MTQ

A Medically Oriented Time Series Query Language

by

Gregory Floyd Cooper

Submitted in Partial Fulfillment

of the Requirements for the

Degree of Bachelor of Science

at the

Massachusetts Institute of Technology

May, 1977

Signature of Author.....
Department of Electrical Engineering
and Computer Science, May 23, 1977

Certified by.....
Thesis Supervisor

Accepted by.....
Chairman, Departmental Committee on Theses

This research was supported by the National Institutes of Health, Department of Health, Education and Welfare (Public Health Service) under Grant Number 1-R01-MB-00107-01

MTQ
A Medically Oriented Time Series Query Language

by

Gregory Floyd Cooper

Abstract

Computer stored medical data is becoming increasingly common. Most medical databases contain a time and a parameter dimension for each individual patient. A current problem is how to access the complex, time-oriented information contained therein. To date only the initial steps have been made towards developing a computer language that specifically addresses the difficult problems of time series searching. Solutions to these problems are sure to open pathways to better maintenance of day to day patient care, as well as to new medical research discoveries dealing with various time aspects of diseases and their treatment.

This thesis introduces a new time-series search language called MTQ. Each aspect of the language is first motivated and then explained in detail. A moderately complex search program is built step by step as new program constructs are introduced. Finally the current view and the future projections of the language are discussed.

Thesis Supervisor: Peter Szolovits,
Assistant Professor of Electrical Engineering
and Computer Science

Acknowledgements

I would like to thank Professor Peter Szolovits for the great deal of time he has spent with me in developing MTQ. He has been kind enough to encourage me when I was heading in the right direction, and wise enough to redirect me when I began to stray. Through it all he has remained cheerful in his support.

A special thanks also to Dr. David Wirtschafter and Dr. Emmanuel Mesel who introduced me to the area of time-series searching, and who supported me in the development of TSSS at the Clinical Information Systems Group at the University of Alabama in Birmingham. Dr. Wirtschafter has also been kind enough to provide the the medical data which has been used as a test database.

Professor Vaughan Pratt has been of considerable help by suggesting that his CGOL language and GOT parser were appropriate to my needs in developing MTQ; he was quite right. He has also given helpful advice on their use.

Finally I would like to thank Byron Davies, Ramesh Patil, Brian Smith, and Bill Swartout for their insightful suggestions on how MTQ might be improved.

Table of Contents

List of Figures

I.

Introduction

For several years now there has been a continuing interest and growth in computer stored medical data. (Fries 1976) Many of the current medical databases are logically structured as a three dimensional matrix with patients, time, and medical parameters as the axes (typically for any one patient, there are several medical parameters recorded each time the patient sees a health care provider). (Fries 1972) As the information content of such resources continues to increase, so does the interest in improving the ease and reliability of methods for accessing their contents. Any improvement in such access would, for example, aid medical researchers who analyze such data when attempting to develop or validate medical hypotheses.

If a database has a time dimension, as mentioned above, then there is a marked increase in the difficulty of both comprehending and completely expressing the information desired when compared to simpler two dimensional databases that contain only a patient and a parameter axis. It has been difficult to provide a time-search programming language which is comprehensible to the average user of the database and yet adequately powerful enough to allow expression of the majority of the search requests.

Several current systems have addressed time series searching issues to a limited extent. In these systems (and MTQ as well) time is assumed to be recorded as discrete events known as visits. A visit is simply a patient-health care provider encounter. If n encounters have been made

then there are n visits, where 1 represents the first visit and n represents the most recent visit. Generally at a visit, vital signs are recorded, as well as other data pertinent to the patient's particular condition. Obviously any system which records time in discrete units is not able to capture the state of the patient at every point in time. However, one must realize that in general this visit information is all that is available, regardless of the medium (computer or otherwise) on which it is recorded. Thus a visit oriented representation of time is a natural one. Furthermore, the time of each visit is recorded so that at least interpolations of parameters at any time between two visits can be made.

One of the earliest time series search systems to be developed was ARAMIS at Stanford University. (Weyl, et al. 1975) ARAMIS prompts the user for the intended search and then the user returns a fixed formatted response. Although this system is very limited in the complexity of expressible searches, it does benefit by being easy to learn and use. Subsets may be constructed that contain patients satisfying some search criteria. Operations such as set union and intersection are provided for combining such sets for use in additional searches. More recently at the University of Alabama in Birmingham a language known as TSSS has been developed. (Wirtschafter 1976) The major concept of TSSS is the use of time subscripted clinical parameters within a general boolean expression. Although TSSS is more powerful than ARAMIS with respect to the complexity of searches expressible, it suffers because in many cases complex search expressions are difficult to construct and comprehend. Also, conditional

searches are not possible; this means that if a given time search predicate has a given value, then based on that value it is not possible to test another search predicate which for example might reference time points established in the original predicate. More will be said about conditional searches in the next chapter. Most recently a language known as COSTUDY has been implemented at Massachusetts General Hospital. (Beaman 1977) COSTUDY is capable of searching the time relationships within a MUMPS database. It is too early to evaluate its performance.

In the chapters that follow a new time series search language called MTQ will be examined. MTQ has been completely implemented as defined in this thesis, using GOT[1] as a parser (Pratt 1973) and MACLISP (Moon 1974) as the target language used to run MTQ programs. One goal of MTQ is to allow differing levels of usage ranging from simple searches requiring practically no programming knowledge, to complex searches requiring moderate programming skills. Additionally, each search developed should be self documenting; someone with no knowledge of MTQ should be able to understand the general intent of the search with little or no explanation. Thus, MTQ can serve as both a time-series programming language, and as a model of the meaning of the natural language expression of the most common time-oriented medical relationships. (For a more complete model of this natural language expression see Bruce 1972).

[1] GOT is an acronym for the Generalized Operator Translator developed by V.R. Pratt.

A final goal is that MTQ should provide a retrieval formalism that is powerful enough to serve as a base component for a higher level, natural language type module which would allow non-programmers to express complex searches.

In the next chapter MTQ will be defined and a search example will be incrementally constructed as new language constructs are revealed.

II.

MTQ Language Definition

II.1 Trees

MTQ is a tree structured language. Using trees allows a wide spectrum of searches to be expressed ranging from the very simple to the highly complex. In most cases the complexity of a program's tree structure will directly reflect the conceptual complexity of the search request it represents.

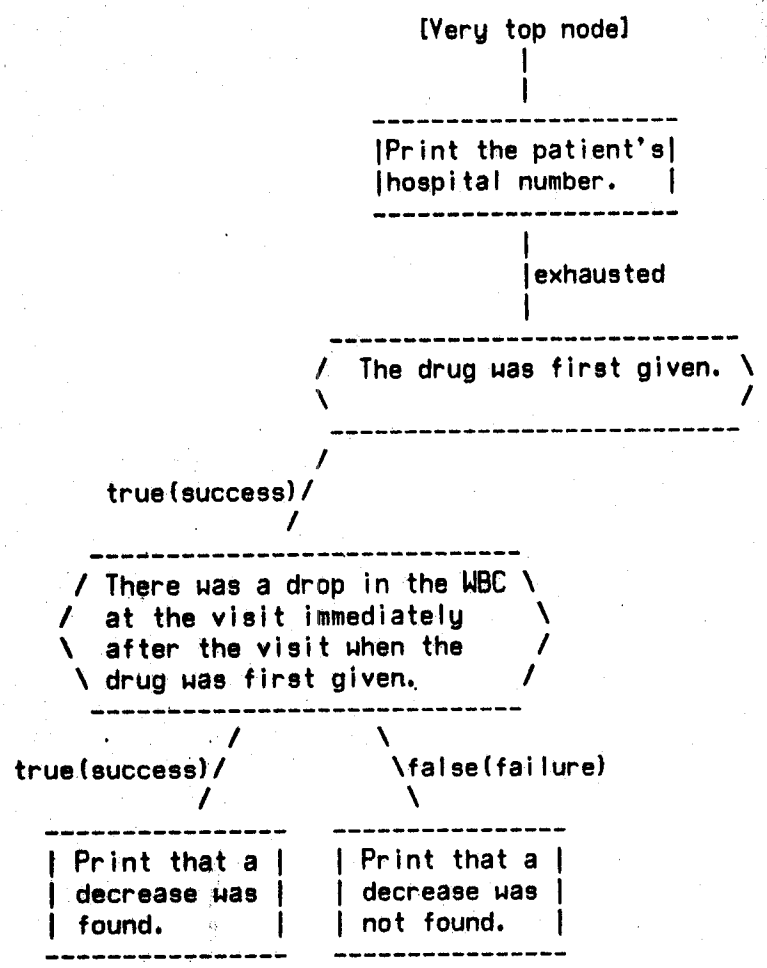
There are two types of nodes used to construct search trees: the search node, and the action node. A search node is used to test the truth of a specified parameter relationship. It has one entry branch and three exit branches. The single entry branch is of course typical of any tree structure, but the exit branches require some explanation. One exit branch is known as the success (or true) branch. This branch is taken when the search expression within the given search node is true. Another branch called the failure (or false) branch is taken if the negation of the search expression is true. Actually there are additional specifications within the search node that restrict when a success or failure branch may be taken, but these slight complications will be discussed later in the chapter. Typically control will return to the search node when some lower node in the tree has completed its operations. The third and final exit branch is known as the exhausted branch. This branch is taken after the

search node has tested its search expression within the context of all the user-requested time patterns. Upon taking the exhausted branch the node is classified as exhausted until reentry occurs from the branch above it.

