Computer Sciences Department
THE UNIVERSITY OF WISCONSIN
1210 West Dayton Street
Madison, Wisconsin 53706

LEARNING TO USE CONTEXTUAL PATTERNS IN LANGUAGE PROCESSING

by

Sara R. Jordan

Technical Report No. 152

March 1972

LEARNING TO USE CONTEXTUAL PATTERNS IN LANGUAGE PROCESSING

Sara Reynolds Jordan

Under the supervision of Professor Leonard Uhr

The research described in this thesis concerns the application of pattern recognition and learning to natural language processing. Using the techniques of learning and pattern recognition, a program has been written to learn and to demonstrate its knowledge of natural language by trying to transform language strings from one form to another, and to answer questions based on the information it has learned.

In a departure from the usual approach to question answering and other natural language processing, this program avoids using built-in linguistic or logical information or techniques. In addition, several types of language behavior are attempted in the same single program, including transformation or translation of language strings, information learning and organization, and question answering. Emphasis is given to this program's ability to learn a memory net structure and to categorize the nodes in it into general behavior classes.

All inputs are in the form of unstructured, unsegmented strings of natural language. In a general and uniform way, the program processes these strings and incorporates its knowledge into a net structure which acts as its permanent memory. Learned language units are interrelated and organized in the net by a general process of categorizing them into classes according to feedback as to "correct" usage received interactively from a human trainer. Weights are used for learning and unlearning relationships.

Using only the information which it has thus learned and represented in the memory net, the program accepts natural language input strings, processes each string according to the requested task, and outputs in natural language a response, which may be a) a translation or other transformation of the input string, or b) an answer to a question input by the human.

The purpose of this research is not to try to produce high quality language translation and question answering; rather, it is to experiment with a memory structure which, with the aid of a set of simple and general heuristics, demonstrates an interesting kind of learning for natural language manipulation.

The program is currently running interactively on the Univac 1108 (Exec 8) Timesharing System at the University of Wisconsin. Coded in Fortran V, the program includes a string-matching list processing language written by the author especially for this research.

[illegible]

Leonard Uhr

Professor in charge of thesis

ACKNOWLEDGEMENT

The research reported in this thesis evolved from some of the work of my advisor, Professor Leonard Uhr. I would like to express my deep appreciation for his helpful guidance and the many hours of discussion devoted to the problem.

In addition, I wish to acknowledge with gratitude the constructive criticisms and comments of my thesis readers, Professors Larry Travis and Frank Baker.

Finally, I am especially grateful to my husband Sam, whose helpful support and understanding made this effort possible.

This research was partially supported by NIH grant number GJ-583.

CONTENTS

CHAPTER 1

INTRODUCTION AND BACKGROUND

1. Introduction

Two basic interests - in learning and natural language processing - converged to produce the language learning program which is described in this thesis. Using the techniques of learning and pattern recognition, the program learns and demonstrates its knowledge of natural language by trying to transform it from one language to another and to answer questions based on the information it has learned.

In a departure from the usual approach to natural language processing, and to question answering in particular, the program makes a point of using no built-in linguistic or logical information or techniques. All its inputs are in the form of unstructured, unsegmented strings of natural language.

In a general and uniform way, the program processes these strings and incorporates its knowledge into a net structure which acts as its permanent memory. Learned language units are interrelated in the net organized by a general process of categorizing them into classes according to observed usage. Weights are used for learning and unlearning relationships. The program can then operate on other input strings by using pattern recognition techniques to enter the memory net and access the learned information in it in order to produce natural language responses.

Pattern recognition and learning techniques provide an interesting approach toward intelligent manipulation of language. Natural language, in its rich complexity, provides a challenging medium for experimenting with learning processes, and has the potential for stretching any learning technique to its limits. On the other hand, learning may be the only way to approach a fairly general language manipulator, one which will work with any language(s) on various tasks.

Our understanding of human language is still incomplete at best, and one cannot hope to program explicit grammar rules and some sort of dictionary for every language which might be seen. Consider the problem of a child learning to communicate with the people around him. He is not supplied with a precise grammar and lexicon package with which to face the world. Instead, he must slowly and haltingly learn to recognize the meaningful units and patterns in the (usually spoken) language that he perceives and to associate them with the objects and acts of his daily life. By observation and experience, he learns to use the patterns of the language with increasing sophistication until he achieves some sort of linguistic maturity, at least as a user of that particular language.

In much the same manner, the program to be described here learns to use the languages it sees. By analysis of its experience, it tries to learn the meaningful patterns of the languages and how to manipulate them correctly.

2. Major Approaches to "Understanding" Natural Language

Language learning might be viewed as learning to "understand" and manipulate language to produce some desired goal of behavior. The learning and understanding of language by computer can be demonstrated in different ways. We will briefly outline several different approaches.

First, the computer can demonstrate its understanding by accepting a natural language input and outputting a paraphrase of its content. This was the approach of Klein [1965], whose program accepted as input short essays and used dependency analysis to produce an essay-type paraphrase which summarized the content of the source text.

The computer can also indicate its knowledge of a language with the ability to translate it into another language. Because of the great complexity of natural language, however, good translation has been an insurmountable task. Few research efforts along these lines have even been attempted in recent years. However, Uhr [1964] described three programs which use pattern recognition in an attempt to translate from one natural language to another. These programs (from which the research described in this thesis evolved) will be described in Section 4.1. Along similar lines, Siklossy wrote a program which tries to translate from a functional language to a natural language; his work also will be discussed in more detail in Section 4.1.

Another interesting method of demonstrating understanding of a language is to relate an input to previous knowledge and discuss and explain its relationship. This is the approach of Quillian's Teachable Language Comprehender [1969] which operates with a semantic network memory capable of representing factual assertions about the world.

The most frequently attempted task which (incidentally and very convincingly) demonstrates knowledge of a language is to answer natural language questions. Question answering (QA) in general is a very large and active area of research, possibly because of the obvious usefulness of any successful large-scale program for handling large information files. There are many aspects of the question answering problem, with much research involved in each area, but our interest centers on the problems of understanding and manipulating natural language. We shall mention several QA efforts in Section 4.3.

3. The Approach Taken in the Current Research

Our approach to the natural language learning problem is different from the other attempts which have been reported in the literature, perhaps because our purpose is different. Although our research is similar in many respects to that of Uhr [1964] and Siklossy [1968], in the current program there is much more emphasis placed on the memory structure, and more types of behavior are attempted in the same program.

The program is given the ability to learn a memory net structure and to categorize the nodes (e.g., words) in it into general behavior classes according to the usage it infers it must make of them. Using only the information which it has learned and represented in the net, the program accepts natural language inputs and tries to perform the requested task, which may be a) to translate to another form or another language (and possibly also to learn the information in the string) or b) to answer a question. The translation or answer is then output in natural language.

Much as with the Uhr programs [1964], the language translation task is approached as a pattern recognition problem with contextual interactions of great complexity. Once the program has gained entry into the memory net (by means of pattern recognition), the traversals from node to node are strongly influenced by contextual constraints. Very simple, the translation or transformation problem is to find a combination of words which "covers" the input and then to transform those words to a target form suitable for output. Some transformations may involve changing the order of the segments as they appear in the string. The program must do all this without the aid of linguistic information about the words it is processing.

The approach to question answering is similar, but slightly more complex. In this case, the program is attempting to find a combination of facts which are related (in the network) and suitably account for all of the question. Since there is no pre-programmed linguistic information, the program cannot know the nature of the relatedness of the two facts, but must use the collection of facts assembled by various heuristics. Also, since there is no logical processing, the program cannot check to see if the correct answer can be logically deduced from the collection of facts. All it can do is output an answer based on the relationships it has learned, and try to correct its information if it discovers that it was wrong.

The purpose of this research is not primarily to try to produce high quality language translation and question answering; rather, it is to experiment with a memory structure which, with the aid of a set of simple and general heuristics, demonstrates an interesting kind of learning for natural language manipulation.

4. Previous Relevant Research

In this section we will survey research in three different areas which are most relevant to the work described in this thesis. The research to be discussed involves language learning, memory networks, and question answering.

4.1 Language learning programs

Research in this area is probably the most relevant in all of artificial intelligence to our research. The first learned behavior sought by this program was an understanding of natural language, as indicated by the ability to translate it into some other form for output. A few other programs have attempted to learn or translate natural language.

4.1.1 Siklossy's ZBIE

The research of Siklossy [1968] is based on much the same idea as the books which propose to teach a language through pictures. By inspecting a picture of a simple situation and studying its description in a foreign language, the student is supposed to learn that foreign language directly from the "real world". Thus bypassing his native language, the student is supposed to avoid the problems of translating from one natural language to another.

In place of the (allegedly) universally understood pictures, Siklossy's program (called ZBIE) represents situations with a functional language in which the "sentences" are usually tree structures. By requiring such a structured functional expression as input for translation to a natural language, ZBIE avoids many of the problems of learning or translating from a natural language which is input as an unstructured string.

ZBIE compares the functional language representation of a situation with the representation of the situation in some natural language, usually Russian. By successive comparisons of situations represented in these two languages (one strongly noninflected, one natural), ZBIE tries to learn how to express other situations in the natural language. Then ZBIE can indicate its learned knowledge of a language by outputting a description of some new situation in that language.

Although ZBIE is an interesting and fairly successful program, it avoids many of the problems which our program tries to solve. Whereas our program accepts unsegmented, unstructured strings in one natural language (L1) for translation into another natural language (L2), ZBIE starts in the middle and accepts for translation only presegmented and structured strings in the special functional language, thus avoiding the problems of L1's idiosyncrasies.

4.1.2 The Autoling system

The Autoling system of Klein's group [Klein _et al_, 1968] learns a language in the sense that it automatically produces a phrase structure grammar of (a subset of) some natural language during teletype interaction with a live informant in that language (the informant must also know English). The phrase structure heuristic learning program accepts as inputs sentences written with spaces between morphological units. The program tries to parse the sentence according to the grammar it has built so far. If no complete parses are found, then Autoling tries to adapt its current provisional grammar so that the sentence will parse correctly. The top nodes of any incomplete parses which were found for the sentence are ordered and used as input for a set of rule building heuristics. Using these heuristics, Autoling tries to modify a relevant grammar rule so that it applies also to the selected top nodes and thus gives a complete parse. Before the provisional grammar is actually changed, however, the program uses the altered portion of the grammar to generate a test sentence for the informant. If the informant approves of the test sentence, then the grammar is actually changed; otherwise, the test sentence is remembered as "illegal" and Autoling tries another heuristic. If all the heuristics fail to satisfactorily adapt existing rules of the provisional grammar, then the top nodes of the best incomplete parse are added as a new rule.

Our present research is similar in many respects to Autoling, especially the techniques of finding the most relevant rule (or node) in the memory and modifying it so as to generalize its usage to the current input case. Although our program can have no explicit grammar, the program is, in effect, "parsing" an input string as it processes it through the implicit grammar of the memory net.

One of the goals of Autoling is to replace the linguistic fieldworker. The program does quickly produce a grammar for a given sample corpus, but the quality and completeness of the grammar do not yet compare with that of the grammars produced by human linguists.

4.1.3 Uhr's pattern string learning programs

The series of three program described by Uhr [1964] form the basis from which the present research evolved. Each program experiences a set of string inputs in some natural language L1 and uses its memory to recognize segments of the strings and translate them into strings of some other language L2. Recognized segments are associated with each other in the memory graph by one-directional learning rules. Classes of segments can also be generated and used to write "restructuring rules" in the memory graph. The structure of the memory is not emphasized, but can be thought of as a graph defined implicitly by the "learning rules", in which pattern nodes are connected by several types of edges representing different relationships.

The second and third programs must learn to recognize segmental patterns in continuous strings that may not have any indication as to where the segmentation should occur. The first program, however, uses rote learning techniques on strings already segmented by spaces. It can learn to recognize and translate any sentence generated from its vocabulary, as long as there are no interactions between words in the vocabulary.

The second program can handle simple interactions between words by putting them into equivalence classes and generalizing a rule learned about a specific pair of words to the class as a whole. Thus it can build up simple combinations of patterns and can classify patterns for purposes of deciding between ambiguous alternates and changing the order of patterns in the string.

The third (projected) program attempts to strengthen the rules of the second program by using as context not just a single word but a group of words. Thus, it combines classes into classword patterns, handles much more complex classes as contexts, and handles discontinuous interactions more efficiently.

The program reported in this paper uses a general memory network and learns all the behavior mentioned above, with certain added abilities, such as unlearning incorrect information. In addition, the new task of question answering is attempted.

4.2 Memory networks

An important element of the research to be described in this paper is its memory structure and the techniques by which the program acquires its knowledge and represents it in a learned semantic memory network. We are interested not in just a data structure suitable for retrieving information, but in the relationships inherent in natural language itself. Two such memory structures will be described below. Although they are not learned structures, both systems have interesting properties which are relevant to this research.

4.2.1 Quillian's semantic memory

After a well-developed background discussion of semantic memory, Quillian [1968] describes his own semantic memory model. The model consists basically of a mass of nodes interconnected by different kinds of associative links. Although each node can be thought of as named by some English word, the important feature of the model is the distinction between the two different ways that a node can be related to the true meaning of its name word. A _type node_ relates directly to the configuration of other nodes that represent the meaning of its name word. A _token node_, on the other hand, refers indirectly to its name word concept by having a certain kind of associative link that points to the name word's type node. For any word concept, there is exactly one type node in memory; the nodes which define the concept of this type node are token nodes, each of which points to its own unique type node. In general, there will be many token nodes pointing back to each type node. Note that no word concept can be thought of as a "primitive" in the system, each is defined in terms of the others.

The node configuration defining a type node's concept can be thought of as a _plane_ in memory. As opposed to this "immediate definition" of a word, Quillian defines a _full word concept_ in his model to be all the nodes which can be reached by exhaustively tracing out through all links from the initial type node of the word, together with all the relationships among these nodes specified by local, within-plane links.

Quillian holds that such a memory organization is useful for semantic tasks and as a reasonable description of the general organization of human semantic memories.

The memory structure described in Chapter 3 is similar in many respects to Quillian's semantic memory; a comparison of the two memories is made in Chapter 3, Section 3.

4.2.2 Lamb's linguistic and cognitive network

According to Lamb [1969], the types of relations and the configurations of relations which have been observed in linguistic data are also present in what he calls "cognitional data", that is, relationships and facts about the real world. To demonstrate this, he describes a single, stratificational network of relationships which accommodates both linguistic and cognitional data.

Starting at the level of the basic phonological components of sound (e.g., labial, closed, voiced) which he calls phonons, Lamb shows how different layers of sign patterns (e.g., at phonemic, morphemic, lexemic, or sememic levels) are connected into one complex network. Crossing the vertical connections of the strata are "planes" of syntactic patterns (e.g., lexotactics, semotactics, morphotactics) which choose among alternate realizations of different network elements and determine the ways in which the elements can combine. A tactic pattern is shown to act as a filter in decoding a sentence, filtering out tactic anomalies and resolving ambiguities to the extent possible at the particular tactic level.

Residing at the top level of a complete linguistic system are the elements for concepts. Lamb argues that if we inspect cognitive data in the same manner as linguistic data, we will find that cognitive data also fall into a network of relationships, starting from simple concepts. Thus, relational cognitive data can be viewed as another stratum lying on top of the linguistic network. Example relational segments are given for the game of baseball and animal taxonomy. Lamb contends that the resulting network can be viewed as a model for (at least some of) the knowledge that a human has stored in his brain.

The memory described in Chapter 3 shares certain characteristics with Lamb's network, since it stores its learned word "lexicon" and factual information in the same single net structure. Further comparison will be made in Chapter 3, Section 3.

