To The Graduate School:

I am submitting herewith a thesis written by Robert H. Kroboth entitled: "The Detailer's Assistant: the Enhancement of a Prototype Expert System Using M.1 and C." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael G. Thomason

Michael G. Thomason, Major Professor

We have read this thesis
and recommend its acceptance:

J. R. B. Corbett

Accepted for the Council:

Vice Provost
and Dean of The Graduate School

THE DETAILER'S ASSISTANT: THE ENHANCEMENT OF A PROTOTYPE EXPERT SYSTEM USING M.1 AND C

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Robert H. Kroboth

June 1987

In Loving Memory of Eileen Kroboth Burke

ACKNOWLEDGEMENTS

I wish to thank the many people who have given me their support and encouragement.

I wish to thank Dr. Gunar Liepins, of the Oak Ridge National Laboratory, for giving me the opportunity to work on this thesis project.

I especially want to thank Dr. Michael Thomason for serving as my thesis advisor, as well as Dr. Robin Cockett and Dr. David Straight for serving on my thesis committee. Thanks is also extended to Dr. Charles Pfleeger and Ms. J. Wallace Mayo for the support and encouragement they gave me throughout my undergraduate and graduate education.

Finally, sincerest gratitude is extended to my wife, Ty Kroboth.

ABSTRACT

The Detailer's Assistant is a prototype expert system designed to recommend Navy personnel assignments. To enable the effective use of the system by Naval personnel, file access, explanation, and user interface facilities were developed and incorporated into the existing system. These facilities were developed by designing software using the languages C and M.1.

The Explanation Network Development System (ENDS) was created to provide a tool to enable the creation and modification of the files used by the explanation facilities.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This thesis concerns the enlargement and enhancement of the Detailer's Assistant prototype expert system. The Detailer's Assistant prototype is an expert system developed by Gunar E. Liepins, Ph.D., of the Oak Ridge National Laboratory. The system is written in M.1 and is designed to run on the IBM PC-AT and compatibles. M.1 is a rule-based language designed for the development of expert systems. The Detailer's Assistant expert system is under development for the Navy Military Personnel Command (NMPC) to assist Navy personnel (detailers) in the assignment of enlisted personnel (sailors) to jobs within the Navy.

The Detailer's Assistant prototype is capable of assigning sailors using a limited set of rules that provides assignment recommendations for personnel within a limited set of circumstances. The purpose of this thesis is to add to the capabilities of the Detailer's Assistant by designing user and file access interfaces to M.1, and to design and develop additional rules in M.1. The interfaces and rules are designed to enlarge the Detailer's Assistant's domain of knowledge, and to provide explanation capabilities that can inform the detailer as to why the prototype expert system recommends a particular assignment.

The Detailer's Job

Within the Navy's assignment system, enlisted personnel are assigned to tours of duty lasting from two to five years. When a sailor nears the completion of a tour, the sailor is assigned to another tour of duty. This reassignment is known as a Permanent Change of Station (PCS). The assignment is designed to maintain acceptable personnel manning levels among the Navy's sea, shore and overseas units. The assignment is also required to take into consideration the qualifications of the individual, PCS funding, and jobs (billets) available to be filled. Additionally, the preferences of the individual can influence the assignment decision. The Enlisted Transfer Manual (ETM) documents the rules and regulations designed to ensure that the above considerations, along with many other factors are considered every time an assignment is made. It is the detailer's responsibility to assign sailors in accordance with the ETM.

The detailer is an enlisted Navy person trained in a particular skill (such as a cook or aircraft mechanic) who is assigned to a two-year tour as a detailer. The detailer is responsible for the assignments of other sailors in his/her same skill category (community). There can be up to several thousand enlisted personnel in any one community. There are two to three detailers for each community. It is

not unusual for one detailer to make one to two hundred assignments per week [8].

New detailers begin their tour of duty with very little knowledge about the enlisted assignment system and must learn detailing while on the job. This can prove to be difficult for the new detailer. The ETM is a large set of complicated, interrelated regulations. The detailer makes and confirms many assignments over the phone with the particular sailor being assigned. This means the detailer must have a clear understanding of a significant portion of the ETM to make the assignment in accordance with Navy regulations, while trying to satisfy the desires of the sailor [8].

Role of the Detailer's Assistant Expert System

There is an overwhelming amount of information that must be analyzed for an assignment decision. It is nearly impossible for new detailers to make sound assignment decisions without the help and training of experienced detailers. It is even difficult for experienced detailers to look consistently at all of the information that is available and to choose the best assignment decision for both the Navy and the sailor.

The Detailer's Assistant is under development to analyze, on a case by case basis, each sailor's assignment

situation and the applicable rules found in the ETM. It will then present to the detailer a list of possible billets, in descending order of desirability, for the assignment of that particular sailor. In addition, the Detailer's Assistant will display a brief explanation of the decisions involved with selecting each listed billet.

Expert systems through computer modeling of expert human reasoning, can solve hard problems that require human expertise [3]. By using expert system technology, the Navy can capture the knowledge of the best experienced detailers and allow all detailers to use this knowledge. Once familiar with the Detailer's Assistant, the novice detailer can use the expertise of an experienced detailer on demand. Experienced users can use the Detailer's Assistant to provide the best assignment recommendations and present these to the constituent as possible assignments.

Organization of the Remaining Chapters

Chapter II - M.1: An Expert System Development Tool. This chapter gives an overview of expert systems and explains differences between rule-based systems and conventional languages. The major components of M.1, the expert system shell used to develop the Detailer's Assistant are presented.

Chapter III - Expansion of the Detailer's Assistant using M.1 and C. This chapter describes the design and development of the Detailer's Assistant's external routines, including the explanation generation system.

Chapter IV - The Explanation Network Development System (ENDS). Output from this system is used by the explanation generation system to present assignment explanation to the detailer. This chapter discusses the design of ENDS.

Chapter V - Summary. This chapter summarizes the facilities developed, and presents areas for future research.

CHAPTER II

M.1: AN EXPERT SYSTEM DEVELOPMENT TOOL

Expert Systems are computer programs designed to solve real world problems normally solved by experts in a particular field. To accomplish this, designers of expert systems endeavor to represent the knowledge of an expert in a computer program. This chapter describes differences between expert system development tools, also known as knowledge engineering tools, and conventional procedural programming languages. An overview of M.1, the expert system shell presently utilized to develop the Detailer's Assistant, and the criteria used in its selection are given.

Differences between Knowledge Engineering Tools and Conventional Programming Languages

Conventional procedural languages, such as PASCAL and C, allow the programmer to write programs that solve specific problems. With these languages, the order of program execution is embedded within the statements of the language and encoded by the developer. Generally, the computer executes the instructions generated from these languages in sequential order. The programmer determines an algorithm that will provide the correct answer and then writes the sequence of code that, when executed sequential-

ly, will provide the correct answer. Also, these languages normally allocate the memory needed for scalar variables and arrays at compile time. During execution, the space available for this data is fixed [3].

Conversely, the execution of programs written in a knowledge engineering language is generally data driven. That is, sections of these programs are executed by situation instead of a planned sequence. These languages employ dynamic allocation of data. This means that the size of an individual data object can change during execution. This is a necessity for list processing languages.[3]

Additionally, many knowledge engineering tools use an inference engine to provide a built-in problem-solving mechanism. Knowledge engineering tools attempt to provide mechanisms to encode knowledge about particular real world situations. The inference engine uses this encoded knowledge, or knowledge base, when executing, to provide answers or advice on situations that are within the scope of the encoded knowledge domain. The knowledge engineering tools allow the system designers to use the heuristics extracted from domain experts to be encoded into the knowledge base.

In addition to providing the problem-solving instructions for knowledge engineering tools, the inference engine generally provides other functions that are not directly related to the knowledge base. Inference engines provide user input and system output facilities. If the knowledge

engineering tool provides file access and linkage to other languages, these facilities are said to be contained in the inference engine [12].

M.1: A Rule-Based Knowledge Engineering Tool

M.1 allows the system designer to represent domain knowledge using rules [10]. Each rule in this type of system generally represents a portion of the domain knowledge. Figure 2.1 shows the general architecture of M.1.

When invoked, the M.1 inference engine seeks to solve a specific problem within the domain of the knowledge base. The inference engine uses the rules and facts in M.1's knowledge base to find conclusions. These conclusions are stored in its working memory, called the cache, for future reference. Additionally, M.1 also provides facilities to read and write from disk to its cache thus providing a means to store and read its working memory.

