DEVELOPMENT OF AN EXPERIMENTAL TEMPLATE DRIVEN EDITOR FOR TREES

By

RONALD DEAN MOORE

Bachelor of Science

Southeastern Oklahoma State University

Durant, Oklahoma

1981

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1984

DEVELOPMENT OF AN EXPERIMENTAL TEMPLATE

DRIVEN EDITOR FOR TREES

Thesis Approved:

James R. Van Doren
Thesis Adviser

Sharilyn A. Thoreson

W. E. Hed

Norman N. Durham
Dean of the Graduate College

PREFACE

The development of a template editor for trees is examined by defining the mechanisms of the template and by discussing the techniques used to coordinate the traversal of two trees. This paper shows the feasibility of editing network and tree structures using an overlay (template) to keeping the data in a defined pattern. I wish to thank the faculty and staff of the Computing and Information Sciences Department for their help and friendship. Special thanks go to my major advisor, Dr. J. R. Van Doren, who answered my questions and also encouraged me in the work, to my committee members Dr. G. E. Hedrick and Dr. S. A. Thoreson for their helpful comments and Finally I express my sincere appreciation to Kathy McMath, who read the early manuscripts, the men of "Pigma Chi", who put up with me when deadlines came near, and others who encouraged me to continue. The biggest thanks go to my family, whom I love very much, and to my parents, to whom this paper is dedicated.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Objectives

Data structures are of concern in any application of computers. The increase of computer usage and the implementation of new sophisticated applications require proper attention to data structures by system designers. Tree structures are a useful group of these structures. Trees are used in compilers to develop structured symbol tables [5]. B-trees are often used when implementing indexed files for information retrieval systems [1].

This project describes the initial building and modifying of tree data structures. It has two major objectives; they are:

(1) to investigate the issues involved in a template driven tree editor; and

(2) to develop a template driven test editor for trees.

The trees used as a work base are the general class of ordered trees as defined by Standish [13]. Such a tree is a hierarchical data structure consisting of a finite set of one or more vertices with one designated vertex as the root. The remaining vertices (nodes) are partitioned into mutually exclusive subsets, each of which is an ordered tree.

Beginning in Chapter II the methods of editing and their relation to template editing is discussed. The chapter introduces the types of trees, the techniques, and the methods that are useful in coordinated processing. Chapter III describes the design of the editor and discusses implementation problems.

Both the editor and the concept of template driven editing are reviewed in Chapter IV. The issues involved in this type of editing are summarized and questions that are beyond the scope of this project are presented along with possible suggestions for further study of template driven editing of trees.

Four appendices are included to aid the user in working with the editor implemented in the project. Appendix A describes the implementation language and discusses the special features and restrictions of the language. Appendix B, a System Guide, contains the conceptual design logic of the system in Program Design Language (PDL) form in addition to a list of variable names and their definitions. Appendix C, a User's Guide, illustrates the proper way to start an editing session and contains comments on the format of displays and messages. The appendix also outlines each command of the editor. Appendix D gives the UNIX command procedure that connects logical files to physical files and performs the execution of the language interpreter.

Survey of the Literature

A number of different approaches to editing are found in the literature. The presentation of the literature is based on the type of design help given.

Conceptual Editor Design

A text editor is one of the basic components of a text-processing system [8]. Meyrowitz and Van Dam [8] describe this class of editors with three groups; line-oriented editors, stream editors and display editors. Line-oriented editors such as IBM's CMS editor (ca. 1967) is an example of a fixed-length line-oriented editor with a textual interface, designed for a time sharing system. Stream editors on a document as a single, continuous chain of characters rather than act upon fixed-length or variable-length lines. TECO, the Text Editor and COrrector (ca. 1970), is an interpreter for a string processing language.

TECO can be used interactively as a stream-oriented editor. The conceptual model considers a document to be a sequence of characters, possibly broken into variable-length virtual pages. Pages may be combined in an in-core editing buffer considered to be a variable-length string capable of expanding to the in-core memory available. Display editors or full screen editors use the Irons [8] conceptual model. In this model, text is conceived as a quarter-plane

extending indefinitely in width and length, with the topmost, leftmost character the origin of the file. The user travels through this plane by using cursor keys making changes when desired.

Structure editors attempt to use the natural structure found in most editing targets [8]. The idea of the ED3 prototype described by Stromfors and Jonesjo [14] is to superimpose a tree structure onto the text. The tree will act as the table of contents of a book. The difference is that changes of the tree will cause the text parts to be correspondingly reordered. Meyrowitz and Van Dam [8] mention the NLS/Augment editor as a structure editor using the same concept. Regardless of the subject matter, all NLS information is stored in a hierarchical outline structure. Statements can be nested an arbitrary number of levels. NLS provides modifiers to reference not only text elements but structure elements as well.

A Language-directed editor, according to Morris and Schwartz [9], combines the text manipulation functions of a general-purpose text editor with the syntax-checking functions of a compiler. Language-directed editors are claimed to allow more productive editing of program texts. They reduce typing effort by providing abbreviations for frequently occurring text elements such as keywords. The editor commands are tailored to the syntactic structure of the text. For example, a single command will locate the next occurrence of a variable X, ignoring occurrences of X

as a string in constants or comments. Language-directed editors are tend to be great consumers of computer resources and impose tight syntactic constraints on the object text.

Fraser [4] describes another type of editor as a syntax-directed editor. A syntax-directed editor accepts a grammatical description of hierarchical data structures and allows the user to enter and edit arbitrary trees having this structure. Fraser's syntax-directed editor, SDS [4,8], extracts all of its structure-dependent parameters from a grammar that resembles grammars accepted by typical compiler-compilers. Following the grammar is section of code used for semantic action. This code and grammar is compiled into a record declaration for the data type and code to check the syntax of input.

Implementation Model for Design Support

In the course of investigating logical design models for designing a template driven editor conventional procedure oriented processes proved inadequate. Essentially this is due to the coordination of processing required between a template and the structure being edited.

A model that is useful in coordinated processes is a group of modules with information items flowing between the modules. Under these conditions each module may be made into a coroutine [2]; that is; it may be made coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Coroutines are

subroutines all at the same level, each acting as if it were the master program when in fact there is no master program. Conway [2] uses this model to implement a high-speed, one-pass, syntax-directed Cobol compiler. Coroutines allow the source code to be input and a message output one at a time. Wang and Dahl [16] give an example to merge the data items contained in two or more binary search trees into a single sorted sequence. The algorithm consists of a set of recursive processes, one for the traversal of each tree, interlocked by coroutine linkage.

Hedrick and Alexander [6] discuss a preprocessor that allows coroutines in Fortran. The preprocessor accepts as input, statements that allow the user to define his own coroutines. A symbol table is constructed for names of the coroutines and a number is assigned as identification. Fortran subroutines are generated with the last statement as a computed GO TO statement which enables execution of the subprogram to resume at the proper place when it is called more than once in the program. Every subroutine which is generated by the coroutine preprocessor has a labelled common block used for communication among the various coroutines. A special subroutine interrogates this common block and/or sets the values in order to sequence the subroutine calls.

It is the intent of this paper to use this model and these designs to develop a prototype template driven editor

for general tree structures. The following is a discussion of this development.

CHAPTER II

METHODS OF EDITING

Introduction

Editors are software systems that attempt to perform text manipulations with speed and accuracy while upholding the overall computer performance. Editors are key elements in an interactive computing environment and exhibit many interesting questions concerning data structures and user interfaces. To review, the four primary types of editors mentioned in the previous chapter are: text editors, structure editors, language-directed editors, and syntax-directed editors.

Of the four types of editors, text editors are the most common. A text editor [8] is one of the basic components of a word processing system or an independent tool concerned with the creation and maintenance of the character strings of the target text. Word processing systems include other features as text formatters and spelling correctors while the text editor performs free style manipulations of text with dynamic changes in style and format.

Structure editors manipulate portions of generic structure. Since target applications have some innate structure (e.g. manuscripts are composed of chapters,

sections, and paragraphs), structure editors take advantage of this "natural" organization to manipulate text in a tree form. One such example is the NLS/Augment editor mentioned by Meyrowitz and Van Dam [8]. NLS uses a hierarchical tree structure to contain text of a hierarchical nature.

The third type of editor is a language-directed editor [9] which combines the text manipulation functions of a general-purpose editor with the syntax-checking functions of a compiler. The editor is combined with the functions of a parser, allowing the creation of programs in terms of the syntactic structure. Text is entered character by character and is parsed token by token as it is being entered. A disadvantage of language-directed editors is the tight syntactic constraints they impose on the object text.

The last type is syntax-directed editors [4]. A syntax-directed editor accepts a grammar description of a hierarchical data structure and allows the user to enter and edit arbitrary trees having this structure. A language grammar description would allow the editing of a parse tree from which program code could be generated. An example of a formatter/editor is SDS [4], where the syntax of a paper is described to the editor. Syntax-directed editing is an aid in program development and is expanded to any structure that can be described by a grammar.

The approach of template driven editing is similar to syntax-directed editing by accepting a description of a

hierarchical data structure. The description is not a grammar but a hierarchical structure with defined mechanisms. The purpose of template driven editing is to provide mechanisms to describe a data structure in a template and allow data structures to be created and edited having this structure.

Definition of Trees Edited

In addition to the ordered tree definition of Standish [13], the trees used for this project are more explicitly defined by the Programming Language for Allocation and Network Scheduling User's Guide [10]. According to this definition, each node has a label consisting of any character string containing no embedded blanks. Labels can be used to store information in the tree or to identify the nature of information in the subtree. Figure 1 shows the graphical format of a labeled tree using a convention that the label of the node is written to the right of the node. The leaf nodes of a tree have a value, which may be a character string, a numeric value, or null. Values are shown below their nodes as in figure 2. While graphical format is convenient for conceptual tree structures it is not used for screen display of this prototype. Figure 3 shows the tree of figure 1 as defined by an indented text format. Each new level is indented three spaces and the values of the leaf nodes are separated by a hyphen (-) character preceded and followed by a blank for visual

purposes. This representation of trees is similar to Hoare records [5] but differ with respect to the ability for dynamic change in structure whereas Hoare records do not. A null label in graphical format is demonstrated by a blank to the right of the node while an at-sign (@) is used to indicate a null label in textual format.

