned with details of syntax and concentrate on the creative aspects of programming. Many of the programs generate code, such as the Cornell Program Synthesizer[Teit 81]. Some of the programming aids generate graphics which illustrate the program structure. Triance discusses a system which draws structures much like the Jackson Design Method depending on how the programmer describes the control flow. Some of the generators help nonsophisticated users by supplying screens which quiz the individual

for information.[Radi 82] As the fields on the screen are filled, the program interacts with a data base to produce a program. These types of generators are best suited to produce standard reports. Many of the programs are language syntax directed and feature extensive error checking combined with diagnostic statements [Akin 82], [Teit 81]. Several of the development tools look at data flow and data modification as the basis for the tool [Akin 82], [Roma 79]. This would seem to agree with research carried out by Chand [Chan 80] and Balzer [Balz 81] which examined program development from the standpoint of how data is modified.

Program generators can increase code production and programmer efficiency, but true automatic programming has not yet been realized. In a review of three program generators Moskowitz [Mosk 83], emphasized that while there are some automatic code generators, no tool has been developed which does creative thinking. One still has to know how to program before the code generator is of much value!

Objective of the System

The process of software development should be enhanced by the development of a system which would provide help menus for program development, make provisions for top-down refinement, and be screen oriented for statement input. A possible way to accomplish these goals is to have a system capable of providing for expansion of any program statement using a methodology similar to Warnier [Warn 74]. This methodology would provide the templates needed for screen input and at the same time allow for decomposition of a problem.

The goal of the test was to have a problem solving tool which would be useful in a variety of ways. That is, we wish to develop a tool that will be useful for all phases of program development, including the initial understanding of the problem, will provide for top-down development , and will also be useful as a documentation instrument. We are interested in providing a development tool which requires little experience, provides internal development instructions, allows for updates, and provides an easy-to-follow display of an algorithm.

CHAPTER TWO - REQUIREMENTS

The operation of the program should be as self-explanatory as possible such that a minimum of preliminary study about the program is necessary. The program should contain the necessary prompts and internal instructions so that the program will be just as useful to a novice as to an experienced programmer.

The system should provide the user with some tutoring in program development and have the capability of providing a menu driven help facility for more specific instructions. The tutoring should come early in the program execution and experienced users should be able to bypass the tutoring. The help facility should be available from anywhere in the program and the return should put the user back to the place from which the help was called.

The program should have the capability of 'expanding' a program statement. That is, the system should be able to produce a new screen on command with which to amplify the meaning of any statement. This expansion should be clearly related to the statement that was expanded and should have some clear numbering system showing the order of expansion.

The user should be able to peruse the screens comprising the algorithm at will and make any changes utilizing a full screen editor. The perusal should be easy for the user to do and perusal should be possible in any direction. Changes made to any screen during the perusal process should be saved automatically.

The program should have the capability of assembling statements into a facsimile of a Warnier-Orr diagram and be able to recover and modify any previous diagram. The assembled diagram should show clear relationships between screens so that the order of expansion is not in doubt. The program should be able to search the current directory for a previous diagram and recover it into the main data structure for examination or modification.

The program should inform the user that very limited modification to an algorithm is allowed between sessions. The user must realize early in the program's execution that to recover previous algorithms, rigid conditions must be met.

The documents produced by the system should have a hierarchical numbering system which will make them useful as a documentation instrument for many phases in program development. The numbering system should clearly show which line was expanded and its relationship to any previously expanded lines in a hierarchy.

CHAPTER THREE - THE DESIGN

To utilize the file accessing capabilities of the UNIX Operating system and provide the desired screen oriented feature, the design decision called for the use of preconstructed screens which would display programming instructions on Warnier-Orr style input templates. These screens are simply files containing the desired instructions or template which could be copied to the screen as needed. UNIX allows one to open a number of files with relatively simple commands during program execution.

A stored algorithm in the file system program can be recovered for the session, but this requires that the file remain substantially unchanged by some system editor in the interim. By substantially unchanged the design dictates that no lines shall be added or deleted since this will change the fixed screen size. The finished diagram also has two header lines which the program bypasses in the recovery process, so those must remain undisturbed.

The user help requirement was met by adopting a system of information screens which would be available at the start of the program supplemented by a menu-driven help command that would be available from anywhere in the program (see Figure 3.0). The information screens can either be perused one at a time or bypassed for the most part if the user doesn't need them. Once in the help area the user may select from a menu of specific screens and examine one or all the screens in any order and when finished the return is to the screen from which the help call was made.

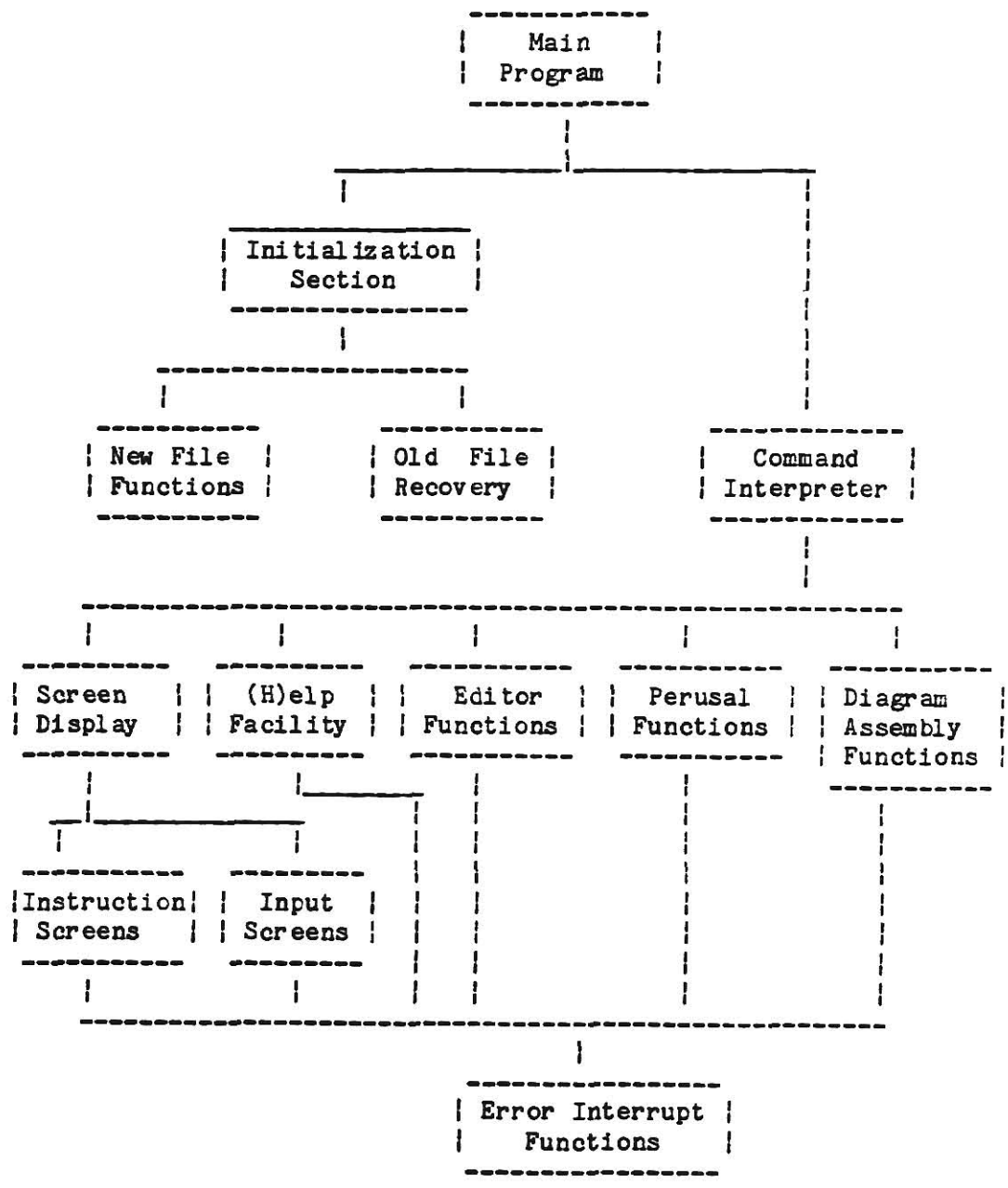

Figure 3.0 Hierarchy Diagram

The fact that the program was to be screen oriented, coupled with the excellent file accessing features of 'C' indicated that the screen display system be table-driven. This quite naturally led to the use of a structure in 'C' called a structure. The information that seemed critical was to store the screen number and the starting location of the screen in the screen storage file (see Figure 3.1). This information would allow for most any operations on the screens such as tracing the hierarchy, sorting the screens, and overlaying a screen in the file if changes were made.

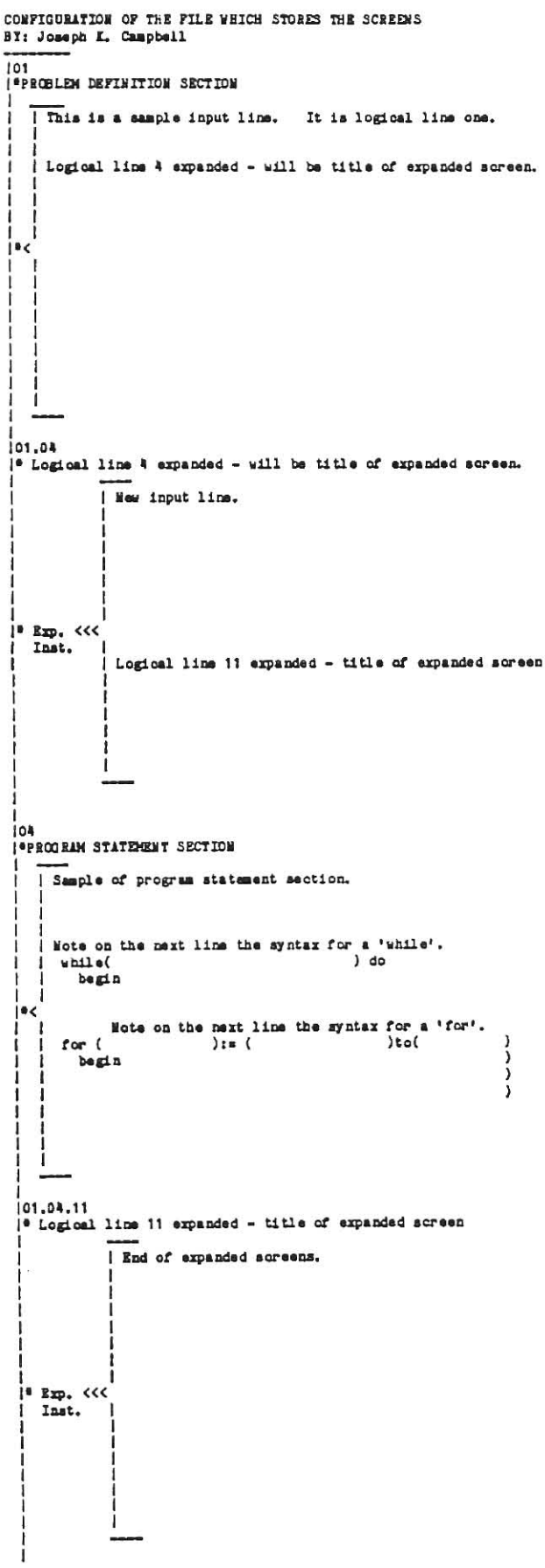

Figure 3.1 Screen Save-File Configuration - Before Sorting

After the problem is defined, a program has three basic parts, input, program statements, and output statements. Therefore the design decision was made to have the input screen hierarchy start in each of those four areas, but provide a common 'home' screen from which the user could develop the program in any of the four parts.