Entries in the cache are of the form

EXPRESSION = VALUE, cf, CF because REASON

and represent conclusions that are interpreted as "**EXPRESSION** has the value **VALUE** with **CF** percent certainty, and was concluded because **REASON**" [10]. There can be multiple

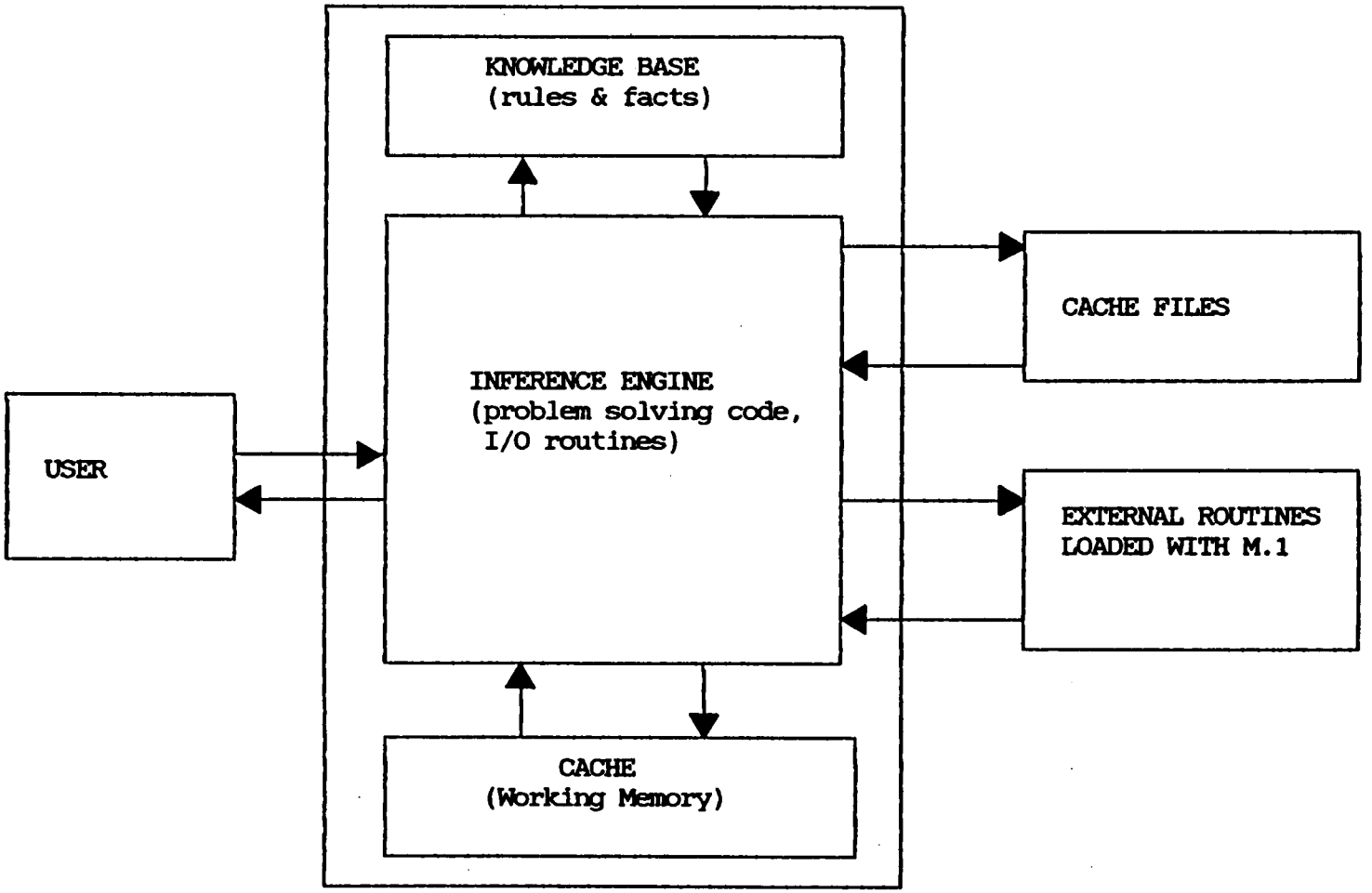

Figure 2.1: The General Architecture of M.1

entries for an **EXPRESSION** stored in the cache, dependent upon whether the expression is single-valued or multi-valued.

Knowledge Base Entries

Facts provide a particular piece of information about the domain knowledge. Facts are knowledge base entries of the form

LABEL: EXPRESSION = VALUE cf **INTEGER.**

and can be used by the inference engine to conclude a value for **EXPRESSION** [10]. For example: if the inference engine sought a value for **largest_animal** and encountered the following fact in the knowledge base

kb-1: largest_animal = whale cf **100.**

then

largest_animal = whale cf **100** because **kb-1.**

would be stored as the last entry in the cache.

Meta-facts provide information to the inference engine regarding how to conclude a value for a particular expression. One such meta-fact, the question meta-fact, has the form

question(**EXPRESSION**) = [**TEXT**].

and will display on the screen the string contained in **TEXT.** This allows the user to input the value for **EXPRESSION** from the keyboard.

Rules are knowledge base entries of the form

```
if PREMISE
then CONCLUSION.
```

where a PREMISE is composed of propositions of the form

```
EXPRESSION = VALUE
```

each joined by a boolean operator [10]. This is illustrated by the following example:

```
if temp = hot and
   sunny = yes and
   chlorine_level = acceptable
then ok_to_swim = yes.
```

The CONCLUSION is also made up of propositions of the form:

```
EXPRESSION = VALUE cf N
```

When the inference engine tests the premise of a rule and finds that it evaluates to TRUE, the conclusion is stored in the cache.

The Inference Engine

Upon activation, the inference engine seeks the expressions found in the goal knowledge base entry. This entry has the form:

```
goal = consult.
```

If this entry actually appears in a knowledge base, M.1's inference engine begins searching to find a value for the expression consult.

If it finds a cache entry of the form

consult = VALUE cf 100 because REASON

it then displays the entry to the user. If there is no such entry, the engine sequentially searches the knowledge base for an entry that might enable it to conclude a value for consult. If the entry is a fact, the search is successful and the value found for consult is displayed to the user.

If the entry is a meta-fact, the engine executes the underlying code of the meta-fact, to determine the value for the expression. In the case of the question meta-fact, the value for the expression, consult, is obtained from the user's response to question's prompt.

If the inference engine finds a rule that contains the following proposition

consult = VALUE cf CF

in its conclusion, it tests each clause in its premise. If all clauses are true, the rule succeeds, so the value for consult is noted in the cache and M.1 displays this value to the user. If any of the clauses in the premise test false, the rule fails and the sequential search continues, from this point, for another knowledge base entry.

If M.1 cannot conclude the value of an expression after searching the cache and the knowledge base entries, it generates, under default conditions, a question for the user to provide a value for the expression it is seeking. If the inference engine encounters the meta-fact

noautomaticquestion(consult).

in the knowledge base, the question is not generated and the search fails. The user is then notified that M.1 could not find a value for consult.

Rule Testing

When testing the premise of a rule, M.1 compares the **EXPRESSION** of each clause with its associated **VALUE**. M.1's inference engine uses the above search procedure to find the value of **EXPRESSION** before testing the clause. In the course of searching for **EXPRESSION**, it may test the premise of other rules that might conclude a value for **EXPRESSION**. This process of seeking a value for **EXPRESSION** by testing other rules is called **backward chaining**. **Backward chaining** is the underlying problem-solving method used by the M.1's inference engine [10].

The Selection of the Detailer's Assistant Development Tools

The development of the Detailer's Assistant began in late 1985. Prior to this time, Dr. Liepins selected the hardware and software to develop the Detailer's Assistant. He used the following criteria to select the hardware and software for the Detailer's Assistant:

1. Hardware - Cost was the major criterion for choosing the hardware for the delivery system.

The U.S. Navy determined that the total cost for hardware and software should not exceed $5000.00 per system. The choice was made to deliver the system on IBM compatible microcomputers.

2. Software - The Assignment problem could be best represented using a frame-based or production rule system. Funding constraints dictated the timely development of a prototype. Time constraints limited the software search to systems equipped with the inference engine and I/O utilities.

CHAPTER III

THE EXPANSION OF THE DETAILER'S ASSISTANT USING M.1 AND C

The Detailer's Assistant was developed using M.1's built-in user interface and file access facilities. As development of the Detailer's Assistant continued, the system's requirements necessitated the modification of these facilities and the development of explanation capabilities not provided by M.1.

M.1 does not permit the direct modification of its capabilities. It does permit executable routines to be linked and invoked, by M.1 at run-time to perform additional tasks. Several routines were written and linked to M.1 to provide the needed capabilities. This chapter discusses M.1's external interface, user interface, and file access facilities with their limitations. The design and implementation of the Detailer's Assistant's new user interface, file access, and explanation facilities written in M.1 and C are also discussed.

M.1's External Interface

Teknowledge, the developer of M.1, has provided an interface from M.1 to external routines written 'C'. These routines can be compiled and linked with interface routines

provided by Teknowledge. The resulting file can then be loaded along with an executable version of the expert system at run-time.

Interfacing an External Routine

When execution begins, M.1 invokes the routine Initext which contains one function call to the routine Addfunction for each external C routine. For example, if it is desired to allow M.1 to invoke the external routine Get_data, a C statement similar to

```
Addfunction("getdata",1,Get_data)
```

would be included in the routine Initext. Addfunction uses the three arguments to build a table of addresses to subroutines. The first two arguments provide a lookup value for use by the External meta_proposition to invoke the routine Get_data. The third argument is the address of the external subroutine to be invoked by M.1.

The External meta-proposition, when executed by M.1, permits execution of external routines. For example, if M.1 sought value for the expression get_age, and M.1 encountered the rule

```
If ssn = SSN and
   external(getdata,[SSN]) = [AGE] and
   not(AGE = error)
then get_age = AGE.
```

in the knowledge base, the inference engine would execute the external function that has the corresponding look-up

entry "getdata" in M.1's external function table. The second argument in the **External** meta-proposition is a list of values, separated by commas that M.1 permits the external functions to access. The external routines are permitted to pass to M.1 the list to the right of the equal sign. The number of values in either list cannot exceed 127.