4.3 Question answering programs

The second task chosen to be learned by the program was question answering. Although the problem is attacked in a way very different from most efforts, a few question answering (QA) systems will be briefly discussed with respect to the use of natural language and the generality of the memory structure.

One of the main difficulties in QA systems is how to deal with the complexity of natural language. Some research projects have simply bypassed the language problem in order to concentrate on the logical processing of information or the internal structure of the data base. For example, the SAMENLAQ II system [Shapiro and Woodmansee, 1969] accepted as inputs only relational triples of the form xRy so that the authors could concentrate on developing a powerful data structure. In some early cases a QA program would accept natural language input, but only in the format of a small subset of English sentences from which key words and relations could be easily picked out. Examples are the SIR [Raphael, 1964] and STUDENT [Bobrow, 1964] systems.

The Protosynthex I system [Simmons, Klein, and McConlogue, 1964] accepted natural language questions and used the content words in them to index into a corpus of English text and extract information-rich sentences from it. Our program uses a technique very similar to this (see Chapter 9). The extracted sentences were then dependency analyzed and put through semantic evaluation before being chosen as answers. The current program does not use these linguistic procedures, however.

More recent attempts to process natural language questions all seem to have made further use of linguistic information, with built-in grammars for processing inputs, aided by lexicons giving linguistic word properties. Examples are the DEACON system [Thompson, 1966; Craig _et al_, 1966], the CONVERSE system [Kellogg, 1968] and Protosynthex III [Simmons _et al_, 1968]. For these systems, a natural language input is parsed by filtering it through coordinated syntactic and semantic processing to some internal form which can access the data base.

The current program also works with natural language inputs, data base and output, but contrary to the above mentioned efforts, it tries to do it all without built-in linguistic information procedures.

Most early question answering systems achieved success by operating on data bases structured specifically for their content. Systems like BASEBALL [Green <u>et al</u>, 1963], SIR [Raphael, 1964], and Lindsay's kinship-relations program [Lindsay, 1963] worked only within a narrow range of subject matter.

Later efforts have been directed at building systems which are useful for data bases with different structures or contents. For example, the DEACON system could handle any subject matter which could be stored in its interrelated ring structures. The CONVERSE system was designed for use with large, formatted information files concerning any suitably organized subject matter. Then the ability to answer a question would depend on how closely it matches the internal structure of the data, and whether the desired information is accessible from that direction.

The question answering system designed by Woods [1967, 1968] achieves independence from the data structures, subject matter, or manner of data retrieval by operating at a higher level in terms of primitive functions and predicates. The user of the system would have to provide the primitives appropriate to his data base. In contrast to this approach, the SAMENLAQ II system system of Shapiro and Woodmansee [1969] tries to achieve maximum generality and question answering power in the data structure itself. The memory used is a net structure with nodes related by labeled, directed edges. The relations used as labels are also nodes themselves, so that the memory can store and use information about its relations as well as the items related.

The memory of our current program is somewhat similar to that of SAMENLAQ II, in that both are node networks connected by labeled, directed edges. In contrast to SAMENLAQ, however, our program has no way to tell which portion of an input information string might be a relation, so all string segments are stored in the same way (see Chapter 8).

5. Summary

The goal of the present research is to learn to transform natural language strings and answer questions, using only learning techniques, the natural language strings of the program's experience, and a memory network which is independent of subject matter. Built-in information and special techniques from logic and linguistics are expressly disallowed. As its main technique, the program is allowed to categorize the nodes of its memory into behavior classes according to the language usage which it observes. Although the methods used are different from those of any system mentioned above, we feel that the resulting behavior provides an interesting study of learning and the relations inherent in natural language.

CHAPTER 2

OVERVIEW

1. General Overview of Program

In this thesis we describe and outline the operation of a large, interactive computer program, which we will refer to as METQA, (for MEchanical Translator and Question Answerer). Running interactively on the computer, METQA accepts intput strings of natural language from a human trainer and, after processing each string, outputs a natural language response. The processing of the string may involve transforming (or translating) it to some other form in the same or another language, or it may involve answering an input question based on information previously learned by the program.

The program can operate in any of three modes at the command of the human trainer. The first mode is the transformation mode, in which METQA acquires vocabulary and uses learned vocabulary to transform strings to some other form or language. While METQA is still in this mode, the human trainer may optionally turn on another mode, which tells METQA to learn the factual information contained in the input strings as well as their transformations. We call this the learning mode. In the third mode of operation, the question answering mode, the input string is marked as a question asked by the human, and METQA uses the information learned into its memory to try to produce an answer.

METQA decides what to learn by comparing the response it has output with any (specially marked) feedback string the trainer may wish to give. A feedback string is taken to be the <u>correct</u> response which METQA should have given to the original input from the trainer. By comparing the feedback with the response, METQA tries to determine which (if any) portions of the original input string were processed incorrectly so that memory modifications can be learned. Learned information and relationships are represented and stored in a memory network which serves as the permanent memory of METQA.

METQA learns by adjusting and reweighting the links which connect the nodes of its memory. New relationships are hypothesized and built into the net with some initial, neutral evaluation weight on each link. The hypotheses are tested by using them to direct future behavior; the links are reweighted according to the success of the behavior. If some hypothesis is downweighted to a certain minimum because of bad behavior, then it is discarded or forgotten.

By thus learning and adapting its memory net from its experience over time, METQA tries to produce a memory which yields intelligent manipulation of language.

2. Program Specifications

METQA is currently running interactively on the Univac 1108 (Exec 8) Timesharing System. The program is written in Fortran V and is about 7000 lines long, including the string-matching list processing language (called SMALL) which was written by the author especially for this program (see Appendix for explanation of SMALL). The system consists of a large driver program commanding about 80 subroutines. In the computer, METQA occupies about 31,000 words of core plus an available space array, which has ranged from one to five thousand words.

The available space array is set up as a doubly-linked list of 2-word cells, in which two available space mechanisms operate, one from each end. METQA's permanent memory is constructed in the lower end of the array, while the upper end is used as an erasable work space by the program (see Appendix for more details on available space).

Except for the distinction between memory space (which holds METQA's memory net) and work space (which temporarily holds the program's work lists), no knolwedge of the SMALL system is required of the reader.

3. Conditions Set for the Program

The main objective of the present research is to see how well an adaptive program can produce "intelligent" manipulation of language without any prior knowledge of it. The program has the abilities to form a general relational net structure and to form classes or categories of the nodes in this net for use in distinguishing different behavior situations. Given only a net structure and the general procedure of categorizing its nodes in order to determine behavior, just how well can METQA learn to handle language tasks such as translation and answering questions?

With the above question in mind, we set certain important conditions for the program. Both input and output information should be minimized: There are no large files for dictionaries or grammars, which would be limited in use to just the language for which they were written. There are no built-in encyclopedias or fact files. METQA tries to learn everything on its own (with rare exceptions to be discussed in Chapter 9, Section 9).

Another important goal was to maximize the generality of the program so that it could adapt easily to different tasks. The program should handle any language (or potentially any form of communication) and any subject matter by trying to acquire the appropriate "world view". It must adapt its memory to reflect whatever experience it has had.

4. Behavior Goals for the Program

Within the above framework, METQA must attempt its various tasks. First, it tries to learn to transform some input string (called the source string) to some other desired form (called the target string) as directed by a human trainer. This might involve translation from one natural language to another, or transformation from active to passive voice within a language, or converting a (digitized) spoken input to a written output -- in brief, transformation between any forms which can be suitably input. METQA's only requirement for its input is that it be a string of discrete symbols.*

In order to correctly transform input strings, the program must learn the different external forms of the same memory node which comply with different contextual situations. For example, this might involve learning a feminine form of some adjective when in the context of feminine words, or perhaps learning a special plural form of a noun when in a plural context. METQA must also learn any segmental permutations necessary for correct transformation between different representations. Such permutations typically occur because of word order differences between languages (e.g., adjective and noun transposition between English and French).

In addition to transformation of strings, METQA must also try to learn the information contained in the strings it processes, so that it can answer questions related to this information. Considering the information retrieval continuum, with document retrieval on one end and sensitive, specific answers to complex questions at the other end, it is of course desirable to come as near as possible to true question answering. METQA is limited here by the fact that it has no knowledge of linguistic structures or word properties, and cannot definitely determine the relationships among a particular set of words.

*This requirement does not preclude two-dimensional pictures or a sound spectrum, however. It requires an initial processing through a two-dimensional pattern recognizer or spectral analyzer, with output of a string of recognized units to be used by METQA as its unknown input.

Consequently, as a beginning, METQA simply tries to output a small set of one or more related facts which include a question's answer, at least implicitly.

METQA is in part a pattern recognition system, too. Thus, it must accept imperfect input strings, allowing for misspelled words or garbled or missing characters and trying to do as well as it can. With the general pattern recognition techniques discussed in Chapter 4, METQA tries to recover the missing information by inspection of the context, much as would a human.

Given all these goals and requirements for operation, how does METQA process any given input string? The procedures involved will be briefly indicated in this chapter, and discussed in much more detail in succeeding chapters.

5. The Learning Technique Used

Perhaps the most important feature of this program is its use of learning. There is no direct attempt made to simulate the human mind, either in the form of semantic memory network used or in the manner in which METQA learns. Instead, we are interested in a system which can represent and store semantic information in a simple and general way, and which can adapt itself, or <u>learn</u>, based on its experience.

As will be explained in Chapter 3, the memory consists of a network of nodes connected by labeled, directed links. A link may have an associated list of restrictions or conditions which must be satisfied before that link can be traversed. Because of METQA's operation as an adaptive system, each link and each restriction on a link has an associated weight that both serves to choose paths and aids in evaluation of its usefulness. At the surface of the memory net, the description of a word as a character string may be composed of several short string segments; here, each segment has an associated weight to evaluate the segment's importance relative to the whole word description.

Thus, for example, the string for "telephone" could be segmented as "TELE(2)-PHONE(8)", where the segment weights are shown in parentheses. Here the weights show that METQA has learned that the second segment is much more important for recognition than the first.

An evaluation weight is a small integer which can be changed by the program to reflect the success of using the weight's associated link, link restriction, or string segment (the readjustment operation is the same for all).

The adjustment of evaluation weights is carried out as follows. If a certain link was traversed and led to an incorrect response by the program, then that link is penalized; its evaluation weight will be reduced by some small amount (we call this downweighting). If METQA traversed this bad link because the input satisfied some restriction on that link, then the evaluation weight of that restriction is also reduced. On the other hand, if METQA needs to reinforce the knowledge of some link which leads to correct behavior, then that link's evaluation weight will be increased (we will refer to this as upweighting the link).

Suppose some link (or restriction) has repeatedly led to bad behavior, and its evaluation weight has been reduced to zero. Then METQA will decide that it was incorrectly learned, and the link (or restriction) will be erased -- completely removed from the memory. This downweighting and discarding is also known as "forgetting".

This weight adjustment technique allows METQA to evaluate what it has learned before, based on the success of its later behavior. All hypotheses are given the same preset neutral value for their initial evaluation weight. If the initial hypothesis was incorrect, then repeated downweighting will ultimately cause that learned item to be unlearned, i.e. erased from memory. A correct hypothesis, however, will produce correct behavior and will be upweighted often enough so that there is no danger of its being "forgotten". Other programs which have successfully used this weighting technique include the pattern recognition systems of Uhr and Vossler [1963] and of Uhr and Jordan [1969].

For this reason, it is quite reasonable for METQA to generate a whole parallel set of multiple hypotheses to explain some experience. Since there is often no way to determine which of several alternate hypotheses is the best, METQA learns all of them and then depends on its later experience to reweight and thus unlearn the bad hypotheses and reinforce the good ones.

This, then, is the general basis of the learning process which will be referred to throughout this paper. After analyzing the current input string from the perspective of its accumulated experience, METQA hypothesizes one or more relationships and incorporates them into the memory. Each hypothesis receives a neutral initial evaluation weight. The learned relationships are tested by using them to direct the behavior on future experience. Then according to the success of its behavior, each hypothesis will be modified, reweighted, or even "forgotten". By this process, the correct and useful hypotheses should survive and the erroneous ones should be discarded.

6. Learning from Labeled Input Pairs

In order to decide what relationships to learn by the above technique, the program makes careful comparisons of pairs of input strings. The human trainer provides labeled input strings for this comparison.

An unmarked input (i.e. with no preceding special symbol) such as "THEDOG"* directs METQA to simply transform that source string through the memory toward some target string which is the translation of the source string. As its response, METQA outputs either the completely translated target string (e.g. "LE CHIEN"), or else a partially translated string, along with an indication of any part of the source string that it could not recognize (e.g. "LE U(DOG)", where "U" means "unknown").

*All input strings are followed by a special termination symbol ("<") which delimits the string for METQA.

The next input received might be another string to be translated, but the trainer may instead wish to give METQA feedback to its response. If so, the string is marked with a special symbol ("=") at its beginning (e.g. "=LECHIEN"), so that METQA will compare this string with the previous one in order to learn transformations (as described in Chapter 6).

Alternatively, any input might be marked as a question to be answered by the program (see Chapter 9) or as a special "metacommand" to change some parameter value in the program (see Chapter 10, Section 4). However, only a feedback string can "connect" with the previous input and initiate METQA's learning procedures.

7. Use of Context Classes

Another very important feature of the program is its use of context classes or categories. METQA groups together into a class or category any one or more nodes whose usage it would like to characterize in a general way. Thus if the program discovers some special behavior pattern in the context of (i.e., in the input string close to) the word "fille", then it will try to characterize this context (or input situation) by forming a class with the word "fille".

METQA could just learn to behave in a certain way in the context of "fille", but such a learned rule is entirely specific and would require learning the special behavior separately for each word which causes that behavior. It is far more efficient to form a general class, say C1, whose first member is the word "fille", and thus to learn a more general rule: to behave that certain way in the context of Cl (i.e. words like "fille"). Then whenever METQA observes the same behavior pattern in the context of some other word W, it can conclude that W is another one of those words like "fille". Thus W would be added to the class Cl as a new member, and the more general rule mentioned above would still apply, but now to a larger class of words.

7.1 Formation of context classes to disambiguate

Context classes are built for two main reasons. The first is for disambiguation -- to enable METQA to choose from among alternate behavior possibilities. For example, suppose METQA knows that "le" is the French translation of "the". Then on some feedback, it discovers that "the" should have been translated to "la" in the present case. This means that there is an ambiguity; "the" can transform to "le" or "la".

In order to determine in the future which alternate should be chosen, METQA tries to find the distinguishing characteristics of the current input. That is, what caused this input to require a different behavior from that which METQA already knew? To answer this question, METQA forms a class for each word near the ambiguous word in the input (two words on each side). Hopefully, the trainer has learned to be sensitive enough to the program so that at least one of these classes contains the necessary distinguishing context. (The other, false inferences will gradually be unlearned as distinguishing contexts, as described in Section 3.)

These new classes are listed together as a context class condition on the link to the new alternate behavior ("1a"). This context class condition acts both as a restriction and a selector. Use of this alternate behavior form ("1a") is restricted to cases where a member of at least one of the context classes listed is present nearby in the input. On the other hand, the satisfaction by the input of that desired contextual situation causes this restricted link to be <u>selected</u> over the other alternate behavior ("1e"). (This use of context classes will be discussed again in Chapter 5, Section 5.)

<u>7.2 Formation of context classes for permutation rules</u>

Context classes are also built to handle the description of input situations where segmental permutations have been discovered to occur. If METQA discovers a difference in word order between input and feedback strings, then it will try to learn a rule governing the permutation of parts so that the string can be output correctly next time. That is, it will try to characterize (by means of context classes) each segment involved in the permutation and indicate the position which that segment should have after the permutation. (Permutation rules are discussed in Chapter 7.)