Encouraging the decomposition of a program is partially accomplished by having the input screen hierarchy begin in one of four areas. Once the user has started the program and is at least as far a screen 2, It was determined that a logical way to provide for top-down development was to provide a new input template on command whenever a statement was determined to need refinement. This template would be similar to the old but have some numbering system which would tie all the templates together. The new template could in turn be expanded and this process continue to some finite point where the statement is decomposed to the desired level of detail.

To provide the 'expansion' feature, the judgement was that the best way was to again utilize the advantages of UNIX and provide a new screen which would have a label linking it to the expanded statement. This label would actually be the statement which was expanded. This label, combined with the hierarchical screen numbering system would show the relationship between two consecutive screens and also allow the expansion sequence to be traced to the original or any subsequent expansion.

To simplify the ordering of the input templates, no deletion of lines or screens would be allowed. This requirement was a result of using a

file to store the screens. When an existing screen is corrected, then the corrected version must replace the old version in the file, thus having the screens of a fixed size made the overlay process quite simple. Screens stored in the file the first time are appended to the end of file.

To speed the operation of the program and to arrange the program functions in a hierarchy, it was decided to have the commands be interpreted in one function with program control always ultimately returning to that command interpreter. This mode of operation is similar to the way UNIX handles editor commands. The only time program control does not reside in the command function is when the user is in the help mode. Having the command control in one function should make it easier to enhance or modify the program.

Originally it was planned to use some form of the UNIX 'vi' editor for input to the templates, but the development of a limited editor was more time-effective. The 'vi' editor is really a large collection of extremely complex programs which are not well documented and are written in very sophisticated 'C' code. By using a package of screen commands called 'curses' it was relatively simple to construct a small, full screen editor and designate protected areas on a screen such that input is only allowed in certain places. In many respects the developed editor behaves like a subset of 'vi', therefore a user familiar with that editor need not learn new commands.

CHAPTER FOUR - IMPLEMENTATION

Because the UNIX operating system is written in 'C', and many routines which the program could utilize without modification were written in 'C', that programming language seemed to be a very natural choice for this system development. By using 'C' the system calls could easily be made and the opening and closing of supporting files was compatible with the operating system. The screen handling package called 'curses' is also written in 'C' so that was a major factor in the decision to use the language. 'C' not only has a structure similar to records in Pascal, but it allows one to use pointers much as one would use an array, except the pointer can actually be incremented, thus in effect one has a dynamic array. 'C' allows one to access files much like an array, but at a slight sacrifice in execution speed. This feature provided an easy way to store and access the template screens. Finally, selection of 'C' provided the author with a good excuse to learn the language, which is really a necessity if one is to do much work with UNIX.

The program has four parts, initialization and recovery, general and specific instruction screens, input to template screens, and Warnier-Orr diagram assembly. Initialization and recovery are accomplished by prompting the user for specific information which the program needs to begin operations. The instruction screens are of two types: general and specific help screens, any of which may be located and perused as the user desires. The template screens actually display a stylized Warnier-Orr brace and input is allowed only within the confines or the brace. Assembly of the final Warnier-Orr diagram is

automatic as part of the finishing function.

The initialization and recovery sections start the user in the correct mode or recover a previous algorithm for modification. First, the program determines if the current session is to be used to develop a new algorithm or if it will be necessary to recover an existing algorithm. This is accomplished by asking the user to respond to numbered questions by pressing the corresponding number. After the program determines the purpose of the session, the appropriate section is executed.

The initialization section requests information from the user to be used as headings for the completed diagram. The cursor moves automatically to the first of two input areas where the user types in his/her name and the second input area records a diagram title. These two headings are stored in a character pointer by incrementing the pointer and then written to the file which will become the final Warnier-Orr diagram.

The initialization section also takes care of the details of recovering a previous diagram and storing the required information in the program's data structure. First the program uses a Stream Editor (SED) command to remove the last line of the Warnier-Orr diagram and to strip the 'Warnier' file of all but the first two lines and move the lines to the internal diagram storage file. The program then uses a 'C' function called 'fseek' which positions the read pointer at a specific byte location in the file, then the screens are examined one at a time till the end is reached. As the screens are displayed, the

screen identification number is copied and stored in the table of screen data. The starting position of that screen in the diagram storage file is also noted. Since each screen has a fixed length, the position to be stored is simply the previous position plus a fixed byte count. This is another reason why the fixed screen size proved to be a good design decision. That is, one can overlay an old screen with a corrected version, or in the case of a new screen simply open the file for appending and write the screen to the file. Each time a new screen is added, the position count is incremented by the screen byte size to mark its beginning location in the save file.

One important function of the program is to provide a method of building the Warnier-Orr diagram, which is accomplished by providing template screens upon which the user can develop program statements. The templates are a stylized version of the original Warnier brace and occupy a different portion of the screen, according to whether the template is first in a hierarchy or an expanded screen. The templates provide visual reinforcement to the concept of top-down design (see Figure 4.0).

```
CONFIGURATION OF THE FILE WHICH STORES THE SCREENS
BY: Joseph K. Campbell
----------
|01
|*PROBLEM DEFINITION SECTION
|  ____
|  | This is a sample input line.   It is logical line one.
|  |
|  |
|  | Logical line 4 expanded - will be title of expanded screen.
|  |
|  |
|  |
|  |
|*<
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  ____
|
|01.04
|* Logical line 4 expanded - will be title of expanded screen.
|          ____
|          | New input line.
|          |
|          |
|          |
|          |
|          |
|* Exp. <<<
|  Inst.   |
|          | Logical line 11 expanded - title of expanded screen
|          |
|          |
|          |
|          |
|          |
|          ____
|
|
|01.04.11
|* Logical line 11 expanded - title of expanded screen
|          ____
|          | End of expanded screens.
|          |
|          |
|          |
|          |
|          |
|* Exp. <<<
|  Inst.   |
|          |
|          |
|          |
|          |
|          |
|          |
|          ____
|
|04
|*PROGRAM STATEMENT SECTION
|  ____
|  | Sample of program statement section.
|  |
|  |
|  |
|  | Note on the next line the syntax for a 'while'.
|  |  while(                                ) do
|  |    begin
|  |
|*<
|  |        Note on the next line the syntax for a 'for'.
|  |  for (              ):= (                )to(          )
|  |    begin                                                 )
|  |                                                          )
|  |                                                          )
|  |
|  |
|  |
|  |
|  ____
|_____
```

Figure 4.0 Sorted and Assembled Warnier-Orr Diagram

A major goal of the program is to encourage the user to decompose the problem. This is partially accomplished by having the input templates begin in four areas: problem definition, input definitions and statements, program statements, and output definitions and statements. After the user has performed the initialization routines and has reached 'screen 2', he/she may request a template for problem definitions by pressing 'D'. The other areas may be reached by pressing 'I' for input, 'P' for program statements, and 'O' for output statements. If the user is working somewhere in the expanded screen hierarchy these screen may also be requested, eliminating the need to peruse one screen at a time till they are reached. Thus the user has the flexibility to move around to different parts of the the program development at will.

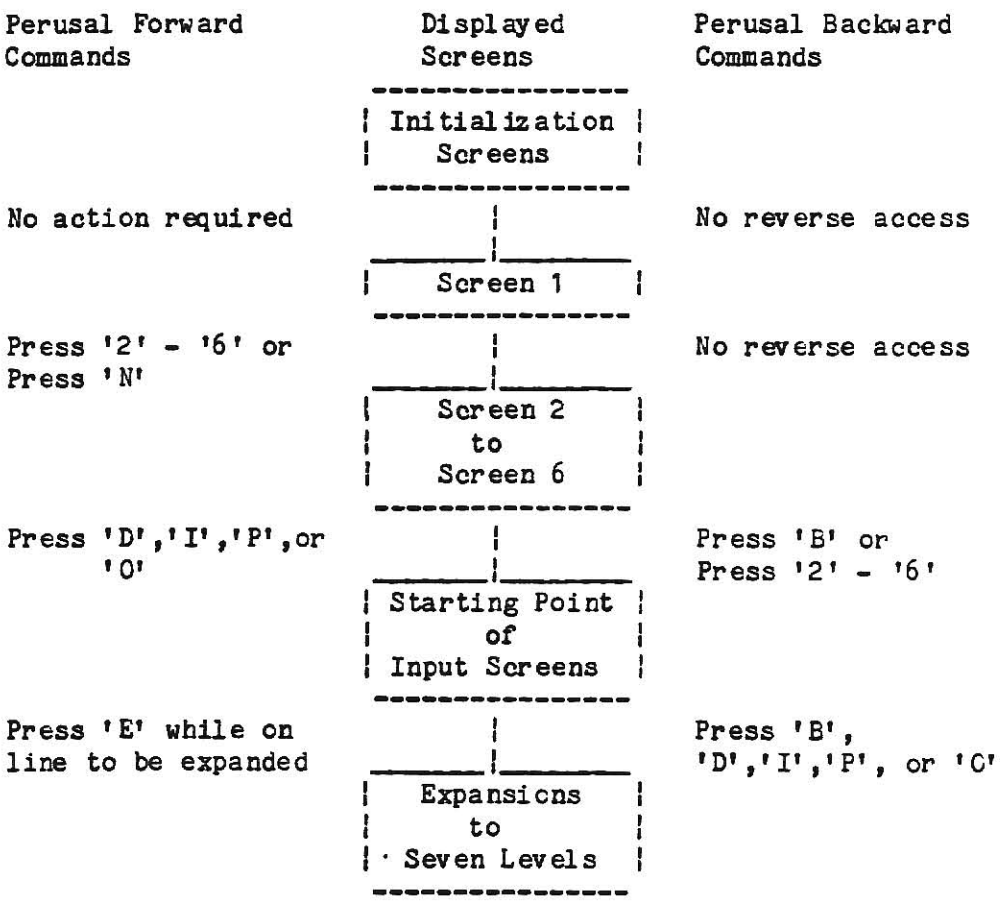

Figure 4.1 Perusal of Screens

When a user wishes to expand a statement to greater detail, a request is made for a new screen, which has the expanded statement as its title. The request command is quite simple. The user simply positions the cursor anywhere on the line to be expanded and presses 'E'. The program then copies the entire line which is to be expanded and the screen identification into two character pointers. The program then checks the screen identification number against the screen table. If the screen has never been expanded then the identification data is stored in the table. The program then checks to see if the expansion of that particular line has been done before. If the expansion is 'new', then the program copies the file containing the expansion template to the screen and the expanded sentence is copied on the expanded screen to serve as a title and provide an additional logical link between screens. If the expansion has been done before, the program uses the data in the screen table to find its location in the screen_file and copies that version to the screen. The expanded sentence is written over the previous title because changes might have been made to the sentence at some point.