External routines use the Import routine, supplied by Teknowledge, to access the values in the **External** meta-proposition's second argument. Import places these values into memory locations that can be addressed through variable reference by external routines. For example, the call

```
import(1,STRING,instr,11)
```

would access the first value in the **External** meta_proposition's second argument and place the first 10 bytes and a NULL character in consecutive memory locations starting at the address instr. If there are other values to be imported into the external routine additional calls to the routine Import are needed.

The routine Export allows the external interface to pass back information to M.1. For example, if the external routine made the call

```
export(1,INTEGER,&age)
```

Export would place the value at the address referenced by age into memory M.1 could address. Import and Export permit the transfer of three scalar types: real, integer, and string.

M.1's File Access Facilities

M.1 permits the reading and writing of cache files. Cache files are ASCII files that consist of zero or more strings in cache entry format that are terminated with a line feed and carriage return.

Within the knowledge base, M.1 uses the Do meta-proposition to read and write cache files. If the inference engine encountered the rule

```
if do(loadcache record.dat)
then record_loaded = yes.
```

while trying to seek the value record_loaded, the inference engine would attempt to open and read the entire contents of the MS-DOS file, record.dat. If the file was not present or the contents of the file was not in cache entry format, M.1 would display a fatal error message and exit to MS-DOS. If there was no error detected, the cache entries in the file would be appended to the cache.

Similarly, M.1's working cache can be written to a cache file using the Do meta-proposition. For example, the entire content of the cache is written to the ASCII file cache.dat when the Do meta-proposition

```
do(savecache cache.dat)
```

is executed by the inference engine.

M.1's file access facilities do not permit access to files in a format other the cache file format. In addition, these file access facilities do not permit testing of

individual cache entries before they are appended to the cache or written to a cache file.

Programs generating files in cache file format for M.1 to read must ensure that the cache entries written into the file are needed by M.1 to complete its searches. If the cache entries read and appended to the cache deplete M.1's working storage, M.1 will display an error and stop execution. This occurs because the availability of M.1's working memory depends on three factors: the amount of MS-DOS addressable random access memory available on the microcomputer, the size of the knowledge base and external code, and the number of cache entries already in M.1's working memory [10].

The Detailer's Assistant's External File Access Interface

During the Detailer's Assistant's execution, the Navy's billets (job openings) must be made available to the expert system to permit the system to determine and display the jobs which represent the best assignment for a sailor. Initially, the Do meta-proposition was used to append a small number of cache entries, representing the job openings, to the cache. Although this was acceptable for the initial prototype, in reality, there could be up to several thousand job openings available at any time. The system

The Record Editing System

It is necessary for the detailer to review each sailor's personnel record and, if needed, modify any field of the sailor's record prior to assignment. The editing system provides a facility that permits the editing of a personnel record and writes this record to the file sailor.dat in cache entry format. This file is accessed by the expert system by executing a Do meta-proposition.

Due to MS-DOS memory restrictions, editing a sailor's personnel record is performed by a separate program. The Detailer's Assistant is executed after editing. The execution of each program is controlled by the MS-DOS batch file, DETAILER.BAT. This file controls the individual program execution for the entire assignment system.

There are two major data structures used by changper-.exe, the personnel record editing program. They are the personnel record and the screen record structure. The personnel record structure contains fields to store individual sailor information. The screen structure, shown in figure 3.1, stores data needed by screen editing routines to display and edit the fields in the personnel record.

would crash if an attempt was made to append all possible job openings to the cache.

To allow access to a file of job openings that contain a realistic number of billets, the Detailer's Assistant's knowledge base was modified to provide a set of arguments that would qualify a billet for further analysis by the Detailer's Assistant. The set of arguments is passed through an External meta-proposition call to an external C routine. The C routine reads each billet, from the file requis.dat, and compares the data in its fields with the arguments passed from the Detailer's Assistant. If the fields match, the billet is appended in cache entry format to the file billet.cac. Billet.cac can then be read in using the Do meta-proposition. Typically less than ten billets are appended to the cache using this method.

M.1's User Interface

M.1's user interface permits the expert system designer to enable user input and output with minimal programming effort. M.1's facilities do not permit the display or editing of multiple fields of information needed by the prototype to allow the detailer to display and edit personnel records. The Record editing system was designed to permit the detailer to edit multiple fields of information. This system is discussed below.

```
typedef struct scr_def
   {
   int number;
   short foreground, background;
   int number_of_fields;
   int row[80];
   int col[80];
   char chars[25][80];
   int atts[25][80];
   int field_length[80];
   int verification_code[80];
   char legal_vals[80][8][15];
   }  Scr_type;
```

Figure 3.1: The Screen Data Structure

The screen structure arrays chars and atts contain the character and attribute for each cursor position of the screen [8]. The static portions of the screen are stored in these arrays. The integer number_of_fields stores the number of data fields to be displayed and edited. The arrays called row and column contain the cursor position for all fields to be displayed on the screen. For example, the cursor position of field 10 is row[10],col[10]. The field_-length array contains the maximum length of each field to be edited. The verification_code and legal_vals arrays provide values used for field type checking.

Changper Execution

Initially, the global record, Screen, of type scr_type, is initialized with the data in file editscr.dat. A user supplied social security number is used to provide the key to the sailor's personnel data file. Its contents are read into the global personnel structure, Sailor.

The routine Draw_fields is then invoked to draw the editing screen. First, Draw_fields displays the static screen by entering a nested loop to display each character and attribute of the screen starting at row-column position 0,0 to row-column position 25,80. Each character and attribute is displayed with the low-level interrupt routine Put_scrn_char.

Next, the routine Set_screen is called to change the foreground and background characters of all characters to printed to have the foreground color Screen.foreground, and the background color Screen.background. Set_screen uses an ANSI sequence to set these colors [4].

The routine Draw_fields then enters a loop to print each data field and its corresponding value from the Sailor structure. A loop control variable is used as an index for access to the row, column, and field_length elements for each field. The loop exits after Screen.number_of_fields have been drawn.

To draw field i, the cursor is placed at Screen.row[i], Screen.col[i]. An editing field is then delimited by printing a field of blanks Screen.fields_length[i] in length. This delimits a section on the screen that is the color Screen.background.

The routine Slr_assign is then invoked with the field number (i) and the address of a destination string as arguments. Slr_assign copies the contents of the field in the Sailor structure that corresponds to the field number argument into the destination string.

Upon return from Slr_assign, the cursor is repositioned to the beginning of field i and the string returned is printed in the field. An example editing screen is shown in Figure 3.2.

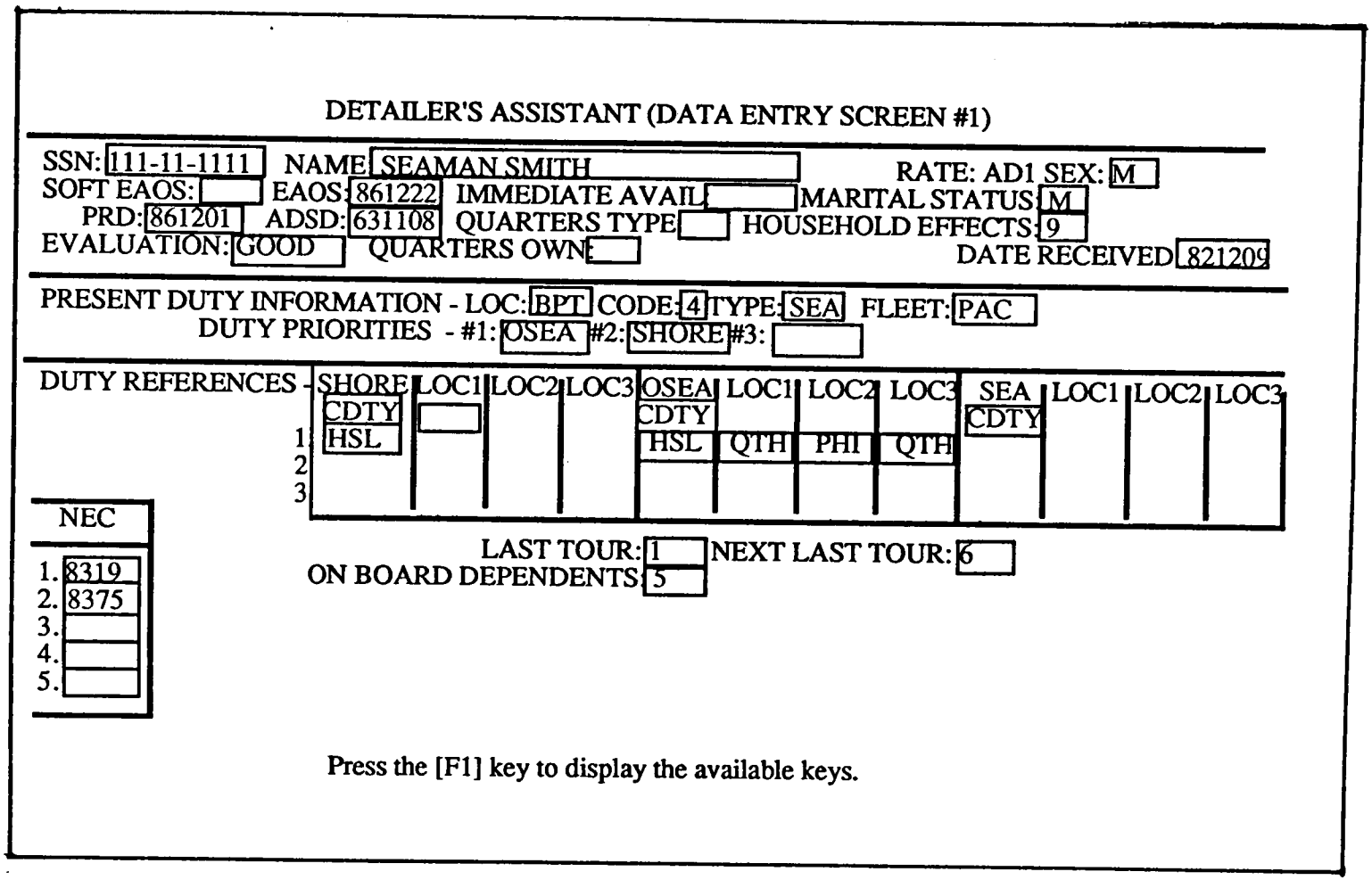

Figure 3.2: A Sample Editing Screen

Editing The Record

After the record has been displayed, the routine Edit_sailor is invoked to permit editing of the data fields. Initially, Edit_sailor displays the characters of the top-left data field blinking on and off. This indicates that the field can selected for editing.