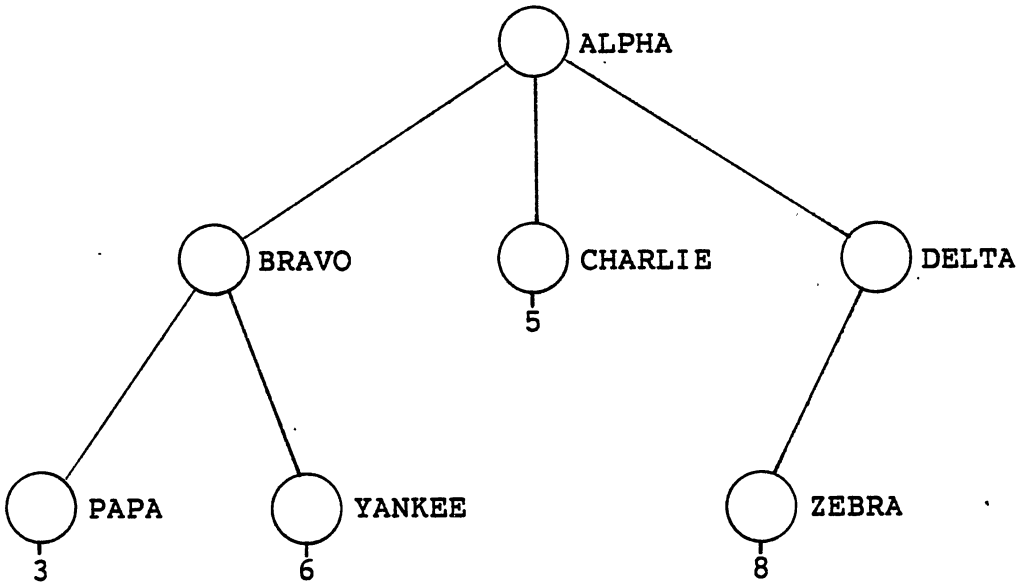


Figure 1. Labeled Tree (with substructures)

Figure 2. Labeled Tree (with values)

```
ALPHA
   BRAVO
      XTRA - 3
      YANKEE - 6
   CHARLIE - 5
   DELTA
      ZEBRA - 8
```

Figure 3. Tree of Figure 1 in Textual Form

Basic Editing Commands

The choice of a command set for a structure editor should be based on functionality and convenience. Stromfors and Jonejo [14] state, "Only a few commands are needed to make it easy for a user to walk around in a tree." Table I shows the command set for the prototype of a template driven editor for trees. The Advance and Back commands move one position across the siblings in a forward and backward direction. The Down command moves to the first child of current node while the Up command moves to the parent of the current node. Figure 4 shows the changes of the current node while performing the above commands. The Root command places the current node at the root of the tree regardless of the current location.

Two commands are used to display the template and data trees. Write Template displays the subtree of template that has the current node as the root and the Write command displays the subtree of the data tree that has the current node as the root. The commands Label and Value change the

information of the current node. The Value command is only valid on leaf level nodes and any changes are subject to a consistency check with the template.

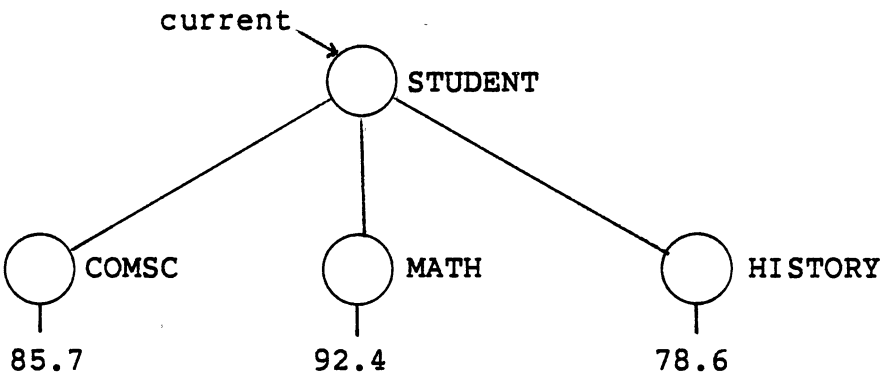


Figure 4a. Tree with root as current node

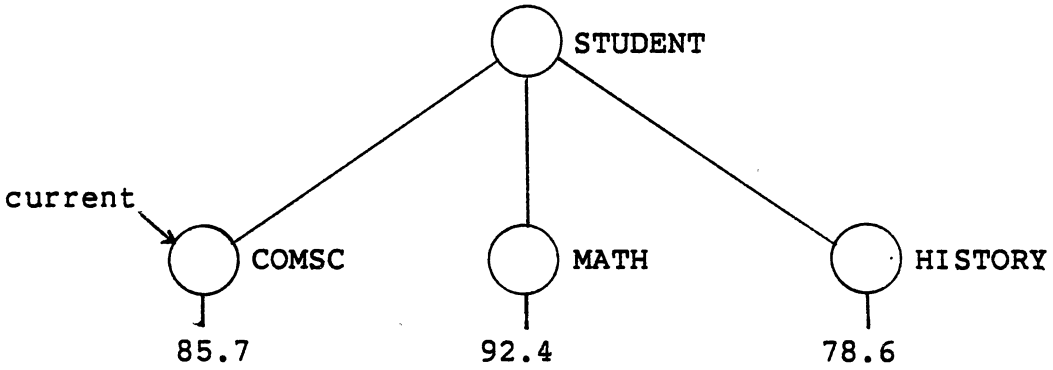


Figure 4b. Tree of (a) after Down command

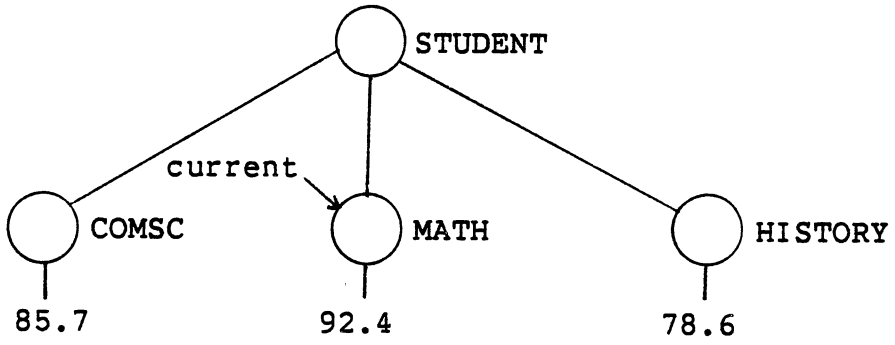


Figure 4c. Tree of (b) after Advance command

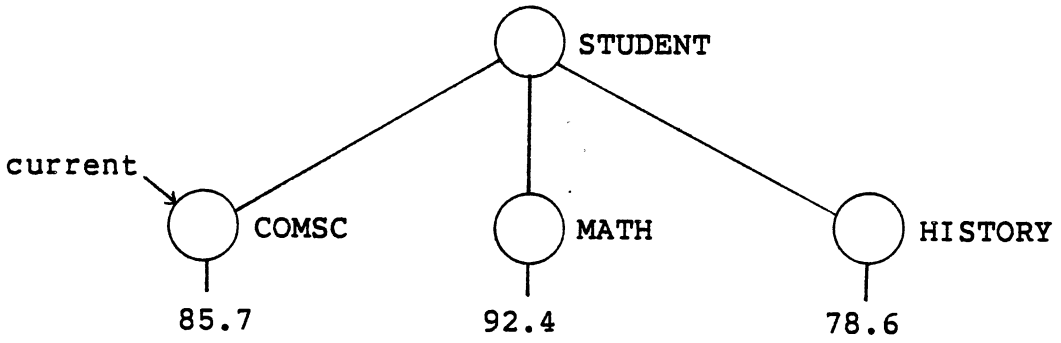


Figure 4d. Tree of (c) after Back command

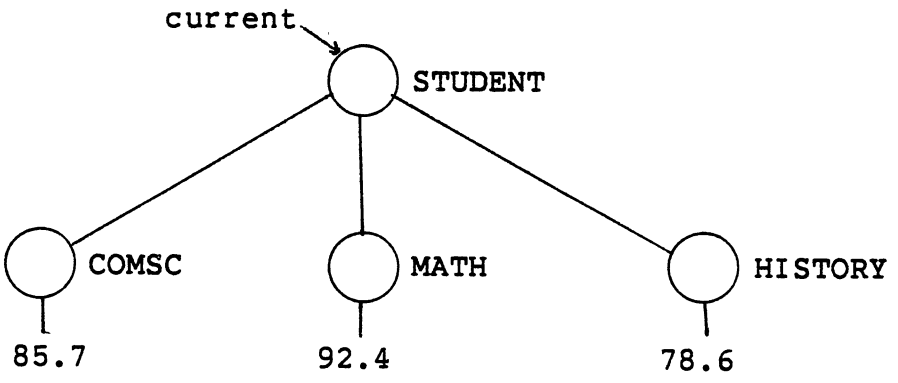


Figure 4e. Tree of (b), (c), or (d) after Up command

TABLE I

COMMAND SET FOR A TEMPLATE DRIVEN EDITOR FOR TREES

Command	Description
Help	list the command set
Advance	advance one position across siblings
Back	back one position across siblings
Down	move down to first child of current node
Up	move up to parent of current node
Root	move to the root of the tree
Write	write current subtree to screen
Write Template	write current template tree to screen
Label	assign label to current node
Value	assign value to current node
Insert Before	insert before current node a blank node
Insert After	insert after current node a blank node
Insert Down	insert down a blank node as last child
QUIT	end current editing session

The three remaining commands are Insert Down, Insert After, and Insert Before. Each of these commands insert nodes in positions relative to the current node. However, the use of these three are severly restricted by the template being used. Additional information on the restrictions can be found in Appendix C - User's Guide and discussion about command sets for structure editors is in [14,15].