The user may input statements within the limits of the Warnier-Orr brace using a limited full-screen editor. Just as one would write statements to the right of a given Warnier-Orr brace when using pencil and paper, the editor behaves in much the same way. When a template appears on the screen, the program moves the cursor to the logical first line of the brace. The editor allows the cursor to be moved anywhere within the confines of the brace to input commands, make corrections, and expand lines.

To make the system as simple to use as possible, commands and other input have single stroke execution. There is one exception to this feature, and that occurs when the user is typing his/her name and the program title, which requires a <CR>. The single stroke limits the mnemonic value of the command to a degree, but single stroke execution is also used in the UNIX 'vi' editor so the intent was to limit the amount of new editing techniques that the user would need to learn. There is no 'look-ahead' buffer for input commands however. By not having that feature, it helps to force the user to carefully consider the next input command.

The program identifies each input template with a unique identification number which allows the screens to be perused and finally assembled into a Warnier-Orr diagram. The numbers begin with either 01, 02, 03, or 04 depending on whether the user is in the problem definition section, the input statement section, the program statement section or the output statement section. As the screens are expanded the identification number indicates the hierachy allowing for the screens to be examined in order. Perusal backwards is accomplished by truncating the last number and then searching the screen table for the identification number and the associated save file location of the screen in question. Because the screens have an ordered identification numbering system, the screens may be put in order to form the final Warnier-Orr.

The template identification is a composite numeral based on the order in which lines of templates are expanded. When a user elects to expand a line, the program determines the logical line on which the

cursor is positioned and appends a dot and the line number to the existing identification number. Thus the user can tell how many expansions a screen has undergone by noting how many times numbers have been appended to any base number. Additionally this provides the mechanism for giving the screen identifications their hierarchy of development. Since the base number will be 01, 02, 03, or 04 depending on the area being developed, it is a simple matter of sorting to put the program parts into their proper relationship in the final diagram.

The numbering system allows the main data structure which holds the screen identifications and locations to be sorted and the screens then assembled to form a Warnier-Orr-like diagram. The program performs a bubble sort on the structure records using the screen identifications as the key. Then with the screens in their proper order, the program copies each screen in order and appends it to a new file called 'Warnier'. When the last screen has been copied, the program calls the Stream Editor and appends the closing line for the last Warnier-Orr brace, finishing the diagram. Of course, the first two lines were written to the file earlier, those being the program title and the author's name.

Screen display, cursor movements, and many of the editor features utilize a collection of programs, collectively called 'curses'. This screen handling package is used to position and move the cursor, define windows for the screen, write information to the screen, write from the screen to a file, and other functions useful in developing programs to manipulate the screen. Curses provides several useful

standard variables, one of which is the standard screen, which was utilized exclusively in this program since use of this variable makes the screen functions behave in a very predictable manner. Many of the curses routines have been taken from various parts of the 'vi' editor, thus it is very compatible with the UNIX operating system.

Because this program is written in 'C', it is composed of a number of short functions totaling about 1300 lines of code. The functions can be classified according to the system requirements they support. The general modules are error interrupt handling functions, initialization functions, editor functions, tutorial functions, and formating functions. Many of the functions perform in several modules, thus the code is not arranged in clear boundary areas.

Summary

Using top-down development as a methodology for teaching programming and as a principle of software engineering has been shown to be valid by research carried out over the past ten years. These findings have important implications to the two areas since they are so closely tied in terms of the best way to approach programming. Software tools have been developed which emphasize top-down design and even generate code. No tool has yet been developed, however that can do the supporting creative thought processes necessary to develop programs. It is clear that the areas of software engineering and how persons acquire programming knowledge are fertile spheres for further research. This

tool fulfills the requirement for creative thought in program development and hopefully it will add to the knowledge of the better ways to approach the teaching of programming.

CHAPTER FIVE - CONCLUSIONS AND EXTENSIONS

This system should be useful as a general tool for initial program development system.

The expansion feature of this system and the ease of use should encourage students to develop programs in a top-down manner. The capability of expanding an instruction provides the user with a two-dimensional editor that allows for easy tracing of the program development. By using the expansion, the user has a simple way of producing a modularized algorithm which quite logically develops into functions and subroutines.

Warnier-Orr-style diagrams, such as this system produces, should have additional value as documentation for several phases of program development. The system encourages the user to rephrase the problem, which is included as part of the final diagram. Thus the diagram is a record of the problem from the definition phase to the algorithm development phase. This sort of development record should be an important component of the overall documentation of any project.

The system could be a valuable research tool if used in a classroom environment to help determine if graphic aids do aid certain types of learners and in what ways. Some studies have suggested that one category of students has difficulty in programming because they have difficulty imposing an ordering scheme on unorganized data. A system such as this should provide the sort of visual organization which those students need to solve a problem logically.

Currently the input screens and associated Warnier-Orr braces are of a common length. The program output would be improved if the braces were of proportional size according to the number of lines they contain.

Because of the storing scheme for screens, they are of necessity fixed length. The ability to insert and delete lines as an editor function would increase the flexibility of the system.

The program could be enhanced in future work by the addition of a code generator based on the developed algorithm. There are several types of code generator programs around but most of them produce code from some type of specialized command which requires some programming knowledge. A code generator developed in connection with this program would have the advantage of producing modularized code as well as well defined module interfaces.

Selected Bibliography

Akin 81 Akin, T.; and LeBlanc, Richard J. "The design and Implementation of a Code Generation Tool," Software-Practice and Experience, XI (July 1981), 1027-1041.

Balz 81 Balzer, Robert. "Transformational Implementation: An Example," IEEE Transactions on Software Engineering, SE-VII (January 1981), 3-11.

Benb 79 Benbasat, I; and Dexter, A.S. "The Human/Machine Interface for Problem Solving," 12th Hawaii International Conference on Systems Science, XIII (1979), 60-70.

Broo 80 Brooks, Ruven E. "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," Communications of the ACM, XXIII (April 1980), 207-213.

Chan 80 Chand, Donald R.; and Yadav, Surya B. "Logical Construction of Software," Communications of the ACM, XXIII (October 1980), 546-555.

Conr 84 Conrow, Kenneth; and Graham, Tom W. "A Language-Independent Program Design Utility", Kansas State University Computing Center Technical Paper, 1984.

Curt 83 Curtis, Bill. "A Review of Human Factors Research on Programming Languages and Specifications," AEDS Monitor, XXI (March/April 1983), 24-30.

Free 76 Freeman, Peter; Wasserman, Anthony I.; and Fairly, Richard E. "Essential Elements of Software Engineering Education," Second International Conference on Software Engineering, 1976, 116-122.

Hass 82a Hassell, Johnette. "Cognitive Style and a First Course in Computer Science: A Success Story," AEDS Monitor, XXI (July/August 1982), 33-35.

Hass 82b Hassell, Johnette; and Law, Victor J. "Tutorial on Structure Charts as Algorithm Design Tool," AEDS Monitor, XXI (September/October 1982), 17-32.

Houg 80 Houghton, Raymond C.; and Oakley, Karen A.(CDS). National Bureau of Standards Software Tools Database, Washington, D.C., 1980.

Hunt 82 Hunt, Earl B. Concept Learning, New York: John Wiley and Suns: 1982.

Jack 75 Jackson,, K.F. The Art of Solving Problems, New York: St. Martin's Press, 1975.

Jens 79 Jensen, Randall W.; and Tonies, Charles C. Software Engineering. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

McKe 81 McKeithen, K.B.; Reitman, J.S.; Rueter, H.H.; and Hirtle, S.C. "Knowledge of Organizational and Skill Differences in Computer Programmers," Cognitive Psychology (1981), 13.

Mill 56 Miller, G.A. "The Magic Number Seven, Plus or Minus Two," Psychological Review LXIII (1956), 81-97.

Mohe 82 Moher, Thomas; and Schneider, Michael G. "Methodology and Experimental Research in Software Engineering," International Journal of Man-Machine Studies. XVI (January 1982), 65-87.

Mosk 83 Moskowitz, Robert. "Three Program Generators," Popular Computing. II (March 1983), 146-150.

Myer 79 Myers, Ware (Editor). The Need for Software Engineering," Computer. II (February 1978), 12-25.

Nati 81 National Bureau of Standards. Proceedings of the NBS/IEEE/ACM Software Tool Fair. October 1981.

Radi 82 Rading, Howard; and Rautenberg, Lee. "Interactive Software Automatically Translates Tasks into Programs," Electronics. LV (August 25, 1982), 101-104.

Scan 77 Scandura, Joseph M. Problem Solving. New York: Academic Press, 1977.

Schn 82 Schneider, Michael G.; Weingart, Steven W.; and Perlman, David M. An Introduction to Programming and Problem Solving With Pascal. (2nd Edition), New York: John Wiley & Sons, 1982.

Simo 83 Simons, G.L. "Fifth-Generation Computers: Features and Research", Software World. XIV (1983), 5-9.

Stee 83 Steele, Anne C.; and White, Kathy B. "Systems Development: The Processing Stages," AEDS Journal. XVI (Spring 1983), 188-195.

Teit 81 Teitelbaum, Tim; and Reps, Thomas. "The Cornell Program Synthesizer: a Syntax- Directed Programming Environment," Communications of the ACM. XXIV (September 1981), 563-573.

Thar 80 Tharrington, J. "New Users Must Tackle Problem Analysis First," Computerworld. XIV (August 1980), 11.

Tria 79 Triance, J.M.; Edwards, B.J. "A Computer Aided Program Design Project," EUROPEAN IFIP. 1979, 621-625.

Warn 74 Warnier, Jean D. Logical Construction of Programs. New York: Van Nostrand Reinhold: 1974.