The user is permitted to select another field to edit by pressing any of the four cursor keys. When one of these keys is pressed, Edit_sailor reads the value of the key and compares the Screen.row and Screen.column positions to find the index of the field that represents a move in the desired direction. The new field is then displayed with its characters blinking and the old field's characters are rewritten in non-blinking mode.

When the return key is pressed, a field is selected for editing. The routine Edit_str is invoked to allow the user to edit the contents of the selected field. Edit_str is a generic string editing routine that is called with four arguments. These are the row and column position of the beginning of the editing field, the address of the beginning of the string to be edited, and the maximum length of the string.

Edit_str provides an overstrike mode for editing and permits the left and right arrow keys to place the cursor in

any position within the string being edited. The routine does not permit cursor movement beyond the last character of the string nor does it permit the entry of a character that would allow the string to be longer than the maximum length argument. This control prohibits the visual corruption of other components of the editing screen. The escape or return key returns the user back to Edit_slr.

Edit_slr then calls the routine Assign_slr to place the contents of the character string just modified into the corresponding field in the Sailor structure. Edit_slr passes the index of the currently selected field and the address of the edited string to Assign_slr. Assign_slr uses this index to find the corresponding field in the Sailor structure to receive the newly edited string.

On-line Help

On-line help is available to the user while either selecting or editing a field. The help screen is presented in the form of a pull-down window that displays the keys available to the user for editing or field selection.

To draw the help windows, the routine Help_screen is invoked when the user presses the F1 key. This routine invokes the interrupt routine Get_scr_char to read each character and attribute of the screen, before it is over-written by a character used to draw the help window. These

characters and attributes are stored in RAM and are used to erase the window by writing each stored character and attribute to its original position on the screen. Help_-screen waits for the user to press a key on the keyboard before it exits.

Interrupt Utilization

It was necessary to use PC interrupt routines to provide the interface described above. These routines permitted access to the PC's memory-mapped video and keyboard services. These ROM-BIOS (Read Only Memory-Basic Input Output Services) routines provide access to specific input and output functions that are not available to applications using normal I/O requests.

The ROM-BIOS routines are accessed through software interrupts that use CPU registers to pass arguments. Before a software interrupt is executed, the CPU's registers are initialized with values needed by the interrupt routine. The interrupt instruction is then executed to perform the desired interrupt routine. After the interrupt returns, return arguments stored in the CPU's registers are accessed by the calling routine.

Computer Innovations, the company that wrote the C compiler used for development, provides the Sysint routine that contains the necessary code to perform ROM-BIOS

interrupts. Sysint accepts three arguments, the interrupt number and two addresses. Each address points to a record containing eight integers used to pass CPU register initialization and return arguments. Table 3.1 shows the utilization of ROM-BIOS services through Sysint calls by several low-level I/O routines written for this system.

The Explanation Generation System (EGS)

M.1 does not have a facility to explain why a particular conclusion is made. The Detailer's Assistant is required to provide assignment explanations that describe how the Detailer's Assistant made a particular assignment. An explanation generation system, written in C is used by M.1 to provide these explanations to the detailer.

Explanation Generation System Overview

After the Detailer's Assistant has found a billet to be appropriate for assignment, the system displays this assignment to the detailer. The system then asks the detailer if an explanation for this assignment is needed by displaying a question on the screen. If the detailer responds by typing in "yes", the rule

```
If do(savecache explain.sym) and
   external(explain,[])=[RESULT] and
   RESULT = ok
then answer_explained = yes.
```

Table 3.1: ROM-BIOS Routines Invoked Using Sysint()

Software Interrupt Number	Service Number	Service Type	Description	Used in Editing Routines	Used in ENDS	Function Name
16	0	Video	Set Cursor Size	yes	yes	Set_cursor
16	1	Video	Set Cursor Position	yes	yes	Cursor
16	2	Video	Read Cursor Position	yes	yes	Get_scr_pos
16	3	Video	Set Active Display Page	no	yes	Set_activescr
16	7	Video	Read Character and Attribute	yes	yes	Get_scrn_char
16	8	Video	Write Character and Attribute	yes	yes	Put_scrn_char
22	0	Keyboard	Read Next Keyboard Character	yes	yes	Get_key
22	2	Keyboard	Get Shift Status	yes	yes	Num_lock

is tested by the inference engine. The testing of this rule causes the cache to be written to the cache file explain.sym. The external routine Explain is invoked when the External meta-proposition is tested. Explain is the entry point to the Explanation Generation System, EGS.

Upon entry to EGS, the file explain.sym is read and an array is created to hold each **EXPRESSION** and its corresponding **VALUE**. EGS uses this array to provide values for symbol substitution during the explanation process.

To display explanations, the EGS traverses a state transition network that has been created and stored by the Explanation Network Development System (ENDS), described in Chapter IV. Figure 3.3 shows an example state transition network.

Initially, EGS prints the text associated with the starting state, and proceeds to test the boolean expression of each exiting arc until an arc expression tests true, or all exiting arcs have been tested. If there are no arcs left to be tested, EGS has reached a terminal state and control is returned to M.1.

If a boolean expression tests true, EGS traverses along the arc to the state entered by the arc. The text associated with the new state is displayed to the detailer, and the new state's arcs are tested. This process is repeated until a state is reached that either has zero

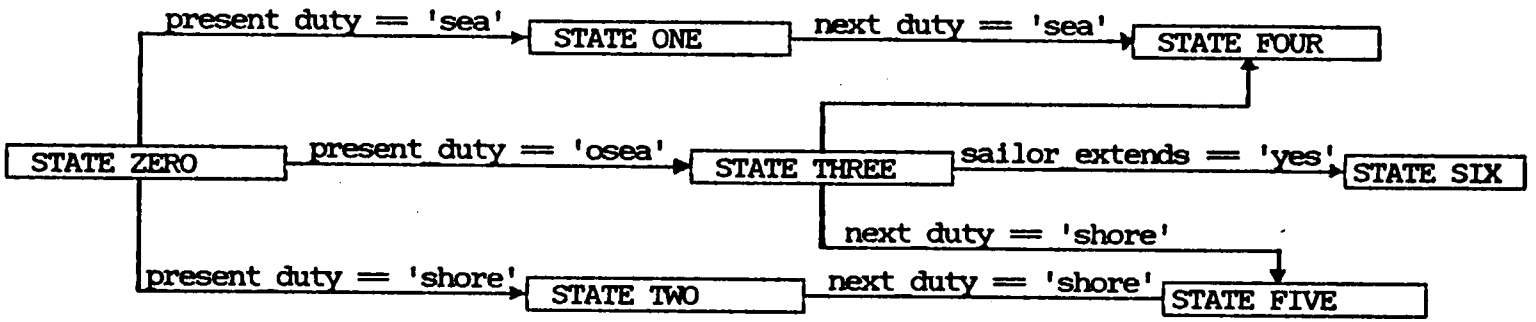

Figure 3.3: A Sample State Transition Network

exiting arcs, or all of its exiting arc expressions test false.

EGS Software Components

When the routine Explain is entered, the routine Build_sym is called to read the file Explain.sym and create a dynamically allocated array of pointers. Each pointer contains the address of structure of type pattern. This structure is listed in Figure 3.4. There is one structure, and one array element pointer for each unique symbol in the cache file. The pointer Symbol contains the address of the first entry in the array of pointers.

When each cache entry is read, its **EXPRESSION** is placed in the symbol field of a pattern structure. The routine Get_token, discussed later, is called to return the **EXPRESSIONS's VALUE** and its type. These are stored in the value and token_type fields of the pattern structure.

The array pointed to by Symbol is sorted using a non-recursive quicksort [2]. The non-recursive quicksort was chosen to provide an efficient sorting algorithm on data in random order with little stack usage overhead. Ordering of the pointers is based on the value of the symbol field in the structure pointed to by each array element. Sorting is performed to permit a non-sequential (binary) search to locate any symbol in the table by other routines in EGS.

```
typedef struct val_struct
   {
   char symbol[40];
   char  value[40];
   int   token_type;
   } Pattern;
```

Figure 3.4: The Pattern Structure

The routine Print_explanations is then invoked to traverse the network and display explanations to the detailer.