Action nodes are the second type of tree nodes. They are generally used to perform some action based on the the branch taken by a search node. In most cases this entails communicating some result to the user via an output device such as a printer. An action node is much simpler than a search node. It has one entry branch and one exit branch. The exit branch is called the exhausted branch. An action node performs all specified actions and then takes the exhausted branch.

As an example, consider a database of patients who were given a chemotherapy agent (BCNU) as treatment for Hodgkin's disease. It is reasonable to suspect that the initial treatment with this powerful agent will cause a drop in the white blood count (WBC) due to the drug's toxicity to the normal body cells as well as the cancerous ones. A simple search program can be written to verify that this in fact occurs for most such patients. If search nodes are represented as diamonds and action nodes as rectangles then figure 1 is a representation of the tree structure for this search. In this example and subsequent ones the WBC will be given in thousands. Notice that the two lower action nodes do not have exit branches. In general if a search node or an action node is missing a branch, then when control would normally pass down that branch, the control is given instead to the first non-exhausted search node encountered along a path inclusively between the current node and the very top node. The effect of this default convention is that search nodes are exhausted

starting from the bottom of the tree and progressing toward the top.



An Example of a Simple Search Tree

figure 1

In figure 1 the search starts at the *very top node*. This node is merely for conceptual convenience and in practice it is not specified. It is defined as a non-exhaustable search node. Each time control returns to the very top node the data of the next patient in the data base is considered, and the search begins anew. If there are no more patients then the search is completed. In figure 1 the very top node passes control to the action node directly below it. The action node prints the patient's hospital number and then passes control down its exhausted branch. The first search node has now been entered. It scans from visit one to the last visit in search of an instance of the drug being administered. Notice that there is no false branch at this node and thus each visit at which medication was not given is ignored. If no instance is found of drug administration then this node is exhausted. But notice that this search node does not have an exhausted branch either. In light of this, the control passes up the tree to the first non-exhausted search node, which is the very top node. The data of the next patient is then considered and the search begins anew. If, however there was an instance of drug administration then the first search node would pass control down its true branch at the first such instance found. The second search node would then have control. This node determines if the WBC decreased at the visit immediately after drug administration (an assumption is being made here that the WBC was in fact recorded at this second visit; this is to simplify the example). If so then the true branch is taken and a message to that effect is printed. If not the false branch is taken and the absence of the decrease is printed. Notice that after either of these two action nodes completes their printout

the control will return to the very top node since the other two search nodes have exhausted the time patterns of interest as specified by the user and neither of them has an exhausted branch.

In the next section visit variables will be introduced which allow expression of the desired time patterns wherewith a search expression is to be tested. The scoping of the variables within any given node extends to all the nodes below it in the tree. If some successor node lower in the tree redefines a visit variable, then that most recent definition will be in effect starting at the node of redefinition and extending to all nodes below it. Any time a node is entered from its entry branch (the branch on top) its visit variables are initialized to their originally defined initial values. Initialization <u>does not</u> occur when control is returned from a lower node in the tree.

The four simple MTQ statements used to construct trees are as follows:

<starting node name> IS THE STARTING NODE$

WHEN <node name> IS TRUE: <node name>$

WHEN <node name> IS FALSE: <node name>$

WHEN <node name> IS EXHAUSTED: <node name>$

The $ symbol is used to terminate all MTQ statements. Matching angle

brackets are metasymbols that must be replaced by the entities described within them. For instance <starting node name> might be replaced by the name TOP, which would then designate TOP as the highest node in the tree. A specification of a node's contents follows its name. As an example, figure 2 shows a representation of the tree of figure 1 using the above constructs. It is important to realize that the <node name> between the colon and the $ delimiter has no semantic meaning to MTQ other than as a reference name for constructing trees, and therefore any apparent semantic content is for documentation purposes only. In later sections the English sentences used in figure 2, such as "Print that a decrease was found", will be converted to MTQ statements. These sentences have been underlined in figure 2 and they each end in a period rather than $.

```
Top is the starting node$
Print the patient's hospital number.

When Top is exhausted:
  Drug_given$
  The drug was given.

  When Drug_given is true:
    Find_WBC_drop$
    There was a drop in the WBC on the visit immediately
    after the visit when the first drug was given.

    When Find_WBC_drop is false:
      Report_failure$
      Print that a decrease was not found.

    When Find_WBC_drop is true:
      Report_success$
      Print that a decrease was found.
```

An MTQ Tree of Figure 1

figure 2

II.2 The DEFINE VISIT Statement

In order to express searches that involve a time dimension there must be some means of specifying the time events of interest. For this reason the visit variable is provided. As an example, each WBC in the database is associated with a particular visit at which the given patient visited the health care provider. To reference some particular WBC for a given patient one might specify:

WBC at visit x

where visit x represents some class of visits previously defined by the user to be of interest to him. In other words, although at any one time visit x will represent one particular visit value, the value of visit x may be defined to change at a prescribed time to another value that is within the constraints of a defined pattern. The patterns so defined are to be used at the node of definition. Any other visit variables in nodes between the current node and the very top node may be referenced, but their values do not change in the current node. This of course is in contrast to the visit variables defined at the current node, which may in fact vary in value so long as control is at the current node.

Before describing in detail the various patterns available to the user by means of the DEFINE VISIT statement, some general definitions need to be given. A *node frame* is defined as a set of visit variable - visit value pairs, with one such pair for every visit variable defined at the

given node. The visit value is an integer representing some visit. A ***valid node frame*** is defined as a node frame in which the value associated with each visit variable in the node frame is such that the criterion of every visit variable definition at the node is satisfied. A ***reference frame*** is defined as the union of the current valid node frame with each current valid node frame of each node from the current node to the very top node. This definition must be qualified, because if two visit variables in separate valid node frames have the same name then only the one defined in the node closest to the current node is retained in the unioned set. A reference frame is used as the visit variable environment within which to test a given search expression or perform a given action. Thus when a visit variable is referenced, the value used in the reference is the value associated with the given visit variable in the current reference frame.

With these definitions completed the DEFINE VISIT statement is now discussed. Conceptually there are two types of visit variable definitions. One is a dependent visit variable definition and it serves as a mere syntactic convenience. It will be discussed later. The other type is a dynamic visit variable definition. It is this type definition that specifies the time patterns of visits to consider in performing a time-series search.

To free the user from having to meticulously specify the time frames (node frames) of interest in performing a search, reasonable assumptions have been built into the meaning of dynamic visit variable definitions. It is important that the user understand these assumptions

and their consequences. As the DEFINE VISIT syntax and semantics are given below, the accompanying assumptions will be clearly stated as well.

Figure 3 gives the syntax of all the dynamic visit variable definitions. Zero, one, or more of the dynamic visit variable definitions may occur at any given node. Figure 4 gives a general semantic interpretation of dynamic visit variable definitions by using conventional programming constructs to build a semantically equivalent representation. Perhaps, however, the WHEN portion of each DO statement should be briefly explained. After a DO variable has been incremented by 1, the WHEN portion of the DO statement checks that its body is true, and if not then the respective DO variable is incremented and this test starts over. This process continues until either <high bound> is exceeded, or the WHEN body is true.

<dynamic visit variable definition> ::= DEFINE VISIT <visit name> SUCH THAT <body>$

<body>	<low bound>	<high bound>	<test>
<test exp>[1]	1	LAST VISIT[2]	<test exp>
IT IS ANY VISIT AFTER <value>	<value>[3] + 1	LAST VISIT	TRUE
IT IS ANY VISIT ON OR AFTER <value>	<value>	LAST VISIT	TRUE
IT IS ANY VISIT BEFORE <value>	1	<value> - 1	TRUE
IT IS ANY VISIT ON OR BEFORE <value>	1	<value>	TRUE
IT IS EXCLUSIVELY BETWEEN $\langle value \rangle_1$ AND $\langle value \rangle_2$	$\langle value \rangle_1$+1	$\langle value \rangle_2$ - 1	TRUE
IT IS INCLUSIVELY BETWEEN $\langle value \rangle_1$ AND $\langle value \rangle_2$	$\langle value \rangle_1$	$\langle value \rangle_2$	TRUE
IT IS BETWEEN $\langle value \rangle_1$ AND $\langle value \rangle_2$	$\langle value \rangle_1$	$\langle value \rangle_2$	TRUE
IT IS ANY VISIT WITHIN RANGE	1	LAST VISIT	TRUE

[1] <test exp> is a truth expression; it will be explained later in section II.4.

[2] LAST VISIT evaluates to the last visit number of the current patient data being searched.