The user interface is also an important factor in editing. Roberts [11] addresses the subject of the user interface relative to the usability of text editors by beginners, novice users and knowledgeable users. According to Roberts, the ideal interface would tutor beginners and advise novices, but not hinder the knowledgeable. For the beginner, the method of progressive disclosure would be suitable. This explicitly shows the user the valid command repertoire and the operation targets at any given time. Another method uses menus to allow the user to select choices rather than to memorize commands. Both of these methods would be helpful for the novice but could be somewhat irritating to the user who already knows what commands are needed and how to use them.

Structure editing lends itself nicely to split-screen display. The top portion of the screen displays the current node and subtree while the bottom portion awaits the command input. Regardless of the method used, the editor should use

descriptive mnemonic commands and a help facility on either user request or upon invalid or disabled commands. The topic of user interfaces is beyond the scope this paper. The content of this paper emphasizes the functionality of template driven editing.

Template Editing

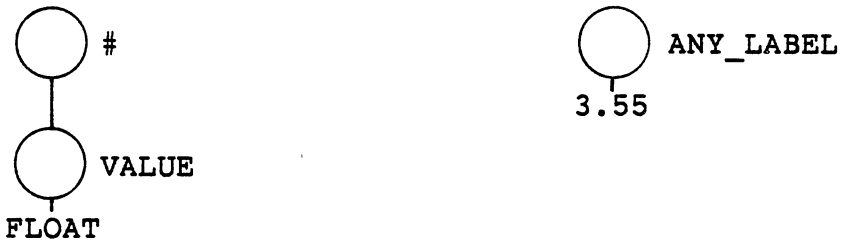
As grammars are used to generate the possible combinations of tokens in programming languages, template trees are used to generate the possible combinations of structures in data trees. Though the template is a special tree created by a template editor, it is used to influence the data tree. The primary goal of template driven editing is to insure structural and content validity in the data tree. The template is built by placing special labels in the tree structure restricting the editor to allow only valid input. The mechanisms to describe the value field of a node are CONSTANT and VALUE. When CONSTANT is the label of the current template node the value of that node is placed in the current data tree node. A label of VALUE in the template causes a value to be placed in the data tree of the data type specified by the value field of the template node. Figure 5 shows how values are described. The four types available are: string, integer, float or null.

Labels are subject to restrictions also. The generalized label is denoted by a pound sign (#), also shown in figure 5, and allows any character string as input. A

more restricted template symbol is the asterisk (*). The asterisk signals the editor that a label may be input from a list of choices and placed in the data tree as in figure 6. The left branch (CHOICE) has as its children the choices available while the right branch exist as a place holder accepting the choice and allowing a possible substructure. A label is considered to be fixed if the template symbol is not a defined mechanism. A fixed label, as shown in figure 7, is required in the data tree if the node exists.

Template tree Data tree

a) general label (with CONSTANT)



Template tree Data tree

b) general label (with VALUE)

Figure 5. Template and Data Tree with general label

The dollar sign ($) signifies the repeat of the exact substructure that is described in the right subtree. The left subtree (REPEAT) is the value of the maximum number of times it can be repeated. The substructure can be repeated any number of times between zero and the maximum. The example in figure 8 shows the maximum to be three but the substructure is repeated only twice. Any legal template mechanisms can be used in the right subtree.

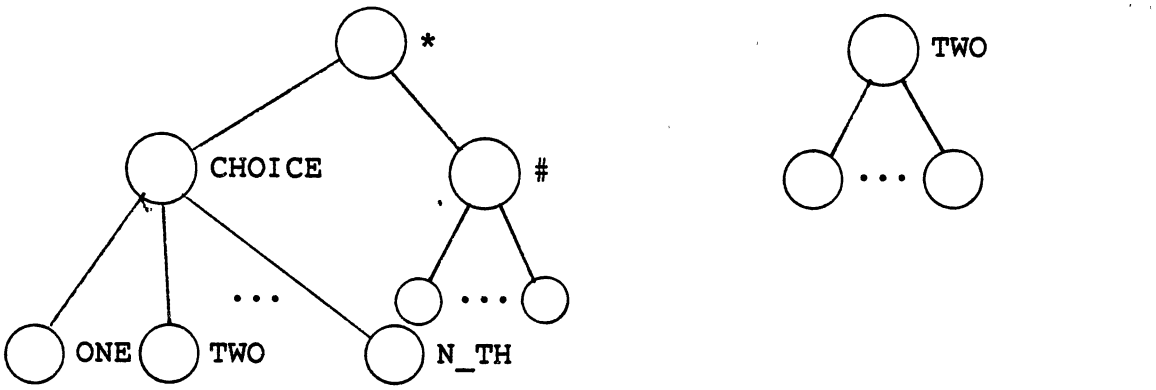


Figure 6. Template and Data Tree with choice of label

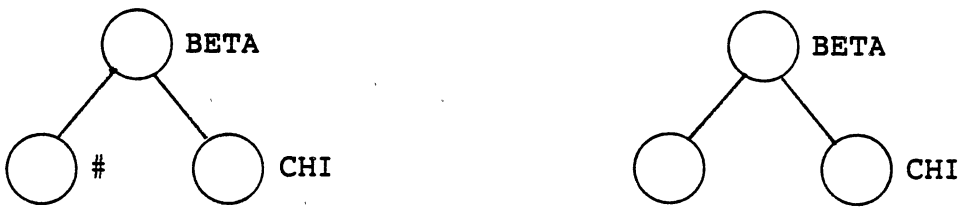


Figure 7. Template and Data Tree with fixed label

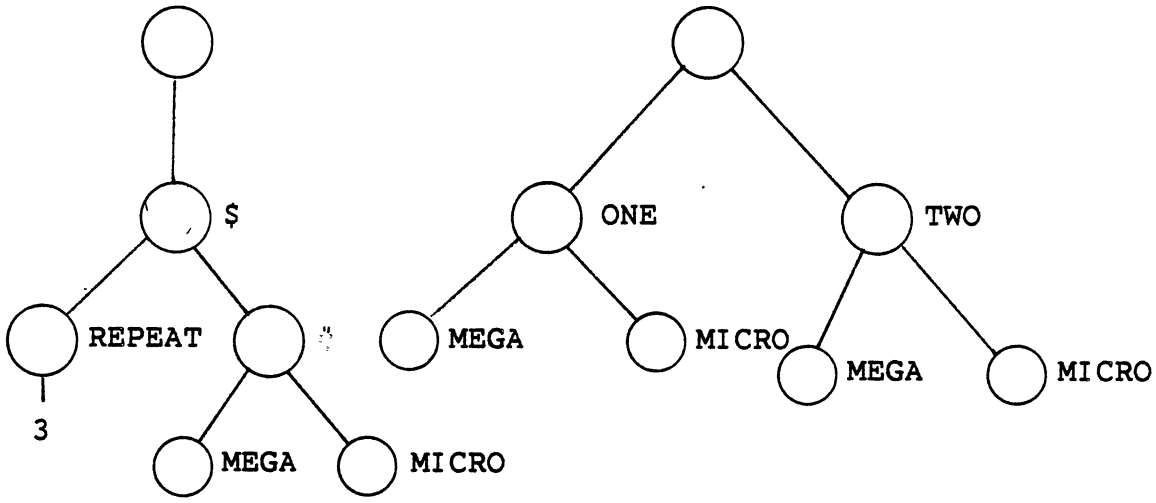


Figure 8. Template and one of four valid Data Trees with horizontal repeat

The template structure of a vertical repeat mechanism is shown in figure 9. This mechanism requires two symbols. The ampersand (&) is the root of this structure, and when it is encountered its location is stored and editing continues with the single child. Upon encountering the circumflex (^) the stored location of the ampersand becomes the current template location thus repeating the substructure vertically.

Table II presents a summary of template symbols. With these description mechanisms and the command set of Table I template driven editing can be accomplished on n-ary ordered trees. The remainder of this paper includes a description of how these symbols and mechanisms are implemented in a template driven editor.

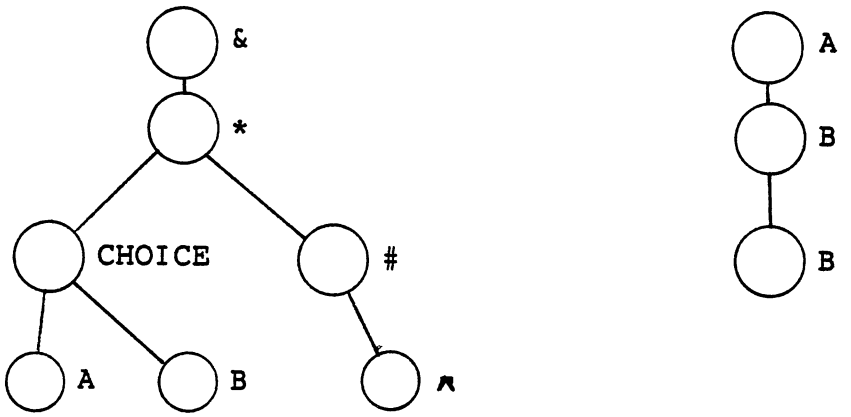


Figure 9. Template and Data Tree with vertical repeat

TABLE II

TEMPLATE MECHANISM SYMBOLS AND ACTIONS

Symbol	Actions
#	any input accepted in label field
*	choice of input accepted in label field
CHOICE	root of subtree with valid choices
<label>	fixed label - any label that is not a template mechanism
CONSTANT	specifies fixed value of data leaf
VALUE	specifies data type of data leaf value
$	possible repeat of substructure vertically
REPEAT	maximum number of substructure repetition
&	possible repeat of substructure horizontally
⋔	placeholder for possible horizontal substructure repetition

CHAPTER III

DESIGN OF AN EDITOR FOR TREES

Introduction

The design of editors, according to Meyrowitz and Van Dam [8], has been ad hoc, with the editor often copying and inheriting poor design from previous systems. Their call for strong design techniques is stated:

> It is time that editor designers, like programming language designers, commit their conceptual models and user interfaces to paper before implementation. This requires extensive search of the literature, analysis of alternatives, and experimental validation of ideas, all traditional actions in science and engineering ... ([6] p.403).