Wass 73 Wasserman, Anthony I. "The Design of 'Idiot-Proof' Interactive Programs", Proceedings, 1973 National Computer Conference, XLII, M34-M38.

Well 81 Wells, M.J.; and Rzevski, G. "Computer Aided Learning of Software Design," Computer Education, XXXIX (November 1981), 20-23.

Wexe 81 Wexelblat, Richard L. "The Consequences of One's First Programming Language," Software-Practice and Experience, XI (July 1981), 733-739.

Appendix A

User's Guide for *Algorithm Development Program*

Joseph K. Campbell

Abstract

This guide describes a system which allows a user to:

-peruse screens which provide tutoring problem definition and program development

-write program statements and refine them to desired detail by means of an 'expansion' feature which provides a new input area on command

-develop a stylized Warnier-Orr diagram which provides for easy tracing of program logic

-print the finished Warnier-Orr diagram if desired

-recover an existing Warnier-Orr diagram for modification.

This system was developed to provide the programmer with visual reinforcement while providing a methodology for encouraging top-down development. The visual reinforcement is accomplished by providing the user with input to a screen on which a facsimile of a Warnier-Orr brace is shown. The top-down development is encouraged by providing screens of instructions on the techniques and a facility for expanding a program statement till the desired detail is reached. Because the program provides so many internal instructions and help commands, very little advance study is necessary to be successful with the system.

ACCESS TO THE SYSTEM

The location of the commands and permissions necessary to run the program may be obtained from the Computer Science Department System Supervisor.

Starting Up

When the program starts the user is shown 'screen 0', displaying preliminary information about the use of the system. The screen contains a prompt asking if the user wishes to continue or not. It is important to mention here that input commands are single stroke execution. If the user attempts to input more than one command at a time the results are unpredictable, the worst case being an undignified exit from the program.

The next screen, 'screen 00', is the start of the initialization process. This screen prompts the user to categorize the session as a first time use, an experienced user constructing a new algorithm, or modification of an algorithm at some earlier session. The sentences are numbered and the user responds with the appropriate number.

Depending upon the choice made, the next screen will be 'screen 0A', 'screen 0B', or 'screen 0C', all of which prompt the user for more specific information needed to start the program. Also included in these screens is information about recovering existing algorithm files developed at at earlier date.

If the user indicates this is the first session, 'form 0A' is displayed and the program asks the user to supply his/her name and a title for the algorithm. When the name and title are entered a carriage control is needed to finalize the input. This is the only time in the program that a carriage return is needed to input a command.

If the user indicates that he/she is experienced in the use of the program and will be starting a new algorithm, 'form 0B' is displayed. This form has a warning message that any existing file named 'Warnier' will need to be renamed because the new algorithm being developed will destroy any existing file in the current directory with that name. If the user has already accomplished the renaming task, the program passes control for 'screen 0A' and the user proceeds with the new algorithm.

When the user indicates that the session will be used to modify some existing algorithm, 'screen 0C' will be displayed. This screen notifies the user that the existing 'Warnier' file must be essentially unchanged and tells the things that must be the same if the file is to be recovered correctly. The program then gives the user an opportunity to exit the program and put the file in the correct order. If the file is unchanged since the last session, the program recovers the file for access by the program.

GENERAL INSTRUCTION SCREENS

After these 'housekeeping' chores are finished, the program displays 'screen 1', which is the start of the general instruction screens. At this point the user has a choice of perusing the instruction screens one at a time using the 'N' command, or requesting a specific screen by pressing '2' - '6' for some specific screen. The screens provide information on how to approach the problem definition phase, a quick review of top-down design, and how to reach and use the input screens. When the user is viewing a particular screen and wishes to view some

previously displayed screen, then perusal may be done in reverse by use of the 'B' command. Note that the system distinguishes between upper and lower case commands.

Screens 2 through 6 provide the following information:

'screen 2' - Gives some general information about the program and how to use the (H)elp, (N)ext, (S)ave, and program exit commands.

'screen 3' - Provides the user with a diagram showing how the parts of the Warnier-Orr are labeled and their relationship to each other.

'screen 4' - Dispays a list of the input screen editor commands and the other display commands.

'screen 5' - This screen is a quick review of techniques for problem definitions.

'screen 6' - This is the 'home screen' from which the user branches into one of the four input screens. This is also the screen to which the user will eventually return if perusal is done in reverse order in the input expansion screens.

SPECIFIC HELP SCREENS

The specific help screens are intended to supplement the general direction screens by providing more detailed help in a variety of areas. Once the user has reached 'screen 1', then the help menu may be requested at any time by pressing 'H'. This moves program control to a 'help area' which means that the user may continue to request help screens as long as desired, including the main menu screen. Return to the screen from which the (H)elp request was made is accomplished by pressing 'r' or any key other than 'H' or '0'-'9'.

The specific help screens are viewed by requesting a menu number

listed in the main help screen. If the user is viewing some numbered help screen and wishes to see the menu again, all that needs to be done is press 'H'.

The specific help screens provide information as follows: '0' - Lists a summary of the editor commands and their functions.

'1' - Discusses some techniques for defining problems in understandable terms.

'2' - Introduces the part of the algorithm development which defines input statements and data definition.

'3' - Shows how the algorithm development program can be used to define output formats and values.

'4' - Writing program statements and using the expansion feature to refine them is introduced on this screen.

'5' - Detailed instructions on the use of the expansion feature are given on this screen along with a diagram showing how the screen numbering system indicates the development hierarchy. The screen also discusses perusing forward through finished expansion screens.

'6' - This screen shows a small hierarchical path of expanded screens and how to peruse screens in reverse order.

'7' - The program will write the syntax of 'while' and 'for' loops on input screens on command. This screen shows how they appear on the screen and the commands necessary to display them.

'8' - This screen has a diagram illustrating how the Warnier-Orr diagram arranges the expanded screens for printing and the command necessary to start the ending sequence.

'9' - Gives directions on how to reach the various starting screens for problem definition, input statements, program statements, and output statements.

USING THE PROGRAM DEVELOPMENT SCREENS

Choosing the Appropriate Input Area

The program provides input screens for algorithm development in four different sections. These screens may be reached by pressing 'D' for the problem definition section; 'I' for the input section; 'P' for the

program statement section; and 'O' for the output section. Once the user has progressed at least to 'screen 2', any of the input screens may be requested by pressing the appropriate key.

Input and Editor Commands for the Input Screens

Once the user has an input screen displayed, the appropriate program statements may be entered to the right of the Warnier-Orr brace shown on the screen. The cursor will be positioned on the first logical line of the brace. The program will not allow the cursor to move anywhere but within the confines of the brace. To actually write characters to the screen press 'i' for input mode, 'esc' key will return from input.

Movement around the input area is accomplished by using the 'h' for left , 'j' for down, 'k' for up, and 'l' for right movements. The backspace key and space bar also provide horizontal movement.

Corrections may be made to existing text in by several simple commands. Insertion in a line is accomplished by pressing 'a', which inserts text in a line and moves the ending text to the right at the same time. To escape from 'a' or append, press the 'esc' key. To delete text one character at a time, press 'x'. To delete a line from the cursor position, press 'z'. Note that these commands are quite similar to the UNIX 'vi' commands.

The program will display the format of a 'while' or 'for' loop on command if there are sufficient lines remaining on the Warnier-Orr brace to do so. To display the 'while' format, press 'w' and 'f' to

display the for loop. These loops are displyed on the cursor position line starting at the first position of the line. Any existing text on the line is destroyed.

Using the Statement Expansion Feature

The expansion feature allows a user to refine any statement to some desired level of detail. This is accomplished by providing a new input screen on command which has a clear link to the 'expanded' statement. A new input screen for a logical line is provided when the user presses 'E' and the cursor is on the line to be expanded. The link between the expanded sentence and the new input screen is established in two ways. The expanded sentence appears as the title of the new input screen near the top of the screen.

The other link is established by the identification number of the screen which is in the upper left hand corner of the screen. This number is directly related to the number of the expanded line. The identification number is constructed by appending the number of the expanded line to the existing identification number. Thus a number such a 01.03.04 shows that the base screen '01', which is the problem definition section, has been expanded two times. The first time line 03 was expanded and then line 04 of that expansion screen was itself expanded. Expansion of any particular line may be done to seven levels. Any line of a Warnier-Orr brace may be expanded in any particular order. The system will assemble them in the correct sequence at the end of the session.

Perusal of the Screens

Perusal may be done in any order. That is, 'forward' or in reverse order. To peruse forward requires the use of three commands, depending on the part of the program in execution. The 'E' command may be used to peruse the instruction screens up to 'screen 6'. At this point the system must be told which input section to display, by pressing 'D', 'I', 'P', or 'O'. After pressing one of these, the forward perusal command is exactly like the (E)xpansion command. To view the desired expansion screen, position the cursor on the appropriate sentence and press 'E'. If that line was previously expanded, the screen will display the old expansion, otherwise a new screen comes on as before. Any changes the user happens to make while perusing will automatically be saved.

Perusal backward is accomplished by using the (B)ack command. Pressing 'B' will display screens in reverse order. If the user is in any of the input screen sections the system will peruse the screens in reverse order, following the order indicated by the screen identification number. When reverse perusal causes 'screen 6' to be displayed, further reverse perusal will display the instruction screens in reverse order. Again, any changes which the user happens to make while perusing will automatically be saved.

Perusing the screens is exactly like perusing a 'tree structure', thus to reach certain 'branches', the user must peruse to the 'base' of the 'tree' and then back out to the desired 'branch'. In this analogy, the 'base' would be 'screen 6', and the 'branches' are the expansion

screens. Perusal and expansion may be made in any order at any time.

FINISHING UP

The Stop Command

After the user is finished with the algorithm or simply wishes to terminate the session, then the (S)top command is used. When the user presses 'S' the system will prompt the user for necessary further commands. It is extremely important not to issue any commands till the the system requests them.

Storing and Printing the Warnier-Orr Diagram

The system assembles the input screens in the correct order to form a Warnier-Orr diagram with a vertical alignment (see figure 3). The user may elect to have a hard copy of simply exit the program. The finished diagram is stored in the user's current directory under the name 'Warnier'. That file may be examined just a one would look at any other file and may even be modified, however the user must be careful not to add or delete any lines or change any identification numbers.

EARLY EXIT FROM THE PROGRAM

The program may be exited at any point in the program by pressing 'Q' or the 'DEL' key. Use of these two exit methods should only be used when the user desires not to save any file. These exit methods destroy any work accomplished during the current work session, thus they should be used with extreme care.

Appendix B

Programmer's Guide for Algorithm Development Program

Joseph K. Campbell

Abstract

This guide describes a system a user to:

-peruse screens which provide tutoring problem definition and program development

-write program statements and refine them to desired detail by means of an 'expansion' feature which provides a new input area on command

-develop a stylized Warnier-Orr diagram which provides for easy tracing of program logic

-print the finished Warnier-Orr diagram if desired

-recover an existing Warnier-Orr diagram for modification.