[3] <value> is an expression that evaluates to a numerical value; in the case above it represents a visit number. It is explained in section II.3.

Dynamic Visit Variable Definition Syntax

figure 3

DEFINE VISIT Sequence

<dynamic visit variable definition>$_1$
<dynamic visit variable definition>$_2$
.
.
.
<dynamic visit variable definition>$_n$

||
||
||
V

Equivalent DO Loop Representation

DO <visit name>$_1$=<low bound>$_1$ TO <high bound>$_1$ BY 1 WHEN(<test>$_1$);
DO <visit name>$_2$=<low bound>$_2$ TO <high bound>$_2$ BY 1 WHEN(<test>$_2$);
.
.
.
DO <visit name>$_n$=<low bound>$_n$ TO <high bound>$_n$ BY 1 WHEN(<test>$_n$);

<do something>

END$_n$;
.
.
.
END$_2$;
END$_1$;

<this node is exhausted at this point>

The Semantics of Dynamic Visit Variable Definitions

figure 4

The DO loops of figure 4 are used to sequence through each valid node frame. Notice that it is within the inner body of the DO loops (i.e. <do something>) that the search expression is tested (or the actions performed). If the search expression is true then the true (success)

branch will be taken. If it is false the false (failure) branch will be taken. When control returns to this node from a lower node it is as though evaluation of <do something> was completed. If however control enters this node from a higher node then execution will begin at the outermost DO loop. Now notice that there are several assumptions made when defining dynamic visit variables. One assumption is that the visit variables (i.e. the <visit name>'s) will increment from some lower bound to some higher bound in increments of 1. This was of course a major design decision. As an example of a pattern that could not be searched in MTQ, consider the following visit variable value sequence (assuming the current patient being searched has 4 visits in the database):

1, 4, 2, 3, 3, 2, 4, 1

Experience with the TSSS search system (Wirtschafter, et al. 1976) has shown however that no practical searches require patterns outside those provided by MTQ.

The second assumption of dynamic visit variable definitions is that the order of the definitions affect the resulting sequence of valid node frames. This is seen in figure 4 by the way in which the order of the define visit variable definitions determine the order of the DO statements in the corresponding representation. In most cases a visit variable defined lower in the sequence will change value several times before a visit variable higher above it in the sequence will change value. Also, notice that any visit variables referenced in the <body> (see figure

3) of a DEFINE VISIT statement should be previously defined if they appear in <low bound> or <high bound>. If they will appear in <test> then they should either be the current visit variable being defined or some previously defined visit variable. It is important to realize that any of the visit variables defined in nodes above the current one are considered to be previously defined.

It should be pointed out that assumption 2 was not made because it would lead to an easy implementation using conventional DO constructs, but rather that this assumption was the most intuitive one known for the given level of programming detail with which MTQ deals. Perhaps the problem is that most people are unaccustomed to dealing with having to describe detailed and unambiguous time relationships. This should serve as a caveat to those who might otherwise define dynamic visit variables at a node without seriously considering the semantic implications. Fortunately, however, in most cases it is not necessary to define more than one dynamic visit variable at a single node. By using fewer such definitions per node one gains greater control over the search and consequently a clearer understanding of its semantic implications.

Dependent Visit Variable Definitions

The second type of visit variable definition is the dependent visit variable definition. As mentioned earlier, this is used as a mere syntactic convenience. Figure 5 shows all definitions of this type.

<dependent visit variable definition> ::=

DEFINE VISIT <visit name> SUCH THAT IT IS <dependent body>$

<dependent body>	<dependent value>
<value>	<value>
<value>$_1$ VISITS AFTER <value>$_2$	<value>$_1$ + <value>$_2$
THE VISIT IMMEDIATELY AFTER <value>	<value> + 1
<value>$_1$ VISITS BEFORE <value>$_2$	<value>$_2$ - <value>$_1$
THE VISIT IMMEDIATELY BEFORE <value>	<value> - 1

Dependent Visit Variable Definition Syntax

figure 5

When a reference is made to a dependent visit variable, it is as if a reference were made to the <dependent value> of its definition as shown in figure 5. The <dependent body> may contain references to any visit variables defined either at the current node or in any of the nodes above the current node, except itself and dependent visit variables which reference it. Thus dependent visit variables are not intended to be recursive. It should now be apparent that the order of dependent visit variables within the DEFINE VISIT sequence of a node is arbitrary, since such variables do not imply a sequence of visit patterns.

A DEFINE VISIT Example

An example should help in understanding how to define visit variables. With this in mind, refer to the Drug_given node of figure 2. Notice that the visit when "the drug was first given" could be any visit from visit 1 to the LAST VISIT. This being the case, a visit variable should be defined which can scan this range. This means that a dynamic visit variable definition is needed. From the choices in figure 3 the last <body> seems to fit the need. For lack of a better name, suppose this visit is called visit a. The definition would then be:

DEFINE visit a SUCH THAT IT IS ANY VISIT WITHIN RANGE$

Now refering back to figure 2, in the Find_WBC_drop node there is also mention of "... the visit immediately after visit a ...". This is a dependent visit, since it is just one visit after the visit a defined above. From the choices in figure 5 the second <dependent body> is the one needed. Suppose this visit is called visit b, then its definition would be:

DEFINE visit b SUCH THAT IT IS THE VISIT IMMEDIATELY AFTER visit a$

Figure 2 is redisplayed in figure 6 with the above visit variable definitions substituted within the Drug_given and the Find_WBC_drop nodes. The respective search expressions have been modified to reference these

two visits.

```
Top is the starting node$
Print the patient's hospital number.

When Top is exhausted:
  Drug_given$
  Define visit a such that it is any visit within range$
  The drug was given at visit a.

  When Drug_given is true:
    Find_WBC_drop$
    Define visit b such that it is the visit immediately after visit a$
    The WBC at visit b is less than the WBC at visit a.

    When Find_WBC_drop is false:
      Report_failure$
      Print that a decrease was not found.

    When Find_WBC_drop is true:
      Report_success$
      Print that a decrease was found.
```

Figure 2 Plus Added Visit Variable Definitions

figure 6

II.3 The Syntax and Semantics of <value>

In earlier sections the <value> entity was referenced without being defined. At this point both its syntax and semantics will be discussed. As mentioned earlier, <value> always evaluates to a number. Figure 7 gives the BNF syntax of <value>; see the appendix for resolution of any ambiguities in the operator-operand association.

```
<value> ::= <number> | <value function> | <complex value>

<complex value> ::= <value> <arithmetic operator> <value> | (<value>) |
                    <sign><value>

<arithmetic operator> ::= + | - | * | / | ^

<sign> ::= + | -

<value function> ::=    VISIT <visit name>|
                        <item reference>|
                        MAXIMUM <item name> <optional range>|
                        MINIMUM <item name> <optional range>|
                        AVERAGE <item name> <optional range>|
                        ENTRIES <item name> <optional range>|
                        DAYS FROM <date> TO <date>|
                        GREATEST DIFFERENCE BETWEEN <value list>|
                        CLOSEST VISIT AFTER <date>|
                        CLOSEST VISIT BEFORE <date>|
                        CLOSEST VISIT TO <date>|
                        LAST VISIT|
                        PATIENT_NUMBER|
                        THE <value>|
                        NUMBER OF SUCCESSES AT <node name>|
                        NUMBER OF FAILURES AT <node name>|
                        NUMBER OF EXHAUSTS AT <node name>|
                        ABS(<value>)

<number> ::= <any MACLISP number>

<item reference> ::= <item name> AT <value>

<item name>, <node name>, <visit name> ::= <character string>

<character string>[1] ::= <a string of alphabetic, numeric, and
                          underscore characters beginning with alphabetic>

<optional range> ::= FROM <value> TO <value> | nil

<date> ::= <value>

<value list> ::= <value list>,<value> | <value>
```

[1] All alphabetic characters are considered capitalized.

The Syntax of <value>
figure 7

An Explanation of Each <value function>

VISIT <visit name>

This returns the value of the visit variable named <visit name> from the currently active reference frame. If, however, <visit name> is a number then that number is returned.

<item name> AT <value>

<item name> is the name of some parameter stored in the database. The value of this parameter at the <value>th visit is returned using the data of the current patient being searched.

In the next four functions if <optional range> is not nil then it has the form: FROM <value>$_1$ TO <value>$_2$. If <optional range> is nil then <value>$_1$ ::= 1 and <value>$_2$ ::= LAST VISIT.

MAXIMUM <item name> <optional range>

The maximum of <item name> inclusively between the <value>$_1{}^{th}$ and the <value>$_2{}^{th}$ visit is returned.

MINIMUM <item name> <optional range>

The minimum of <item name> inclusively between the <value>$_1{}^{th}$ and the <value>$_2{}^{th}$ visit is returned.

AVERAGE <item name> <optional range>

The arithmetic average of <item name> inclusively between the <value>$_1$ th and the <value>$_2$ th visit is returned.

ENTRIES <item name> <optional range>

The number of entries of <item name> inclusively between the <value>$_1$ th and the <value>$_2$ th visit is returned. An entry occurs if the given parameter was given a value at a visit in the range. In many cases certain parameters are not entered at a visit since they are not pertinent to the patient's condition at that visit.