Print_explanations reads the first state into a Shell_state structure named Current_state from the file T1.NET. The file T1.NET contains the state and arc records comprising the network. A state is a structure of type shell_state, shown in Figure 3.5, and an arc is a structure of type shell_arc, shown in Figure 3.6. The file T1.NET contains each state record, followed by zero or more arc records that represent the state's exiting arcs. The file storage of the network displayed in Figure 3.3 is shown in Figure 3.7.

Next, Print_explanations multiplies the Current_state.-state_num field by the size of Shell_sent. The Shell_sent structure (see Figure 3.8) is used to store the text that is displayed to the detailer. The result of the multiplication provides a byte offset to the beginning of the sentence record in file T1.SNT that contains the correct text for display. An example of T1.SNT is found in Figure 3.7. To access this record, the lseek routine is passed the calculated offset to correctly position the file pointer. The record is then read into the Shell_sent structure, Sent_buffer.

The Sent_buffer.sent field is an array of character strings that contains the text to be displayed for a state. Starting with the first string in the array, each character

```
typedef struct shell_state_struct
   {
   int                  state_num;
   char                 state_name[21];
   int Number_of_entering_arcs;
   Shell_arc            *arc_list;
   struct shell_state *next_state,
   } Shell_state;
```

Figure 3.5: The Shell_state Structure

```
typedef struct shell_arc_struct
   {
   char                 boolexp[75];
   struct Shell_arc     *next_arc,
                        *nack_arc;
   int                  next_statenum;
   struct Shell_state *next_state;
   } Shell_arc;
```

Figure 3.6: The Shell_arc Structure

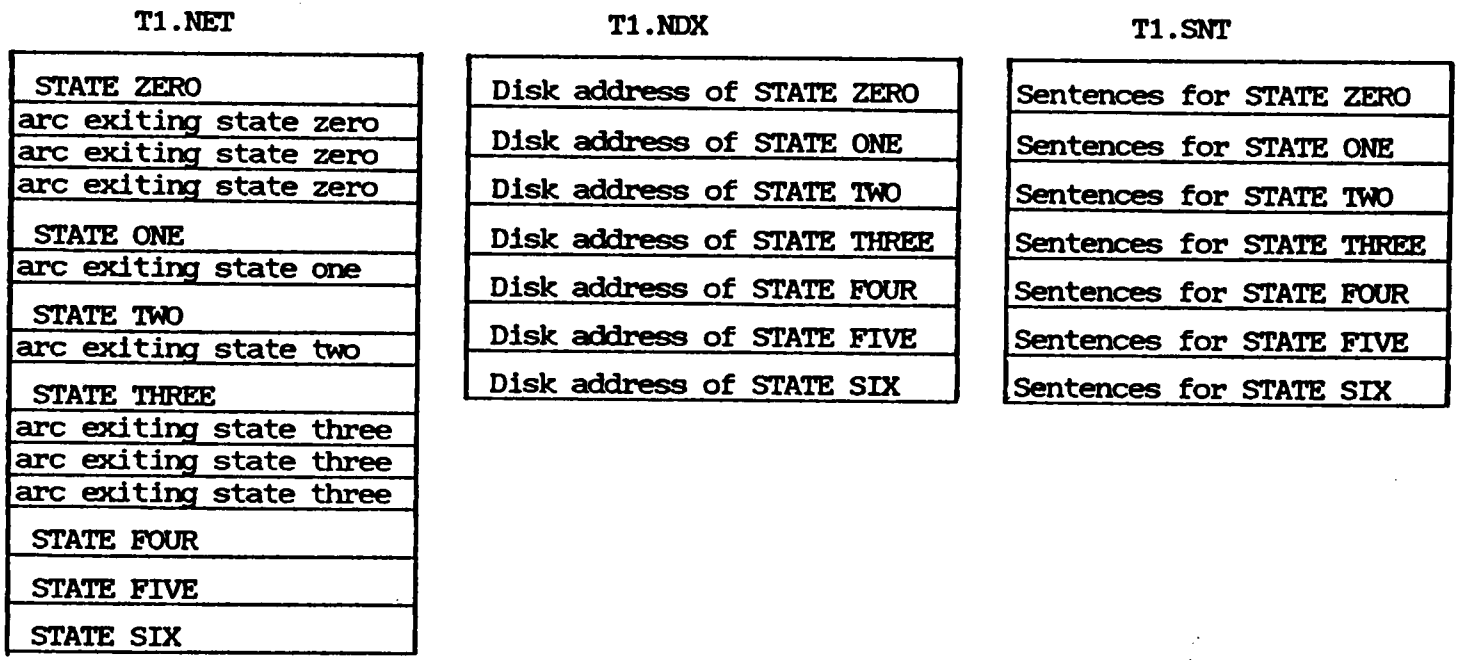

Figure 3.7: File Storage For the Example State Transition Network

```
typedef struct sent_struct
   {
   int                  state_num;
   char                 sent[8][79];
   struct sent_struct   *next_sent;
   } Shell_sent;
```

Figure 3.8: The Shell_sent Structure

is individually printed until the NULL character, or a symbol delimiter character (decimal value four) is encountered. If the NULL character is encountered the next string in the array is processed.

If the symbol delimiter is found the routine Print_sym is invoked. Print_sym concatenates all characters found up to but not including the next symbol delimiter, into a local string variable. Print_sym passes this string as an argument to the routine Find_sym. Find_sym performs a binary search on the array of symbols to return a pointer to the structure of type pattern that contains a value in its symbol field that is identical to the string argument. If Find_sym returns a non-zero address, the address points to a structure of type pattern that contains the value field that is displayed instead of the symbol and its delimiters. If Find_sym returned a NULL value, indicating the search failed, the symbol and its delimiters are displayed to the user.

After the array of strings has been completely processed, Print_explanations determines if any arcs exit Current_state by testing the Current_state.arc_list field. If this field contains the NULL value, this state does not have exiting arcs and EGS returns to the M.1.

If the value tested does not equal NULL, the structure Arc, of type shell_arc, is sequentially read from the file T1.NET. The Arc.boolexp field is evaluated to determine if

the number of the next state to be processed is contained in the Arc.next_statenum field.

The routine Eval is invoked to evaluate the boolean expression stored in Arc.boolexp. Eval is a recursive evaluation function that makes successive calls to the routine get_token, to return each token of a relational expression and an integer representing its token identifier. Figure 3.9 shows the token identifiers that can be returned by Get_token. Eval calls itself recursively to evaluated sub-expressions that are valid boolean expressions. These expressions are discussed in Chapter IV.

Once a complete relational expression has been reduced to tokens, the tokens comprising the expression and their respective identifiers are passed to the routine Compare, to return the value of the relational comparison.

Compare checks the token identifiers of the left and right relational expression operands to determine if either are of type SYMBOL. If either operand is of type SYMBOL, the routine Find_symbol is again called to return a pointer to a structure of type Pattern. This pointer value is assigned to the local pointer t_sym. The values of the t_sym->value and t_sym->token_type fields are substituted into the token and identifier field of the relational operand. If Find_sym returns NULL, Compare exits returning a zero to Compare.

```
#define    EOLN       0
#define    SYMBOL     1
#define    LITERAL    2
#define    EQUAL      3
#define    LTHAN      4
#define    GTHAN      5
#define    LTHANEQ    6
#define    GTHANEQ    7
#define    AND        8
#define    OR         9
#define    OPAREN     10
#define    CPAREN     11
#define    NOTEQUAL   12
#define    CINT       13
```

Figure 3.9: Integer Token Identifiers

After the substitution process is successfully completed, Compare tests the new token identifiers to determine if a string or integer comparison is needed. Compare returns the value returned from a call to the routine Comp_ints, if the operands are of type CINT or Comp_lits, if the operands are of type LITERAL. These routines return the value of the relational comparisons represented by the relational expression token identifier.

Eval can return three values , -1, 0, and 1. A -1 returned by eval indicates an error was found in the boolean expression. The values 1 and 0 represent the values true and false. If eval returns a 1 to Print_explanations, the Arc.next_statenum field is multiplied by the size of the system's long integer (4 bytes). The result of this multiplication provides a byte offset into the file T1.NDX. An example of this file is shown in Figure 3.7.

Next, T1.NDX's file pointer is set to point to the calculated offset and a long integer is read from T1.NDX. This long integer contains the byte offset of the state represented by the Arc.next_statenum field. The next state can then be processed by positioning T1.NET's file pointer to this byte offset and reading the next state into the structure Current_state. This state is then processed as described above.

If Eval returns a 0, the value of field Arc.next_arc is compared with the NULL value. If it does not equal the NULL

value, the next arc can be read in and processed. If it equals NULL, EGS exits to M.1.

Memory Utilization

ENDS runs as a stand-alone system and reads the states, arcs, and sentences comprising the network into RAM before it processes the network. When EGS is invoked by M.1, over 550k bytes of RAM are used by MS-DOS and the Detailer's Assistant's code. This leaves approximately 90k of working memory for the EGS dynamic memory allocation.

To permit the EGS system to process networks larger than those that could be stored entirely in its RAM (that can easily be created by ENDS) the file access method described above was designed. This permits EGS to process all arcs, states, and sentences of an entire network stored by ENDS with three buffers, one for each file.