The conceptual model used here in developing a prototype editor for trees is based on the principle of a template driven method of input and validity checking. The template used in a template driven editor for trees describes the basic structure of the data tree. The editor is expected to compare the data tree and the template tree to assure that a data tree is consistent with the template used. The label and value requirements are also to be checked for satisfaction of the template tree rules mentioned in Chapter II. A template driven editor, as with any editor, needs the ability to move the current viewer window around in the target data. With the target being a tree this causes the

movement mechanisms to be modeled after a natural tree traversal. The traversal of the data tree and template tree requires a coordinated technique to keep the correspondence of data tree nodes and template definitions.

In an interactive computing environment the purpose of an editor is to create and change information structures. The editor for trees must allow input. This input could be limited by the structure imposed by the template, but should be allowed nonetheless. The extent of what could be inserted is addressed on an individual basis during the editing session under the considerations of the template used for that session.

The development of an editor for trees is a topic that brought about many unanswered questions. The limited definitions of template nodes and ignoring the possibility of a deletion (pruning) of a subtree were not studied in detail because of time constraints. The prototype developed should exhibit enough functional characteristics to show its usefulness and test its results.

Conceptual Model of Editor Design

The underlying concept of template driven editing is the need to keep track of which node in the template corresponds with a node of the data tree. The model used in developing a template driven editor for trees is one of a recursive coroutine system [1,2,14].

Coroutines were first introduced by Conway [1] while designing a compiler that would receive one line of input and

output a message. These coroutines require subprograms to act as if each were the calling program. The technique of coroutine processing has become increasingly useful in the coordinated processing required by some operating system techniques. Wegner [15] suggests that the same kind of coordination can be used in search trees. Template driven editing decodes the template, take actions required by template mechanisms, and then repeat the cycle.

Coroutines, as the name implies, are routines that cooperate with one another. Subroutines have a relationship to the calling procedure like one of a master-slave. Figure 10a shows the master-slave relationship of the main procedure and two subroutines. The subroutine is subordinate and is started at the first of the block and continues until the end or some return point. The call passes control to the subroutine but control must be returned to the main procedure before control can be passed to another subroutine. Figure 10b shows the same procedures with a coroutine structure allowing control to be passed through subprograms until the task is completed. The RESUME statement is the method by which control is passed between coroutines. Further explanation of coroutines and the RESUME statement is found in [2,14]. Each coroutine can be viewed as the calling procedure block. This allows coroutines to process one section of code, pass control to another coroutine which does the same and then return control to the original to continue processing at the point immediately following transfer of control or pass control to another coroutine.

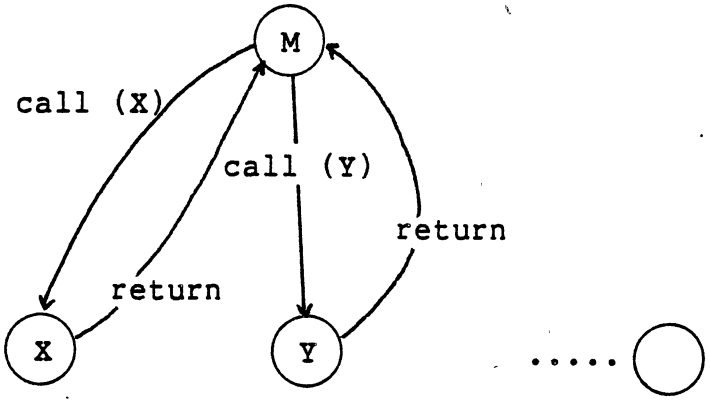


Figure 10a. Master-slave relationship of subroutines.

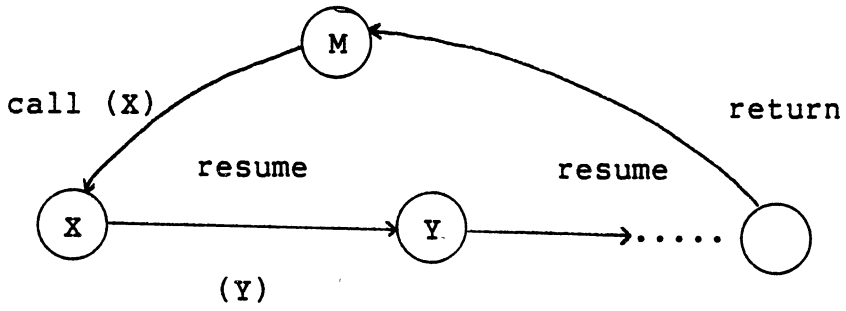


Figure 10b. Cooperative relationship of coroutines.

Recursion is used to aid in the coordinated processing by allowing a subtree to be processed as an independent tree. After the subtree is traversed processing can be continued along siblings of the current node which is the root of the subtree. This is similar to the local instance discussed by Dahl,

> ... the scanner [traversal] can be accomplished by a recursive procedure attribute, local to the relevant instance of the coroutine. ([2] p. 195).

By using recursion to produce these local instances and coroutines to traverse along a level, template driven editing can be accomplished. The importance of coordinated traversal should be seen due to the one to one correspondence of the template mechanism structures and individual data nodes.

The definitions included in the template are primarily limited to labels and values. Three compound symbols exist for choice of label, vertical repetition, and horizontal repetition. The set of definitions (see Table II) exhibit enough power to test the concept of template driven editing for trees while still keeping the complexity of definitions to a minimum.

Subsystem Design of the Editor for Trees

There are three major subsystems of the template driven editor for trees. The movement subsystem has the responsibility of coordinated traversal of the template and

data trees. The validity checking subsystem insures that the data tree is consistent with the template being used. The editing subsystem interprets the command input by the user and executes the command. This execution could be accomplished in the editing subsystem or by a call to another subsystem (i.e. movement or validity checking subsystems).

Movement Subsystem

The movement subsystem has the responsibility of coordinating the traversal of the template tree and the data tree. The recursive coroutine system is made up of a recursive external routine with two internal coroutines. One coroutine has the function of traversing the data tree. This is done by assuming that the only direction desired is that of left to right along siblings of the same level. The advance along the sibling is completed and control is resumed in the subsystem requesting the movement. The other coroutine must make a decision concerning the movement direction in addition to performing the movement in the template tree. The information needed for this decision is provided by the subsystem requesting the movement. If the direction of the movement is along the current level then an advance is made in the same fashion as with the data tree. If the direction of the movement is down, then the template traversal coroutine checks both trees for substructure,

which is required for downward movement, and issues a recursive call to the global routine.

By using recursion for the depth traversal, conceptually, a new set of local coroutines are used. Each call to the global routine temporarily freezes the variables of the current level and traverses the subtree as an individual tree. This provides a convenient method of handling the nested structures which trees create.

Validity Checking Subsystem

The validity checking subsystem is invoked upon entering a editing session or by an input user command. This subsystem has two major roles: building the data tree with the required parts of the template and to print a list of data tree nodes that are not consistent with the coordinated template node. The building role is discussed in the editing subsystem because of the close functional relationship. The checking role of the subsystem checks only the tree for which the current node is the root and issues requests to the movement subsystem to perform a generalized inorder traversal of the n-ary tree pair, the current data tree and the template used to edit the data tree. Because the process is for the entire tree, special comparisons are needed for repeat substructures in the template. If an error occurs while the template is at the root of a repeat substructure, the current template node is

advanced to the next sibling in the event that the repeat was optional. This allows misplacement of an error but is required if optional repetition is desired.

Editing Subsystem

The editing subsystem displays current information, prompts the user for commands, and executes the commands. At any idle time of the editing session the current data tree node with either the value or the number of children is shown. Immediately following the current information a prompt for the next user command is displayed.

The execution of user commands may require interaction with the other subsystems or could be processed in this subsystem. For example, any commands displaying or changing current node information are done in the editing subsystem (Write, Write Template, Label, Value, etc.). Movement commands (Advance, Down, etc.) require issuing requests of the movement subsystem. Commands used in building the tree (Insert After, Insert Down, Insert Before) must check the template to see if the command is valid and issue a request to the validity checking subsystem to build the node and any required substructure associated with the node. The building of the substructure keeps the editor consistent with the definition of a template or overlay of another structure.

Implementing a Recursive Coroutine System

The recursion of a part of a recursive coroutine system should be done with an implementation language that supports recursion. Otherwise, the stack manipulation would become tedious. The foundation of the logic used for coroutine implementation is discussed by Hedrick and Alexander [4]. They developed a preprocessor to handle coroutine processing in Fortran. The principle is to implement the conceptual model of coroutines by using Fortran subroutines. For each coroutine, a subroutine is generated with a labeled common block. The data in this labeled common block is used for communication among the various coroutines; in particular, the Coroutine Control Routine (CCR) which interrogates and/or sets the values that are used to sequence the coroutines. The RESUME statement is then implemented by assigning a new code to the variable designating the next routine to be called and returning control to the CCR.