DAYS FROM <date>$_1$ TO <date>$_2$

<date> is a number of the form YYMMDD where YY, MM, and DD are a two digit year, month, and day respectively. The positive number of days from <date>$_1$ to <date>$_2$ is returned, assuming <date>$_1$ is a date on or before <date>$_2$. If this is not the case then the negative of the days difference is returned.

GREATEST DIFFERENCE BETWEEN <value list>

The value returned is |max(<value list>) - min(<value list>)|.

CLOSEST VISIT AFTER <date>

<date> is of the form described above. The visit number of the next visit after <date> is returned. If no such visit exists then a ??? is returned (this will be discussed in section II.5).

CLOSEST VISIT BEFORE <date>

<date> is of the form described above. The visit number of the visit most immediately preceding <date> is returned. If no such visit exists then a ??? is returned (this will be discussed later).

CLOSEST VISIT TO <date>

<date> is of the form described above. The visit number that is closest in time to the given <date> is returned. If the <date> is equidistant between two visits then the more recent visit number is returned.

LAST VISIT

This returns the visit number of the last recorded visit for the patient whose data is currently being searched.

PATIENT_NUMBER

This is an item associated with each patient for identification purposes.

THE <value>

<value> is returned.

NUMBER OF SUCCESSES AT <node name>

This returns the number of times the success (true) branch has been taken by <node name> since control was last passed to <node name> by the node above it. If this <node name> has no success branch then the returned value is the same as if such a branch existed and the nodes lower in this

branch always return control to this node.

NUMBER OF FAILURES AT <node name>

This returns the number of times the failure (false) branch has been taken by <node name> since control was last passed to <node name> by the node above it. If this <node name> has no failure branch then the returned value is the same as if such a branch existed and the nodes lower in this branch always return control to this node.

NUMBER OF EXHAUSTS AT <node name>

This returns the number of times the exhausted branch has been taken by <node name> since control was last passed to <node name> by the node above it. If this <node name> has no exhausted branch then the returned value is the same as if such a branch existed. This returned value will be either 0 or 1.

ABS(<value>)

The arithmetic absolute value of <value> is returned.

II.4 The SEARCH Statement

The search expression is probably the single-most important construct in MTQ since it allows expression of the time qualified parameter relationships of interest to the user. The syntax and semantics of the search expression are given in this section. Figure 8 gives its syntax; see the appendix for resolution of any ambiguities in the operator-operand associations.

```
<search expression> ::= <search function> | <complex search>

<search function> ::=   TRUE |
                        FALSE|
                        THERE ARE NO MORE PATIENTS|
                        IT|
                        <item reference> IS NORMAL|
                        <item reference> IS ABNORMAL|
                        <object> IS UNKNOWN|
                        <object> IS KNOWN|
                        NOT <search expression>|
                        BETWEEN <value> AND <value> EACH <item name>[1]
                         CHANGES FROM THE PREVIOUS <item name>[1]
                         BY <value> TO <value>|
                        <value> ~ <value> +OR- <value>

<complex search> ::= <relation>|
                     <search expression><logical op><search expression>|
                     <special logical expression>|
                     (<search expression>)

<relation> ::=  <value> <relational op> <value> |
                <value> = '<character string>' |
                '<character string>' = <value>

<relational op> ::= IS <standard relational op>|
                       <standard relational op>

<standard relational op> ::= > | >= | = | ne | <= | <

<logical op> ::= AND | OR

<special logical expression> ::= <relation><another comparison>

<another comparison> ::= <special logical op><relational op><value>|
                         <another comparison><another comparison>

<special logical op> ::= ,AND | ,OR

<object> ::= <value> | <search expression>
```

[1]These should be the same <item name>.

The Syntax of a Search Expression

figure 8

Note that now <test exp> of figure 3 may be defined as

<test exp> ::= <search expression>

For the most part a search expression is like a standard boolean expression found in many conventional programming languages. The major components of <search expression> that need explanation are <search function> and <special logical expression>. After describing these in detail below, the discussion will return to the topic of how to use <search expression> as the search expression within a search node.

An Explanation of Each <search function>

TRUE

This is the boolean true primitive.

FALSE

This is the boolean false primitive.

THERE ARE NO MORE PATIENTS

This returns true if the current patient, whose data is being searched, is the last patient in the database.

IT

This returns the value of the closest <value> to the left of IT that is used on the left side of a <relational operator>. This is used as a convenience when writing relations. For example one could use the following expression:

the WBC at visit x is > the WBC at visit y and IT is < the WBC at visit z

instead of the more conventional and tedious expression:

the WBC at visit x is > the WBC at visit y and the WBC at visit x is < the WBC at visit z

<item reference> IS NORMAL

Within the database there is a range of values associated with each item (parameter), and this is defined as the normal value range for that item with respect to the patient population stored in the database. This function returns TRUE if the <item reference> is within this normal range for that particular item; otherwise FALSE is returned.

<item reference> IS ABNORMAL

This returns NOT(<item reference> IS NORMAL).

<object> IS UNKNOWN

In the next section on handling unknown values a complete discussion will be given on what constitutes an unknown object. Until then, suffice it to say that TRUE is returned by the above function if <object> is unknown, otherwise FALSE is returned.

<object> IS KNOWN

This returns NOT(<object> IS UNKNOWN).

NOT <search expression>

If <search expression> evaluates to TRUE then FALSE is returned. If <search expression> evaluates to FALSE then TRUE is returned. If <search expression> evaluates to an unknown value then an unknown value is returned; more will be said about this in the next section on handling unknown values.

BETWEEN <value>$_1$ AND <value>$_2$ EACH <item name> CHANGES FROM THE PREVIOUS <item name> BY <value>$_3$ TO <value>$_4$

<item name> is the same item name in both cases. <value>$_1$ is a lower bound visit and <visit>$_2$ is a higher bound visit. The value of <item name> at each visit within this range is subtracted from the value of <item name> at the following visit (if that visit is within the range). For each such subtraction the result must be no less that <value>$_3$ and no greater than <value>$_4$. If this is not the case then FALSE is returned, otherwise TRUE is returned.

<value>$_1$ ~ <value>$_2$ +OR- <value>$_3$

This is the approximate equality operator (Kahn 1975). ~ is the standard mathematical approximation sign. The above has the same semantic meaning as the following:

(<value>$_1$ >= <value>$_2$ - <value>$_3$)

AND

(<value>$_1$ <= <value>$_2$ + <value>$_3$)

An Explanation of the <special logical expression>

This expression is provided as another convenience in the construction of a search expression. Consider the example associated with IT in the section above on <search function>. That expression could also be written as:
the WBC at visit x is > the WBC at visit y ,AND < the WBC at visit z

In a similar manner ,OR can be used. A sequence of ,AND or ,OR can also be used as in
<value> is = 1,OR = 2,OR = 3

An equivalent semantic representation of

<relation> ,OR <relation op> <value>

is given by

(<relation> OR IT <relation op> <value>)

Similarly an equivalent semantic representation of

<relation> ,AND <relation op> <value>

is given by

(<relation> AND IT <relation op> <value>)

The <special logical expression>'s are left associative and ,OR has the same precedence as ,AND. It is suggested however that ,OR and ,AND not be used together in the same <special logical expression> as this may lead to conceptual confusion.

Now that <search expression> has been defined, it can be used in discussing how to construct a search statement. Actually the syntax is quite simple, as it is given by:

SEARCH: <search expression>$

If <search expression> evaluates to true then control passes down the true (success) branch of the current node. If <search expression> is false then control passes down the false (failure) branch of the current node. If <search expression> is an unknown value (as discussed in the next section) then control is maintained at the current node and the next valid node frame is instantiated before testing <search expression> again.

As an example, figure 9 shows figure 6 with the English search expressions converted to MTQ statements.

```
Top is the starting node$
Print the patient's hospital number.

When Top is exhausted:
  Drug_given$
  Define visit a such that it is any visit within range$
  Search: Rx_given at visit a = 'yes'$

  When Drug_given is true:
    Find_WBC_drop$
    Define visit b such that it is the visit immediately after visit a$
    Search: The WBC at visit b is < the WBC at visit a$

    When Find_WBC_drop is false:
      Report_failure$
      Print that a decrease was not found.

    When Find_WBC_drop is true:
      Report_success$
      Print that a decrease was found.
```

Figure 6 with Added MTQ Search Expressions

figure 9

II.5 Handling Unknown Values

In the previous sections references have been made to "unknown values". This section will clarify the meaning of unknown values, as well as discuss the manner in which they are handled by MTQ (Belnap 1977). An unknown value can be the value of either <value> or of <search expression>. It is represented by the symbol ???. An unknown value (i.e. ???) usually originates with an item (parameter) at a given visit which has not been entered, and thus has a default value of ???. Also, if the visit number in an item reference is out of range with respect to the current patient's data then ??? is returned. This ??? value can then propagate throughout the expression in which it is contained. The remainder of this section is devoted to describing this propagational effect. Notice that this section is augmenting much of what has been stated previously about the value of functions, operators, and expressions; there is now an extra case in which ??? may be their resulting value.