CHAPTER IV

THE EXPLANATION NETWORK DEVELOPMENT SYSTEM (ENDS)

The ENDS system was developed to provide a tool for the designers of the Detailer's Assistant to create, edit and save state transition networks used by EGS. This chapter presents an overview of ENDS and discusses the major ENDS software components.

ENDS Overview

ENDS begins execution by either creating a new network or reading an existing network from disk and building the lists representing the network. The new or existing network is modified by permitting the user to select functions which add, delete and print the states and arcs for the user. When ENDS is exited, the user can elect to save all changes made to the network on disk. This saved network can be used by the EGS to generate explanations, or it can be read and modified by ENDS.

The ENDS State Transition Network

The ENDS state transition network can be graphically represented by a directed graph, as shown in Figure 3.3.

The vertices of the graph are referred to as states, and arcs are the edges connecting the states. The information for each state in the network is stored in structures of type shell_state (see Figure 3.5). The information relevant for each arc is stored in structures of type shell_arc (see Figure 3.6).

The shell_state structure contains information that allows ENDS to access each state and the state's exiting arcs. The fields of the shell_state structure follow:

state_num: an integer value, used internally by ENDS to uniquely identify a state.

state_name: a string that permits the user to uniquely identify a state.

number_of_entering_arcs: an integer representing the current number of arcs pointing to this state.

arc_list: a pointer to the list of shell_arc structures that contain the information for each arc exiting the state.

next_state: a pointer to type shell_state. ENDS can access each state in the network by traversing the list built with this pointer.

The shell_arc structure contains information allowing the access and testing of each arc exiting a state. Information allowing access to the state entered by the arc

is also stored. The following is a description of the shell_arc's fields:

boolexp: a character string holding the boolean expression to be tested.

back_arc: a pointer to the previous arc in the list, pointed to by arc_list.

next_arc: a pointer to the next arc in the list pointed to by arc_list.

next_statenum: an integer used to uniquely identify the state entered by the arc. This value is used by ENDS, when the network is built in RAM and when traversed by EGS.

Reading and Building a Network

The user can edit an existing network by entering the system name and a file prefix, at the MS-DOS command line. For example, to edit the T1 network, the user would enter

ENDS T1<Return>

after the MS-DOS system prompt to load and execute ENDS. ENDS invokes the routine Get_net to open files T1.NET and T1.SNT. If no error is detected, the routine Read_net is called to read and build the network.

First, Read_net allocates memory for a shell_state structure, and its address is assigned to the global pointer Start_state. Next, a shell_state record is read into the

allocated structure from the file T1.NET. The arc_list structure of this state is tested to determine if this state has exiting arcs that need to be read.

If the value stored in arc_list is not equal to NULL, memory is allocated for a structure of type shell_arc and its address is assigned to the state_structure's arc_list field. The arc record is read into the allocated arc structure from the file T1.NET. The next_arc field of this arc is then tested to determine if another arc must be read and appended to the arc list. If so, memory is allocated for an arc structure and the structure's address is assigned to the last arc's next_arc pointer. The address of the last arc is then assigned to the new arc's back_arc field. Arcs are sequentially read and appended as described until a NULL value is encountered in the next_arc field of arc just read.

Next, the state structure's next_state field is tested to determine if another state is to be read from the file T1.NET. If so, memory is allocated for another state. The next_state field of the last state processed is assigned the address of allocated structure. This state is processed as described above. The building process continues until Read_net encounters a state that contains a NULL value in its next_state field.

Next, the Start_state list is traversed, stopping at each state to traverse its arc list. Each arcs's next_state field is assigned the new memory location of the state

entered by the arc. This permits direct access to the state structure referenced by the arc during network editing.

Finally, the sentences associated with each state are read into shell_sent structures and appended to a list pointed to by the global pointer Start_sent. An example of the lists constructed for a state transition network is shown in Figure 4.1.

Creating A Network

If an MS-DOS command line argument is not detected by ENDS, the system prompts the user with a question, asking the user if a new network should be created. If the user enters the character 'N', ENDS exits to MS-DOS.

If the user enters a 'Y', the user is prompted to enter the filename prefix of the new network. If the prefix is valid, memory is allocated for a shell_state structure, its pointer and numeric fields are initialized to NULL, and the state_name field is initialized to the string "START STATE". The address of this structure is assigned to the global pointer Start_state. A shell_state structure is created and its address is assigned to the global variable Start_sent.

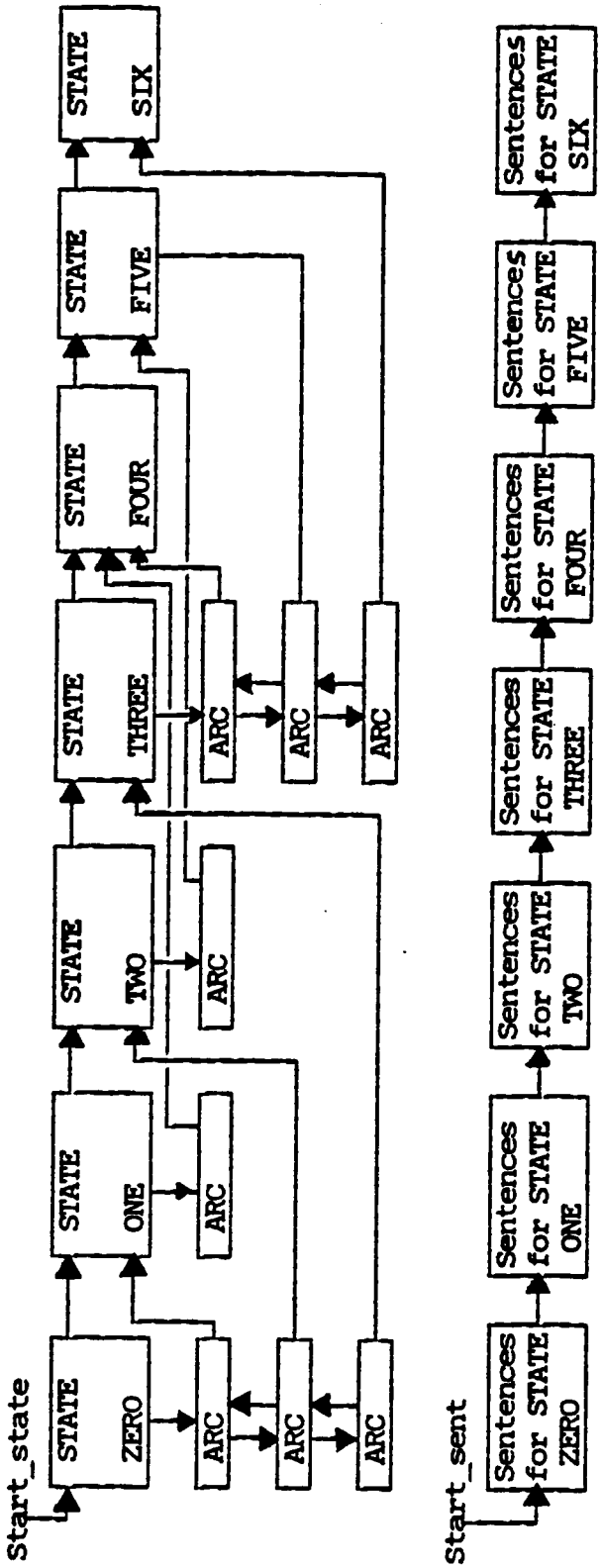

Figure 4.1: Network Lists Built By ENDS

Editing a Network

When ENDS successfully builds or creates a new network, it invokes the routine Edit_network. Edit_network is a high-level routine that controls network modifications. First, Edit_network invokes the routine Paint_screen to display the non_data components of the state editing screen. Next, Edit_network passes the pointer Start_state to the routine Display. The Display routine fills in the data fields with the values obtained from the Shell_state structure stored at the address passed to it. A sample state editing screen is shown in Figure 4.2.

The three major components of the state editing screen are: the arc and next state display windows, the text display and editing window, and the status window.

The arc and next state display windows display information concerning a state's exiting arcs. The arc display window presents the first 35 characters of the each arc's boolean expression. The state_name field of the state pointed to by an arc's next_state field is displayed to the right of each arc. These windows can display up to seven exiting arc expressions and next state names at a time. The text display and editing window displays the character strings found in the displayed state's corresponding shell_sent structure.