For example, suppose METQA receives the input string "BROWNDOG" and outputs as its response "BRUN CHIEN". Next the feedback string "CHIENBRUN" is received, and METQA discovers that the two segments should have been permuted. Therefore METQA will try to use contexts to describe this input in order to build a permutation rule.

Context classes will be formed to contain "brun" and "chien"; then these (internal) class names will be used in the following rule (let "M/C(word)" be read as "some member of the class containing 'word'"):

> If M/C(brun) is immediately followed by M/C(chien), then permute the segments so that the second segment comes first (i.e., switch the two segments around).

Since METQA operates from any source language to any target language, a complementary rule would be built, from output to input language, for M/C(dog) followed by M/C(brown).

As explained in Chapter 7, a permutation rule is altered and generalized in further experience so that, hopefully, its context classes will approach the actual linguistic classes involved (e.g. French adjectives, French nouns).

8. Generalization of Context Classes

When METQA needs to characterize an input string for description or disambiguation purposes, it uses already existing context classes wherever possible. We refer to this as generalization of learning. In this way, the total number of classes is minimized, and the classes become more general in usage.

Generalization occurs in two types of situations. In the first case, METQA generalizes some class membership to a new node because it behaves like a member of that class. For example, suppose METQA knows that "the" transforms to either "le" or (in the context of class C1, which at the moment contains only "fille") "la". Then it would transform "THEWOMEN" to "LE FEMME". When feedback indicates that "the" should transform to "la" in this case also, METQA must adjust the memory so that a context of "femme" also elicits "la".

Instead of forming a new class to contain "femme", METQA generalizes membership in class C1 to "femme" for the following reason. If the link restricted to C1 should have been followed, that is if "femme" should have caused the same behavior as a member of class C1, then perhaps "femme" should be a member of C1. As a result, class C1 now has at least two members ("fille" and "femme"), and METQA is on its way to forming a class of feminine words. Note that the condition for transforming from "the" to "la" is still the same.

In the second type of generalization, METQA generalizes a new context condition for the usage of some word form (that is, for the traversal of some link) from some already classified word which (feedback says) should elicit that word form. This type of generalization changes not the class membership of a node, but the conditions for traversal of some link.

For example, suppose METQA knows both class C1 from the previous example and that "small" transforms to "petit". Then an input of "SMALLGIRL" will be transformed to "PETIT FILLE", but the feedback string will be "PETITEFILLE". After comparison of the strings, one of METQA's hypotheses is that a new form "petite" occurs in the context of "fille". (For more details see Chapter 6, Section 5.2.)

Hence a new node for "petite" is built and connected to the other "small" nodes, but its use must be restricted to this contextual situation. As we have seen, "fille" has already been assigned a class membership (in C1) for other reasons, so METQA will generalize from that class membership and hypothesize that class C1 must be the distinguishing context which should here elicit the form "petite". Hence the link to "petite" will be restricted to contexts of class C1, and the class membership of "fille" is unchanged.

Notice the result of this generalization. Even though METQA has not seen the input "SMALLWOMAN" before, it will now output the feminine form of "small" in its response "PETITE FEMME" because "femme" is now a member of class C1. Thus as a result of learning a new form (or a new usage of an old form) in the context of one classified word, that learned behavior will generalize across the whole class.

Of course, it is possible to generalize too much. For example, if "fille" had membership only in the class of French nouns, then METQA would have learned that "small" translates to "petite" in the context of all French nouns, which is obviously false. If such an erroneous generalization does occur, METQA has to unlearn it through experience, downweighting and ultimately discarding the bad context restriction. Meanwhile, it will extract other class memberships from other words, or generate new ones if necessary, and eventually arrive at a better set of context restrictions. (Examples and discussion of generalization will be found in Chapters 6 and 7.)

9. Uses of Context Classes for Question Answering

Besides the major uses of context classes for disambiguation and description of contexts, there are some convenient "side effects" for question answering as a result of the categorization of METQA's vocabulary.

In Chapter 3 (Section 2) and Chapter 9 (Section 4) we explain the concept of an extended meaning, which is very useful in extracting from the net a richer selection of information about some word than just those facts that the word alone could. Briefly, the extended meaning of some word W includes not only W, but all words which are class members of W and all words of which W is a class member. Thus the extended meaning of "fruit" might include "orange", "peach", "food" and "plant", as well as "fruit". Hence any context classes which METQA forms can cause its knowledge (or literally, the nodes in its net) to become more interrelated and thus more accessible from different directions.

Another important use of word categories is in the processing of interrogative or "question" words (such as "who", "what", "when"), which appear in questions to be answered by METQA. Using the class memberships known for the question word itself, METQA searches for some word which has one of the same class memberships to use as the referent of the question word. Thus, if METQA has a fairly well developed context class structure, the referent for the question word "who" might be "Mary" or "boy" or "aunt" (perhaps because all are members of the class of persons), but it could not be "table" or "small".

10. Preview of Coming Chapters

The next chapter will discuss METQA's semantic memory net structure and explain the various types of links and nodes in it. In Chapter 4, the pattern recognition technique used to recognize character strings is explained and illustrated. Several methods for learning string patterns are discussed, along with the problem of recognizing relevant information.

Chapters 5 through 9 explain the actual operation of the program for its tasks of translation and question answering. In Chapter 5, the transformation process is examined. Word patterns are recognized in the source string and the words are combined to form "parses" of the string. Then each recognized word follows some path through the nodes of the memory net (as allowed by contextual requirements) toward the desired target string. Appropriate transformations and segmental permutations occur along the way.

The learning of transformations is explained in Chapter 6. Assume some source input string has been processed to its target string and its word paths are still threaded through the memory network. A feedback string is processed in similar fashion, but the goal of each feedback word path is to quickly join or intersect a path from the source input. The reason for this is that METQA wants to know if it was correct in its processing of the source string. If it was, then the source input and feedback paths should overlap immediately. It is these intersected paths that METQA uses to associate and learn string transformations.

Since there are often differences in word order between two languages or between two equivalent strings in the same language, there must be rules to describe the conditions under which string segments should be permuted. In Chapter 7 we discuss the learning and use of these permutation rules, with attention to how METQA can generalize the rules.

Next, we turn to METQA's task of answering questions. Chapter 8 explains how information is learned and placed into the net as "facts". Then in Chapter 9 we examine the process by which METQA finds an answer to a question. Using the words found in a question, METQA selects an initial set of facts from the memory net. The initial facts suggest other facts, and these in turn suggest other information, and so on. The facts are linked together into "chains" of information which METQA assembles in an attempt to "account for" (with information) each word in the question. Next comes an explanation of the special treatment of interrogative words, followed by examples showing the use of these techniques.

Various topics are covered in Chapter 10. Special problems are discussed, along with proposed remedies. In particular, we discuss the problems which arise when METQA is transforming in only one language and in more than two languages. Also examined is the issue of "metalanguage"--commands directed at METQA and not at its memory net. Such commands are used to change program parameters and switches; if it is advisable, they could also be used to modify the memory itself. Among the future extensions discussed are the treatment of pronominal reference, special types of questions, and question answering feedback.

Finally, the results of the work so far will be shown in Chapter 11, with sample runs illustrating how METQA learns.

11. Comments About the Examples

In order to illustrate METQA's procedures, many examples are given, with some showing small segments of a memory net. The reader should note that, although only a few nodes may be shown, the sample net may be just a small part of a very large memory net. Usually, only the nodes relevant to the example are shown.

The translation examples are mostly between English and French, with a little German here and there. Although METQA is programmed to handle any language, in its current state it gets confused about the target direction when working with more than two languages at the same time. This confusion may be only temporary (until it has learned the necessary context classes), but we found it more convenient to use only two languages at a time and to concentrate on the problems of learning and generalization.

12. Present State of the Program

It should be emphasized that this thesis is a report on a *continuing* research project. The algorithms for the entire program are described in detail; they have been completely programmed and are now running on the computer. However, the question answering routines have not been tested out and debugged. The translation routines are operating as shown in Chapter 11, but both parts of the program are quite large and complex and hence will need more thorough testing and exploration as the work with METQA progresses.

CHAPTER 3

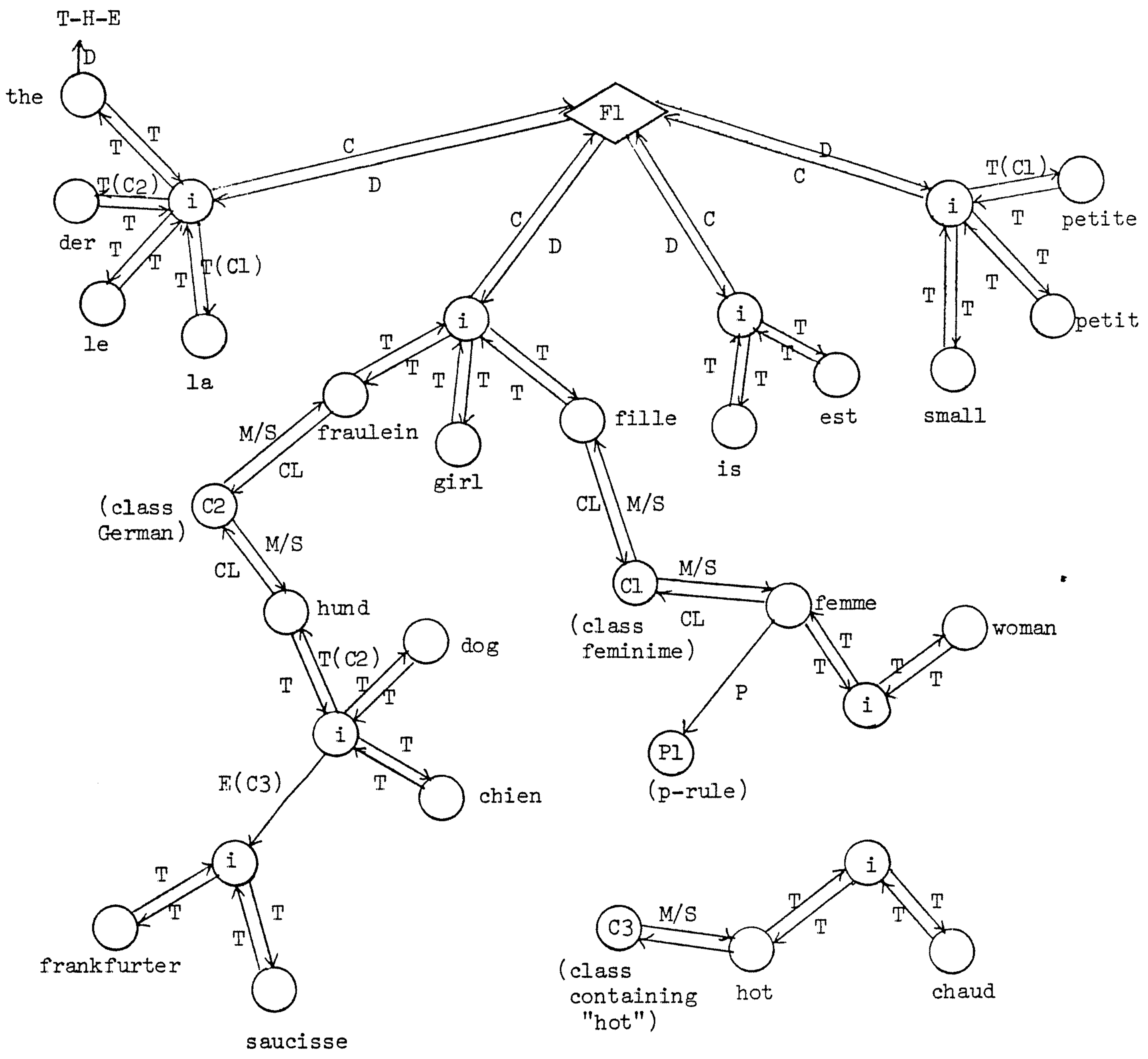
THE MEMORY STRUCTURE

1. General Description of the Memory Net and Its Links

In order to achieve as much generality as possible, the only built-in structures given to the program are the capacity to build a node network connected with an expandable set of labeled, directed links, and the general procedure of categorizing memory nodes into classes according to their behavior and usage. All the permanent knowledge gained by METQA's experience is represented in this one semantic memory structure. The permanent memory net (and its net directory list containing the addresses of all surface nodes) should not be confused with the entire available space list, which holds all METQA's temporary, erasable work list space, as well as the memory net (see Chapter 2, Section 2).

Each "node" in the memory network is a list representing a state of some word or phrase during its processing. The nodes are connected by directed links that represent various basic relationships indicating class structures, transformation possibilities, how a word has been used, and so on. A sample memory net segment is shown in Figure 3-1.

Perhaps the most common link used represents the transform relationship. Transform links connect nodes representing the same word or idea, but in different states or external forms. For example, the memory shows that the English word "dog" can be transformed



Link relationships shown:

T - transforms to
C - combines to form fact
M/S - member/subset of this class
CL - class membership in
E - equivalence; can be replaced by
D - description
P - permutation rules suggested by use of this node

Figure 3-1. Sample Memory Net

into the French word "chien" or the German word "hund". That is, by following just transform links one can "translate" one of the above words to either of the other two by passing through the central node representing the internal idea of "dog". Unlike other relations, transform links between nodes usually appear in pairs, allowing traversal in either direction.

Combination links are used to connect the internal nodes of various words to one node that represents a fact. _Description_ links are used to connect fact nodes to their component words and to connect each word node to a list containing the string description (that is, the literal character string) of the word involved. _Class_ links connect any word node to nodes representing the classes or categories which METQA has formed including that word. (The terms _class_ and _category_ will be used interchangeably throughout this paper.) Similarly, a class node is related to all its member or subset nodes by _membership_/_subset_ links. A _permutation_ rule link relates a node to each permutation rule (represented as a node list) which might apply to situations including this node (see Chapter 7). Another type of link (_equivalence_) connects a node to some other, different node by which it can be replaced in usage (a synonym, for example). See Figure 3-1 for examples of these links.

Any link (except a description link) can be modified, or _restricted_, by a context requirement which limits use of this link to certain situations. A context requirement list can have several options, at least one of which must be satisfied for traversal of that link. If METQA can find, nearby in the input string, words whose class memberships satisfy all the context class requirements of some option, then that link can be followed.

Thus for example, context requirements can prohibit transforming to the French word "la" unless the input has nearby a member of some usage class, say class C1, containing feminine words (see Figure 3-1). In this manner, the program can learn to choose among multiple links at some node by discovering the disambiguating contextual word classes which typically appear for each situation. This process seems similar to the one by which a human, on perceiving the word "dog", would only in the context of "hot" think of food.

Each "node" of the memory could be thought of as an attribute-value list whose attributes are some selection from the set of relations. The values are then pointers to other nodes having the attribute's relation to the original node. Most of the attributes (again excluding the description relation) can have more than one value, that is, more than one pointer to other nodes or lists. Of course this is necessary, since any word can combine to form more than one fact, or can have any number of class memberships, and so on.

A node in the semantic memory does not have a particular identity or name as such, although conceptually a node can often be thought of as a given word or idea.* Actually, every node is just

*Throughout this paper we shall name or refer to terminal nodes by using the lower case word which is represented in the node's description string. That is, "DOG" represents the string of characters D,O,G; "dog" represents the terminal node which can be entered by matching the string "DOG".

a collection of pointers which show certain relationships to other nodes and which indicate the usage of the node in METQA's experience. Thus any node, say for a word or phrase, is defined not explicitly, as in a dictionary entry, but by its usage in the information set; that is, it is defined by the links which can be followed from it to other word, phrase or class nodes.