OVERVIEW

This system is a form display program which accepts input to specific areas on the screen. The main data structures are the external supporting files, a table which keeps track of how screens are stored in a file, and a group of global variables that coordinate the activities performed on the data structures. A key feature of the program is its capacity to 'expand' a given input line such that a new input screen appears that is conceptually linked to the input line. The system also supports a small editor with full screen capability. When the user indicates he/she is ready to stop, the program sorts the screen table and constructs a Warnier-Orr diagram, stores it in a file named 'Warnier', and prints a copy if the user so desires.

FILE SUPPORT SYSTEM

The files which support the program are those which permanently reside in the directory and those created by the program.

Permanent Files

The permanent files are those which display information to the user and files that will be used for input. All of the support files are of a fixed length, exactly twenty-three lines. The files are of fixed length to simplify the storage of completed input files which will later form the Warnier-Orr diagram. The program uses the same function, called 'filecopy' to copy files from the file system to the screen. This function is called from some function which opens a specific named file, then passes a file pointer to 'filecopy'. Note that when characters are written to the screen, a function called 'addch' must be used. If some other function is used, when the screen is subsequently copied to a file and later recopied to the screen, an error is generated by the system.

Created Files

The program creates three files while in execution. The first is the file, called 'scrnfile', in which the input screens are stored. The second is a file in which the current screen is stored if the user requests to view help screens. The last is really a permanent file, named 'Warnier', in which the completed algorithm is stored and placed in the current directory.

SCREEN TABLE

The screen table is a global array of 'structures' which is much like an array of records in Pascal. An individual structure contains two elements, a character pointer, 'expid', and an integer, 'expos', which in this case is a special 'C' language integer of type 'long'.

In 'C' pointers have some very special capabilities and some unexpected restrictions. A pointer may be incremented in 'C' which in effect gives the user a dynamic array. However, one must be careful to allocate new memory for each pointer used, especially in a structure, or 'C' will write all information to a common address. What this means to the user is that if the pointers in a structure must have new memory allocated for each one. If this is not done, every time new data is put in a pointer, all the pointers will be changed to the same thing.

The long integer is used to keep track of where a screen is stored in the file. The number indicates the beginning position of the first byte of the stored screen. 'C' has a function called 'fseek' by which the user may position the file pointer anywhere in a file, then read from and write to the file. Thus a file may be treated much like an array but with slightly slower access.

GLOBAL VARIABLES

The global variables are used to control the interaction of various functions and to control limits on certain data structures. The global

variables and their properties are:

1. Screen Control Variables (integer)

   lines - the permanent record of vertical cursor position

   cols - the permanent record of horizontal cursor position

   xp1 - the left limit of horizontal cursor movement

   xp2 - the right limit of horizontal cursor movement

   yp1 - the upper limit of vertical cursor movement

   yp2 - the bottom limit of vertical cursor movement

2. Table Pointers (integer)

   delta - the amount of bytes added to 'expos' of the screen table each time a screen is added to the screen file

   newexpos - the starting position for saving screens in the screen file

   curpos - the starting position of a screen to be copied from the screen file or an updated screen overwriting a previously stored screen

   lastpos - the starting position of the last screen in the screen file

3. Miscellaneous Global Variables (integer)

   true - used to simulate a boolean true condition

   false - used to simulate a boolean false condition

   nestlvl - the memory block size to be allocated for each new character pointer

   btriger - tells the system if a screen file has been created during this session so it may be erased at the conclusion of the program

   form - tells the program which screen is being displayed and when the user has reached the input screens

   explvl - informs the system of how many expansions an instruction has undergone

   goingback - informs the system that perusal to the home screen is coming from the input screens or the information screens

   cammvcur - cursor can only be moved around the screen and

editor commands used when displaying the input screens

done - a simulated boolean which is set as a condition for the program to exit the command interpreter function and exit
the program.

SYSTEM OPERATION

Initialization Operations

The initialization routines require some user input so that the program can determine if a previous file needs to be opened or a new one created.

The scenario when a new file is to be created is as follows:

The user responds to a question which indicates that this session is to be used for a new algorithm. After opening a file named 'scrnfile', the system displays a screen with two input zones, accepts the inputs and stores the information in character pointers. The information is displayed for the user to check and if correct the program uses a special function which appends the pointers to 'scrnfile'.

If the session is to be used to update and existing file the actions are:

The system opens the files 'Warnier' and 'scrnfile', calls the Stream Editor (SED) to erase the last line of 'Warnier', which is

the closure of the last brace, transfers the remaining file to 'scrnfile', and removes 'Warnier'. The program then uses 'fseek' and 'ftell' to determine the length of the file. The file pointer is then positioned at the start of the first stored screen, which can only be after the two header lines.

The program then displays each stored screen so that the necessary information may be extracted from each one and the data stored in the table structure. The data to be extracted are the screen identification and the beginning position of the screen in the storage file. The screen identification is read into a character pointer by a function called 'mkstr', which copies the screen id, allocates new memory, and calls a function to store the pointer in the table structure. As each screen is copied the structure integer 'expos' is incremented by 'delta' thus the program knows the starting position of each screen in the file as well as its identification number.

Displaying the Information Screens

After the initialization process the program displays 'screen one' and transfers control to the command interpreter function, 'homecopy'. This function interprets all subsequent commands from the keyboard unless the user has asked for help by pressing 'H'. From 'screen 1' the user can give the command 'N'. This command increments the global 'form', and displays the file corresponding to the value of 'form'.

Displaying the Input Screens The 'N' command may be used to reach

'screen 6'. This is a special screen because it serves as the home screen for the input screens which are identified by the function they perform in building a program. There are four ways in which to progress once 'screen 6' is reached. The user may press 'D' for the problem definition sequence, 'I' to develop input statements, 'P' to write and expand program instructions, or 'O' to structure output. In any case, the behavior of the program is exactly the same for all areas, thus we will examine the 'D' sequence.

When the user presses 'D', the system opens the appropriate file and copies it to the screen. The expansion level global 'explvl' is incremented so that the cursor may now be moved and editor commands used. The cursor screen limits are defined and the cursor positioned on the screen. The display template on the screen is a likeness of a Warnier-Orr brace, and input is allowed to the right of the symbol. (see the section of this guide titled 'Editor Commands') The user writes statements to develop the algorithm on the screen.

If any statement needs to be amplified, the user presses 'E' while the cursor is on the line and the program displays a new input screen. The steps leading to a new screen display are as follows:

1. The screen identification is copied into a character pointer and this is compared to the screen identifications in the table structure to see if the current screen has ever been saved. If it has never been saved, the screen identification is put in the table and the screen appended to the end of 'scrnfile'. If it was previously saved, the position of the

starting location is noted during the comparison and the old saved version is overwritten.

2. The number of the line being expanded is then appended to the current screen identification to form the expanded screen identification. This number is then checked to see if it is 'new' or 'old' just as was done in the first step. The sentence which is to be expanded is also copied into a character pointer. If the line has never been expanded, the system opens one of the external files 'forme' which is a Warnier-Orr template with the brace offset to the right to indicate a subordinate brace to the one displayed when 'D' was pressed. This file is copied to the screen and the new identification number and expanded sentence are also copied to the screen.

   If the line has been expanded before, the program searches 'scrnfile' and copies it to the screen. To finish the display, the expanded sentence is copied to the screen, overwriting any previous sentence that was displayed.

Each time a line is expanded, the operations involved in the original expansion sequence are repeated. Thus a given line may be expanded to any level of detail allowed by the program. The global variable 'nestlvl' controls how many expansions may be done on a given line. This global integer is the size of memory requested from the 'C' function 'malloc'. Each expansion increases the character length of the identification number by three (a '.' and the number of the line

being expanded), so to determine the number of expansions allowed, the nestlvl is divided by three. Currently the program allows seven expansions of any one line.

Perusal of the Screens

Perusal of the screens may be done in any direction. The perusal functions are different depending on what sections are in execution. Perusal of the information screens is done by the program examining the current status of the global variable 'form', then opening and displaying the external file corresponding to the value.

Perusal of the input screens in a 'forward' manner is accomplished by using the same functions as the 'E' command explained earlier. This has the added advantage of automatically saving any changes that may have been done.

Perusal in reverse involves several steps and always uses the command 'B'. First the current screen identification number is compared to those in the screen table. If it has never been saved, it appended to the end of 'scrnfile'. If it was saved previously, the starting position is noted, the file pointer is positioned by 'fseek', and the old screen is overwritten. Again, by using this procedure any changes made will automatically be saved.

To located the next screen backwards in the hierarchy, the current screen identification is truncated by the three end characters and the result from compared to the identification numbers in the screen

table. When the identification number is matched, the position of the screen in 'scrnfile' is noted, and 'fseek' positions the read pointer in the file. The twenty-three lines comprising the screen are displayed and the process is repeated for each 'B' command.

When the backward perusal reaches the screen with the identification number '01' (or '02', '03', or '04) the global variable 'explvl' will have reached zero. This is a signal to the program that further backward perusal will be based on the value of 'form'. The program uses the global variable 'goingback' to determine from which direction the home screen is being approached.

The user may also peruse directly back to the '01', '02', '03', or '04' starting screens by pressing 'D', 'I', 'P', or 'C'. The command interpreter function calls the functions necessary to save the current input screen, then locates the appropriate screen in 'scrnfile' if it has been used. If the particular starting screen has not yet been used, the program opens the external file and copies it to the screen.

Storing the input screens

Several 'C' functions make it possible to store the screens quite easily. The function 'fseek' will move the file pointer to anywhere one desires in a file, and reading or writing may begin at that point. The pointer may be positioned at the beginning, end, or at byte location in the file. The 'C' function 'rewind' should be used before each seek to insure proper positioning in the file. Another handy

function is 'ftell' which returns the byte position of the pointer.

Actual storage of the files is done in two ways. By comparing the screen identification to the screen table, one can determine if the screen was ever saved. If it is a 'new' file, 'fseek' positions the pointer at the end of the file, and the screen is appended.

If the screen is 'old', the location of the screen is noted during the comparison to the screen table identifications, and 'fseek' positions the file pointer at the starting point of the screen. Then a fixed number of lines are either read to, or from the file.

Construction of the Warnier-Orr Diagram

The input screens display a template which is a stylized Warnier-Orr brace. When the user has completed a session, the 'S' command causes a series of functions to be activated which produces a Warnier-Orr diagram.