Implementation of this prototype editor for trees is written in the Programming Language for Allocation and Network Scheduling (PLANS). PLANS is a recursive block structured language that uses trees as the primary data structure. However, PLANS does not have a method of implementing external variables (as in the Fortran common block) or variable entry points (as in Fortran entry specifications) which could be useful. For more information

on the PLANS language refer to Appendix D and [8].

The problem of implementing external variables was solved by making all routines (control and conceptual) local to a recursive procedure. An integer variable was used to contain the entry point information. Upon entering the routines, the entry point variable is queried by if-then-else-if statements to decode the label at which processing should continue. Exiting the routine by a RESUME statement updates the entry point variable that would be used the next time the routine was called (resumed).

The conceptual model of recursive coroutines is a possible solution to design the coordinated traversal needed to accomplish template editing of trees. The movement subsystem controls the traversal while the editing subsystem and validity checking subsystems interact with the user and insure the consistency of the data tree.

CHAPTER IV

SUMMARY AND RECOMMENDATIONS

The need for an "editor for trees" is apparent when systems are used that make use of dynamic tree structures. For example, scheduling and allocation problems like those encountered by NASA with space shuttle flights [8] are instances in which large trees were used to arrive at schedules with no regard to how or where the trees were originally built. Updating these large trees presents problems unless the necessary tools are available; i.e., editors for trees. In hierarchical data base systems the use of a tool to populate these systems could be less time consuming than conventional data base input programs. With the state of the art of computing science trees are used in many different applications. Editing of these tree structures will allow trees to become data structures which are as common as files but have the added features that make trees efficient and practical.

The general language features desired in the development of template driven editors for trees include direct coroutine implementation, arrays of tree variables, and external storage of trees during an editing session. Direct coroutine implementation would bring the physical

code of the editor closer to the conceptual model. Pointer arrays are useful in allowing depth repetition during editing of trees. The feature of external storage would allow very large trees to be edited (either large template trees or data trees) without using a large amount of the computer's main storage. Presently trees used in the editing process of the prototype must be kept in memory.

This project was undertaken with two major objectives in mind: First, to investigate the difficulty of a template driven tree editor; and secondly, to develop an editor for trees in which these ideas could be tested.

The results of the first objective show that template editors are useful and possible to implement. However, many questions were raised during the work and will need to be addressed before a production environment can be considered.

One question raised concerns optional substructures. Is there an instance when a substructure could be optional or are all substructures required? This may possibly be application dependent, but if the editor is a generalized template editor for trees, then any tree can be created for any specific application. Another question along the same line has to do with the use of a Prune command to delete nodes and connected substructures. A prune command could be needed if the substructures were allowed to be optional but would restructure the data tree otherwise. Removal of nodes or substructures requires the right sibling of the deleted node to be checked against the template structure of the

deleted node.

Another area that raised questions was how much consistency checking should be done automatically and how much should be left to the user. The option taken in this project is that automatic traversal should occur when entering the editing session. If the data tree exists, then the traversing causes only messages to be printed concerning the inconsistencies. If the data tree does not exist, then the automatic traversing would build a tree with the structure and required labels and values of the template. When an insert command is used, the same building process is done. Pattern matching and error detection questions arise when errors occur in optional repeat structures. When using the optional repeat structure, the consistency check will try the next template node upon an error causing the program to assume that the repeat structure was not used. This sometimes causes the error message to be misplaced and possibly may cause more errors.

The final question is one concerning potential ambiguity in a template. This question covers many avenues of study and may have a wide variety of answers. The ambiguity does not cause the editor to create an invalid tree but exists due to modifications made by means other than the editor. If all the ambiguity is removed, then the restrictions on a template will require separate editors for templates and data trees. If ambiguity is allowed (as in this prototype), then again error detection and consistency

checking becomes difficult.

Template driven editing is a tool that could prove to be productive in the expansion of information retrieval systems. Many techniques used in these systems are hierarchical in nature allowing the idea of template generation to apply itself to data base generation. A more immediate use of template editing is the population of data base systems and the changing of the structure of these systems. Another extension could be with network structures. These structures would benefit from a tool to build the network while adhering to a defined pattern.

The second objective has been accomplished through the design and implementation described in the previous chapter. The editor provides the user with the capability to edit a wide range of trees. The general template allows the editor to function conceptually as basic tree editor [13]. Using the editor to generate additional templates allows the user to create templates which meet specific requirements. The attribute that makes a template editor useful is the ability to build a tree structure with all required substructures, fixed labels, and fixed values. This insures the validity of the tree and the usefulness of the tree in the intended application.

Further work could be done concerning these questions. A selected list of topics is given to invoke thoughts of future work. 1) The user interface of a template driven editor for trees. 2) The development of additional template

mechanisms to allow more generalized structures. 3) The removal of template ambiguity contrasted with editing flexibility. 4) The performance of an template driven editor compared with other methods of tree creation and modification.

SELECTED BIBLIOGRAPHY

[1] Bradley, J., File and Data Base Techniques. Holt, Rinehart and Winston, New York, 1981.

[2] Conway, M.E., "Design of a Separable Transition-Diagram Compiler" Communications of the ACM, Vol. 6, No. 7 (July 1963), 396-408.

[3] Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., Structured Programming, "Hierarchical Program Structures" pp. 175-220. Academic Press, London (1972).

[4] Fraser, C.W., "Syntax-Directed Editing of General Data Structures." Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation" Vol. 16, No. 6 (June 1981). ACM, New York, 1981. pp. 17-21.

[5] Gries, D., Compiler Construction for Digital Computers. John Wiley and Sons, Inc., New York, 1971.

[6] Hedrick, G.E., and Alexander, B.R., "Coroutine Programming in Fortran". The Australian Computer Journal, Vol. 4, No. 2 (May 1972).

[7] Medina-Mora, R.I., Syntax-Directed Editing: Towards Intergrated Programming Environments. Ph. D. dissertation Carnegie-Mellon University (DA8215892) 1982.

[8] Meyrowitz, N., and Van Dam, A., "Interactive Editing Systems: Parts I and II". Computing Surveys, Vol. 14, No. 3 (September 1982).

[9] Morris, J.M., and Schwartz, M.D., "The Design of a Language-Directed Editor for Block-Structured Languages." Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation Vol. 16, No. 6 (June 1981). ACM, New York, 1981. pp. 28-33.

[10] Ramsey, R.H., Willoughby, J.K., and Kullas, D.A., User's Guide to the Programming Language for Allocation and Network Scheduling, Technical Report SAI-77-068-DEN, Science Applications, Inc., Englewood, Colorado, 1977.

[11] Roberts, T.L., Evaluation of Computer Text Editors, Ph.D. dissertation Stanford University (8011699), 1981.

[12] Samual, C.A., A Software Design for the Programming Language PLANS, Master's Thesis, Oklahoma State University, 1982.

[13] Standish, T.A., Data Structure Techniques. Addison-Wesley Publishing Company, Inc., Philippines, 1980.

[14] Stromfor, O., and Jonejo, L., "The Implementation of Experiences of a Structure-Oriented Text Editor." Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation Vol. 16, No. 6 (June 1981). ACM, New York, 1981. pp. 22-27.

[15] Van Doren, S.R., Unpublished user's guide for PLANS tree editor, Oklahoma State University, Computing and Information Sciences.

[16] Wang, A., and Dahl, O.J., "Coroutine Sequencing in a Block Structured Environment." BIT, Vol. 11, No. 4 (1971). pp. 425-449.

[17] Wegner, P., Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, New York, 1968.

APPENDIX A

IMPLEMENTATION LANGUAGE DESCRIPTION

The implementation language for the experimental template editor of trees is the Programming Language for Allocation and Network Scheduling (PLANS). PLANS is a high level language which supports dynamic manipulation of tree data structures. The language was developed to reduce the cost and time span of developing and maintaining software which support scheduling and resource allocation tasks. The language is under continuing development at the Computing and Information Sciences Department of Oklahoma State University. The installation consists of a Perkin-Elmer 3230 operating under the UNIX[TM] operating system.

PLANS supports ordered trees, but the trees are stored in memory as binary trees. The left pointer points to the first child of the current node while the right pointer points to the next sibling. Recursion is available in PLANS which is used to traverse down and up through trees. An advance function allows the traversal of nodes left to right along one level. PLANS does not have local static variables and therefore must be placed in a higher scope block. External variables also are not included in PLANS. This requires procedures that use external variables to be made internal to the universal block. This technique of achieving external variables is used when implementing coroutines to traverse two trees in a coordinated manner. Another language feature restricting coroutine

---

[TM] UNIX is a trademark of Bell Telephone Laboratories

implementation is the absence of a computed goto or label variable used to enter routines at the location following the last statement executed.

The emphasis of PLANS is on data structures, access methods, and manipulative operations. In the developing of an editor for trees, a language that treats trees as a basic data type is a natural luxury. The access methods of tree nodes make PLANS a useful language in developing an editor for trees. The use of tree pointers to point to subtrees allow traversal to continue even if unusual circumstances occur in the template descriptor mechanisms. These circumstances develop because PLANS does not have string functions which would allow multiple fields in the label of a node. The template symbols for repeating structures ($,&) and for choice of labels (*) use substructures to give additional information to the routines. Indirect access methods are used to access special function subtrees (CONSTANT, VALUE, CHOICE). The manipulative operations of PLANS allow insertion of nodes either before or after the current node which changes the pointers to reflect the insertion. The trees are actual as well as conceptual to the user.

In developing an editor for trees, the need for nested depth structures is apparent. To nest the substructures several utilities were written to accomplish a push down stack. Because PLANS does not facilitate tree pointer arrays, this was accomplished by converting the pointers to

integers and storing the integers in an array with the last element of the array as the top of stack pointer. Conversion was done by sending a tree pointer as an actual parameter to a procedure with a dummy parameter as an integer. The inverse was done when converting back to tree pointers. The size of the push down stack limits the amount of vertical repetition. If trees are nested a large number of times the stack size will need to be increased. Programming changes in the push and pop utilities and the size declaration in the main procedure are required.