2. Different Types of Nodes

Again using Figure 3-1, let us look briefly at the different types of nodes which appear in the memory net. Terminal nodes, or surface nodes, are the only nodes through which one can enter or leave the memory net. Each terminal node has a description which contains the external, character representation of the particular word; this character string must be matched (by pattern recognition techniques described in Chapter 4) before one can "enter" the node. Transform link(s) connect it to the internal memory nodes associated with that word. No terminal node has combination links to form facts, however; only interior nodes may combine. In Figure 3-1, only the terminal node for "the" shows its description string.

An idea node is the internal node providing the common link between the various external representations of the "same" word. It has transform links to all the external forms it can assume. For example, the internal idea of a domesticated canine might be represented by a node which can transform to the nodes for "dog", "chien", or "hund" (if those are the forms which METQA has experienced and learned). In addition, an idea node has combination links to all the known facts which use that word or idea. Thus terminal nodes link to idea nodes, and idea nodes are the entry points to the information area of the net -- the facts which are used to answer questions.

A <u>fact</u> node is described by the string of idea nodes whose words would externally state that fact. Any fact may in turn have combination links up to some larger fact whose representation includes it. In general, a fact node does not have any transformation links; to "translate" the fact to some external form, METQA must go down (description links) to the idea nodes of the component words and follow transform links from there. Thus a piece of information learned in some language is always accessible by way of any language METQA learns.

A <u>p-rule node</u> (permutation rule) is a list describing (by means of context classes) a situation in which string segments should be permuted; see Chapter 7 for an explanation of p-rules.

Virtually any node in memory can have class links to nodes representing any usage classes in which METQA has placed that node. Also, there might be equivalence links to other nodes which can replace this node; these might be synonyms or implications of the original node, such as small/equiv/little, or hot + dog/equiv/frankfurter.

A concept which has proved very useful in METQA's operations has been that of the <u>extended meaning</u>. In a simple search for the facts using some word node W, we would consider only the combinations of W into fact nodes. If we want a richer and more powerful selection of information from the net, however, we might consider the <u>extended meaning</u> of W. This would include not only W, but also the nodes which are connected to W by other types of links (class, membership/subset, equivalence). Using the combinations of all these additional nodes, we get a much larger but still related set of information.

<u>3. Memory Content</u>

The content of the memory network is the only information available to METQA. Hence at any given time, METQA's knowledge is only as complete as its prior experience. The absence of a desired piece of information from the net means not that it is false, but only that METQA does not know it. If some "fact" is false or badly learned, successive downweighting because of bad responses would cause it to be discarded. By the same process, some fact which is consistently useful in producing good (reinforced) responses would be upweighted to the extent that it might be regarded as a basic tenet. In this way, even though the memory can contain mistaken information for a time, experience will tend to direct it steadily toward the reinforced and hopefully consistent world view presented by the human trainer.

<u>3.1 Comparison with Quillian's memory model</u>

The structure of this semantic memory is similar in many respects to Quillian's memory model [Quillian, 1967], but has several major differences. Whereas Quillian's memory model is built by humans who encode entries from a dictionary, aided by their own semantic memories, METQA learns its semantic memory. A word in Quillian's model has its own delimited entry or definition plane, but the meaning of a word in METQA's memory is the more amorphous (and perhaps more human-like) collection of all that has been said about or with the word. Whereas Quillian's memory model differentiates between type nodes (which define meaning) and token nodes (which refer indirectly to their parent type nodes), there is one and only one node for any word in METQA's memory; token appearances are not separate nodes, but form part of the node itself. One might possibly think of the link pointers within some node to other nodes as token appearances of the other nodes, however.

As the reader may have noticed, the extended meaning in METQA's memory is similar to the "full word concept" of Quillian's memory model. But since METQA's task is to retrieve specific information to answer some question, the limited extended meaning is more manageable and useful than the unlimited full word concept. For a discussion-type question, however, METQA can follow all links out from a node with an effect similar to the full word concept.

3.2 Comparison with Lamb's linguistic and cognitive network

The linguistic and cognitive network of Lamb [1969] has a conceptual similarity with the memory network of METQA. Lamb argues that linguistic and cognitive data exhibit similar types of relationships and can be incorporated into a single, stratified network of relationships which can be viewed as a model of (at least some of) the knowledge of a human. He discusses his network from the level of phonological components of the phonemes up through the different strata to the concepts, which are interrelated to form the cognitive stratum.

Although Lamb's network has more levels than METQA's, there is a strong similarity between them. In METQA's memory, the character representations of the terminal nodes function, like the phonemes of Lamb's network, as the basic entry points of the net. Going up through different levels (strata), we reach the idea nodes, which correspond to the concepts at the top of the linguistic portion of Lamb's network. The idea nodes, or concepts, are in turn related to the fact nodes, or cognitive stratum, which hold factual information about the world. Finally, the context requirement restrictions on the links of METQA's memory network have a very similar function to that of the syntactic pattern "planes" which cross the vertical connections between strata in Lamb's network.

In conclusion, the hope is that this semantic memory is simple and general enough to be produced and manipulated by the learning program, but still complex and rich enough to develop a useful "world view" and to provide interesting behavior.

CHAPTER 4

PATTERN RECOGNITION

1. Introduction

In order to enter and traverse the memory net, METQA must first recognize some string of characters as a known unit. For each terminal word or segment in memory, there is a description list showing the external character string representation of that word. The description takes the form of an ordered n-tuple of characters, each tuple part having associated with it certain information (explained below). The program performs a threshold match on the n-tuple character pattern and uses its matching score as weight for its attempts to follow transform links from that node. After consulting a directory for a list of which nodes should be matched against (typically because of same initial letter), METQA attempts to match all the nodes indicated and retains for processing any node which matched sufficiently well to transform. The details of further processing will be discussed in Chapter 5.

2. Pattern Recognition Method Used to Enter Net

The pattern match operation is essentially the same as the n-tuple threshold match for two-dimensional patterns discussed in a previous paper [Uhr and Jordan, 1969]. These programs learn and adjust their own pattern characterizers, in the tradition of the

Uhr and Vossler programs [1963]. Each part of the tuple has several bits of information used to determine its treatment. There is an indication of where to look for the string hunk (along how many characters) and METQA is allowed to slide its search template back and forth a certain amount (a "wobble" tolerance) to allow for garbled, added, or deleted characters. Each part has a weight which is adjusted over time to indicate this part's importance, relative to the whole pattern.

For example, suppose we have a terminal node for "bicycle", whose description is the 3-tuple BI(4)-CY(2)-CLE(3); the weight of each part is in parentheses following the part's string segment. If a pattern subpart matches the input stream in the position indicated, the part's weight is added to a cumulative weight for the match. After all the pattern's subparts are attempted, the accumulated weight in the tally indicates the success of the match. Thus given the input string "BICICLE", METQA would match the first and third tuple parts for a total match tally of $4 + 0 + 3 = 7$. Given the string "BYCYCLE", however, the last two parts would match, resulting in a tally of $0 + 2 + 3 = 5$.

Next comes the match judgment (using the combined weights) on the node METQA just attempted to recognize. Every transform link from a terminal node has an associated threshold value; this is the minimum weight the match must achieve in order to be eligible for that link. One by one, the possible transform links emanating from this node are consulted as to what threshold of weight must be exceeded in order to traverse that link. If at least one link's threshold requirement is met, then the node is considered to have been recognized.

Suppose there is only one transform link from the terminal node for "bicycle", and its threshold value is 6. Then the input string "BICICLE", having a tally of 7, would exceed the transform threshold and would be recognized and transformed. The input string "BYCYCLE", however, would fail. Its weight tally of 5 would not meet the threshold requirement of 6, so that the string would not even be considered recognized.

3. Multiple or Overlapping Matches

As mentioned above, METQA will attempt to match any number of nodes, starting at a given character position. It is possible and quite acceptable that more than one match will occur at a given point. If the input is ambiguous, we want to recognize and pursue all interpretations. If, on the other hand, a spurious match occurs, it will probably be overruled later, either by a node which gives a better "fit" of the input string, or by lack of the proper context to traverse some restricted link associated with the badly matched node.

These pattern recognition attempts are made at every character position throughout a given input, so that METQA becomes aware of <u>every</u> node which matches the input, whether the node strings overlap or not. In the input string "THEREDEAR", for example, METQA might successfully match the nodes for "the", "there", "her", "here", "red", "dear" and "ear". In the human, this corresponds to the initial, unconscious unscrambling of a message necessary to determine all possible meanings before our built-in semantic and syntactic parsing "weeds out" the spurious messages. Similarly, at a later step, METQA "parses" by finding the best combinations of the matched nodes according to spatial and contextual constraints.

4. Importance Given to Pattern Recognition in This Program

In a program dedicated mainly to pattern recognition, much learning effort can be directed at extremely fine adjustment of the weights and separate parameters associated with each small tuple part. Our experience with these methods is documented in the paper mentioned previously [Uhr and Jordan, 1969]. Although METQA has very general pattern recognition capabilities, most of the actual testing has been directed at operations _within_ the memory net, so fairly simple pattern representations, where words are always correctly spelled, have been used.

The method used for learning the node description patterns can also vary with the emphasis of the program. During the testing of METQA, the pattern learning has been programmed to consist of simply bundling an entire word or meaningful unit into a single tuple part, with initial weight and link threshold requirements determined by the character string length (typically, weight=2×(number of characters), threshold=.8×(weight)). But in order to really test the power and generality of METQA's pattern _recognition_ procedures, more sophisticated pattern learning procedures must be used. Let us briefly discuss several possible such procedures (which would require moderate programming changes).

5. String Learning Procedures

One simple technique for learning string patterns is to allow METQA to segment the pattern string impartially and automatically into small character hunks of a certain length, say two characters long, or a randomly chosen length. Each hunk could initially receive the same weight, but the weights could then be modified according to the program's experience to reflect the relative importance of the different hunks. Thus to continue with the "bicycle" example, METQA might automatically segment the string into BI(3)-CY(3)-CL(3)-E(1) or BIC(5)-YCL(5)-E(2). In addition, the program might try different partitioning lengths and decide based on its experience which is the most successful hunk length to use at a given time. Such testing and evaluation of learned characterizer sizes would be analagous to the tuple variations studied by Bledsoe and Browning [1966]. Hunks could also be made to overlap. This method gives some tolerance for misspelling of input words, but is still a somewhat artificial way to segment a string.

Another segmentation procedure which is more responsive to the actual character string under consideration would depend on the program's continuing analysis of the statistical distribution of different characters, using the idea that rarely occurring characters or or character combinations convey more information than characters which are frequently seen. (Used for whole words, this same idea is a basic feature of the question answering routines discussed in Chapter 9.) One could build into the program a letter frequency table based on some of the information retrieval studies [Luhn, 1958; Maron and Kuhns, 1960; Meadow, 1967, pp. 86-103], but this would assume that the frequencies will not vary for the different languages to be processed, and that the program will be concerned with independent frequencies of single letters. If METQA were trained on a large enough corpus of words, it could calculate the frequencies for its own experience.

A more pleasing (but probably very slow) approach would be to allow the program to discover which characters or clusters appear rarely enough to be considered meaningful and important. Such discovered information would then be used to isolate and highly weight very distinctive portions or a word, while giving the less important characters a correspondingly lower weight. Thus "bicycle" might ultimately be segmented as B(4)-I(1)-CYC(6)-LE(2), and "refrigerator" as R(4)-E(2)-FRIG(9)-E(1)-RA(4)-TO(2)-R(2). Perhaps the program could also learn not to break apart by segmentation some sequence of characters which are very distinctive when they appear in a certain order in a string. Examples might be -ngth (as in length, strength), -vis- (as in visible, television), etc., which are important since relatively few words contain the full sequence.

Another possible pattern learning procedure would be to extract for the n-tuple the most indicative segments of a word (perhaps the consonants) and some directions for their proper placement and weighting. Our "bicycle" and "refrigerator" examples might thus be represented as B(4)_C(3)_CL(3)_ and R(3)_FR(4)_G(3)_R(2)_T(3)_R(2). Thus the unimportant characters would not have any weight in the recognition. Here a problem would arise, however, when METQA attempts to output that node's string, with only an incomplete designation of the string. Since METQA's net must function for operations in any direction, it seems best to always have the complete pattern available. Hence one solution would be to include the entire string in the description, but to give a zero weight to the unimportant characters.

6. Indirect Reference to Pattern Descriptions

Whatever particular method of pattern extraction is used, METQA tries to make use of already known patterns if possible. If some part of the pattern string was matched by a known node, the newly learned pattern will refer indirectly to the description of the already learned (sub)pattern. This is done not only to make efficient use of memory space, but also because it seems desirable to use old knowledge to learn new words or concepts. At the proper place in the pattern description being learned, instead of a literal character string there is a reference to some other node's description string, with an appropriate weight attached. See Figure 4-1 for an example of indirect reference.

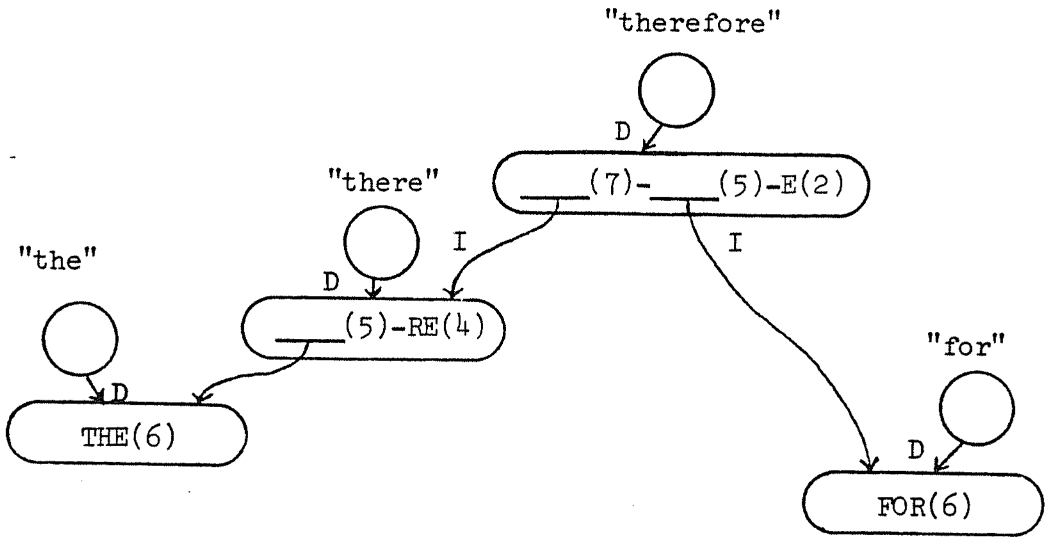


Figure 4-1. Example of indirect reference for pattern description strings. The four terminal nodes shown have nested indirect referencing. "D" marks description links; "I" signifies indirect reference pointer.

During an attempt to recognize such a node, the discovery of an indirect reference causes a recursive push-down in order to recognize the embedded node. When the whole embedded node description has been attempted, then METQA pops back up to tally the success weight of the match and to finish processing the original (calling or referring) pattern.

7. Recognizing Relevant Information

The use of pattern recognition to find memory nodes containing information relevant to a given question is quite different from the procedure for recognizing nodes by which to enter the net. In the string matching case we seek a close, ordered match, and the nodes themselves (by certain weight requirements) determine whether or not they have been matched by a string.

The matching of relevant information, however, is a much looser process. Using a "Pandemonium" type procedure [Selfridge and Neisser, 1963] each content word in the question being processed is polled as to which information nodes it considers relevant, and to what extent (weight). Each component word of a fact node contributes a certain amount of weight to it. All the suggested nodes are merged into one list and are ordered according to their combined weights. Then METQA chooses the most relevant (highly weighted) information to process in trying to produce an answer to the question. A cut-off limit is used at this point, typically requiring a fact to be suggested by words comprising at least half of its total possible weight in order to be relevant enough to use in trying to find an answer. (The question answering process will be described in Chapter 9.)