First the program checks to see if the current screen has been saved and opens the permanent file 'Warnier'. Next the screen table structures are sorted in ascending order keying on the screen identifications. Then the program copies the first part of 'scrnfile' to the screen and the first two lines, containing the title and programmer's name, are read and appended to 'Warnier'. The screens are transferred to 'Warnier' in hierarchical order by examining each structure in the screen table in order, locating the screen, and appending it to 'Warnier'. After the transfer is complete the user is

asked is a hard copy is desired. If so the program uses a 'C' system call to print the file.

FUNCTIONING OF THE HELP COMMAND

The help command is part of the operation of the program, but the commands involved are interpreted by another function, thus the discussion is separate. When the user requests help by pressing 'H', the system opens a temporary file and stores the current screen in the file. Then control is transferred to a function called 'helparea' and an external file containing the menu of help commands is displayed. There are ten external files of specific help hints and instructions which the user can display by pressing '0' - '9'. These may be viewed over and over because the program remains in 'helparea' till the user presses 'r'.

When the user is through and has pressed 'r', the temporary file is opened and the current screen is again displayed. The temporary file is then closed and removed from the system.

ERROR INTERRUPTS

The program 'catches' the error interrupts as they occur and allow a graceful exit from the program. These error routines are located in the source code near the start of the program.

MODIFICATION OF THE PROGRAM

Several properties of the program would probably require modification to fit local needs. If more expansion levels are required, the global variable 'nestlvl' would need to be changed. Its current value is 21, which means that the most expansions of any one line is limited to seven.

The total number of expansions of all screens is determined by the length of the structure array. This is determined by 'maxlen', which is currently defined as 250.

The basis for most modification would include changing the function 'homecopy' in some way. This is the command interpreter which is a loop that may be exited by going through the finish sequence or by pressing 'Q' or 'del'. These last two commands cause the program to exit without any file being saved in any form.

SPECIAL PRECAUTIONS TO BE NOTED WHEN USING 'C' AND 'CURSES'

When using character pointers to store a character string, be sure to malloc new memory for the anticipated size of the character string. As mentioned, 'C' will write information to a common address if character pointers are used for the same task, such as storing character strings in a structure.

'C' also supports a file access function called 'seek', but 'fseek' should always be used when the file access pointer is a character.

Files should always be closed after an operations such as 'read' and

then reopened for a 'write'. This is common sense but easily forgotten when constructing programs.

The function 'rewind' should be used prior to any 'fseek' to insure one is starting from the beginning of the file.

The position used by 'fseek' is in bytes. Note that a terminal line contains 81 bytes, so to position at a desired line multiply the desired line by 81.