First consider the functions defined by <value function> and <search function>. Of these consider the functions MAXIMUM, MINIMUM, AVERAGE, ENTRIES, and GREATEST DIFFERENCE BETWEEN. Each of these four functions uses a range (or list) of values as its input. Each function ignores any ??? arguments in the range (or list) to which it is applied. If no arguments remain after this exclusion process, then ??? is returned as the function's value. As an example, in the case of AVERAGE WBC, the average of those WBC's <u>entered</u> from visit 1 to LAST VISIT is returned.

Now consider the approximate equality operator which has the form <value>$_1$ ~ <value>$_2$ +OR- <value>$_3$. If <value>$_1$ equals <value>$_2$ then TRUE is returned regardless of the value of <value>$_3$. Otherwise if one of the three <value>'s is ??? then ??? is returned.

Any other functions in <value function> or <search function>, other than the five just mentioned, will return ??? if any of their arguments have the value ??? . For example, consider the function reference:

DAYS FROM <date>$_1$ TO <date>$_2$

If either <date>$_1$ or <date>$_2$ has the value ???, then ??? is the value returned by this function.

Now consider the arithmetic and relational operators. If any argument of a relational operator has the value ???, then ??? is the returned value. This is also the case for both the infix and prefix versions of the operators + and - . The operators *, /, and ^ return ??? if any of their arguments are ???, unless it can be determined from a non-??? argument that a non-??? result should be returned. For example if one argument to * is ???, and the other is zero, then zero will be the value returned.

Finally, consider the operators AND and OR. The tables below give the various possibilities.

AND

arg1\arg2	TRUE	FALSE	???
TRUE	TRUE	FALSE	???
FALSE	FALSE	FALSE	FALSE
???	???	FALSE	???

OR

arg1\arg2	TRUE	FALSE	???
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	???
???	TRUE	???	???

The only remaining MTQ operators not covered in this section are ,AND and ,OR. To determine the way these operators handle unknown values, refer to their equivalent semantics in section II.4.

II.6 The USE Statement

At this point the DEFINE VISIT and the SEARCH constructs have been completely defined. There remains but one other search node construct, and it is called the USE construct. Recall that until now a view has been taken that for each search node there is an associated sequence of valid node frames. Each valid node frame is used in turn to establish a visit variable reference frame as discussed in section II.2. For each reference frame the search expression at the given node is tested. If that test is true then the true (success) branch is taken; if it is false the false (failure) branch is taken; if it is ??? then the next valid node frame is instantiated and this testing process continues. Notice that this process assumes that there is an interest in each and every success and failure of the search expression. In many cases this is just not true. For instance, in many situations the desire is to find the existence of some condition and if found then print some pertinent data. In this case only the first success is of interest. The USE construct has been developed in order to allow control over which successes and failures will be used at a given node, or in other words, in which cases the success or failure branch should be taken from the node.

In the remainder of this section the USE construct will be explained with respect to controlling successes. One should realize that control of the failures is similarly attained by using the word failure (or failures) instead of success (or successes) in the statements below.

The successes and failures of a given node are considered to be

zero at and only at the moment that node is given control from the node above it. The successes and failures then increase incrementally as the success and failure branches are taken respectively. In the case that the USE statement restricts the success (or failure) branch from being taken, then the success (or failure) at the node will not be incremented. In what follows the syntax of each type USE statement will be given, followed by its semantics.

USE THE FIRST <value> SUCCESSES$

<value> must evaluate to an integer. This specifies that the success branch is to be taken from the current node of definition, up to and including (but not beyond) the first <value> successes. After <value> successes have been made it is as though there was no success branch at all at this node (and this condition persists for as long as the number of successes is greater than <value>).

USE THE INITIAL SUCCESS$

This is equivalent to: USE THE FIRST 1 SUCCESSES$

USE ZERO SUCCESSES$

This is equivalent to: USE THE FIRST 0 SUCCESSES$

In other words the success branch is never taken from this node. This is the default if no USE SUCCESS statement is explicitly specified at the node. This default forces the user to specify a USE statement if he intends for control to ever pass down the success branch of the current

node; such a convention will hopefully lead to better documentation of the search program.

USE EVERY SUCCESS$

This says that every time the search expression evaluates to TRUE the success branch will be taken.

USE THE LAST <value> SUCCESSES$

First imagine that instead of the above statement, the statement USE EVERY SUCCESS$ had been specified. Then assume that n successes resulted. The USE THE LAST <value> SUCCESSES$ statement specifies that the last <value> of these n successes should be used. If n is less than or equal to <value> then all n successes are to be used. If n is greater than <value> then it is as though the success branch did not exist for the first n - <value> successes, and then for the remaining <value> successes it does exist and therefore is used as normal.

USE THE FINAL SUCCESS$

This is equivalent to USE THE LAST 1 SUCCESSES$

USE THE SPECIAL <function> SUCCESSES$

Whenever the search expression evaluates to TRUE, <function> will be called and if it returns FALSE the success branch is not taken but rather the next valid node frame is instantiated. If it does not return FALSE then the success branch is taken. Presently there are no <function>'s

available, but its existence makes expansion of the USE construct relatively easy. (Note that the word TRUE above must be replaced by FALSE in the case of USE THE SPECIAL <function> FAILURES$).

This completes the USE statements dealing strictly with controling successes. There is one other USE statement (other than the "failure" version of the statements just described). It is as follows:

USE EITHER THE INITIAL SUCCESS OR THE INITIAL FAILURE$

This statement is used to specify that either the success branch or the failure branch is to be taken, depending on the first known search expression value. When control returns to this node the exhausted branch will immediately be taken. Thus, this is like a conventional conditional statement. In most cases this statement is used only in a search node which has no dynamic visit variable definitions, and thus has only one reference frame with which to test its search expression.

Although only single USE specifications have been shown above, actually a list of such is permissible. For example one could state:

USE THE FIRST 2 , THE LAST 3 SUCCESSES$

The meaning of this list of specifications is that the union of the

successes meeting each specification determines which successes are to be used, and furthermore each success in that union is used at the point at which the associated search expression is found to be true. If a later list element specification contradicts an earlier one in the statement then the later one is used. Only one USE SUCCESS statement should appear per node. The complete syntax for the USE construct is given in figure 10.

```
<use statement> ::= USE <class list> <type>$|
                    USE EITHER THE INITIAL SUCCESS OR THE INITIAL FAILURE

<class list> ::= <class list>,<class> | <class>

<class> ::= THE FIRST <value> | THE LAST <value> |
            THE SPECIAL <function> | THE INITIAL | THE FINAL |
            EVERY | ZERO

<type> ::= SUCCESSES | SUCCESS | FAILURES | FAILURE

<function> ::= <the name of a MACLISP function in the environment>
```

The Syntax of USE

figure 10

As an example, refer back to figure 9 and recall that in the Drug_given node, of all the possible cases in which the search expression might be true, only the first one (i.e. the first success) is of interest. This is of course an apparent instance of the need to utilize:

USE THE INITIAL SUCCESS$

In the node Find_WBC_drop, the intent was to perform a conventional conditional test and use either the initial success or the initial failure. This is then an instance of the need to utilize:

USE EITHER THE INITIAL SUCCESS OR THE INITIAL FAILURE$

Figure 11 shows figure 9 with these new additions to the search nodes.

```
Top is the starting node$
Print the patient's hospital number.

When Top is exhausted:
  Drug_given$
  Define visit a such that it is any visit within range$
  Search: Rx_given at visit a = 'yes'$
  Use the initial success$

  When Drug_given is true:
    Find_WBC_drop$
    Define visit b such that it is the visit immediately after visit a$
    Search: The WBC at visit b is < the WBC at visit a$
    Use either the initial success or the initial failure$

    When Find_WBC_drop is false:
      Report_failure$
      Print that a decrease was not found.

    When Find_WBC_drop is true:
      Report_success$
      Print that a decrease was found.
```

Figure 9 with the Addition of USE Statements

figure 11

As another example of the USE statement, suppose that one wanted to know the initial and final dates on which a drug was given to a patient (assume as before that this is a special patient population in which only one drug is being given as treatment). In this case a visit variable is defined so as to be any visit within range (call this visit e). The search expression should test that the drug was administered. The USE statement would be:

USE THE INITIAL , THE FINAL SUCCESSES$

This search is shown in figure 12.

```
.
.
.
Find_initial_and_final_drug_dates$
Define visit e such that it is any visit within range$
Search: Rx_given at visit e = 'yes'$
Use the initial , the final successes$

When Find_initial_and_final_drug_dates is true:
  Output_the_drug_date$
  Print the drug date.
```

A Second Example of Using USE

figure 12

II.7 Action Nodes

Although the concept of an action node was introduced in section II.1, up until now the main focus of attention has been on building search nodes. Action node construction will now be discussed. Recall that the typical use of an action node is to communicate the result of a search node.