As mentioned in Chapter 2, a limiting factor on METQA's ability to answer questions will be its ignorance of linguistic structures and its consequent inability to determine relationships (other than cooccurrence) among the words of an input. Furthermore, word order alone is unreliable from language to language, or sometimes even from sentence to sentence within a language. For these reasons, word order is not currently used in recognizing information. Instead, METQA must depend on cooccurrence of words, contextual restrictions to weed out erroneous choices, and word permutation rules to put things into proper order.

In the event that this proves inadequate, it might be necessary to try considering local word order in choosing information. That is, we might consider different pairs or triples of content words from the question and note their relative order. Then a fact containing a given triple of words in the proper order would receive a higher relevance weight than if the order was different.

It is not clear to what extent this measure would help. Consider the question "Did Tom give Jerry's apple to Mary?". Using local word order with the triple (Tom, apple, Mary) would enable METQA to choose the fact "Tom gave the book and the apple to Mary" over, say, "Mary and Jerry like Tom's apples". But unfortunately, the sentence "Tom and Jerry throw apples at Mary's dog" would also pass the local order test, and we would again be dependent on the other methods to select the desired information.

In conclusion, METQA uses pattern recognition techniques as a means of finding the proper starting positions in the memory net before initiating other processes. That is, before beginning to transform or translate an input, METQA must know which nodes to transform from. And before finding the answer to a question, METQA must recognize which information should be used to build the answer. In each case the problem reduces to one of recognizing the proper patterns.

CHAPTER 5

THE TRANSFORMATION PROCESS

1. Formation of Paths through Memory

Suppose METQA receives an input string of characters in some language. Starting at each character position, METQA consults the net directory to see which nodes might possibly be matched from there. For every letter of the alphabet, the directory contains a list of pointers to every learned node which begins with that letter.

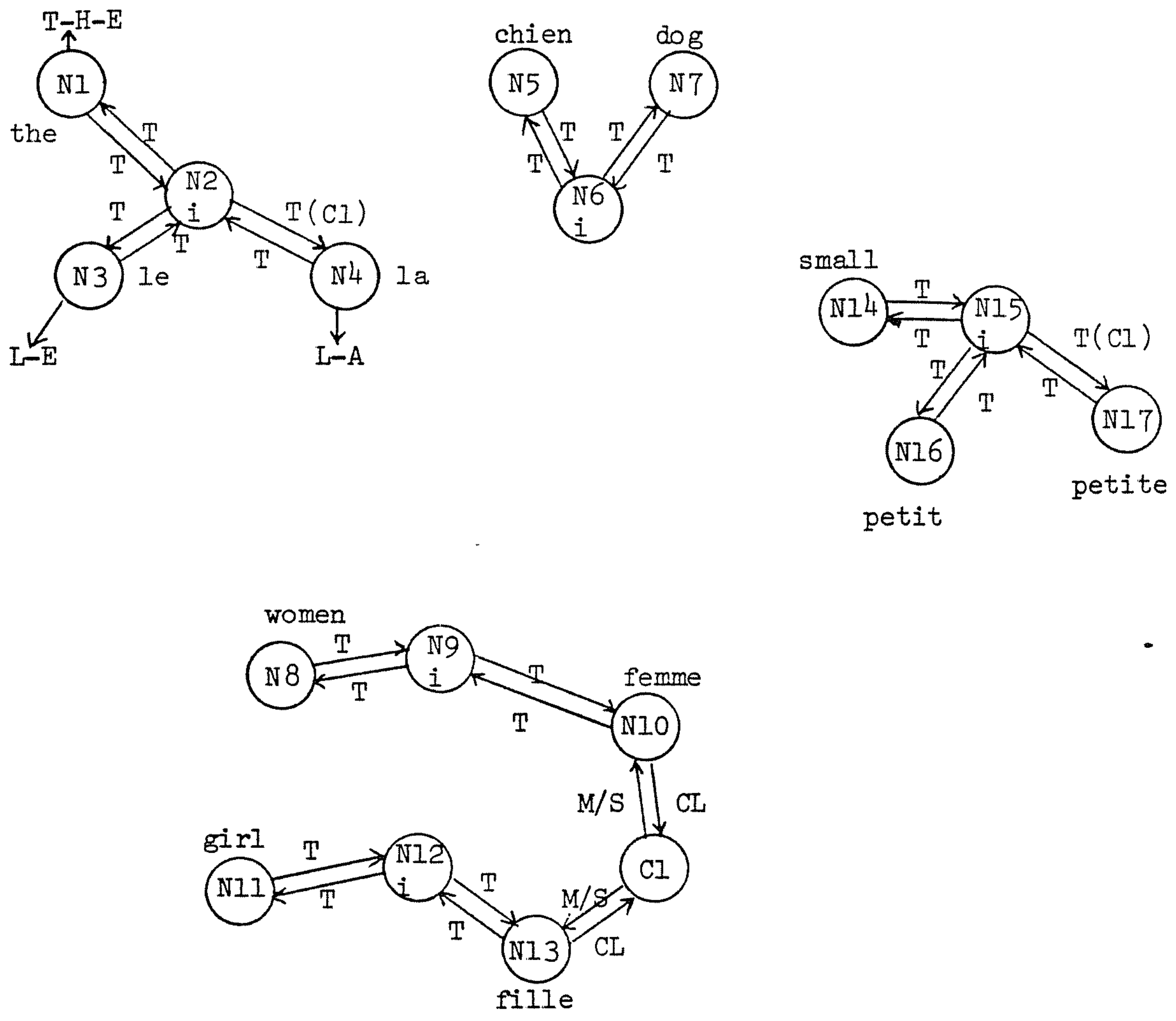
Given the list of suggested nodes, METQA attempts to match each one, using the n-tuple pattern, threshold match techniques described in Chapter 4. By looking for all possible node patterns at every possible character position, METQA exhausts the node match possibilities throughout the input string.

Now METQA starts processing every node which matched the input well enough to satisfy its transform link's threshold value. In order to keep a record of where it has been in the memory net, METQA forms what we will call a _path_ list. A _path_ is a list of the nodes traversed by following transform links from node to node in memory, as allowed by the context in which the path appears. For every matched node, METQA will keep a special path list in the "working space" of its memory. Note that every path has an associated position, the position of its matched node representation in the input string.

Now let us examine the formation of paths by means of an example. Suppose we have a very small memory net, as shown in Figure 5-1. METQA at this point knows five words, with as many as three representations for each, and one context class which has two members. The resulting directory contains a node pointer list for the nine initial letters. (A later example will show how such a memory might have been formed through learning.) Although there are a dozen terminal nodes, for only one group (nodes N1 - N4) do we show the description pointers to their external representations "THE", "LE", and "LA".

Given an input string "THEDOG", METQA consults the node directory for every letter. Starting from the "T", node N1 is successfully matched against the first three characters. No words are known which start with "H" or "E". Starting from the "D", METQA successfully matches node N7 ("dog"). No known words start with "O", and there is not enough room in the input string to match the suggested node N11 ("girl") for the final "G". Thus METQA is able to form two paths, one for "THE" and one for "DOG".

Now the paths can follow transform links. Node N1 transforms to node N2, the "idea" node for different forms of the definite article, but from that point there are two possible links to follow. The (newer) transform link to N4 is restricted to context of class C1; that is, a member of class C1 must be found "nearby" (currently, this means within two paths in either direction) in the input in order to follow that link. Since the desired class member is not present (the "dog" nodes have no class membership), the path from N2 can go only to node



Directory

C → N5
D → N7
F → N10,N13
G → N11
L → N3,N4
P → N16,N17
S → N14
T → N1
W → N8

Links

T - transform
CL - class
M/S - membership/subset
D - description

Figure 5-1. Formation of paths in a small memory.

Sample transform paths through net:

Input: THEDOG

```
Path1  THE  N1 - N2 - N3  →  LE
Path2  DOG  N7 - N6 - N5  →  CHIEN
```

Input: THEWOMAN

```
Path3  THE  N1 - N2 - (C1) - N4  →  LA
Path4  WOMAN  N8 - N9 - N10  →  FEMME,  FEMME  ∈  C1
```

Figure 5-1. (Continued)

N3, which is terminal with the external representation "LE". Similarly, a path is formed from "DOG" to "CHIEN".

Now consider another input string "THEWOMAN". METQA matches and forms paths for "THE" and "WOMAN". Note that in the "woman" group of nodes, node N10 ("femme") is a member of context class C1. Thus when the path for "THE" reaches node N2 and must choose between two links, the class C1 context restriction on the link to N4 will be satisfied. Thus METQA will note the context class and extend the path to node N4 as shown, and hence on to the external representation "LA". Thus the formation of class C1, which contains words of feminine gender, has allowed the program to choose the correct (feminine) form of the definite article. (Chapter 6, Sections 3 and 7 will show how this class can be formed.)

2. Comment on Input Strings

As the reader may have noticed, the input strings given to the program by the human trainer contain no spaces between words. This mode of input was chosen in an effort to be as general and realistic as possible. The program was written to handle any language in any form which can be suitably input as a string of symbols. These forms might include printed words, digitized speech sounds, or recognized hand printed characters. Only in written language are spaces provided to isolate words for the reader. In speech there are no such aids; a whole sentence can sound like one unintelligible stream of sound, if the language is unfamiliar. Just as a young child must learn to recognize the meaningful units and grammatical patterns in the language he hears around him, so METQA must operate on the input strings it receives. Unsegmented input makes the problem more difficult, but results in a more powerful and useful program. METQA inserts spaces between the words it outputs, however; this is not necessary, but is done for convenience and clarity to the human.

Note that METQA will learn from strings with spaces. It will learn the space as a word, and this will serve to segment strings, helping it to recognize and learn other words.

3. Combination of Paths into Covers

After paths have been formed for all the matched nodes anywhere in the input, then METQA must decide how all the pieces fit together in order to best account for or "cover" the input string. We will define a cover as an ordered collection of paths (gathered into a work list) which account for nonoverlapping segments of the input string. Any single path can be used in more than one cover list. The only restriction is that no character of the input may be used by more than one path in the same cover; that is, paths may not overlap.

There may be characters left unaccounted for between two matched paths in some cover; we will refer to these characters which are unrecognized and untransformed by a cover as garbage. In assembling a cover, the goal is to minimize the number of garbage characters.

Finding a complete cover of an input without any garbage is analagous to finding a parse of a sentence; notice, however, that an input is often not a sentence.

Using all the available paths, METQA considers all possible covers of the input. Only the best covers, those with the least amount of garbage, are kept in the work space for further processing.

As a contrived example, consider the input shown in Figure 5-2. Given a current vocabulary which includes the indicated list of words, suppose the input string "THEREDDOGMAYBARK" is received. METQA recognizes and forms a path for each word in the list, then starts to build covers.

```
Input:   THEREDDOGMAYBARK
         _____X_____X____
         _____X___ ___ ____
         ___ ___ ___ ___ ____
         X____X_____X____
         X____X___ ___ ____
```

Current vocabulary includes:

bark	may
dog	the
dogma	there
here	

Possible covers:		Garbage count:
CV1.	THERE U(D) DOGMA U(Y) BARK	2
CV2.	THERE U(D) DOG MAY BARK	1
CV3.	THE RED DOG MAY BARK	0
CV4.	U(T) HERE U(D) DOGMA U(Y) BARK	3
CV5.	U(T) HERE U(D) DOG MAY BARK	2

U( ) means enclosed string is unknown or unrecognized.

Figure 5-2. Formation of covers for an input string.

As a rule, from any given position, the path with the longest matched string is used first. Thus the first cover found is "THERE U(D) DOGMA U(Y) BARK", with two characters unaccounted for. (Note that "U( )" means that METQA could not recognize the enclosed string.) Any later covers are therefore allowed at most two garbage characters. The second cover built is "THERE U(D) DOG MAY BARK", which reduces the maximum garbage allowed to one character. But later, cover CV3 ("THE RED DOG MAY BARK") is built with no garbage at all. Any succeeding covers will be rejected if they have <u>any</u> unaccounted for characters. Thus covers CV4 and CV5, although perhaps acceptable before cover CV2 was found, would now be rejected immediately. As a result, METQA has only two covers to process through memory.

It is possible and quite acceptable to have more than one perfect cover (i.e. with no garbage) of an input string. In such cases of ambiguity, <u>all</u> the "parses" are developed and processed through memory.

The few covers resulting from the formation process are then cycled repeatedly through a group of procedures to check covers for satisfaction of required context, to transform all paths another level farther, and to try to apply any suggested permutation rules.

4. Check for Needed Context

At every cycle, a context check is made in case some cover path has been transformed along a link which was restricted to certain contexts. For each cover, the latest node of each path is inspected for presence of a new context requirement. If such a list of context class requirements has appeared, the rest of that cover is searched for node(s) having the desired class membership(s). Failure to find the necessary contextual situation in the input means to METQA that this cover of the input is anomalous and must be destroyed. A successful search, however, means that transformation to the restricted link's node was consistent with the context classes of the cover, and this cover remains useful.

5. Transform Down to Another Level

So that the progress of a cover through the memory net is careful and even, with status checks at every turn, each path is transformed down only one link per cycle. If there is only one link to follow from the current node, the path is simply extended to that node. In case of multiple links leading from a node, however, METQA tries to resolve the ambiguity and choose the one best link.

In the initial stages of testing, METQA was allowed to pursue all possible links from each node in order to obtain all possible covers and to check for consistency later on. But this resulted in such a time-consuming proliferation of nonsense and/or identical parses, that METQA was constrained to disambiguation node by node.

Whenever there is more than one link to follow, METQA makes its choice according to a hierarchy of disambiguation rules. The first criterion applies only if this string is marked as a feedback to the previous input. In that case, as will be explained in Chapter 6, METQA is looking for an intersection of paths from this feedback path with some path from the original input. That is, if one of the link choices from a feedback path node leads to a node which was occupied by a path of the original input, then that intersection link would be chosen above all others. Path intersections receive first priority because they end the uncertain groping through memory for the correct path. The paths formed for original input and for feedback should ideally be just reverses of each other, so the earlier a path intersection is found, the better.

To illustrate, let us use the small segment of a memory net shown in Figure 5-3. Suppose that METQA had previously processed the input string, "THEBOOKISTOOSHORT" and had output, say, "LE REGISTRE EST U(TOO) COURT". Then there is a cover path saved from node N7 ("book") through N6 to N8 ("registre"). The next input is the feedback string, "LELIVREESTTROPCOURT". Now when METQA recognizes the string "LIVRE" at node N5 and starts to transform from there, it must decide between the two links to nodes N9 and N6. Since the current string is feedback, the first step is to look for a path intersection. Node N9 was not a member of any input cover path, but node N6 was. Consequently, METQA chooses the link to N6 and has thus connected a feedback word with its corresponding input word. (See Chapter 6 for a more complete discussion of feedback and path intersections.)