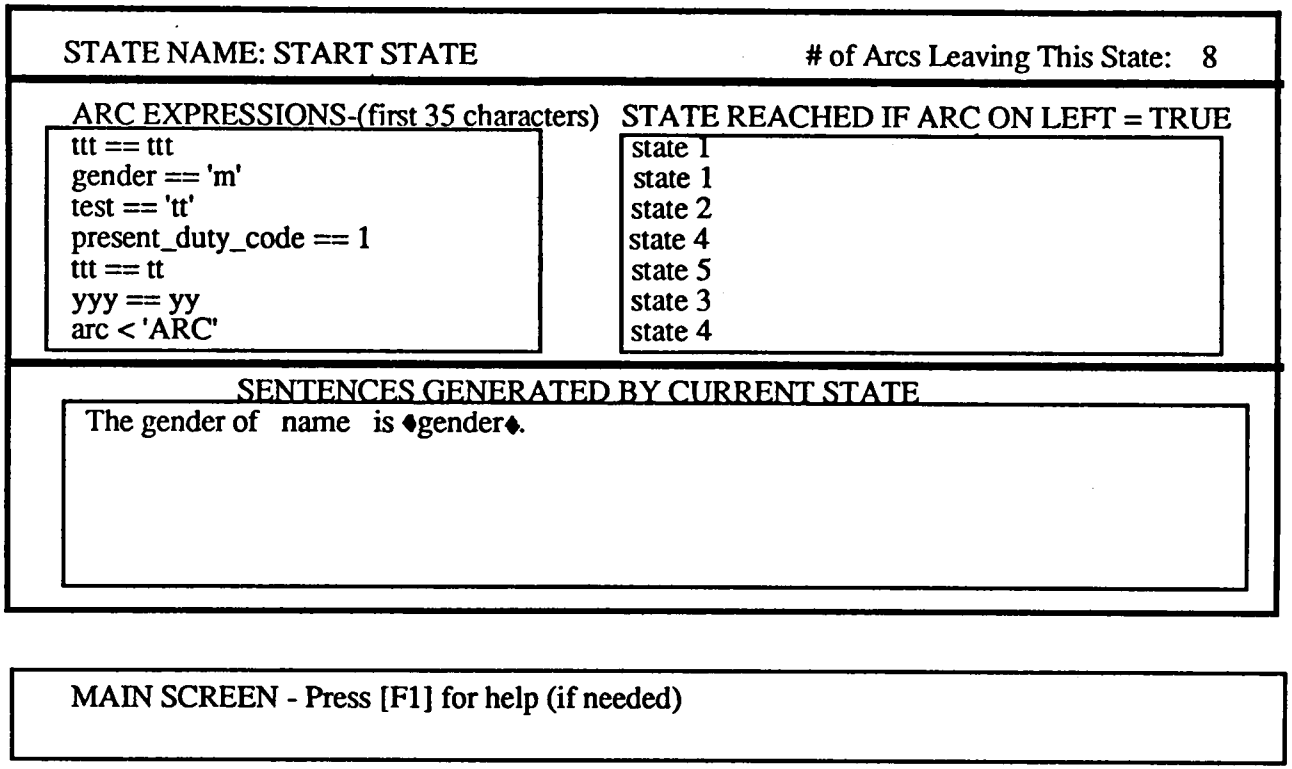

Figure 4.2: The State Editing Screen

The box outlined at the bottom of the state editing screen is the status window. The status window provides an area for the system to display prompts, and accept user responses.

After a state is displayed, the routine Edit_state is invoked by Edit_network to allow the user to make network modifications to the arcs and text associated with the displayed state. Edit_network passes the address of the state structure to be edited, initially the address of Start_state, to Edit_state. Edit_state returns a pointer to the next state for Edit_network to display and edit, or the NULL value, indicating the user's desire to exit the system.

Editing a State

Upon entry, Edit_state assigns its pointer argument to the variable Current_state. It then assigns the address in the Current_state.next_arc field to Present_arc. Next, Edit_state enters a loop that reads a keystroke, and determines which key is pressed by executing a C switch statement. The switch statement compares the decimal equivalent of the keystroke with values in each of the switch's case statements. If the keystroke value is equal to the case statement's test value, the code following the case statement is executed. This code is usually a call to an editing function. Each editing function is activated by

pressing one of ten PC function keys. The function keys are located on the PC's far left keypad. If the user selects another state to edit, or presses the Esc or Return key, Edit_state returns to Edit_network.

Editing Functions

The first function key, F1, provides help for the user. The routine help is invoked to overlay a help window on the screen which displays each editing function key and its associated editing function. This help window is shown in Figure 4.3.

The user can view any other state in the network by pressing the F2 function key. When selected, the status window prompts the user to enter the name of the state to be viewed. The routine Edit_str (discussed in Chapter III) is called to permit the entry and editing of an input string. Edit_state then invokes the routine Jump_to to traverse the state list, starting with the state Start_state.

At each state, Jump-to compares the state's state_name field with the string entered by the user. If they are identical, the Display routine is called to display the contents of this state on the screen. The Display routine is invoked to redraw the original state after the user presses any key. If a state is not found, the user is

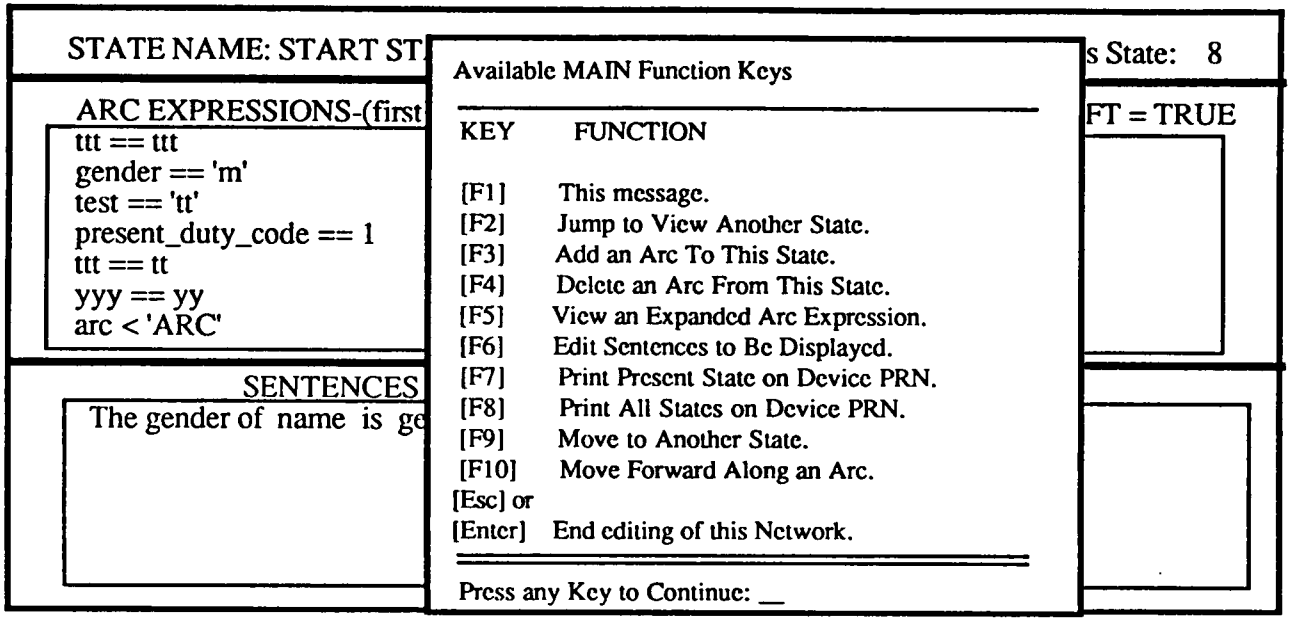

Figure 4.3: The ENDS Main Help Window

notified that the desired state does not exist. Jump_to then returns to Edit_state.

Arc Addition Overview

Edit_state permits the user to add an arc to the displayed state's arc list when the F3 key is pressed. The routine Select_arc called to allow the user to select where in the list to insert the arc. The positioning of the arc in this list determines the order of arc testing by EGS. Next, the routine Valid_arc is called to control arc addition.

Valid_arc prompts the user to enter a string representing the boolean expression for testing. The routine Correct_expr is called to determine if the string entered is a properly formed ENDS boolean expression. If it is, the user is prompted to enter the name of the state that will be entered by EGS if the boolean expression evaluates to true. The name entered is placed in the local string state_name.

Next, memory is allocated for the new arc and the arc is properly positioned into the list pointed to by the current state's arc_list field. The entered boolean expression is then assigned to the new arc's boolexp field. The routine Find_state is then called to return the address of the shell_state structure that contains a state_name

field equivalent to the local string state_name. If there is not a state with this name, Find_state creates a new state and appends it to the Start_state list.(see Figure 4.1) Finally, the next_state and next_statenum fields in the new arc structure are initialized to reference the state_structure returned by Find_state. The arc and next state display windows are then updated.

Boolean Expression Testing

The routine Correct_expr is responsible for ensuring that the boolean expression entered for each arc, can be evaluated by EGS. Correct_expr places the string for testing into the global variable parse_str and then calls the routine Eval to parse this string. This Eval routine is very similar to the Eval routine used by the EGS system. The difference lies in the ENDS Compare routine. The ENDS Compare routine checks the tokens passed to it to determine if they represent a well formed relational expression. Compare returns a -1 if the expression is not properly formed.

The following rewrite rules describe the type of expressions Eval recognizes:

```
BEXPR -> (BEXPR) | BEXPR logop BEXPR | REXPR
REXPR -> operand relop operand
```

Eval (discussed in Chapter III) calls the routine Get_token to extract the next token from parse_str and

return its token identifier to Eval (see Figure 3.9). The majority of these tokens fall into the three classes listed below:

logop: the logical operators &&, ||.

relop: the relop operators ==, |=, <, <=, >, >=.

operands: literals - up to 39 characters enclosed in single quotes
constants - integers
symbols - up to 39 alphanumeric characters and the '_' and '-' characters.

Eval tests each token and returns a -1 if the token's position in the expression will not permit the EGS to evaluate a well formed boolean expression. Eval will also return a -1 if Get_token or Compare return the ERROR value (-1). The expression

```
gender == 'm' && (present-duty == 2 || last-duty != 1)
```

is an example of a valid boolean expression.