The primary statement in an action node is

ACTION: <actions>$

or

ACTIONS: <actions>$

where <actions> is a list of actions. The list is evaluated from the left to the right. The complete syntax of <actions> is given in figure 13.

```
<actions> ::= <actions>,<action> | <action>

<action> ::= BACKUP TO <node name> |
             CONSIDER THE NEXT PATIENT|
             HALT|
             OUTPUT(<output values>)|
             <CGOL code>[1]

<output values> ::= <output values>,<output value> | <output value>

<output value> ::= <value> | '<character string>'
```

Action Node Syntax

figure 13

A description of each <action> is given below.

BACKUP TO <node name>

<node name> should be some search node between the current action node and the very top node. The search node returned to then resumes operation as if control had been passed to it from a node directly below it. Note that if the node <node name> is exhausted then control returns to the first non-exhausted search node encountered from <node name> to the very top node.

CONSIDER THE NEXT PATIENT

When this is encountered in an action list, none of the remaining members of the action list will evaluated. Instead, a BACKUP TO <very top node> is performed and the search tree begins execution anew using the data of the next patient in the database. If there are no more patients then HALT is executed as described below.

HALT

When this is encountered in an action list, none of the remaining members of the list will be evaluated. Instead a BACKUP TO <very top node> is performed, the search is stopped, and MTQ returns control to the user's terminal in a mode ready to accept a new search tree or modifications to any existing tree in the environment.

OUTPUT(<output values>)

First a carriage return is performed on the current output device, which is typically the user's terminal. OUTPUT then evaluates each member of its argument list from left to right. After a <value> in the list is evaluated it is sent to the current output device separated by a space from the previous <value> that was output.

<CGOL code>

This action gives the user the power to execute any number of CGOL statements. CGOL is an algorithmic language with a syntax similar to Algol, but with the semantics of MACLISP. (Pratt 1976) Any of the actions described above may be used within CGOL statements, however the control effects of HALT, BACKUP TO <node name>, and CONSIDER THE NEXT PATIENT will not take place until after the <cgol code> has been completely evaluated. The functions and operators in <value> and <search expression> may also be used. Notice that by using CGOL If-Then statements the user can even write search expressions within an action node. Thus, if a CGOL program is better suited to solving a particular search than the normal MTQ formalism, then the user has the ability to use CGOL without sacrificing the loss of useful functions and operators within <value> and <search expression>.

As an example of action nodes, figure 14 shows the MTQ representation of action nodes previously described in English in figure 11 of section II.6.

```
Top is the starting node$
Action: Output('patient_number' , patient_number)$

When Top is exhausted:
  Drug_given$
  Define visit a such that it is any visit within range$
  Search: Rx_given at visit a = 'yes'$
  Use the initial success$

  When Drug_given is true:
    Find_WBC_drop$
    Define visit b such that it is the visit immediately after visit a$
    Search: The WBC at visit b is < the WBC at visit a$
    Use either the initial success or the initial failure$

    When Find_WBC_drop is false:
      Report_failure$
      Action: Output('the_initial_decrease_did_not_occur')$

    When Find_WBC_drop is true:
      Report_success$
      Action: Output('the_initial_decrease_occured')$
```

Figure 11 with Added MTQ Action Statements

figure 14

As another example, figure 15 shows an MTQ representation of the action nodes of figure 12.

```
.
.
.
Find_initial_and_final_drug_dates$
Define visit e such that it is any visit within rang
Search: Rx_given at visit e = 'yes'$
Use the initial , the final successes$

When Find_initial_and_final_drug_dates is true:
  Output_the_drug_date$
  Action: Output('Rx_given_at' , date at visit e)$
```

Figure 12 with Added MTQ Action Statements

figure 15

The only other construct allowed in an action node besides ACTION is DEFINE VISIT. Any visit variable definition allowed in a search node is also allowed in an action node. However, the interpretation is different. With a search node the visit variable definitions specify a sequence of valid node frames (and thus reference frames), and for each such frame the search expression is tested. Depending on the result of that test the success branch may be taken, the failure branch may be taken, or the next valid node frame may be instantiated. With an action node, however, instead of testing a search expression, the action list at the node is evaluated. After evaluation is completed, the next valid node frame is instantiated to give rise to a new reference frame. This process of instantiation and evaluation continues until all the specified valid node frames have been instantiated. At this point the action node is exhausted and the exhausted branch is taken. (If there are no dynamic visit variable definitions within an action node, the ACTION statement is

evaluated within the existing reference frame and then it is exhausted so the exhausted branch is taken). Figure 16 is an example of using a DEFINE VISIT statement in an action node to print the patient number and visit number for each visit at which the WBC is abnormal.

```
Abnormal_WBC_finder is the starting node$
Define visit x such that the WBC at visit x is abnormal$
Action: Output(patient_number , visit x)$
```

An Example of Using a DEFINE VISIT Statement within an Action Node

figure 16

II.8 The END and RUN Statements

Two simple MTQ statements not yet mentioned are END and RUN. END is an optional statement that may be used for documentation purposes to define explicitly where a node terminates. It has the form:

END <node name>$

In most cases it is not used and in no case is it needed. The RUN statement has the form:

RUN <starting node name>$

where <starting node name> is the node at the top of some search tree. RUN is used to actually initiate execution of the search.

III.

An MTQ Search Example

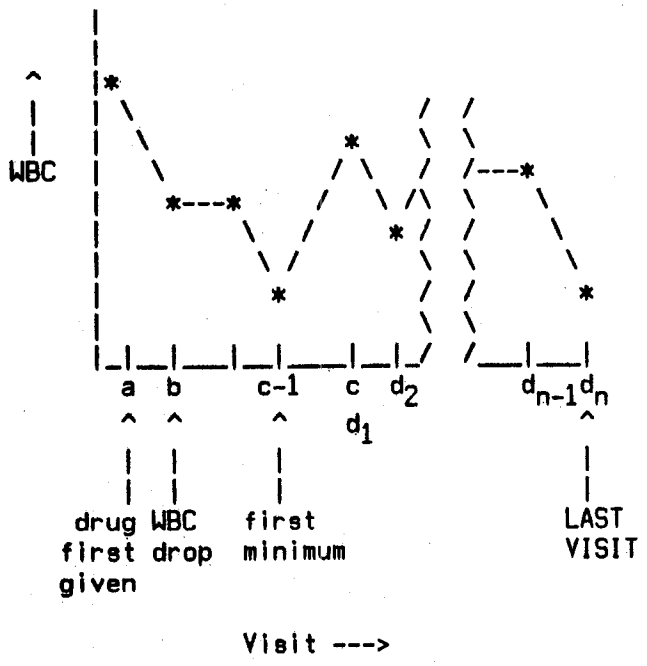
In this chapter a complete MTQ search example will be developed and the results obtained from running the search on a small database will be shown. The purpose of this example is to demonstrate an MTQ program, and not necessarily to perform a medically useful search. The database used has been developed for the sole purpose of testing MTQ and consequently it contains data on only two patients. This data was taken from the Southeastern Cancer Group Hodgkin's Disease database (Wirtschafter and Carpenter 1975) which contains data on several hundred patients with approximately 300 items associated with each visit. The test database to be used here contains only 8 items per visit. Each item is listed in figure 17 along with its unit of measurement (most of these items have numeric values; with minor modifications MTQ could handle complex text string searches as well). There is also a ***header item*** known as the patient_number. A ***header item*** has no time component. In a production database there would be many more header items such as the patient's name, birth date, current address, etc. Obviously there would be additional time oriented items too.

Item Name	Units
Date	YYMMDD
Temp	Degrees Fahrenheit
BCNU	Mg/D
HCT	%
Platelets	x1000
WBC	x1000
Uric_acid	Mg%
Rx_given	Yes-No

Items in the MTQ Test Database

figure 17

The search example here will use the search nodes from figures 14 and 15. Recall in figure 14 that the search was to find the initial drug administration, and this was immediately followed by a visit at which the WBC had declined from the WBC at the time of the drug administration. Figure 15 is a search to locate and print the initial and final drug dates of the patient. Before discussing how these nodes fit into the rest of the search example, a graphic representation of the overall search pattern is shown in figure 18.