If the current input string is not feedback (or if no intersections were found), the next criterion for disambiguation is choice

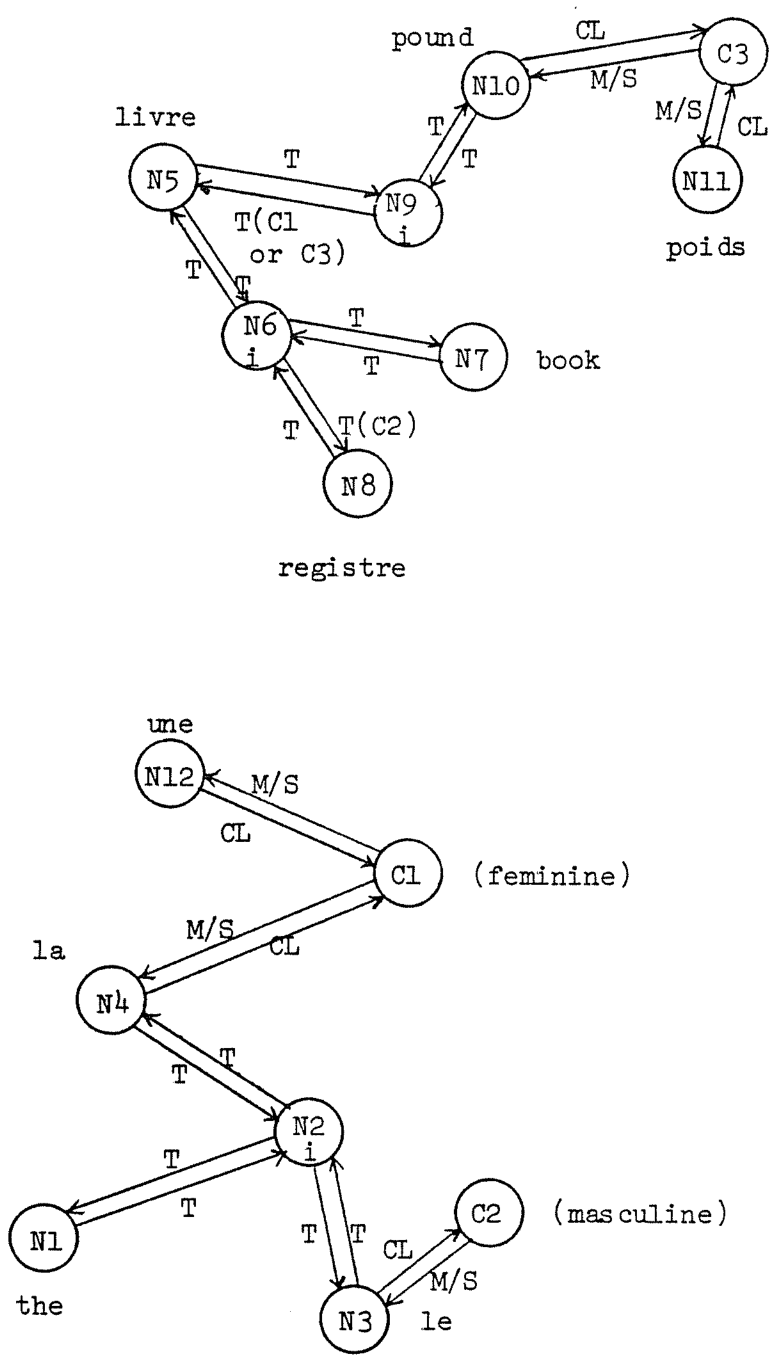


Figure 5-3. Segment of memory net with ambiguous "livre"

by contextual "selection". That is, if there is a contextually restricted link whose context requirements are satisfied by the input, then that link has priority over other choices. Whenever there is a choice between an unrestricted path and contextually restricted path, the presence of a member of the desired context class(es) causes the restricted path to be chosen. Thus the context of an input _chooses_ a restricted path if possible. The reasoning behind this is that satisfaction of a required context implies some previous knowledge of this type of situation and therefore of the correct transformation. It may happen that on this basis, the wrong branch is chosen because of incorrect or insufficient context restrictions. In that case, METQA will consult the feedback string and adjust the restrictions on both the correct and incorrectly chosen links.

As an example, let us again consider the memory segment in Figure 5-3. Suppose METQA receives the input "LALIVREESTUNEMESURE DUPOIDS". When the string "LIVRE" is recognized, there is an immediate ambiguity. The input was not a feedback, so METQA tries the second criterion. The link to N9 is contextually restricted, so METQA searches the input string for a member of one of the desired classes (C1 or C3). Adjacent to "LIVRE" in the string is "LA", which is discovered to be a member of C1 (class of feminine words). This context selects the link to node N9, so METQA extends the

"LIVRE" path on to that node, and later on to node N10 ("pound").

If the multiple links include some contextually restricted link(s), and METQA is currently unable to satisfy the context required by any restricted link, then the path is flagged as "waiting". It is possible that some path will reach a multiple link decision point before its companion paths in the cover have traversed the net far enough to discover the context classes needed to disambiguate. Then transformation of that path is suspended temporarily so that it can wait while the other paths in the cover proceed through the transformation cycles. It can happen that a deadlock is reached, with several paths in a cover caught waiting on each other, and no further transformation possible. In that case, METQA throws a switch and forces the choice of the highest weighted link for each path in order to avoid indefinite looping.

Suppose that an ambiguity occurs and neither of the first two criteria can apply; that is, the input is not feedback and METQA was unable to select a link by the cover's context. Then the only alternative is to choose from the remaining transforms that one which is most highly weighted. Thus by traversing the link which has been most successful or most frequently correct in the past, METQA tries to make the best choice now with insufficient information.

6. Try to Apply Any Suggested Permutation Rules

As will be explained in Chapter 7, METQA learns permutation rules (p-rules) to govern the segmental permutations noted between input and feedback strings. Any node along a path may at some time during translation have been involved in a word order permutation. Then every time that node is used again, METQA is alerted that the indicated permutation might be involved here again. Thus during each cycle, it is possible that several permutation rules may be implied or suggested by the nodes traversed. METQA remembers all these, as well as the rules suggested by previous cycles. Then after the transformation phase is completed, it tries to apply the rules to the newly extended covers. If the class lists required by some p-rule are successfully filled by the input, the indicated permutation of the segments is performed. Then the program finishes checking other p-rules and continues cycling.

7. Criteria for Stopping Cycles

The program repeats the above cycle of operations while processing the covers of the input. Context checking is followed by transforming each path down a node; next, any indicated permutation rules are checked for applicability. Then the covers are again checked for contextual consistency. This process continues until each cover reaches some desired status determined by the current linguistic task.

If METQA is attempting to translate an input string, then cycling must continue until each path in each cover has been transformed from the matched source terminal node all the way through memory to the target terminal node with its desired external representation. Thus the translation process goes from one (input) surface character string to another (output) surface character string, perhaps with some segmental permutations performed along the way. In the event that some input string segments are not recognized or matched by any node, they are carried along and output without change.

If the current input is <u>feedback</u> to the previous input, then METQA operates on it as if translating it back toward the original input, but always searching for path intersections between the original input and feedback strings. Thus for translation feedback, cycling continues until every path either has intersected an input path or has emerged at the surface of the net again at a terminal node.

Suppose METQA is attempting to answer a question. In this case, the goal is not some other terminal string, but access to the information contained in the memory net. Thus the covers should be transformed only until every path has reached its idea node, from which the memory's facts are accessible. At that point, control is transferred to the question answering portion of METQA (described in Chapter 9) in an attempt to produce an answer to a question.

8. Output of Results

After all the covers have been transformed to terminal node level again, METQA is ready to output the translation(s) of the input. Although one cover is chosen and output as the best, all the different translated parsings are shown. The best cover is chosen according to the closeness of the fit (with minimum garbage), and is saved in the work space by METQA in case the next input is feedback to this one. In that event, the saved cover is compared with feedback for use in learning correct translations. The transformation learning process will be described in the next chapter.

CHAPTER 6

THE LEARNING OF TRANSFORMATIONS

1. Learning by Comparison of Strings

The learning of transformations between terminal strings is based on a comparison of two strings provided by a human teacher or trainer. The original input string is received and processed through the current state of the memory net. METQA then outputs the result, which we will refer to as the response (response = transformed input).

The next string can be something other than feedback (another original input or a question, perhaps), in which case no transformation learning will occur. The appearance of a feedback string (marked as feedback by a special symbol "=" at its start), however, begins the learning process.

To learn, METQA must decide which segments of the two strings (which we will refer to as input and feedback) should be associated with each other. This chapter will be devoted to an explanation and discussion of the techniques used to accomplish this association.

2. How Feedback is Processed

When a feedback string is received, certain flags are set to establish a learning mode, but otherwise the transformation process begins as if for any input. METQA tries to match segments of the feedback string against nodes in memory, and forms paths and covers from these nodes as described in Chapter 5. Because of the feedback learning mode, however, the goal is not so much to push through the memory net to another surface, as it is to establish points of contact or association between the input and feedback strings. Thus each feedback path steps through memory searching at each node for an opportunity to intersect an input path. Whenever there is a choice of links to traverse from some node, METQA gives highest priority to any link which leads to a node occupied by a path from the original input, and which would therefore result in a path intersection (examples will be shown in Figure 6-1). If such an intersection occurs, that feedback path is considered ready for the learning process and needs no more transformation during the cycling phase. When processing feedback, the criterion for stopping the cycle of transformation and status checking is that all paths of all covers have either intersected or reached terminal level. At that point, control is passed to a routine which compares the transformed input and feedback strings and tries to learn the transformations involved.

3. Explanation of Path Intersections and Their Meaning

The significance of a path intersection between input and feedback strings is the establishment of an association between two string segments. The joining of two transform paths at some common node means that here is a direct transform route between the terminal strings involved. METQA gives links to path intersections the highest priority because it seeks the shortest, most direct route through memory between two terminal character strings. If METQA bypassed an intersection, then it is likely that any later intersection (therefore by a longer path) found would not give such a short, direct path route. It is even possible that no later intersection would occur at all, if METQA took a wrong turn and completely missed the connection with the original input.

Path intersections are of two general types, terminal and nonterminal. A terminal intersection has occurred if a feedback path joins an input path immediately, as soon as the surface (intersection) node is matched by the feedback string. Such an immediate match means that the input path was correct all the way to the target terminal level string output in the response. Hence no memory adjustment is made, since METQA is too cautious to tamper with the transform links if all is going well.

We say that a nonterminal intersection has occurred if the intersection occurs on any feedback path node in the interior of the net, past the terminal node recognized in the feedback string. Finding a nonterminal path intersection at some node A means that METQA processed the original input string segment correctly up to node A, but at that point of ambiguity took a wrong link and arrived at an incorrect terminal node. METQA must then adjust the links emanating from node A, so that given the same situation again, METQA would take the correct link toward the feedback string segment.

In general, this adjustment would involve lowering the weight of the badly chosen transform link by some small amount as well as lowering the weight of any context requirement used to choose the bad link. In addition, METQA will raise the weight of the good link toward the feedback string by a similarly small amount, and will add or upweight the context which would have caused the good link to be selected.

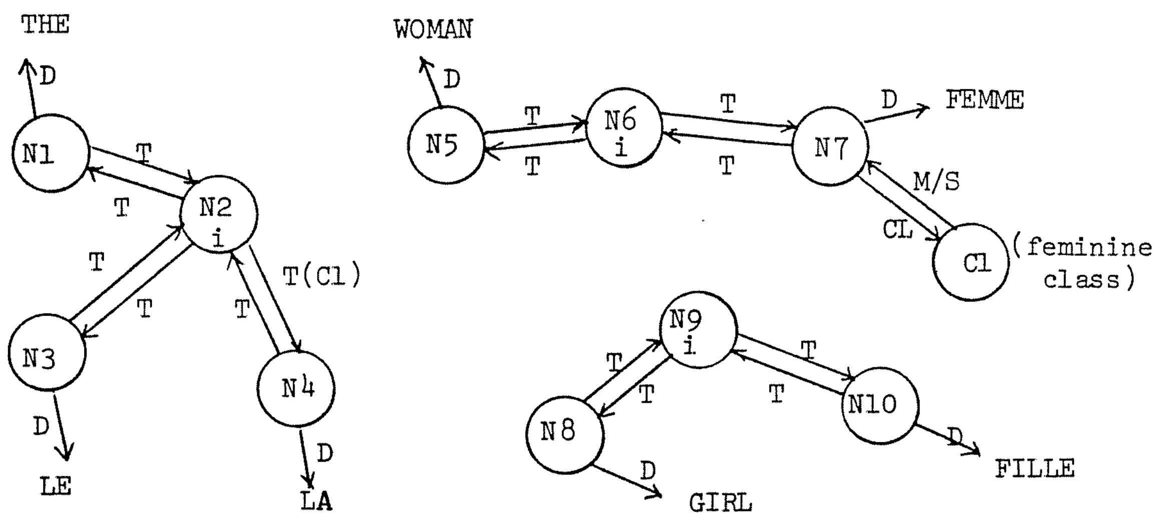
If the good link G already has a context requirement list C, but the current string's paths have no class memberships, then METQA generalizes memberships of the context class list C to the current string. The reasoning behind this goes as follows. The current path should have traversed the good link G. Link G is restricted to contextual situations described by context list C. Then perhaps the context of this current cover also should be described by the list C. Therefore the words in the current cover should be given the same context class memberships described by list C. This procedure will be illustrated shortly with an example.

At times, METQA must add a new context requirement option to the existing restriction list of the good link toward the node matched by the feedback string; this requirement should reflect the context of the intersected path P in this particular cover, so that, given the current situation again, the correct link would be chosen. When such a context list is needed, METQA extracts a local context list from the cover being processed. The context comes from combining the class memberships of the path strings surrounding path P; if there are no class memberships to use, METQA creates new node classes and assigns the node neighbors of P as their members. These new classes are then used as the needed context.

To illustrate the use of path intersections, let us consider the example input sequence in Figure 6-1. Suppose the memory net contains at least the nodes shown. There is a usage class C1, with node N7 ("femme") as its only member. Given the string Input A, "THEWOMAN", METQA forms Path1 and Path2 as shown. Path1 goes from N2 to N4 because the needed context class C1 was present (in N7) and caused that link to be selected. METQA's response is "LA FEMME".

Next comes the string Feedback A, "LAFEMME". The string "LA" of node N4 is matched; METQA initiates Path3 and checks for path intersections. Immediately there is an intersection with Path1 at the terminal node N4 whose string representation "LA" was in the response. The terminal intersection indicates to METQA that the response segment output from Path1 was correct, and no learning is needed there. Similarly, the feedback string segment "FEMME" is matched by node N7, resulting in another terminal intersection, this time with Path2. Since only terminal intersections were found (and the order was not changed), METQA decides its response was entirely correct.

The next string received is Input B, "THEGIRL". METQA matches the string against nodes N1 and N8 and forms Path5 and Path6 as shown. Since there was no class membership in the cover paths to allow passage of the restricted link to N4, Path5 had to go to node N3 ("le") from N2. The response is "LE FILLE", followed by the input



Links: T - transform
D - description
M/S - membership/subset
CL - class

Input A: THEWOMAN

```
Path1  THE    N1 - N2 - (Cl) - N4 → LA
Path2  WOMAN  N5 - N6 - N7 → FEMME,  N7 ∈ Cl
```

Response A: LA FEMME

Feedback A: LAFEMME

```
Path3  LA     N4  →  intersects Path1 terminally
Path4  FEMME  N7  →  intersects Path2 terminally
```

Input B: THEGIRL

```
Path5  THE   N1 - N2 - N3  →  LE
Path6  GIRL  N8 - N9 - N10  →  FILLE
```

Response B: LE FILLE

Feedback B: LAFILLE

```
Path7  LA     N4 - N2  →  intersects Path5 nonterminally
Path8  FILLE  N10  →  intersects Path6 terminally
```

Figure 6-1. Path intersections

of a Feedback B string, "LAFILLE". Node N4 matches the feedback segment "LA", and Path7 is initiated. METQA looks for a path intersection, but finds N4 in neither Path5 nor Path6 of Input B. The string "FILLE" is matched by node N10 and Path8 is discovered to have a terminal intersection with Path6. Next, METQA transforms Path7 to the adjacent node, N2. Here, an intersection is found in the interior of Path5, which had incorrectly transformed to "LE". Faced with this nonterminal intersection at node N2, now METQA must adjust the memory so that if Input B ("THEGIRL") was received again, Path5 would transform to "LA" instead of "LE". Path5 should have gone to node N4 from N2, but was prevented by lack of the required context class C1. METQA conjectures that the string Input B, "THEGIRL", _should_ have satisfied the context requirement; therefore it _assigns_ membership in the class C1 to the only other path's node in the feedback, N10 ("fille"). Thus since "the" should transform to the feminine form "la" when in the context of "fille" as well as of "femme", METQA assigns "fille" to the same class C1 (of feminine words) which "femme" belongs to. Hence, METQA will build a class link from N10 to C1 and a membership link from C1 to N10. In addition to this adjustment, the links involved will be reweighted as described earlier.