The size of the trees are also hampered with PLANS. PLANS trees must be sequentially input and reside in main memory. Since an template editor requires two trees of varying size, if insufficient memory remains for one of the trees, the editor loses effectiveness.

More information about PLANS is in [8] and [10].

APPENDIX B

SYSTEM PROGRAMMER'S GUIDE

This is intended to provide the reader a description of the conceptual design of the experimental editor for trees to aid in understanding the internal program structure of the editor. This is done in two sections. The first is the description of variables used in the program while the second is the Program Design Language of the conceptual model of coroutine procedures. For discussion on languages features of PLANS see Appendix A. For a copy of the implemented source code send a written request in care of the author to P.O. Box 695, Atoka, Oklahoma 74525, U.S.A.

Variables and Descriptions

Table III lists the variables of the program in alphabetic order and a description of uses for each. Variables beginning with a dollar sign ($) signafies a tree variable. Tree variables are also used for character variables because of the absence of character type variables in PLANS. The remaining variables are of type integer or float following the default specifications of Fortran 77.

Program Design

The following is a description of the design of a prototype template driven editor for trees. The psuedo code given and the actual implementation of the editor differs from the discussion in Chapter III. When implementing

coroutines the movement subsystem was duplicated, with minor changes, internal to the remaining two subsystems because of scoping problems in the PLANS language.