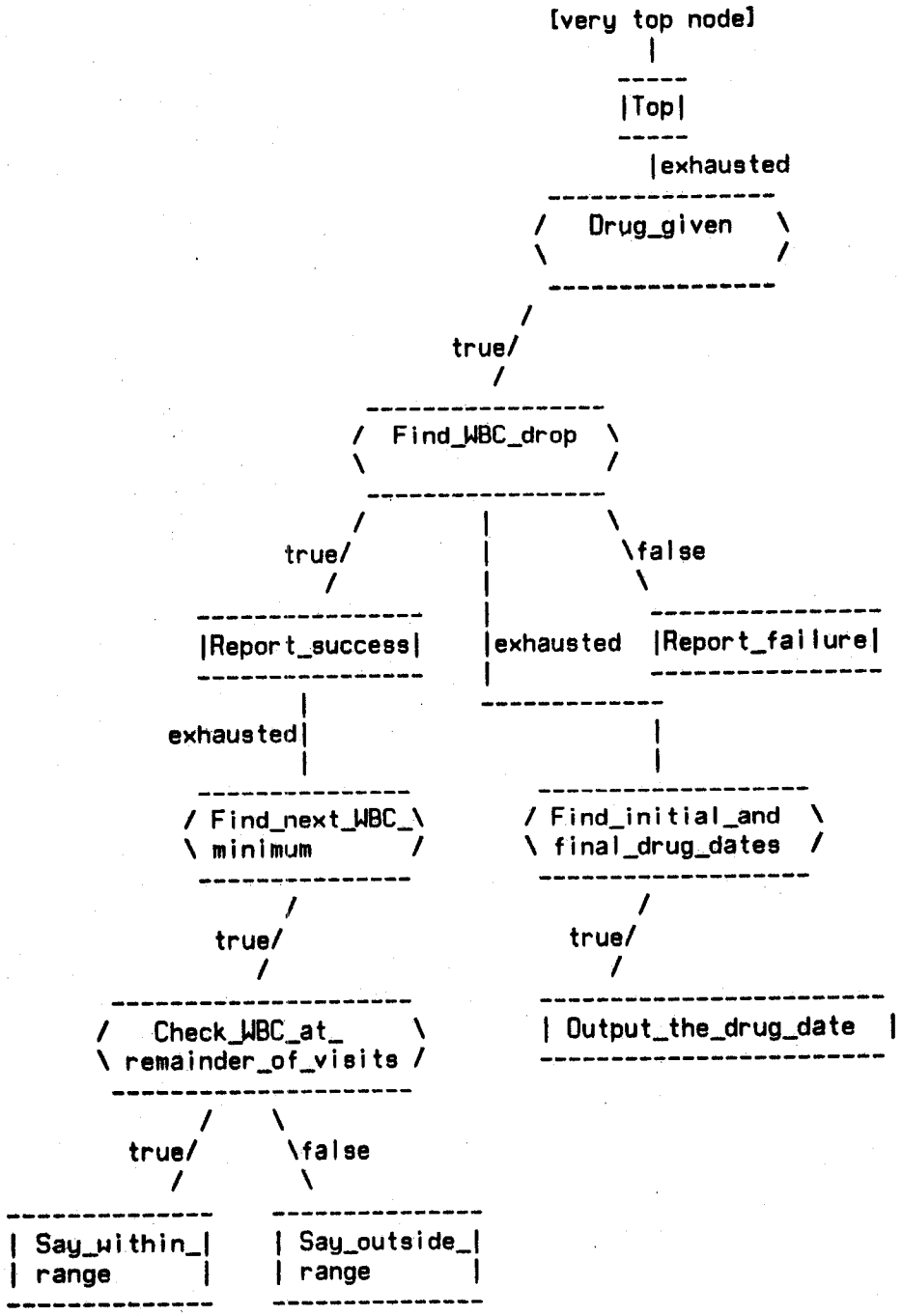


A Graphic Representation of the Search Example

figure 18

The drug is first given at visit a, and at the next visit, visit b, the WBC has decreased. Visit c-1 is the visit at which the first WBC minimum occurs after visit b. Visit c is of course the visit immediately following visit c-1. The WBC at each of the visits on or after visit c is then compared with the WBC at visit c to determine if it is within 3 (3000) counts of the WBC at visit c.

Figure 19 shows a tree of the node names of the example search.



A Tree of the Node Names of the Search Example
figure 19

Notice that figure 14 defines the nodes: Top, Drug_given, Find_WBC_drop, Report_success, and Report_failure.

Figure 15 defines the nodes: Find_initial_and_final_drug_dates and Output_the_drug_date.

Find_next_WBC_minimum and its true branch subtree are represented as MTQ code in figure 20. The entire search program is shown in figure 21. The output of this program, when run using the test database, is shown in figure 22. Notice that both patients 1 and 2 completely fit the pattern sought. Observe also that for patient 2 the WBC was "outside of range" at visit 9 and at visit 12, but that no mention is made of visits 10 and 11. This is because the WBC at visits 10 and 11 are not known and consequently neither the success branch nor the failure branch was taken(see Handling Unknown Values in section II.5).

```
When Report_success is exhausted:
  Find_next_WBC_minimum$
  Define visit c such that it is any visit after visit b$
  Search: The WBC at visit c is > the WBC at (visit c  - 1)$
  Use the initial success$

  When Find_next_WBC_minimum is true:
    Check_WBC_at_remainder_of_visits$
    Define visit d such that it is any visit on or after visit c$
    Search: The wbc at visit c ~ the WBC at visit d +or- 3$
    Use every success$
    Use every failure$

    When Check_WBC_at_remainder_of_visits is true:
      Say_within_range$
      Action: Output('within_range_at_visit' , visit d ,
                     'on_the_date' , date at visit d ,
                     'with_WBC_at' , the wbc at visit d)$

    When  Check_WBC_at_remainder_of_visits is false:
      Say_outside_range$
      Action: Output('outside_range_at_visit' , visit d ,
                     'on_the_date' , date at visit d ,
                     'with_the_WBC_at' , the wbc at visit d)$
```

The MTQ Representation of Find_next_WBC_minimum and Its True Subtree

figure 20

```
Top is the starting node$
Action: Output('patient_number' , patient_number)$

Top is exhausted:
  Drug_given$
  Define visit a such that it is any visit within range$
  Search: Rx_given at visit a = 'yes'$
  Use the initial success$

  When Drug_given is true:
    Find_WBC_drop$
    Define visit b such that it is the visit immediately after visit a$
    Search: The WBC at visit b is < the WBC at visit a$
    Use either the initial success or the initial failure$

    When Find_WBC_drop is false:
      Report_failure$
      Action: Output('the_initial_decrease_did_not_occur')$

    When Find_WBC_drop is exhausted:
      Find_initial_and_final_drug_dates$
      Define visit e such that it is any visit within range$
      Search: Rx_given at visit e = 'yes'$
      Use the initial , the final successes$

      When Find_initial_and_final_drug_dates is true:
        Output_the_drug_date$
        Action: Output('Rx_given_at' , date at visit e)$

    When Find_WBC_drop is true:
      Report_success$
      Action: Output('the_initial_decrease_occured')$
```

An MTQ Representation of the Complete Search Example

figure 21

(continued on the next page)

```
When Report_success is exhausted:
  Find_next_WBC_minimum$
  Define visit c such that it is any visit after visit b$
  Search: The WBC at visit c is > the WBC at (visit c  - 1)$
  Use the initial success$

  When Find_next_WBC_minimum is true:
    Check_WBC_at_remainder_of_visits$
    Define visit d such that it is any visit on or after visit c$
    Search: The WBC at visit c ~ the WBC at visit d +or- 3$
    Use every success$
    Use every failure$

    When Check_WBC_at_remainder_of_visits is true:
      Say_within_range$
      Action: Output('within_range_at_visit' , visit d ,
                     'on_the_date' , date at visit d ,
                     'with_WBC_at' , the wbc at visit d)$

    When Check_WBC_at_remainder_of_visits is false:
      Say_outside_range$
      Action: Output('outside_range_at_visit' , visit d ,
                     'on_the_date' , date at visit d ,
                     'with_the_WBC_at' , the wbc at visit d)$
```

Continuation of figure 21

```
PATIENT-NUMBER 1
THE-INITIAL-DECREASE-OCCURED
WITHIN-RANGE-AT-VISIT 4 ON-THE-DATE 721103 WITH-WBC-AT 5.1
WITHIN-RANGE-AT-VISIT 5 ON-THE-DATE 721114 WITH-WBC-AT 4.4
OUTSIDE-RANGE-AT-VISIT 6 ON-THE-DATE 721128 WITH-THE-WBC-AT 10.8
WITHIN-RANGE-AT-VISIT 7 ON-THE-DATE 721212 WITH-WBC-AT 7.7
WITHIN-RANGE-AT-VISIT 8 ON-THE-DATE 721226 WITH-WBC-AT 5.2
WITHIN-RANGE-AT-VISIT 9 ON-THE-DATE 730109 WITH-WBC-AT 4.8
WITHIN-RANGE-AT-VISIT 10 ON-THE-DATE 730123 WITH-WBC-AT 3.0
WITHIN-RANGE-AT-VISIT 11 ON-THE-DATE 730206 WITH-WBC-AT 4.1
OUTSIDE-RANGE-AT-VISIT 12 ON-THE-DATE 730220 WITH-THE-WBC-AT 1.6
WITHIN-RANGE-AT-VISIT 13 ON-THE-DATE 730306 WITH-WBC-AT 4.1
WITHIN-RANGE-AT-VISIT 14 ON-THE-DATE 730313 WITH-WBC-AT 4.0
RX-GIVEN-AT 721013
RX-GIVEN-AT 730313

PATIENT-NUMBER 2
THE-INITIAL-DECREASE-OCCURED
WITHIN-RANGE-AT-VISIT 4 ON-THE-DATE 720908 WITH-WBC-AT 5.6
WITHIN-RANGE-AT-VISIT 5 ON-THE-DATE 720915 WITH-WBC-AT 5.3
WITHIN-RANGE-AT-VISIT 6 ON-THE-DATE 721013 WITH-WBC-AT 5.6
WITHIN-RANGE-AT-VISIT 7 ON-THE-DATE 721114 WITH-WBC-AT 3.4
WITHIN-RANGE-AT-VISIT 8 ON-THE-DATE 721121 WITH-WBC-AT 4.5
OUTSIDE-RANGE-AT-VISIT 9 ON-THE-DATE 721205 WITH-THE-WBC-AT 2.3
OUTSIDE-RANGE-AT-VISIT 12 ON-THE-DATE 730103 WITH-THE-WBC-AT 1.7
WITHIN-RANGE-AT-VISIT 13 ON-THE-DATE 730116 WITH-WBC-AT 3.8
WITHIN-RANGE-AT-VISIT 14 ON-THE-DATE 730123 WITH-WBC-AT 5.0
WITHIN-RANGE-AT-VISIT 15 ON-THE-DATE 730130 WITH-WBC-AT 4.6
WITHIN-RANGE-AT-VISIT 16 ON-THE-DATE 730227 WITH-WBC-AT 4.0
RX-GIVEN-AT 720817
RX-GIVEN-AT 730227

SEARCH/-COMPLETED
```