4. Segmentation and Association of Transformed Strings

The basic problem of learning or discovering node transformations is to determine which segments should be associated with each other. METQA's transformation learning routine operates by comparing the paths from transformed input and feedback strings. Using the terminal and nonterminal intersections found, it partitions the strings into successively smaller segments. A work list is built to show explicitly all the information about the intersections found and the string hunks among them, and about how the intersections and hunks pair off together or are left unclaimed between other intersections. This list is then used to do the actual learning.

For convenience, we refer to terminal intersection partitions as _walls_ and to nonterminal intersection partitions as _fences_. Terminal intersections represent correctly transformed inputs, so walls are more substantial indicators than fences. A nonterminal intersection means only that the input was matched and started through the net correctly; there is still more to be learned. Hence a fence is not as strong a partition as a wall.

We shall refer to the regions of the string which lie between walls as _areas_; an area can be empty. Any fences which lie in an area divide that area into _subareas_. These subareas can be empty or can be composed of any combination of unrecognized character strings, unintersected paths, and possibly other intersected paths. They are inspected and, by various means, paired off to be learned as transforms. An intersection or hunk in the transformed input or feedback strings which does not "match up" with its partner string within an area or subarea (usually because of a permutation of string segments) will be described as _unclaimed_.

A set of contiguous nonintersecting paths or unknown character strings in a subarea will be loosely referred to as a <u>hunk</u>. The unrecognized character strings are any string segments which were not accounted for by the cover. Nonintersecting paths occur as follows; suppose some (input or feedback) string segment matches some node and is transformed all the way through memory to another surface string. If the match is erroneous, there will be no associated match in the other string (feedback or input) for this path to intersect with, and thus we will have a nonintersecting path. Hence the recognized but unacknowledged string segment must just be lumped in with the rest of the hunk to be learned as a new word. Typically, an unintersected path occurs when some very short word (e.g. "le", "is", "at") is recognized while embedded in a larger word that METQA is about to learn (e.g. "table", "rise", "hat"). METQA simply treats the whole subarea as one character string hunk.

Within the walled-off areas, matched nonterminal intersections (fences) are learned with some adjustment of the existing memory nodes and links. Subareas between fences may most simply be matched hunks, which are learned as direct transformations. Each hunk is learned as a terminal node; an idea node is created; and the terminal nodes are connected to each other by transform links via the internal idea node for that word pair. Any other forms of the same word(s) will be attached to the same idea node by means of transform links in both directions. If there are unclaimed intersections within a subarea, any hunk lying on the left or right end of the subarea has arbitrary priority to match with the hunk in the corresponding subarea. In any case, all unclaimed intersections are marked for later permutation learning.

Consider Figure 6-2 for an example of straightforward learning. METQA receives the input string "THETHINWOMANEATSBREADALSO". Suppose that after transforming the recognized nodes as well as possible with the current memory, the response output is "LE MAIGRE FEMME U(EATS) PAIN U(ALSO)".

Input: THETHINWOMANEATSBREADALSO

Response: LE MAIGRE FEMME U(EATS) PAIN U(ALSO)

Feedback: LAFEMMEMAIGREMANGEPAINAUSSI

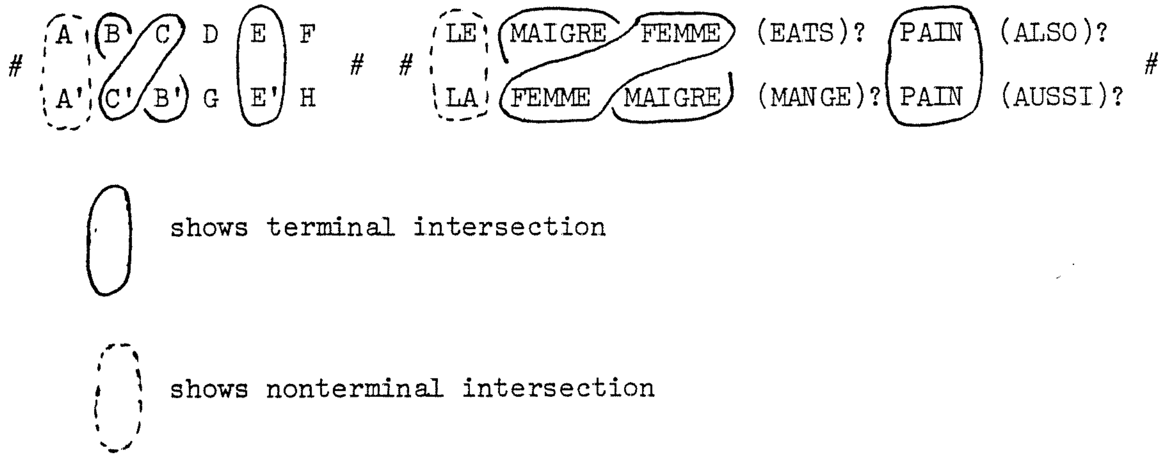


Figure 6-2. Learning transforms. Example of string segmentation and association.

The feedback string received is "LAFEMMEMAIGREMANGEPAINAUSSI". (The trainer has simplified the translation by omitting the "DU" of

"DUPAIN".) Several feedback segments are recognized and path intersections are found to associate the strings as shown in the drawing. The nonterminal "LE" - "LA" intersection causes the usual readjustment of those nodes. Terminal intersections partition off the rest of the strings. METQA notes the permutation of the terminally intersected adjective and noun paths; later a special routine will attempt to learn the permutation, as described in Chapter 7.

The partitioning isolates two pairs of unknown character strings, one on each side of the "PAIN" wall. For each pair, a terminal node is made for each character string and a new idea node connects them via transform links. In this simple manner, METQA is able to isolate and learn any new words which are string hunks conveniently "walled off" by path intersections.

5. Heuristics for Unclaimed Segments

Not all new word situations are so convenient, however. A subarea, or even a whole area between walls, may of course be empty. If only one subarea in a given input-feedback pair is empty, METQA must seek some means of associating the remaining unclaimed hunks with other segments for learning. There is a hierarchy of possibilities from which one or more hypotheses can be chosen.

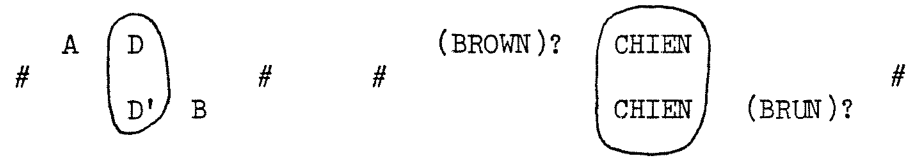
5.1 Permutation

First, an unclaimed unknown hunk looks for a complementary (i.e., in the other transformed string) unclaimed hunk in the adjacent subareas to left and right, to pair off with and be learned as a transformation. A typical case where this heuristic is useful is shown in Figure 6-3. Here the unknown character strings "BROWN" and "BRUN" will be learned as transforms, since they were left unclaimed in adjacent subareas.

Input: BROWNDOG

Response: U(BROWN) CHIEN

Feedback: CHIENBRUN



(Note: # means end of string.)

Figure 6-3.

5.2 Discontinuous transforms

Failing the first possibility, the unclaimed unknown hunk next looks in the adjacent subareas for matched unknown hunks which would have already been learned directly as transforms of each other. If successful, the program hypothesizes a split-merge transform in the context of the intervening fence or wall segment.

Split-merge transforms are METQA's way of handling discontinuous morphemes; they are used in cases where one segment in language L1 is transformed to two segments in language L2, and vice versa. The transform links among the three segments' nodes explicitly indicate how the paths must be manipulated. The link from the L1 node directs the splitting of this path into two parts, each transforming to a separate node of L2. Meanwhile, the links coming from the two L2 nodes in the other direction indicate that their paths must be merged to form one path and ultimately, one string segment in L1.

Input:	NOTEAT
Response:	U(NOT) MANGE
Feedback:	NEMANGEPAS

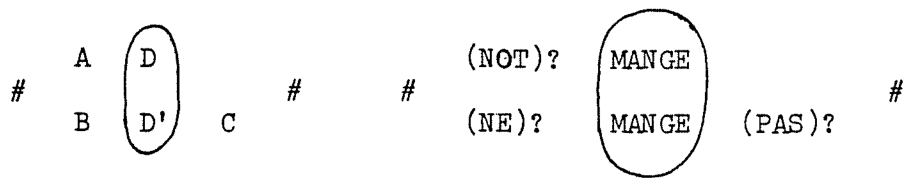


Figure 6-4.

Figure 6-4 shows a typical split-merge situation. The hypothesized relationship assumes that if segment A appears in the context of known segment D (wall or fence), then A's path would split into two forks, B and C, which appear in the context of D', the transform of D. Similarly, if paths B and C appear in the context of path D', they would merge into one path A, which would have the transformed context D. Thus given the situation in Figure 6-4 with unknown hunks A, B, and C, the program would hypothesize not only $A \rightarrow B$ and $B \rightarrow A$, but also A (in context of D) $\rightarrow B + C$ and $B + C$

(in context of D') → A. All three rules are put into memory to be confirmed or denied by subsequent experience. By this means, transforms such as NOT → NE...PAS may be learned, but the program cautiously waits to test its split-merge hypotheses on another input before also hypothesizing the actual permutation rules involved.

Suppose that the above possibilities have failed because both subareas adjacent to some unclaimed unknown hunk are completely empty. In this case, METQA associates the hunk with the fence(s) or wall(s) surrounding it, hypothesizing that in the context of one fence, the other fence, say F, transforms to a form containing the unclaimed hunk as well as the already learned transform of F.

Input:	PRETTYHOUSE
Response:	JOLI MAISON
Feedback:	JOLIEMAISON

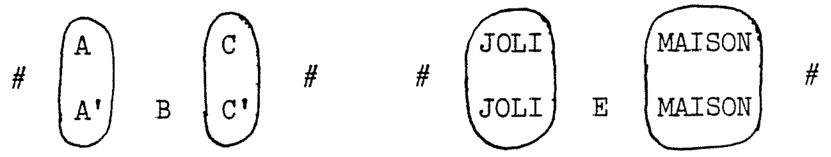


Figure 6-5.

Thus if unclaimed hunk B appears between two walls A-A' and C-C' as shown in Figure 6-5, the program will hypothesize:

A	(in context of C)	→	$\underline{\text{A'B}}$
C	(in context of A)	→	$\underline{\text{BC'}}$
$\underline{\text{A'B}}$	(in context of C')	→	A
$\underline{\text{BC'}}$	(in context of A')	→	C

(Note: Underlining here implies representation as one node in memory.) Hence METQA would learn that "PRETTY" in the context of "HOUSE" is transformed to "JOLIE" instead of "JOLI". Of course it will also hypothesize that "HOUSE" in the context of "PRETTY" transforms to "EMAISON"; but METQA depends on experience and reweighting to help "unlearn" bad hypotheses. Using this last option for learning, the program is able to discover words which change forms to agree with their context, such as:

MANGE	→	EAT,EATS
PRETTY	→	JOLI,JOLIE
THE	→	LA,LE,LES
FISH	→	POISSON,POISSONS

6. Learning and Association of Permuted Intersections

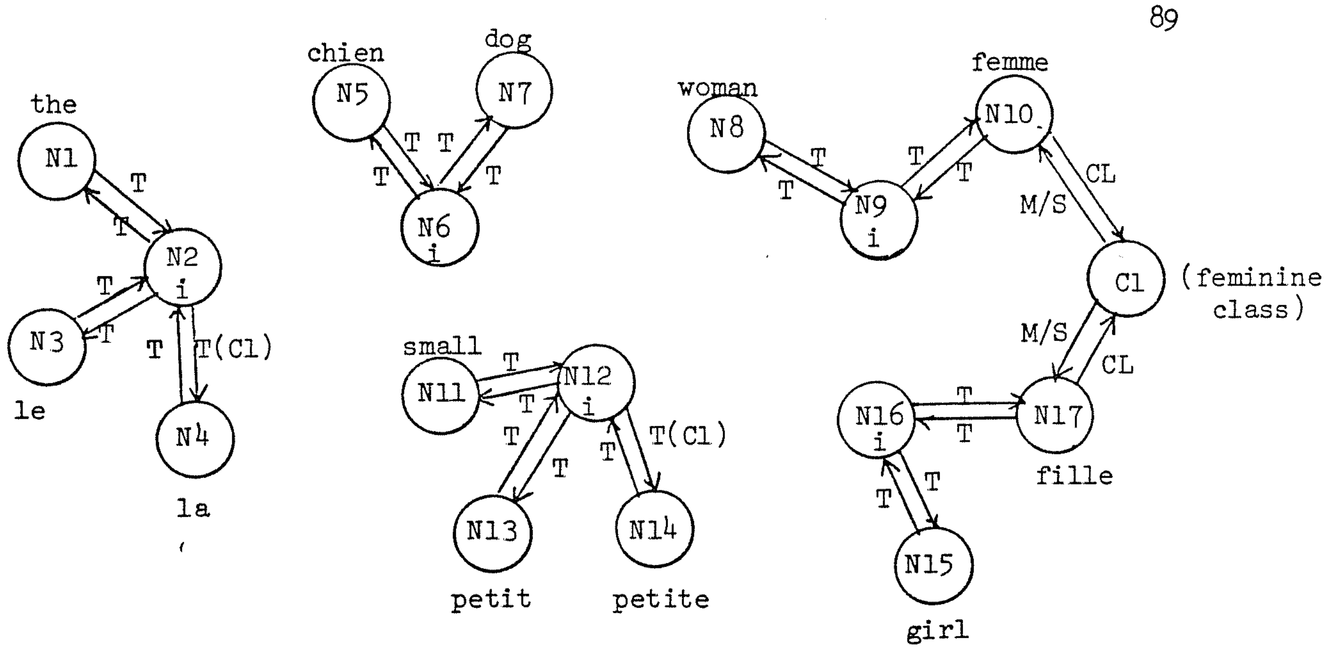
After all the unknown hunks in the work list have been learned, then METQA discards them and concentrates on those segments which were recognized well enough to have at least nonterminal intersections with the transformed feedback. Any permutations involving the newly learned hunks will be dealt with on a future encounter, when the program feels more sure of itself (and the unknown hunks have graduated to the status of intersecting paths through memory).

Next METQA finds and labels the partners for any intersections which were out of place and therefore unclaimed before. Since the construction of the paths includes their histories and intersections, this task is merely one of inspection and search along the work list. The nonterminal intersections thus matched are learned by altering memory as explained before, so that the next time that situation occurs, the correct choice will be made. Finally all that remains to be learned are the rules governing any segmental permutations which were noted between the input response and feedback strings. This last stage of learning for the translation task will be discussed in Chapter 7.

7. Example Learning Sequence

To conclude our discussion of transformation learning, let us consider the example input sequence in Figure 6-6. Starting with an empty memory, the example shows how the illustrated net segment can be learned. The initial nodes and links are built up in a fairly straightforward manner. When METQA does not recognize some string segment, it is enclosed in parentheses and marked U (for "unknown") in the response. Contextual link restrictions are not used until there is some need to distinguish between multiple links. Then, as shown in line 4, METQA extracts or creates a context class from the current cover in order to categorize the cases where this new link should be followed. Thus node N10 for "FEMME", the only other word in the input cover, becomes the sole member of a new context class, C1. Then the new form "LA" receives a new node N4, but the link to it is restricted to situations involving the context class of C1 (feminine) words.