Appendix C

Source Code

```
/* THE INCLUDES FOR I/O, INTERRUPTS
   AND SCREEN HANDLING */
#include <stdio.h>
#include <signal.h>
#include <curses.h>

/* THE GLOBAL VARIABLES */

/* number of screens possible */
#define maxlen 250
/* Fake booleans */
#define true 1
#define false 0
/* screen size in bytes */
#define delta 1863
/* determines how many times a screen can be directly */
/* expanded. number = nestlvl div 3                   */
#define nestlvl  21
/* Tells program an expansion has occured */
int btriger = false;
/* Global cursor position */
int lines,cols;
/* The protected areas of the screen  */
int xp1 = 0;
int yp1 = 0;
int xp2 = 79;
int yp2 = 23;
/* The current info screen */
int form;
/* the current number of expansions of the display screen */
int explvl;
/* The screen table */
   {
struct explocs
/* The identification of the screen */
   char *expid;
/* Sets terminal characteristics for curses */
initscr();
```

```
/* Error interrupt signals */
signal(SIGINT,die);
signal(SIGSEGV,die2);
signal(SIGBUS,die3);
signal(SIGILL,die4);
/* Single stoke */
crmode();
/* Don't echo input */
noecho();
findstart();
homecopy();
if (btriger == true)
   system("rm scrnfile");
wclear(stdscr);
mvcur(0,LINES,0,COLS);
wrefresh(stdscr);
endwin();
exit(0);
}

/* THE ERROR INTERRUPT ROUTINES */

die()
{
signal(SIGINT,SIG_IGN);
if (btriger == true)
   system("rm scrnfile");
wclear(stdscr);
wrefresh(stdscr);
printf("Died due to signal interrupt.\n");
mvcur(0,LINES,0,COLS);
endwin();
exit(0);
}

die2()
{
signal(SIGSEGV,SIG_IGN);
system("rm scrnfile");
mvcur(0,COLS-1,LINES-1,0);
printf("Died due to segmentation violation.\n");
endwin();
exit(0);
}
die3()
{
signal(SIGBUS,SIG_IGN);
system("rm scrnfile");
mvcur(0,COLS-1,LINES-1,0);
printf("Died due to bus error.\n");
endwin();
exit(0);
}

die4()
```

```
{
signal(SIGILL,SIG_IGN);
system("rm scrnfile");
mvcur(0,COLS-1,LINES-1,0);
printf("Died due to illegal instruction.\n");
endwin();
exit(0);
}

/* THE INITIALIZATION ROUTINES */

findstart()
{
form0();
next();
form00();
/* up to now the preliminary inst. screens are displayed */
form1();
findfirst();
}


form0()
{
FILE *fp, *fopen();
fp = fopen("/usr/joseph/project/form0","r");
filecopy(fp);
fclose(fp);
}

next()
{
char c;
c = getc(stdin);
switch(c){
  case 'y' : return;
             break;
  case 'n' : die();
             break;
           }
}

form00()
{
char c;
FILE *ff, *fopen();
ff = fopen("/usr/joseph/project/form00","r");
filecopy(ff);
fclose(ff);
c = getc(stdin);
switch(c){
  case '1': form0A();
            break;
  case '2': form0B();
            break;
```

```
  case '3': form0C();
            break;
  default : form00();
          break;
           }
}

/* RECOVERY ROUTINES */

form0A()
{
char c;
char *title;
char *name;
char *response;
FILE *fa, *fopen();
fa = fopen("/usr/joseph/project/form0A","r");
response = "Is that correct? (y/n)";
title = malloc(70);
name = malloc(70);
filecopy(fa);
fclose(fa);
cols = 5;
lines = 12;
nonl();
wmove(stdscr,lines,cols);
wrefresh(stdscr);
title = heading(title);
cols = 5;
lines = 17;
name = heading(name);
mvwaddstr(stdscr,19,0,title);
mvwaddstr(stdscr,20,0,name);
mvwaddstr(stdscr,22,0,response);
wrefresh(stdscr);
c = getc(stdin);
switch(c){
  case 'y': puttitles();
          break;
  case 'n': form0A();
          break;
  default : form0A();
          break;}
}

char *heading(ff)
char *ff;
{
char i = 0;
int c;
wmove(stdscr,lines,cols);
wrefresh(stdscr);
while (((c = getc(stdin)) != '\015') && (cols <= 67)){
    switch(c){
        case '\010' : wmove(stdscr,lines,cols-1);
```

```
                wrefresh(stdscr);
                cols--;
                i--;
                break;
          default: mvwaddch(stdscr,lines,cols,c);
                   wrefresh(stdscr);
                   ff[i] = c;
                   i++;
                   cols++;
              break;}
                 }
ff[i] = '\0';
return(ff);
}

puttitles()
{
FILE *fg, *fopen();
fg = fopen("scrnfile","a");
lines = 19;
writeoneline(fg);
lines = 20;
writeoneline(fg);
fclose(fg);
}

writeoneline(fg)
FILE *fg;
{
int c;
int i;
for ( i = 0; i <= 79; i++){
   wmove(stdscr,lines,i);
   c = winch(stdscr);
   putc(c,fg);
   if (i == 79)
     putc('\n',fg);}
}


form0B()
{
char c;
FILE *ff, *fopen();
ff = fopen("/usr/joseph/project/form0B","r");
filecopy(ff);
fclose(ff);
c = getc(stdin);
switch(c){
     default: form0A();
              break;
              }
}

form0C()
```

```
{
FILE *ff, *fopen();
FILE *fg, *fopen();
char *fi;
char c;
long end;
long i = 162;
ff = fopen("/usr/joseph/project/form0C","r");
filecopy(ff);
fclose(ff);
c = getc(stdin);
switch(c){
    default: break;}
system("sed  -f /usr/joseph/project/cmdfile Warnier > scrnfile");
system("rm Warnier");
fg = fopen("scrnfile","r");
fseek(fg,0,2);
end = ftell(fg);
rewind(fg);
while(i < end){
   fseek(fg,i,0);
   filecopy(fg);
   fi = malloc(nestlvl);
   fi = mkstr(fi);
   scrndata[lstpos].expid = fi;
   scrndata[lstpos].expos = i;
   rewind(fg);
   lstpos++;
   i += delta;
   newexpos = i;
      }
rewind(fg);
fclose(fg);
}

form1()
{
FILE *fp, *fopen();
fp = fopen("/usr/joseph/project/form1","r");
filecopy(fp);
fclose(fp);
return;
}



findfirst() /* locate the desired instruction form 2-6 possible */
{
char d;
d = getc(stdin);
switch(d){
   case '2' : form = 2;
           route();
           break;
   case '3' : form = 3;
```

```
            route();
            break;
   case '4' : form = 4;
            route();
            break;
   case '5' : form =5;
            route();
            break;
   case '6' : form = 6;
            explvl = 1;
            route();
            break;
   default :  form =2;
            route();
            break;
            }
}

/* THE COMMAND INTERPRETER */

homecopy()
{
char c;
int i;
crmode();
noecho();
while ((c = getc(stdin))  != 'Q' && done == false){
switch(c) {
   case 'B' : goback();
            break;
   case 'E' : if (explvl > 1)
          expand();
              break;
   case 'H' : helparea();
            break;
   case 'N' : goahead();
            break;
   case 'i' : if (canmvcur = true)
               inchar();
            break;
   case ':  /* backspace key symbol <control> <H> but nonprintable */
   case 'h' :

   case 'j' :

   case 'k' :

   case 'l' : if (canmvcur = true)
                  optmove(c);
            break;
   case 'D' : formD();
              break;
   case 'P' : formP();
          break;
   case 'I' : formI();
```

```
          break;
   case '0' : form0();
              break;
   case 'S' : finishup();
             break;
   case 'x' : if (explvl >= 2)
            delchar();
          break;
   case 'a' : if (explvl >= 2)
            apendch();
               break;
   case 'z' : if (explvl >=2){
            wclrtoeol(stdscr);
            wmove(stdscr,lines,cols);
            wrefresh(stdscr);}
                break;
   case 'w' : putwhile();
            break;
   case 'f' : putfor();
            break;
   default  : break;
   }
}
}

/* THE EDITOR FUNCTIONS */

inchar()
{
char d;
nonl();
while ((d = wgetch(stdscr)) != '\033')
  {

     switch(d){

        case '\010' : if (cols == xp1) {
                  system("beep");
                  break;}
                        else {
                    wmove(stdscr,lines,cols-1);
                    wrefresh(stdscr);
                    cols--;
                    break; }
        case '\015' : if (lines < yp2) {
                    wmove(stdscr,lines+1,xp1);
                    wrefresh(stdscr);
                             cols=xp1;
                    lines++;
                    break; }
                else system("beep");
           default:
           if (cols < xp2) {
                     mvwaddch(stdscr,lines,cols,d);
                  wrefresh(stdscr);
```

```
                    cols++;
                break; }
                  else system("beep");
            }
        }
}

char *delchar()
{
char *ff;
int i;
int delcol;
delcol = cols + 1;
wmove(stdscr,lines,delcol);
for (i=0; i<=(xp2-cols); i++){
   ff[i] = winch(stdscr);
   delcol++;
   wmove(stdscr,lines,delcol);
   }
ff[i]='\0';
wmove(stdscr,lines,cols);
wclrtoeol(stdscr);
mvwaddstr(stdscr,lines,cols,ff);
wmove(stdscr,lines,cols);
wrefresh(stdscr);
}

apendch()
{
char *fp;
char c;
int i;
int inscols;
int startcol=cols;
int countch = 0;
int addit = false;
while((c = wgetch(stdscr)) != '\033'){
   inscols  = cols;
   if (((xp2 - inscols) > 0) || (c == ') || (c == '\015')){
      for(i = 0; i <= (xp2 - cols); i++){
      fp[i] = winch(stdscr);
      inscols++;
      wmove(stdscr,lines,inscols);}
      fp[i] = '\0';
      if (countch > 0){
         wmove(stdscr,lines,cols);
         wclrtoeol(stdscr);}
      if ((c  == ') || (c == '\015')){
         if (countch > 0){
            mvwaddstr(stdscr,lines,cols-1,fp);
            cols--;
            countch--;}
            else
            system("beep");}
      else{
```

```
            mvwaddch(stdscr,lines,cols,c);
          mvwaddstr(stdscr,lines,cols+1,fp);
          countch++;
          if (cols < xp2)
             cols++;
             else
             system("beep");}
          wmove(stdscr,lines,xp2);
          wclrtoeol(stdscr);
          wmove(stdscr,lines,cols);
          wrefresh(stdscr);
                          }
      else
          system("beep");
          }
}

optmove(c)
char c;
{
switch(c){
  case 'k' : up();
             break;
  case 'j' : down();
             break;
  case ':        /* a control H is here but non-printable */

  case 'h' : left();
             break;
  case 'l' : right();
             break;
  default  : break;
  }
}

up()
 {
 if (lines > yp1 ){
    wmove(stdscr,lines - 1,cols);
    wrefresh(stdscr);
    --lines;
    }
 else
    {
    system("beep");
    mvcur(lines,cols,lines,cols);
    }
 }

down()
 {
 if (lines < yp2){
   wmove(stdscr,lines+1,cols);
    wrefresh(stdscr);
    ++lines;
```

```
    }
else
    {
    system("beep");
    mvcur(lines,cols,lines,cols);
    }
}

right()
{
if (cols < xp2){
    wmove(stdscr,lines,cols + 1);
    wrefresh(stdscr);
    ++cols;
    }
else
    {
    system("beep");
    mvcur(lines,cols,lines,cols);
    }
}

left()
{
if (cols > xp1){
    wmove(stdscr,lines,cols - 1);
    wrefresh(stdscr);
    --cols;
    }
else
    {
    system("beep");
    mvcur(lines,cols,lines,cols);
    }
}

putwhile()
{
char *fw;
char *fb;
fw = "while(                                ) do";
fb = "begin";
mvwaddstr(stdscr,lines,xp1+1,fw);
mvwaddstr(stdscr,lines+1,xp1+3,fb);
wmove(stdscr,lines,cols);
wrefresh(stdscr);
}

putfor()
{
char *ff;
char *fb;
ff = "for (         ):= (          )to(         )";
fb = "begin";
mvwaddstr(stdscr,lines,xp1+1,ff);
```

```
mvwaddstr(stdscr,lines+1,xp1+3,fb);
wmove(stdscr,lines,cols);
wrefresh(stdscr);
}

/* HELP FACILITY */

helparea()
{
FILE *ff, *fopen();
FILE *fg, *fopen();
char c;
int end = false;
fg = fopen("saveit","w");
writefile(fg);
fclose(fg);
ff = fopen("/usr/joseph/project/help","r");
filecopy(ff);
fclose(ff);
while (((c = getc(stdin))  != 'r') && (end != true)){
   switch(c){
   case '0': ff = fopen("/usr/joseph/project/help0","r");
          filecopy(ff);
             fclose(ff);
          break;
   case '1': ff = fopen("/usr/joseph/project/help1","r");
          filecopy(ff);
             fclose(ff);
          break;
   case '2': ff = fopen("/usr/joseph/project/help2","r");
          filecopy(ff);
          fclose(ff);
          break;
   case '3': ff = fopen("/usr/joseph/project/help3","r");
          filecopy(ff);
          fclose(ff);
          break;
   case '4': ff = fopen("/usr/joseph/project/help4","r");
          filecopy(ff);
          fclose(ff);
          break;
   case '5': ff = fopen("/usr/joseph/project/help5","r");
          filecopy(ff);
          fclose(ff);
          break;
   case '6': ff = fopen("/usr/joseph/project/help6","r");
          filecopy(ff);
          fclose(ff);
          break;
   case '7': ff = fopen("/usr/joseph/project/help7","r");
          filecopy(ff);
          fclose(ff);
          break;
   case '8': ff = fopen("/usr/joseph/project/help8","r");
          filecopy(ff);
```

```
            fclose(ff);
            break;
    case '9': ff = fopen("/usr/joseph/project/help9","r");
            filecopy(ff);
            fclose(ff);
            break;
    case 'H': ff = fopen("/usr/joseph/project/help","r");
            filecopy(ff);
            fclose(ff);
            break;
    default : end = true;
            break;}
    break;
    }
    fg = fopen("saveit","r");
    filecopy(fg);
    fclose(fg);
    wmove(stdscr,lines,cols);
    wrefresh(stdscr);
    system("rm saveit");
    }

/* SCREEN DISPLAY FUNCTIONS */

route()
{
switch(form) {
  case 1    : form1();
            break;
  case 2    : form2();
            break;
  case 3    : form3();
            break;
  case 4    : form4();
            break;
  case 5    : form5();
            break;
  case 6    : form6();
            break;
  default   : break;

  }
}

form2()
{
FILE *fq, *fopen();
cammvcur = false;
fq = fopen("/usr/joseph/project/form2", "r");
filecopy(fq);
fclose(fq);
}

form3()
{
```

```
FILE *fr, *fopen();
fr = fopen("/usr/joseph/project/form3", "r");
cammvcur = false;
filecopy(fr);
fclose(fr);
}

form4()
{
FILE *fs, *fopen();
fs = fopen("/usr/joseph/project/form4", "r");
cammvcur = false;
filecopy(fs);
fclose(fs);
}

form5()
{
FILE *ft, *fopen();
ft = fopen("/usr/joseph/project/form5", "r");
cammvcur = false;
filecopy(ft);
fclose(ft);
}

form6()
{
char *fg;
FILE *ft, *fopen();
if (goingback == true){
   ft = malloc(nestlvl);
   ft = mkstr(ft);
   doisave(ft);}
goingback = false;
explvl = 1;
ft = fopen("/usr/joseph/project/form6","r");
cammvcur = false;
filecopy(ft);
fclose(ft);
}

/* THE INPUT SCREEN DISPLAYS */

formD()
{
FILE *fp, *fopen();
char *fq;
char *fc;
int i;
btriger = true;
cammvcur = true;
if (explvl > 2) {
    fc = malloc(nestlvl);
    fc = mkstr(fc);
    doisave(fc);}
```

```
fq = "01\0";
explvl = 2;
i = chkstr(fq);
if (i == 0){
   fp = fopen("scrnfile","r");
   fseek(fp,scrndata[curpos].expos,0);}
else
   fp = fopen("/usr/joseph/project/formD","r");
filecopy(fp);
rewind(fp);
fclose(fp);
startcur();
}

formI()
{
FILE *fp, *fopen();
char *fq;
char *fc;
int i;
btriger = true;
cammveur = true;
if (explvl > 2) {
    fc = malloc(nestlvl);
    fc = mkstr(fc);
    doisave(fc);}
fq = "02\0";
explvl = 2;
i = chkstr(fq);
if (i == 0){
   fp = fopen("scrnfile","r");
   fseek(fp,scrndata[curpos].expos,0);}
else
   fp = fopen("/usr/joseph/project/formI","r");
filecopy(fp);
rewind(fp);
fclose(fp);
startcur();
}

formP()
{
FILE *fp, *fopen();
char *fq;
char *fc;
int i;
btriger = true;
cammvcur = true;
if (explvl > 2) {
    fc = malloc(nestlvl);
    fc = mkstr(fc);
    doisave(fc);}
fq = "04\0";
explvl = 2;
i = chkstr(fq);
```

```
if (i == 0){
   fp = fopen("scrnfile","r");
   fseek(fp,scrndata[curpos].expos,0);}
else
   fp = fopen("/usr/joseph/project/formP","r");
filecopy(fp);
rewind(fp);
fclose(fp);
startcur();
}

form0()
{
FILE *fp, *fopen();
char *fq;
char *fc;
int i;
btriger = true;
cammvcur = true;
if (explvl > 2) {
    fc = malloc(nestlvl);
    fc = mkstr(fc);
    doisave(fc);}
fq = "03\0";
explvl = 2;
i = chkstr(fq);
if (i == 0){
   fp = fopen("scrnfile","r");
   fseek(fp,scrndata[curpos].expos,0);}
else
   fp = fopen("/usr/joseph/project/form0","r");
filecopy(fp);
rewind(fp);
fclose(fp);
startcur();
}

filecopy(ff)
FILE *ff;
 {
int dwn,across;
int c;
wclear(stdscr);
wrefresh(stdscr);
for (dwn = 0; dwn < 23; dwn++){
   c = ' ';
   for (across = 0; across <= 81 && c != '\n'; across++){
       c = getc(ff);
       mvwaddch(stdscr,dwn,across,c);
       }
   }
wrefresh(stdscr);
}

writefile(ff)
```

```
FILE *ff;
{
int c;
int dwn,across;
for (dwn = 0; dwn < 23; dwn++)
   for (across = 0; across <= 79; across++) {
      wmove(stdscr,dwn,across);
      c = winch(stdscr);
      putc(c,ff);
      if (across == 79)
      putc('\n',ff);
      }
}

/* PERUSAL FUNCTIONS */

goback()
{
explvl--;
goingback = true;
if (explvl <= 0)
     explvl = 0;
switch(explvl){
  case 0: form--;
       if (form <= 2)
          form = 2;
          route();
       break;
  case 1: form = 6;
       route();
       break;
  default: searchback();
        break;
}
}

searchback()
{
FILE  *ff, *fopen();
static char *ft;
int i;
int d;
btriger = 1;
ft = malloc(nestlvl);
ft = mkstr(ft);
doisave(ft);
ff = fopen("scrnfile","r");
for(curpos = 0; curpos <= lstpos; curpos++){
   if((i = strncmp(ft,scrndata[curpos].expid,strlen(ft)-3)) == 0){
      d = fseek(ff,scrndata[curpos].expos,0);
     break;}}
filecopy(ff);
rewind(ff);
fclose(ff);
if (explvl >= 3)
```

```
  stcur2();
else
  if (explvl == 2)
     startcur();
}

goahead()
{
char *ft;
ft = "N cannot be used to progress into expanded forms - use D,O,I,or
P.";
if (explvl >= 1){
   mvwaddstr(stdscr,23,0,ft);
   wrefresh(stdscr);
   return;}
else {
   form++;
   if (form >= 6){
       form = 6;}
   route();}
}

char *cpyln(fln)
char *fln;
{
int tempx;
char c;
int i;
fln[0] = '*';
if (explvl < 3)
   tempx = 6;
else
   tempx = 14;
for (i=1; i < 76; i++){
    wmove(stdscr,lines,tempx);
    c = winch(stdscr);
    fln[i] = c;
    tempx++;}
return(fln);


doisave(ff)
char *ff;
{
int i;
i = chkstr(ff);
savescr(i,ff);
}

startcur()
{
yp1 = 4;
yp2 = 21;
xp1 = 7;
xp2 = 75;
```

```
lines = yp1;
cols = xp1;
wmove(stdscr,lines,cols);
wrefresh(stdscr);
}

/* EXPANSION FUNCTIONS */

expand()
{
char *fid,*fln,*fid2;
int c;
/* copies the line being expanded for title of new screen */
fln = cpyln(fln);
/* copies the present screen id */
fid = mkstr(fid);
fid2 = malloc(nestlvl);
strcpy(fid2,fid);
explvl++;
/* see if present screen has ever been saved */
c = chkstr(fid);
/*save the current screen at its old place,
if found, or at the end of the file otherwise */
savescr(c,fid2);
/* add the no. of the expand line to the screen id */
fid = addln(fid);
/* see if line has prior expansion */
c = chkstr(fid);
switch(c){
/* match so screen exists in scrndata */
  case 0 : oldexp(fln);
        break;
/* no match - must be a new screen */
  case 1 : newexp(fln,fid);
        break;
        }
}

char *mkstr(fz)
char *fz;
{
int getln, getcol;
char c;
int i = 0;
getcol = 3;
getln = 1;
wmove(stdscr,getln,getcol);
c = winch(stdscr);
while(c != ' '){
    fz[i] = c;
    getcol++;
    i++;
    wmove(stdscr,getln,getcol);
    c = winch(stdscr);}
fz[i] = '\0';
```

```
return(fz);
}

chkstr(fid)
char *fid;
{
int i;
if (curpos == lstpos)
   return(1);
for (curpos = 0; curpos <= lstpos; curpos++){
    if ((i = strcmp(fid,scrndata[curpos].expid)) == 0)
       return(0);
}
return(1);
}

char *addln(fid)
char *fid;
{
switch(lines){
case 4:  strcat(fid,".01");
     break;
case 5:  strcat(fid,".02");
     break;
case 6:  strcat(fid,".03");
     break;
case 7:  strcat(fid,".04");
     break;
case 8:  strcat(fid,".05");
     break;
case 9:  strcat(fid,".06");
     break;
case 10: strcat(fid,".07");
     break;
case 11: strcat(fid,".08");
     break;
case 12: strcat(fid,".09");
     break;
case 13:strcat(fid,".10");
     break;
case 14: strcat(fid,".11");
     break;
case 15: strcat(fid,".12");
     break;
case 16: strcat(fid,".13");
     break;
case 17: strcat(fid,".14");
     break;
case 18: strcat(fid,".15");
     break;
case 19: strcat(fid,".16");
     break;
case 20: strcat(fid,".17");
     break;
case 21: strcat(fid,".18");
```

```
     break;
default: break;
    }
return(fid);
}

newexp(fln,fid)
char *fln,*fid;
{
FILE *fp,*open();
int c;
fp = fopen("/usr/joseph/project/forme","r");
filecopy(fp);
fclose(fp);
wrefresh(stdscr);
putid(fid);
wrefresh(stdscr);
putln(fln);
wrefresh(stdscr);
stcur2();
}

stcur2()
{
xp1 = 15;
xp2 = 76;
yp1 = 4;
yp2 = 20;
lines = yp1;
cols = xp1;
wmove(stdscr,lines,cols);
wrefresh(stdscr);
}



putid(fid)
char *fid;
{
mvwaddstr(stdscr,1,3,fid);
wrefresh(stdscr);
}

putln(fln)
char *fln;
{
mvwaddstr(stdscr,2,3,fln);
wrefresh(stdscr);
}


oldexp(ft)
char *ft;
{
FILE *fp,*open();
```

```
int c;
fp = fopen("scrnfile","r");
fseek(fp,scrndata[curpos].expos,0);
filecopy(fp);
rewind(fp);
fclose(fp);
wrefresh(stdscr);
putln(ft);
wrefresh(stdscr);
xp1 = 14;
xp2 = 76;
yp1 = 4;
yp2 = 20;
lines = yp1 +1;
cols = xp1 + 1;
wmove(stdscr,lines,cols);
wrefresh(stdscr);
}

savescr(i,fid)
/*we are saving the current screen before we bring up an expansion */
/*or go back to a previous screen */
char *fid;
int i;
{
FILE *fp, *fopen();
fp = fopen("scrnfile","a");
if (i == 0)
       fseek(fp,scrndata[curpos].expos,0);
else{
   fseek(fp,0,2);/* never been saved - append it to the end of the
savefile */
   scrndata[lstpos].expid = fid;
   scrndata[lstpos].expos = newexpos;
   lstpos++;
   newexpos += delta;}
writefile(fp);
rewind(fp);
fclose(fp);
}

/* FINISHING ROUTINES */

finishup()
{
int i;
char c;
char *ff;
char *fmsg1;
char *fmsg2;
char *fmsg3;
char *fmsg4;
char *query;
char *fsend;
char *fprnt;
```

```
ff = malloc(nestlvl);
ff = mkstr(ff);
doisave(ff);
fprnt = fopen("Warnier","a");
fsend = fopen("scrnfile","r");
fmsg4 = "Sorting and formatting - wait for further instructions.";
wclear(stdscr);
mvwaddstr(stdscr,0,0,fmsg4);
wrefresh(stdscr);
system("sleep 8");
fseek(fsend,0,0);
fileccpy(fsend);
lines = 0;
wmove(stdscr,0,0);
writeoneline(fprnt);
lines = 1;
wmove(stdscr,1,0);
writeoneline(fprnt);
query = "Do you want a hard copy of your algorithm?(y/n)";
fmsg1 = "Your Warnier algorithm is being printed on the line
printer.";
fmsg2 = "Your algorithm is saved in file 'Warnier'.";
fmsg3 = "Done with the finishing tasks - hit any key to exit
program.";
sort_structure();
rewind(fsend);
for (i=0; i<= lstpos-1; i++){
    fseek(fsend,scrndata[i].expos,0);
    filecopy(fsend);
    writefile(fprnt);
    rewind(fsend); }
fclose(fsend);
fclose(fprnt);
system("sed -n -f /usr/joseph/project/endcmd Warnier >> Warnier");
wclear(stdscr);
mvwaddstr(stdscr,0,0,query);
wrefresh(stdscr);
c = getc(stdin);
switch(c){
  case 'y': wclear(stdscr);
            wrefresh(stdscr);
          mvwaddstr(stdscr,0,0,fmsg1);
          mvwaddstr(stdscr,1,0,fmsg2);
          mvwaddstr(stdscr,2,0,fmsg3);
          wrefresh(stdscr);
          system("lpr Warnier");
          break;
  case 'n': wclear(stdscr);
          wrefresh(stdscr);
          mvwaddstr(stdscr,0,0,fmsg2);
          mvwaddstr(stdscr,1,0,fmsg3);
          wrefresh(stdscr);
          break;}
done = true;
}
```

```
sort_structure()
{
char *temp;
int i;
int sorted = false;
int no_left = lstpos -1;
temp = malloc(nestlvl);
while((sorted==false) && (no_left != 1)) {
   sorted = true;
   for (i = 1;i <= no_left;i++){
      if ((strcmp(scrndata[i-1].expid,scrndata[i].expid)) > 0){
       temp = scrndata[i-1].expid;
       scrndata[i-1].expid = scrndata[i].expid;
       scrndata[i].expid = temp;
       sorted = false; }
       }
   no_left--;
   }
}

transferfile(fsend,fprnt)
FILE *fsend;
FILE *fprnt;
{
int down,across;
int c;
for (down=0;down<23;down++);{
   c=' ';
   for (across=0;across<=81 && c != '\n';across++){
       c = getc(fsend);
       putc(c,fprnt);
       if(across==79)
      putc('\n',fprnt);}
       }
}
```