TABLE III

LIST OF VARIABLES AND DESCRIPTIONS USED IN A PROTOTYPE EDITOR FOR TREES

Variable	Description
$blank	temporary node used during insertions
$current_data	current node in data tree
$current_desc	current node in template tree
$data	data tree for current session
$desc	template tree for current session
$flag	LABEL used to perform two pass commands
$mode	LABEL used to check consistency
$parent	pointer to template structure mechanisms
$reply	LABEL receiving yes/no keyboard input
$temp	temporary node used for stack operations
flag	flag set to 1 if $parent is in use
ichild	ordinal number of $current_data
ipoint	integer array used to implement stack
irepeat	maximum number allowed for vertical repeat
jchild	ordinal number of $current_desc
kount	current count of vertical repeat
level	level of current node
m	temporary integer for parameter passing
n	temporary integer for parameter passing
none	integer constant representing 1
norepeat	flag for error recovery in CHECK
num-data	number of siblings along current level
num-desc	number of template nodes along current
valid	flag signaling match in choice structure

```
PROC main;
   SET ipoint(21), level, none <-- 0, -1, 1;
   WRITE message, 'MAIN PROGRAM STARTUP';
                , 'shift to caps for yes/no responses and
                   editing commands';
   READ $desc /* template tree for current edit session */;
   WRITE message, 'Is the data tree a new tree?(Y/N)';
   GET $reply;
   IF $reply = 'Y' THEN;
      $data <-- $NULL /* $NULL is a null tree */;
      $mode <-- 'build';
      CALL check;
   END;
   ELSE;
      READ $data;
      $mode <-- 'check';
      CALL check;
   END;
   CALL list_commands;
   CALL newlevel;
   WRITE /* to file */ $data;
   WRITE message, 'MAIN PROGRAM TERMINATION';
END main;
```

```
PROC check (IN $desc: TREE, $data: TREE, $mode: CHAR,
            INOUT num_desc: INT, num_data: INT, level: INT,
                  ipoint: INT ARRAY[1,21]);
   DEFINE $current_desc AS $desc;
   DEFINE $current_data AS $data;
   SET ichild, jchild, flag, kount <-- 1, 1, 0, 0;
   level <-- level + 1;
   CALL match;
   level <-- level - 1;
   return;
END check;
```

```
COROUTINE match;
   /* coroutine internal to check */
   /* reminder that coroutines continue after last
         executed statement                          */
one:
   IF LABEL($current_desc) = '&' THEN;
      DEFINE $parent AS $current_desc;
      DEFINE $current_desc AS $current_desc(1);
      flag <-- 1;
      CALL pushptr(ipoint,$parent);
      GOTO one;
   END;
   ELSEIF LABEL($current_desc) = '¬' THEN;
      /* check if depth substructure exist */
      IF NUMBER($current_data) > 0 THEN;
         CALL popptr(ipoint,$temp);
         DEFINE $current_desc AS $temp;
         GOTO one;
      END;
   END;
   ELSEIF LABEL($current_desc) = '$' THEN;
      irepeat <-- $current_desc.REPEAT;
      norepeat <-- 1;
      kount <-- kount + 1;
      IF kount <= irepeat THEN;
         DEFINE $parent AS $current_desc;
         DEFINE $current_desc AS $current_desc(2);
         flag <-- 1;
         GOTO one;
      END;
      ELSE;
         norepeat <-- 0;
      END;
   END;
   ELSEIF LABEL($current_desc) = '*' THEN;
      IF $mode = 'check' THEN;
         valid <-- 0;
         DEFINE $choice AS $current_desc.CHOICE;
         DO FOR ALL SUBNODES OF $choice USING $ptr;
            IF LABEL($ptr) = LABEL($current_data) THEN;
               valid <-- 1;
            END;
         END;
         IF valid =1 THEN;
            DEFINE $parent AS $current_desc;
            DEFINE $current_desc AS $current_desc(2);
            flag <-- 1;
            GOTO one;
         END;
         ELSE;
            RESUME err;
         END;
      END;
      ELSE /* $mode = 'build' */;
```

```
            DEFINE $parent AS $current_desc;
            DEFINE $current_desc AS $current_desc(2);
            IF jchild > 1 THEN;
               ADVANCE $current_data;
               jchild <-- jchild + 1;
               num_data <-- num_data + 1;
            END;
         END;
      END;
      ELSEIF LABEL($current_desc) = '#' THEN;
         IF $mode = 'check' THEN;
            IF LABEL($current_desc(1)) = 'CONSTANT' THEN;
               CALL scalor;
            END;
            ELSEIF LABEL($current_desc(1)) = 'VALUE' THEN;
               CALL numeric;
            END;
         END;
         ELSE /* $mode = 'build' */;
            IF jchild = 1 THEN;
               ADVANCE $current_data;
               jchild <-- jchild + 1;
               num_data <-- num_data + 1;
            END;
         END;
      END;
      ELSEIF LABEL($current_desc) = LABEL($current_data) THEN;
         IF LABEL($current_desc(1)) = 'CONSTANT' THEN;
            CALL scalor;
         END;
         ELSEIF LABEL($current_desc(1)) = 'VALUE' THEN;
            CALL numeric;
         END;
      END;
      ELSE;
         IF $mode = 'check' THEN;
            RESUME err;
         END;
         ELSE /* $mode = 'build' */;
            IF jchild = 1 THEN;
               ADVANCE $current_data;
               jchild <-- jchild + 1;
               num_data <-- num_data + 1;
            END;
            LABEL($current_data) <-- LABEL($current_desc);
         END;
      END;
   two:
      RESUME trav_template;
   three:
      RESUME trav_tree;
   END match;
```

```
COROUTINE trav_template;
   /* coroutine internal to check */
   IF NUMBER($current_desc) > 0 THEN;
      IF NUMBER($current_data) = 0 & $mode = 'build' THEN;
         LABEL($blank) <-- '';
         $current_data(next) <-- $blank;
      END;
      IF NUMBER($current_data) > 0 THEN;
         m <-- NUMBER($current_desc);
         n <-- NUMBER($current_data);
         CALL check;
      END;
      ELSEIF LABEL($current_desc(1)) = 'CONSTANT' |
             LABEL($current_data(1)) = 'VALUE' THEN;
      END;
   END;
   IF flag = 1 THEN;
      DEFINE $current_desc AS $parent;
      flag = 0;
      IF LABEL($current_desc) = '$' THEN;
         RESUME match;
      END;
   END;
   IF ichild < num_desc THEN;
      ichild <-- ichild + 1;
      ADVANCE $current_desc;
      RESUME match;
   END;
   ELSE;
      return /* out of coroutine -- up one level */;
   END;
END trav_template;
```

```
COROUTINE trav_tree;
   IF jchild < num_data THEN;
      jchild <-- jchild + 1;
      ADVANCE $current_data;
      RESUME match;
   END;
   ELSE;
      return /* out of coroutine -- up one level */;
   END;
END trav_tree;
```

```
COROUTINE err;
   IF norepeat = 0 THEN;
      WRITE message, 'level-->',level;
                     'LABEL-->',LABEL($current_data);
                     'does not agree with template being used';
   END;
   ELSE;
      norepeat <-- 0;
   END;
   RESUME match;
END err;
```

```
PROC numeric;
   IF TYPE($current_data) °= $current_desc.VALUE THEN;
      WRITE $current_data;
      WRITE message, 'type does not match template';
   END;
END numeric;
```

```
PROC scalor;
   IF $mode = 'build' THEN;
      LABEL($LABEL) <-- LABEL($current_data);
      $current_data <-- $current_desc.CONSTANT;
      LABEL($current_data) <-- LABEL($LABEL);
   END;
   ELSE /* $mode = 'check' */;
      IF $current_data °= $current_desc.CONSTANT THEN;
         WRITE $current_data;
         WRITE message, 'constant required but not found';
      END;
   END;
END scalor;
```

```
PROC list_commands;
  WRITE message '     Experimental template editor for trees';
  WRITE message '                help utility';
  WRITE message '  ';
  WRITE message 'h   -help!:  list the commands.';
  WRITE message 'a   -advance one position across siblings.';
  WRITE message 'd   -move down to first child of current node.';
  WRITE message 'u   -move up to parent of current node. ';
  WRITE message 'r   -move to the root of the tree.';
  WRITE message 'w   -write current subtree to screen.';
  WRITE message 'l   -assign LABEL to current node.';
  WRITE message 'v   -assign value to current node.';
  WRITE message 'ib  -insert before current node a blank node.';
  WRITE message 'ia  -insert after current node a blank node.';
  WRITE message 'id  -insert down, as last child, a blank node.';
  WRITE message 'wt  -write current template tree to screen';
  WRITE message 'quit-quit editing the current tree.';
  WRITE message ' ';
  WRITE message 'return to resume editing';
  get edit($ans) (a1);
  return;
END list_commands;
```

```
PROC pushptr (IN j: INT, INOUT i: INT ARRAY[1,21]);
   icount <-- i(21);
   IF icount >= 20 THEN;
      WRITE message, 'cannot store another pointer';
      return;
   END;
   ELSE;
      icount <-- icount + 1;
      i(icount) <-- j;
      i(21) <-- icount;
      return;
   END;
END pushptr;


PROC popptr( IN i:INT, OUT $t:TREE);
   icount <-- i(21);
   IF icount = 0 THEN;
      WRITE message, 'cannot repeat -- subtree missing';
      return;
   END;
   ELSE;
      j <-- i(icount);
      icount <-- icount - 1;
      i(21) <-- icount;
      CALL toptr;
   END;
END popptr;


PROC toptr;
   $j <-- $i;
   return;
END toptr;
```

```
PROC newlevel (IN $desc: TREE, $data: TREE, $mode: CHAR,
            INOUT num_desc: INT, num_data: INT, level: INT,
                  ipoint: INT ARRAY[1,21]);
   DEFINE $current_desc AS $desc;
   DEFINE $current_data AS $data;
   SET ichild, jchild, flag, kount <-- 1, 1, 0, 0;
   $flag <-- '';
   level <-- level + 1;
   CALL compare;
   level <-- level - 1;
   return;
END newlevel;
```

```
COROUTINE compare;
one:
   IF LABEL($current_desc) ='&' THEN;
      DEFINE $parent as $current_desc;
      DEFINE $current_desc as $current_desc(1);
      flag=1;
      CALL pushptr(ipoint,$parent);
      GOTO one;
   END;
   ELSEIF LABEL($current_desc)='¬' THEN;
      WRITE message, 'Wish to repeat the subtree?(Y/N)';
a1:
   INPUT $reply;
   IF $reply='Y' THEN;
      CALL popptr(ipoint,$temp);
      DEFINE $current_desc as $temp;
      GOTO one;
   END;
   ELSEIF $reply='N' THEN;
      IF $flag='ADVANCE' THEN;
         RESUME template;
      END;
      RESUME command;
   END;
   ELSE;
      WRITE message, 'invalid reply, try again-(Y/N)';
      GOTO a1;
   END;
   RESUME command;
   ELSEIF LABEL($current_desc)='$' THEN;
      irepeat <-- $current_desc.REPEAT;
      IF kount = 0 THEN;
         WRITE message, 'Wish to use this structure? (Y/N)';
         WRITE ($current_desc(2));
      END;
      ELSE;
         WRITE message, 'Wish to repeat structure?(Y/N)';
a2:
         INPUT $reply;
         IF $reply='N' THEN do;
            RESUME template;
         END;
         ELSEIF $reply='Y' THEN;
            kount <-- kount + 1;
            IF (kount <= irepeat) THEN;
               DEFINE $parent as $current_desc;
               DEFINE $current_desc as $current_desc(2);
               flag <-- 1;
               GOTO one;
            END;
         END;
```

```
         ELSE;
            WRITE message, 'invalid, try again-(Y/N)';
            GOTO a2;
         END;
   END /* repeat structure */;
   ELSEIF LABEL($current_desc)='*' THEN;
      valid <-- 0;
      DEFINE $choice as $current_desc.CHOICE;
      IF LABEL($current_data)='' THEN;
         WRITE message, 'enter LABEL from choices below:';
         WRITE ($choice);
         INPUT $nLABEL;
         LABEL($current_data) <-- $nLABEL;
      END;
a3:
      DO FOR ALL SUBNODES OF $choice USING $ptr;
         IF LABEL($ptr)=LABEL($current_data) THEN;
            valid <-- 1;
      END /* do for all */;
      IF valid = 0 THEN;
         WRITE message, 'enter LABEL from choices below:';
         WRITE ($choice);
         INPUT $nLABEL;
         LABEL($current_data)=$nLABEL;
         GOTO a3;
      END;
      DEFINE $parent as $current_desc;
      DEFINE $current_desc as $current_desc(2);
      flag <-- 1;
      GOTO one;
   END /* LABEL is choice of several */;
   ELSEIF LABEL($current_desc)='#' THEN;
      IF LABEL($current_desc(1))='CONSTANT' THEN;
         CALL constant;
      END;
      IF LABEL($current_desc(1))='VALUE' THEN;
         CALL type_check;
      END;
      RESUME command;
   END /* LABEL accepted to any string */;
   ELSEIF LABEL($current_data)='@'
         |LABEL($current_data)='' THEN;
      LABEL($current_data) <-- LABEL($current_desc);
      GOTO one;
   END /* exact LABEL required */;
   ELSEIF LABEL($current_desc)=LABEL($current_data) THEN;
      IF LABEL($current_desc(1))='CONSTANT' THEN;
         CALL constant;
      END;
      IF LABEL($current_desc(1))='VALUE' THEN;
         CALL type_check;
      END;
      RESUME command;
   END;
```

```
   ELSE;
      RESUME command;
   END;
END compare;
```

```
COROUTINE command;
one:
   WRITE 'level = ',level;
   IF LABEL($current_data)='' THEN;
      WRITE 'current node LABEL-->@';
   END;
   ELSE;
      WRITE 'current node LABEL-->',LABEL($current_data);
   END;
   IF NUMBER($current_data)=0 THEN;
      WRITE 'current node value -->',$current_data;
   ELSE;
      WRITE 'number of children = ',number($current_data);
   END;
   WRITE 'enter command';
   INPUT $cmd;
   /*
      identify and process with IF-THEN-ELSE-IF logic
   */
   /*  Help command */
   IF $cmd='H' THEN;
      CALL list_commands;
      GOTO one;
   END;
   /*  CHECK command */
   IF $cmd='CHECK' THEN;
      $mode = 'check';
      CALL check;
      GOTO one;
   END;
   /*  QUIT by returning */
   ELSEIF $cmd='QUIT' THEN;
      $flag='QUIT';
      return /* to NEWLEVEL */;
   END;
   /* return to ROOT level */
   ELSEIF $cmd='R' THEN;
      IF level > 0 THEN;
         $flag='ROOT';
         return /* to NEWLEVEL */;
      END;
      ELSE;
         WRITE message, 'already at root node';
         GOTO one;
      END;
   END;
   /*  Write current template subtree  */
   ELSEIF $cmd='WT' THEN;
      WRITE $current_desc;
      GOTO one;
   END;
   /*  Write current subtree  */
   ELSEIF $cmd='W' THEN;
      WRITE $current_data;
```

```
      GOTO one;
   END;
   /*  change Label of current node */
   ELSEIF $cmd='L' THEN;
      WRITE message, 'enter new label';
      INPUT $nlabel;
      LABEL ($current_data)=$nlabel;
      RESUME compare;
   END;
   /*  assign a Value to the current node, avoiding
       unwanted destruction of substructure. */
ELSEIF $cmd='V' THEN;
   IF number($current_desc)=0|
         LABEL($current_desc(1))='VALUE' THEN;
      IF number($current_data) > 0 THEN;
         WRITE 'caution: value axes substructure';
         WRITE 'do you wish to proceed? (Y/N)';
         INPUT $reply;
      END;
      ELSE;
         $reply <-- 'Y';
      END;
      IF $reply='Y' THEN do;
         WRITE message, 'enter new value';
         INPUT $nvalue;
         LABEL($label) <-- LABEL($current_data);
         $current_data <-- $nvalue;
         LABEL($current_data) <-- LABEL($LABEL);
         RESUME compare;
      END;
      ELSE;
         WRITE message, 'template not at leaf level';
         GOTO one;
      END;
   END;
   /*  Insert a node Before the current node
                and move to the new node */
   ELSEIF $cmd='IB' THEN;
      IF level>0 & LABEL($current_desc)='$' THEN;
         LABEL($blank) <-- '@';
         INSERT $blank BEFORE $current_data;
         num_data <-- num_data+1;
         IF flag=1 THEN;
            DEFINE $current_desc as $parent;
            flag=0;
         END;
         $mode='build';
         CALL check;
         RESUME compare;
      ELSE;
         WRITE message, 'insert before not valid here';
         GOTO one;
      END;
   END;
```

```
/*  Insert a node After the current node
            and move to the new node */
ELSEIF $cmd='IA' THEN;
   IF level>0 & LABEL($current_desc)='$' THEN;
   END;
   ELSEIF level>0 & ichild=num_data THEN;
      $flag <-- 'ADVANCE';
      RESUME template;
   END;
   ELSE;
      WRITE message, 'insert after not valid here';
      GOTO one;
   END;
   LABEL($blank) <-- '@';
   DEFINE $tmp as $current_data;
   ADVANCE $tmp;
   INSERT $blank BEFORE $tmp;
   IF $tmp IDENTICAL TO $null THEN;
      PRUNE $tmp;
   END;
   ADVANCE $current_data;
   jchild <-- jchild + 1;
   num_data <-- num_data+1;
   $mode='build';
   CALL check;
   RESUME compare;
END;
/*  Insert a node as next child of the current node and
    initiate down to new level */
ELSEIF $cmd='ID' THEN;
   IF NUMBER($current_desc)=0|
       LABEL($current_desc(1))='VALUE'|
       LABEL($current_desc(1))='CONSTANT' THEN;
      WRITE message, 'cannot insert below template leaf';
      GOTO one;
   END;
   IF number($current_data)>0 THEN;
      WRITE message, 'insert down not valid here';
      GOTO one;
   END;
   LABEL($blank) <-- '@';
   $current_data(next) <-- $blank;
   $flag <-- 'DOWN';
   $mode <-- 'build';
   CALL check;
   RESUME template;
END;
/*  move Up level by returning */
ELSEIF $cmd='U' THEN;
   IF level>0 THEN;
      return;
   END;
   ELSE;
      WRITE message, 'cannot move up beyond root node';
```

```
      END;
   /*  Advance to next sibling */
   ELSEIF $cmd='A' THEN;
      $flag <-- 'ADVANCE';
      IF level>0 THEN;
         RESUME data;
      END;
      WRITE message, 'cannot advance while on root node';
      END;
      RESUME template;
      RESUME data;
   END;
   /*  Down to first child of next level */
   ELSEIF $cmd='D' THEN;
      $flag <-- 'DOWN';
      RESUME template;
   ELSE;
      WRITE message, 'command not valid';
      GOTO one;
   END;
END command;
```

```
PROC type_check;
   IF type($current_desc.VALUE)>0|
        type($current_desc.VALUE)<4|
        $current_desc=0 THEN;
      IF $current_data = $null THEN;
a3:
      IF $current_desc.VALUE=1 THEN WRITE 'enter string';
      IF $current_desc.VALUE=2 THEN WRITE 'enter integer';
      IF $current_desc.VALUE=3 THEN WRITE 'enter float';
      IF $current_desc=0 THEN WRITE 'enter new value';
         LABEL($label) <-- LABEL($current_data);
         INPUT $nvalue;
         $current_data <-- $nvalue;
         LABEL($current_data) <-- LABEL($label);
      END;
      IF $current_desc.VALUE ^= type($current_data) THEN;
         WRITE message,'data type of leaf not
                        consistent with template';
         GOTO a3;
      END;
   END /* data type check */;
   ELSE;
      WRITE message, ' invalid - value at non-leaf node';
   END;
END type_check;
```

```
PROC constant;
   LABEL($label) <-- LABEL($current_data);
   $current_data <-- $current_desc.CONSTANT;
   LABEL($current_data) <-- LABEL($label);
    END constant;
```

```
PROC template;
one:
   IF $flag='ADVANCE' THEN;
      GOTO two;
   END;
   IF NUMBER($current_desc) > 0 THEN;
      IF NUMBER($current_data) > 0 THEN;
         m=NUMBER($current_desc);
         n=NUMBER($current_data);
         CALL newlevel;
         IF $flag='ROOT' & level>0 THEN;
            return;
         END;
         ELSEIF $flag='QUIT' THEN;
            return;
         END;
         ELSE;
            RESUME command;
         END;
      ELSEIF LABEL($current_desc(1))='CONSTANT'|
             LABEL($current_desc(1))='VALUE' THEN;
         WRITE message, 'cannot move down from leaf node';
         RESUME command;
      END;
   END;
      WRITE 'node does not exist';
      RESUME command;
two:
   IF flag=1 THEN;
      DEFINE $current_desc as $parent;
      flag <-- 0;
      IF LABEL($current_desc)='$' THEN;
         RESUME compare;
      END;
   END;
   IF ichild < num_desc THEN;
      ichild <-- ichild + 1;
      ADVANCE $current_desc;
      RESUME command;
   ELSE;
      WRITE message, 'cannot advance beyond template';
      RESUME compare;
   END;
END template;
```

```
PROC data;
   IF jchild < num_data THEN;
      jchild <-- jchild +1;
      ADVANCE $current_data;
      RESUME command;
   END;
   ELSE;
      WRITE message, 'cannot advance past right most node';
      RESUME compare;
   END;
END data;
```