Output of the Search Example

figure 22

IV.

Discussion

MTQ has been only minimally tested to date. Although programming bugs are likely to appear, the more interesting issue is whether the language as defined will prove to be useful if extensively used in a production environment. The limited testing that has been done seems to indicate that it is relatively easy to understand the meaning of a isolated node. The more difficult task, however, is understanding how the flow of control passes between nodes. This seems to be due to the way in which the flow of control is rather implicitly shared by the DEFINE VISIT, SEARCH (or ACTION), and USE statements. To understand to a first approximation their combined intent is usually easy; to understand in detail all of their interacting implications sometimes requires serious thought. This is usually the case whenever a computer language moves the control structure to a higher level of abstraction. One improvement to MTQ would be the addition of a programmer's aid module that would be able to answer restricted questions about MTQ's interpretation of the search program. Since MTQ has few constructs, it may not be unrealistic to attempt a simple, first level approach to such an aid. On the other hand perhaps the understandability problem is due to the unorthodox control structure. If so then it will require more use before it becomes "natural" to use. However, the goal of MTQ as stated at the outset has not been to provide a totally naive user (in a programming sense) with a tool to perform arbitrarily complex searches, but rather that a programmer

should perform such searches, which then can be interpreted with relative ease by the naive user. Hopefully, after sufficient exposure the naive user will become less naive and able to code more difficult searches himself. It does seem likely that even a very naive user will be able to construct searches which have one search node and a few action nodes, but this has not been tested. One additional aid for the naive user would be to have a book of prefabricated search trees wherein the user could essentially "fill in the blanks" to create his own tailored version of a search. With clearly written instructions this method would probably be quite successful. It may even be worthwhile to build a special prompter program which when given the name of a prefabricated search tree, would prompt the user for the "fill in the blank" parameters, would construct the tailored search tree, and finally would run the search.

There are several other immediate improvements that can be made to MTQ. One is the ability to input and edit search trees online. Presently a search tree is read from a data file. Editing capabilities would allow copying portions of a tree from one location in the tree to another. It would also be helpful to be able to display the tree structure of a program in a format similar to that of figure 19 previously shown. Another graphic aid would be the ability to enter the time relationships sought via a graph. This input would be similar to that of figure 18. This capability is probably very difficult to implement; also it is not presently clear if common search patterns could be specified unambiguously with the use of a graph. As mentioned above, a question-answer module would be quite useful. A first step could be made in that direction by

simply having MTQ type the search expression back to the user in a fully parenthesized form so that the user can assure himself that MTQ has correctly interpreted the incompletely parenthesized input expression as was intended. Another improvement would be the addition of error handling routines, since currently there are almost none. Errors could be flagged and corrective suggestions made at both the point of program entry and execution.

The list of possible improvements continues. If the control provided by the USE statement is not sufficient it may be necessary to expand the USE options available. A spelling corrector and synonym dictionary would greatly improve the ease of using MTQ. MTQ is an interpreter and presently it is in the form of MACLISP code which is also being interpreted. This means that MTQ is slow. One immediate and easy improvement is to compile the MACLISP code. This would eliminate one level of interpretation. There are currently bottlenecks in MTQ which have to do with switching states between nodes. With a small amount of effort these problems could be significantly minimized. Other efficiency improvements include optimization of the DO loops used to establish valid node frames. After these improvements are made the efficiency should be well within the limits of production usage standards.

As MTQ evolves there will no doubt be new functions and operators added. There is even now a preceived need to have MTQ convert between differing units of measurement. For example, each item presently has a standard unit of measurement; it would be convenient to be able to reference an item with a differing unit and have MTQ convert this to the

standard. (The standard would of course serve as the default if no unit modifier was used). For example a reference to

THE TEMP IN ^{O}C AT VISIT X

would be converted to the equivalent standardized temperature in degrees fahrenheit. There are other type functions that are needed as well. In particular there is an acute need for more elaborate output functions. These functions and many others are easily defined in GOT, which is used as the parser for all MTQ input. GOT has proved to be ideal for the job. It is powerful, elegant, and easy to use; a combination rarely found.

In the future hopefully MTQ will be tested on a trial basis with a large medical database, and if it proves to be adequate then production usage would follow. Such use would then allow data to be gathered concerning the type searches most often requested. This could then be used in designing a more English-like language on top of MTQ so that a casual user would have more of the full power of MTQ at his disposal.

References

Beaman, P.D.: A Special Study System for COSTAR: COmputer STored Ambulatory Record, M.I.T. S.M. thesis, May, 1977.

Belnap, Nuel D. : How a Computer Should Think, lecture at M.I.T., March 11, 1977.

Bruce, Bertram: Representation and Processing of Sequences of Events, CBM-TR-4, Rutgers University, May, 1972.

Fries, James F.: Time-Oriented Patient Records and a Computer Databank, Journal of the American Medical Association, 222:1536-1542, 1972.

Fries, James F.: A Data Bank for the Clinician?, The New England Journal of Medicine, 294:1400-1402, 1976.

Kahn, Kenneth M.: Mechanization of Temporal Knowledge, M.I.T. Project MAC Technical Report 155.

Moon, David A.: MACLISP Reference Manual, April, 1974.

Pratt, Vaughan R.: CGOL - an Alternative External Representation for LISP Users, M.I.T. A.I. Working Paper 121.

Pratt, Vaughan R.: Top Down Operator Precedence, ACM Sigact/Sigplan Symposium on Principles of Programming Languages, October, 1973.

Weyl, Stephen; Fries, James F.; Wiederhold, Gio; and Germano, Frank: A Modular Self-Describing Clinical Databank System, Computers and Biomedical Research, 8:279-293, 1975.

Wirtschafter, David D. and Carpenter, J.T.: Analysis of the Uniformity in the Care Process in Hodgkin's Disease Protocol, Proceedings of the American Society for Clinical Oncology (Abstract), San Diego, California, 1975.

Wirtschafter, D.D.; Cooper G.F.; Russo, A.; and Mesel, E.: A Retrieval System for a Time-Oriented Database, The Sixth Annual Conference of the Society for Computer Medicine (Abstract), Boston, Massachusetts, 1976.

Appendix

Function and Operator Precedence

In the table that follows the precedence is given for each function and operator in MTQ. From henceforth the word function will be used to mean either a function or an operator since the distinction is not critically important. Only the first word of the function name is given in the table. For instance GREATEST is used to represent the function GREATEST DIFFERENCE BETWEEN <value list>. The left binding power and the right binding power columns pertain only to the leftmost and rightmost respective operands of the function, since these are the only operands that can possibly be ambiguous with respect to which of two functions gets the operand. If there is competition for an operand then the function with the highest binding power gets it; left association is favored in a draw. If either the left or the right binding power is irrelevant to the function then a dash is entered.

In general when unsure of the precedence, use parentheses to structure the expression.

left binding power	name	right binding power
-	ABS	-
8	AND	8
8	,AND	10
36	AT	36
-	AVERAGE	22
-	BETWEEN	9
-	CLOSEST	14
-	ENTRIES	22
-	FALSE	-
-	GREATEST	14
10	IS NORMAL	-
10	IS ABNORMAL	-
10	IS KNOWN	-
10	IS UNKNOWN	-
-	IT	-
-	LAST VISIT	-
-	MAXIMUM	22
-	MINIMUM	22
-	NOT	9
-	NUMBER	35
7	OR	7
8	,OR	10
-	PATIENT_NUMBER	-
-	THE	40
-	THERE	-
-	TRUE	-
-	VISIT	35
10	<relational op>	10
20	infix +	20
20	infix -	20
21	*	21
21	/	21
22	^	22
-	prefix +	20
-	prefix -	20

Function and Operator Precedence Table

figure 23