AN ALGORITHM DEVELOPMENT PROGRAM USING WARNIER-STYLE BRACES

by

JOSEPH KENT CAMPBELL

B.S., Kansas State University, 1971
M.S., Kansas State University, 1976

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

This system provides programmers with an easy-to-use software development tool. The system provides tutorial screens of information concerning use of the program and problem solving in general. Help screens containing specific infcrmation may be perused during program execution to aid the programmer in development.

This system provides the programmer with methodology for encouraging top-down development. This is accomplished by providing the user with input to screens on which a facsimile of a Warnier brace is shown. Top-down development is encouraged by inclusion of an 'expansion' feature, which permits the user to develop any statement. The 'expansion' command provides a new input screen for statement refinement to some desired level of detail. Input and corrections to the Warnier screens are made with editing commands requiring single stroke execution. The commands are quite easy to master and behave much like the UNIX 'vi' commands. The program will also supply 'while' and 'for' loop syntax constructs on command.

When the user is finished with the session, the completed or partially completed algorithm is sorted to the correct hierarchical order by the system. The sorted screens form a Warnier diagram with a vertical alignment as opposed to the traditional horizontal configuration of a Warnier diagram. The diagram is stored a file in the user's current directory under the file name 'Warnier'. The system will query the user to determine if a printed copy is wanted before the program exits.