APPENDIX C

USER'S GUIDE

This User's Guide is a brief description of how to start an editing session, what messages and prompts will appear, and a review of the command set that is available.

The command to run the editor is TED, an acronym for Template EDitor. TED can have up to two parameters and must have at least one. If only one parameter is used the program assumes that the file specified is to be used as the data tree and the generic template, which allows any substructure or labels to be built, as the template. The structure of the generic tree is shown in figure 11. If two parameters are found, then the first is used as the template while the second is used as the data file. The

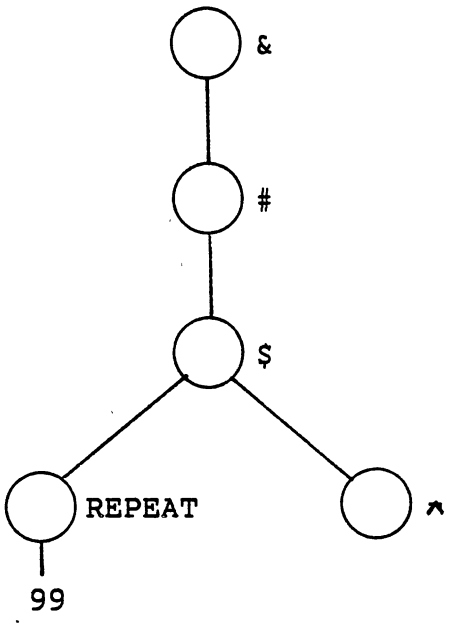


Figure 11. Generic Template for Template Driven Editor

template must exist but the data file can be created at the start of a editing session.

The error messages are classified as informative. Informative messages are those that relay the problem and continue processing, ignoring the command that caused the error. For example, if an invalid command were entered the message would be as follows:

```
invalid command
enter command
```

Many commands are not valid through out the entire session. The editor will inform the user of the invalid command and wait for input of a legal command. Other prompts that appear often during editing are those concerned with repeat structures. Both vertical and horizontal repeats prompt the user to answer if the structure should be used and awaits a yes or no input.

The command set is dynamic, in that some commands are invalid determined by the location of the editor in the template tree. The Help command is valid at all times and displays on the screen a summary of the entire command set available.

Advance is valid only when a right sibling exists.

Down is valid only when substructure exists below current node.

Up is valid at any level in tree except the root node.

Root is valid at any location in the tree and the root becomes the current data tree node.

Write and Write Template are valid at all times.

Label and Value are valid when in agreement with the template used.

Insert Before is only valid when the node to be inserted is a part of a horizontal repeat structure.

Insert After is only valid when the node to be inserted is a part of a horizontal repeat structure.

Insert Down is only valid when a vertical repeat structure is used or when no substructure exists for the current node.

QUIT is valid at any time.

APPENDIX D

UNIX COMMAND PROCEDURE

On the following pages is a sample of a command procedure written for the UNIX operating system. PLANS uses Fortran unit numbers, which are difficult to distinquish when listed in a catalog or directory. The command procedure allows the editor to be called using logical file names as parameters. The procedure checks for existence of the file and associates the logical file name with the physical Fortran unit number.

```
#
#           experimental template editor for trees
#                 shell script
#
#     $1 - name of template tree used for edit session
#     $2 - file name of data tree used for edit session
#
#***
#**   determine which template and data files
#***
echo "  "
if ($#argv == 1) then
   echo 'default template file - GENERIC will be used'
   set tfile = GENERIC
   set dfile = $argv[1]
else
   echo $argv[1] 'template file will be used '
   set tfile = $argv[1]
   set dfile = $argv[2]
endif
#***
#**   link files to fortran equivalents
#***
if (! -r $tfile) then
   echo 'template file is not available'
   exit (102)
endif
cp $tfile fort.1
if (! -r $dfile) then
   echo 'new file '
else
   cp $dfile fort.2
endif
#***
#**   check to see if runfile is there
#***
if (! -r editor.run) then
   echo 'run file is not available'
   exit (103)
endif
cp editor.run RUNFILE
#***
#**   check to see that the interpreter error file is there
#***
if (! -r /u2/pjrv/exe/INTERR) then
   echo 'required interpreter error message file not found'
   exit (104)
endif
#***
#**   check to see that the interpreter is there
#***
if (! -r /u2/pjrv/v2.0/exe/pint.ex) then
   echo 'the interpreter is not available'
   exit (105)
```

```
endif
echo " "
#***
#**   run the interpreter
#***
cp /u2/pjrv/v2.0/exe/INTERR INTERR
/u2/pjrv/v2.0/exe/pint.ex
rm INTERR  RUNFILE
#***
#**
#***
echo " "
echo 'Would you like to save the data tree? '
set resp = ¢gets¢
if ( $resp == "Y" || $resp == "YES") then
   echo 'What would you like to name it? '
   set tree = ¢gets¢
al:
      if ( -e $tree) then
         echo $tree 'already exits.  Want to overwrite it? '
         set res = ¢gets¢
         if ( $res == "Y" || $res == "YES") then
            mv fort.3 $tree
         else
            echo 'New file name: '
            set tree = ¢gets¢
            goto al
         endif
      else
         mv fort.3 $tree
      endif
else
   rm fort.3
endif
rm fort.1 fort.2
```

VITA

Ronald Dean Moore

Candidate for the Degree of

Master of Science

Thesis: DEVELOPMENT OF AN EXPERIMENTAL TEMPLATE DRIVEN EDITOR FOR TREES

Major Field: Computer Science

Biographical:

Personal Data: Born in Atoka, Oklahoma, July 20, 1960, the son of Mr. and Mrs. David Moore.

Education: Graduated from Atoka High School, Atoka, Oklahoma, in May, 1978; received Bachelor of Science degree in Computing Science from Southeastern Oklahoma State University in December, 1981; completed requirments for the Master of Science degree at Oklahoma State University in May, 1984.

Professional Experience: System Operator at S.E.O. Computer Services; Durant, Oklahoma, November, 1979 to December, 1981. Programmer/Consultant at Dicus Corporate Offices; Ada, Oklahoma, May, 1982 to August, 1982. Graduate Teaching Assistant, Department of Mathematics, Oklahoma State University, Stillwater, Oklahoma, January, 1982 to December, 1982. Graduate Teaching Assistant, Department of Computing and Information Sciences, Oklahoma State University, Stillwater, Oklahoma, January, 1983 to January, 1984.