Arc Deletion

When in Edit_state the F4 function key allows the user to delete any arc that exits the state currently being edited. To select an arc to be deleted, Select_arc is called to modify Present_arc to point to the arc selected for deletion. The routine Delete_arc is called to remove the arc.

First Delete_arc tests the number_of_entering_arcs field of the state structure pointed to by the present_arc.-next_state pointer. If the value of this field is 1, the arc selected for deletion is the only arc entering this state. If the selected arc is then deleted, this state would not have any entering arcs, and its associated text could not be displayed by EGS. To prevent this situation, Delete_arc will unlink the state and its associated sentence structure. It then releases the memory used by these structures to the system. If the number_of_entering_arcs field is greater than 1, it is decremented by 1.

Next, the arc is removed from the list and the memory it occupied is released. The arc display windows are rewritten to represent the modified arc list.

Text Editing

The user is permitted to enter and edit the text generated for this state by pressing the F6 function key. Edit_state calls the routine Edit_text to control editing. Edit_text permits the user to enter and change printable characters anywhere in the text display and editing window. On-line help is available by pressing the F1 key. The help window, shown in Figure 4.4, displays the special cursor movement and editing keys that can be used while editing.

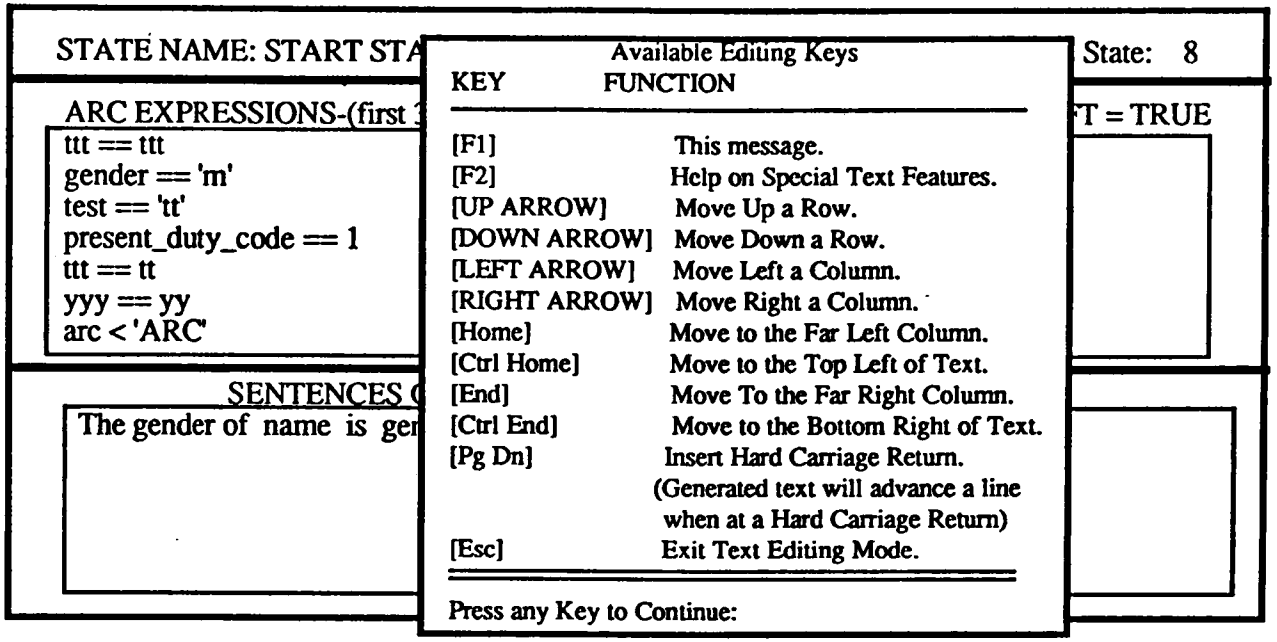

Figure 4.4: The ENDS Text Edititng Help Window

The user is permitted to delimit strings that can be substituted with values from the EGS symbol array at explanation generation time. To properly delimit a string for substitution, the user places a symbol delimiter immediately before and after the string. The delimiter is produced by pressing the Ins key on the PC's numeric keypad. The text display and editing window in Figure 4.2 shows the string "gender" properly delimited for substitution.

Selecting A State to Edit

The first state displayed for editing is always the state pointed to by Start_state. There are two function keys that permit the user to move to, and edit another state. These are the F9 and F10 function keys.

The F9 key permits the user to move to any state in the network by entering the value of the desired state's state_name field. Starting with Start_state, the next_state list is traversed until the state with the correct state_-name field is found, or the NULL pointer is reached. If the state is found, Edit_State returns this pointer to Edit_network. If the desired state is not in the network, the user is notified and editing continues.

When the exact name of the state is not known, the user can move to another state by traversing one of Current_-state's exiting arcs. When the F10 key is pressed, Select_

arc is invoked to allow the user to select the arc to be traversed. After this arc is selected, the arc's next_state value is returned to Edit_network.

Saving The Network

Before exiting to MS-DOS, the user is asked if the edited network should be written to disk. If the user does not want the network stored, ENDS exits to MS-DOS.

If the user wants the network's modifications to be stored, the routine Write_net is called to save the network. Write_net creates new files from file specifications formed from the valid filename entered by the user at the beginning of the session. The filename is prepended to the following 3 filename extensions

.NET, .SNT, and .NDX

to form the network file specifications.

The file filename.NET contains the states and arcs of the network. The list pointed to by Start_state is traversed and each state structure is written to the file followed by each of its exiting arc structures. The sentence structure associated with each state is written to the file filename.SNT.

Before each state is written, the function ltell is called to return the value of the file pointer for the file filename.NET. This value represents the logical disk

address of the next state to be written. This value is written to the file filename.NDX. EGS uses this value to access the stored states.

After all states are written and the three files are closed, ENDS exits to MS-DOS. These files can then be used by the EGS or ENDS systems.

CHAPTER V

SUMMARY AND RECOMMENDATIONS FOR FUTURE RESEARCH

This research has resulted in the creation of data editing, file access, and explanation routines that are utilized by the Detailer's Assistant prototype expert system. The resulting system can access and use a greater number of facts about the assignment, and can be used by an inexperienced user without lengthy training.

The Explanation Network Development System was designed and written to provide a tool that permits the creation and modification of the files used by the explanation routines. This tool can be used by future developers to customize the Detailer's Assistant's assignment explanations without recompiling the existing system.

Recommendations for Further Study

One area for further research consideration is the programming of the PC to access the personnel and job opening data from an on-line database. The Navy is completing the installation of an interactive query system that is accessed through IBM's System Network Architecture (SNA). Routines could be designed and written to interface with the SNA model to provide access to the query system. These

routines could then be substituted, or integrated, into the file access routines that have been written for the Detailer's Assistant. M.1 could access the on-line data through its existing **External** and Do meta-propositions in the Detailer's Assistant's knowledge base.

A second area for further research consideration is the design and development of rules to encode additional knowledge concerning the assignment process. The Detailer's Assistant contains a subset of the knowledge needed to properly assign Navy personnel in every situation. The additional rules would enable the system to make intelligent assignment decisions in a greater number of situations.

LIST OF REFERENCES

LIST OF REFERENCES

1. Aho, Alfred and Ullman, Jeffery., Principles of Compiler Design, Reading, Massachusetts: Addison-Wesley, 1977.

2. Baase, Sara., Computer Algorithms: Introduction to Design and Analysis, Reading, Massachusetts: Addison-Wesley, 1978, pp. 58-59.

3. Barr, Avron and Feigenbaum, Edward., The Handbook of Artificial Intelligence, 3 vols. Reading, Massachusetts: Addison-Wesley, 1982, v. 2, pp. 30-82.

4. C86 C Compiler. Tinton Falls, New York: Computer Innovations Inc., 1985.

5. England, Robert E., "Nexus: a Grammar Based, Menu Driven Extension for the NITPACK Decision Tree Software." Masters thesis, University of Tennessee, 1985.

6. IBM DOS Technical Reference Manual. Boca Raton, Florida: International Business Machines Corporation, 1985.

7. Kernigham, Brian, and Ritchie, Dennis., The C Programming Language, Englewood Cliffs, New Jersey: Prentice Hall, 1978.

8. Liepins, Gunar., "An Expert System Application to Personnel Assignment", Paper written for the Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1986, pp. 1-3.

9. McKeown, Kathleen R., Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text, Cambridge, New York: Cambridge University Press, 1985.

10. M.1 Reference Manual, Palo Alto, California: Teknowledge, 1986.

11. Norton, Peter., The Programmers Guide to the IBM PC, Bellevue, Washington: Microsoft Press, 1985, pp. 80.

12. Waterman, D. A., A Guide to Expert Systems, Reading, Massachusetts: Addison-Wesley, 1986, pp. 20-50.

13. Winston, Patrick., Artificial Intelligence, Reading, Massachusetts: Addison-Wesley, 1984.

VITA

Robert H. Kroboth was born in Pittsburgh, Pennsylvania. He was graduated from Bethel Park Senior High School in Bethel Park, Pennsylvania, in June 1976. He received a Bachelor of Arts degree in Computer Science in March 1985 from the University of Tennessee. In June 1987 he received a Master of Science degree in Computer Science.