Learning is again straightforward until line 6 of the example. There a new transformation of "SMALL" appears, and METQA must decide



Sample input sequence showing how the above net segment might have been learned:

	Input	Response	Feedback	Learned Behavior	New Nodes
1.	THE	U(THE)	LE	THE ↔ LE	N1,N2,N3
2.	THEDOG	LE U(DOG)	LECHIEN	DOG ↔ CHIEN	N5,N6,N7
3.	FEMME	U(FEMME)	WOMAN	FEMME ↔ WOMAN	N8,N9,N10
4.	THEWOMAN	LE FEMME	LAFEMME	THE $\dashv$(Cl)→ LA	N4
				FEMME $\in$ Cl	Cl
5.	PETIT	U(PETIT)	SMALL	PETIT SMALL	N11,N12,N13
6.	SMALLWOMAN	PETIT FEMME	PETITEFEMME	SMALL $\dashv$(Cl)→ PETITE	N14
7.	GIRL	U(GIRL)	FILLE	GIRL ↔ FILLE	N15,N16,N17
8.	SMALLGIRL	PETIT FILLE	PETITEFILLE	FILLE $\in$ Cl	

Figure 6-6. Example learning sequence.

how to categorize the situations where this new form should be used. The only other word in the cover, "FEMME", already has a contextual category assigned to it, so METQA uses that class C1. A node N14 is built for the new form, "PETITE", but the link to it is restricted to contextual situations involving class C1.

One might question the advisability of the above technique of generalizing usage classes or categories across different input situations. In general, it seems to be a good practice; as in the example, French feminine nouns act as French feminine nouns in a wide variety of grammatical situations. If some category is used too broadly, however, the resulting bad response will cause new categories to be learned and used, and the incorrectly assigned class memberships will be downweighted and ultimately discarded.

Continuing with the example, in line 8 METQA receives the input "SMALLGIRL". Since the "girl" nodes have no class membership yet, the "SMALL" path is forced to take the restriction-free link to node N13 ("petit"). Feedback shows that the correct form was "PETITE" from node N14. Since somehow the path was supposed to traverse a link restricted to context of class C1, METQA hypothesizes that perhaps the current context really is in the same category C1; METQA just did not know it yet. For this reason, node N17 ("fille") is made a member of class C1, so that on future encounters "FILLE" will occur with the feminine form of "SMALL".

Thus we see how METQA can learn and generalize from experience, and in so doing get concepts of the similarity in behavior of whole groups of words in certain circumstances. Different forms of any word can be learned, along with the different situations in which to use them. Of course, what METQA learns depends strongly on the input and feedback training sequence chosen by the human trainer, who can alter the planned sequence or pause to repeat, if METQA stumbles in its learning. As METQA categorizes more and more words, the classes are not always intuitively pleasing. It is still interesting, however, to watch how a learning "intelligence" structures the language it experiences from the human trainer.

CHAPTER 7

THE LEARNING AND USE OF PERMUTATION RULES

1. Description of a Permutation Rule

Probably every person who has ever learned more than one language has noticed that the word orders of different languages are not identical. METQA, too, must learn and be able to reproduce these different word orders. Thus in the final phase of learning from feedback, METQA learns rules to govern any segmental permutations which were noted between the input and feedback strings. The goal is to categorize the situation in which a word order difference occurs and describe the actual permutation so that it can be effected in the future.

A _permutation rule_ (p-rule) consists basically of an ordered sequence of class lists or "slots", each of which has a position number associated with it. The precursor of the p-rule is the "restructuring rule" of the Uhr programs [1964]. A very simple example of a p-rule is:

Rule0: C(French adjectives)(2), C(French nouns)(1).

This means that if a member of the class of French adjectives is found immediately preceding a member of the class of French nouns, then the (slot 1) adjective is moved to position 2 and the (slot 2) noun is switched to position 1. Each class list is a disjunctive set of one or more class options (weighted for learning evaluation);

each option is a conjunctive list of one or more usage classes. For convenience, throughout this chapter we will omit any indication of the weights on each option. The construction of these class lists will be explained in more detail as we proceed.

For a given slot of a permuation rule to match an input, all the classes listed in any one of its class options must be found on the corresponding input node's usage class membership list. If all the slots of a permutation rule are thus matched by consecutive input hunks, then the p-rule is executed; that is, the segment matching each slot is moved from its original position in the sequence to the position indicated by the slot's position number.

These permutation rules, corresponding to the transformations in transformational grammars [Harris, 1964; Chomsky, 1964, 1965], are necessary for the treatment of discontinuous morphemes (such as ne...pas) and for dealing with differences in order between different languages. But no p-rules are built into the program; they are learned and reweighted according to their behavior, just as the other relationships between memory nodes are learned. When a permutation of segments is noticed between METQA's response and the feedback string, the appropriate p-rules are inferred from the input and feedback string segments. Whenever possible, METQA attempts to combine and generalize existing rules to handle new cases of segmental permutation. The goal is to arrive at a minimal set of rules which adequately govern all the permutations necessary for the language(s) in use.

2. Preliminary Sketch of how a Pair of p-rules is Learned

The reader should recall from Chapter 6 that a learning work list is built to explicitly associate the input and feedback strings. METQA uses this list first to learn transformations and then to learn any permutations discovered. Let us assume that METQA has already learned all the necessary transformations and that the work list now contains only the path intersections, ordered as they appeared. Suppose that there is indeed a segmental permutation involved, which METQA will now attempt to learn.

First, METQA must isolate the permuted region; that is, the group of segments which were permuted must be delimited and extracted from the work list for convenient inspection. Then the numerical "order" of the permutation is determined. For the simple example Rule0 in the preceding section, the numerical order is (2,1), since two segments are switched. Using this permutation order as an initial distinguisher, the program searches through all permutation rules already in the permanent memory for one which could reasonably apply to the current situation. If a given rule permutes segments to the correct order, then the program attempts to match the class requirements of each of the rule's position slots against the class memberships known for the corresponding segment in the region under inspection. If at least one slot matches (this requirement can be made more stringent), then the rule is assumed potentially applicable to the current permuted region. For example, the sample Rule0 mentioned earlier would be considered potentially applicable to some newly discovered permutation if the noun slot matched, but no class memberships were yet known for the other segment. Hence METQA proceeds to alter the given rule so that it could have been used to produce the desired permutation on the current region. This alteration procedure will be described shortly in Section 2.2. But first, let us consider the case where there is no existing rule which is applicable to the permutation currently being learned.

2.1 Forming a new p-rule

If no existing rule had the correct permutation order, or if no rule of the correct order was judged potentially applicable, then the program builds a whole new p-rule by induction from this input. In order to characterize this situation which requires permutation, METQA must use the appropriate segment of the input string to produce a class list for each position slot in the p-rule. If possible, existing class membership information is used to set up the class requirement list for a slot. However, suppose a particular input segment has no class memberships listed in its terminal node because there has been no need to distinguish or characterize it before. Then in order to build the needed class requirement to describe this situation in the p-rule, the program sets up a new usage class and designates this segment's node as its first member. The new class then becomes the requirement for the particular p-rule slot. Finally the p-rule is assembled, the class list for each slot being associated with a position number as determined by the permutation order.

To illustrate the construction of a new permutation rule, let us use the simple example in Figure 7-1 with intuitive class names for easy understanding (METQA will generate its own internal names). For convenience, we will assume that METQA has already learned all the words involved. Suppose METQA receives the input string "HEBREAKSTHEGREENCUP", transforms it, and outputs the response "IL BRISE LA VERTE TASSE". The feedback string received is "ILBRISELA TASSEVERTE". METQA finds terminal intersections for each segment, confirming its knowledge of the words involved.

But there is a difference in the word order between response and feedback. METQA isolates the permuted area and sees that "VERTE" + "TASSE" should be permuted to "TASSE" + "VERTE". Now suppose "verte" is a member of the classes C(French) and C(adjective); let "tasse" be a member of C(French), C(noun), and C(feminine). Then METQA would use these memberships to build the rule:

Rulel: order(2,1)/[C(French).and.C(adjective)](2),
[C(French).and.C(noun).and.C(feminine)](1).

This means that a French adjective followed by a French, feminine noun would be permuted so that the adjective is in position 2 and the noun in position 1. We will call this newly constructed p-rule Rulel, so that we can refer to it later.

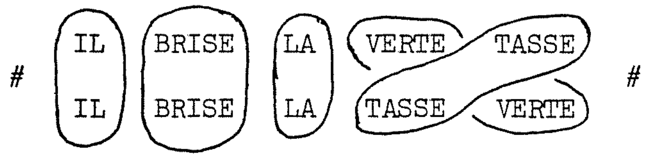
2.2 Altering an old p-rule

Now we have seen how new p-rules are built. Let us return to

Input: HEBREAKSTHEGREENCUP

Response: IL BRISE LA VERTE TASSE

Feedback: ILBRISELATASSEVERTE



Relevant class memberships:

verte ϵ C(French), C(adjective)

tasse ϵ C(French), C(noun), C(feminine)

Resulting new rule:

<u>Rule1</u>: order(2,1)/[C(French).and.C(adjective)](2),
[C(French).and.C(noun).and.C(feminine)](1).

Figure 7-1. Learning a new permutation rule.

the situation where METQA is searching for an existing rule which might apply to some permuted area isolated in the current input. If some existing rule R permutes segments to the correct order, and the current permuted area satisfies the class list of at least one p-rule slot, then METQA decides that R is applicable and proceeds to alter it so that it would have correctly permuted the current string region.

This alteration of an old p-rule consists of adding a new option to the class list of any p-rule slot which was not satisfied by the input segments. If possible, existing class memberships are used. The new class option is the conjunction of all those classes of which the particular segment's node is a member.

Suppose some input segment has no class memberships to use. Then METQA follows the usual procedure of creating a new usage class with this segment as its first member. This new class is then added as the new class option needed to make the p-rule fit the current situation.

Before adding a new option to the class list for a given slot, METQA first tries to generalize the slot's class requirements as follows. If the new option has (arbitrarily) two or more classes in common with some older option, both the options are consolidated into one more general option consisting of only the classes common to the two. Thus the program attempts to derive more generally useful permutation rules.

Let us illustrate the alteration of an old p-rule with another simple example in Figure 7-2. Suppose METQA receives the input string "AHAPPYDOGJUMPSONTHEBOY". After recognition and transformation, the program outputs the response "UN HEUREUX CHIEN SAUTE SUR LE GARCON". The feedback is "UNCHIENHEUREUXSAUTESURLEGARCON". On comparison, METQA discovers that all the words were transformed correctly, but that there is a difference in word order between response and feedback. After isolating the permuted areas, METQA determines that "HEUREUX" + "CHIEN" should be switched to "CHIEN" + "HEUREUX"; a segmental permutation of order 2,1 must be learned.

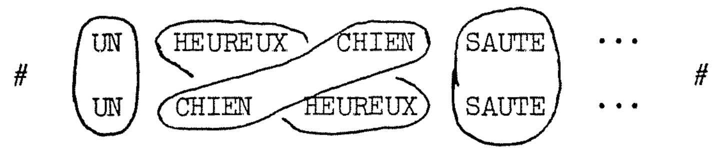
Before building a new p-rule, METQA searches through the permanent directory list of the rules already in existence for one which has the same permutation order and might (with modification) govern this permutation. Rule1 (learned in the example of Figure 7-1) has the needed permutation order; it, too, switches around two segments. Then METQA must check the class requirements of each slot in Rule1 against the current input. Now suppose "heureux" has class memberships in C(French), C(adjective), and C(goodthings) as shown; let "chien" belong to C(French), C(noun), and C(animal). The first slot of Rule1 requires a member of C(French) and C(adjective); obviously, "heureux" satisfies this slot. The second and final slot needs a node which has memberships in C(French), C(noun), and C(feminine); "chien" cannot satisfy the full requirement.

Since at least one of the slots of Rule1 was satisfied, METQA decides to generalize the p-rule to apply to the current situation, too. The first slot needs no alteration, since it already matches

Input: AHAPPYDOGJUMPSONTHEBOY

Response: UN HEUREUX CHIEN SAUTE SUR LE GARCON

Feedback: UNCHIENHEUREUXSAUTESURLEGARCON



Relevant class memberships:

heureux ϵ C(French), C(adjective), C(goodthings)

chien ϵ C(French), C(noun), C(animal)

Previously existing rule:

<u>Rule1</u>: order(2,1)/[C(French).and.C(adjective)](2),
[C(French).and.C(noun).and.C(feminine)](1).

Rule after modification:

<u>Rule1</u>: order(2,1)/[C(French).and.C(adjective)](2),
[C(French).and.C(noun)](1).

Figure 7-2. Altering an old permutation rule.

the current input segment. The second slot must be altered to include "chien" as well as French feminine nouns.

Using the existing class memberships of "chien", a new option can be added to the slot to yield the following class list:

```
[(C(French).and.C(noun).and.C(feminine)).or.
(C(French).and.C(noun).and.C(animal))].
```

However, before replacing the old option with this new list, METQA checks to see if the requirements can be generalized. The new option is compared with the old, and an overlap of two classes is discovered. Consequently, both options will be replaced by a new combined option consisting of only the overlap classes. Hence the generalized class list for the slot will be simply

```
[C(French).and.C(noun)].
```

Thus METQA alters the old p-rule Rule1 so that it can now be represented as

```
Rule1:    order(2,1)/[C(French).and.C(adjective)](2),
          [C(French).and.C(noun)](1).
```

Very simply, this means to METQA that in French, adjective + noun permutes to noun + adjective. Of course this rule has exceptions in the real world; METQA will stumble onto these later, and try to adapt to them by using the same techniques described above, developing new classes to handle these exceptions.

2.3 Complementary p-rules

When the alteration or construction of a rule is finished, the memory nodes involved with the current input are adjusted so as to indicate on future encounters that use of this p-rule should be considered. The program is still not finished with its learning, however. Since translation tranformations are two-directional relationships, each permutation rule must have a complementary p-rule to handle segmental permutations during transformations in the opposite direction--from the (current) target language to the (current) source language. The usage classes may be quite different for different languages, depending upon to which grammatical concepts (gender, case, etc.) are represented in the external form of the language. In addition, the numerical permutation order may differ for opposite directions, as can be seen in the following example: ABC $\rightarrow$ CAB has permutation order (2,3,1), while CAB $\rightarrow$ ABC has order (3,1,2).

The complementary rules are stored together, so that if the program has just finished altering an existing rule to fit the input, it has only to retrieve the complementary rule and go through the same alteration process to make this second rule applicable to the feedback segments. Similarly, if the program has just finished building a new p-rule by induction from the input, the complementary rule must now be built in the same fashion by induction from the feedback segments. The resulting pair of p-rules is stored together in the list of all existing p-rules, ready to be tested, reweighted, and evaluated according to future experience.

Finally at this point, the program has learned all it can from the input and feedback strings, and is ready to accept a new input from its trainer. Perhaps it has even "learned" too much, and some of its hypotheses are erroneous, but the constant reweighting of all the program's knowledge according to its behavior will tend to cause the downweighting and eventual rejection ("forgetting") of bad hypotheses and the achievement of a more "intelligent" memory.

3. How a p-rule is Used in Transformation

The preceding sections have shown how permutation rules are learned; now let us look at how they are used when METQA is transforming some input string.

As indicated in Chapter 5, Section 6, a word node "remembers" if it is ever used to learn or alter a permutation rule. Then any time that same word is used again, it suggests to METQA that the p-rule might be applicable for this new input. At that phase of the transformation cycle where the suggested p-rules should be tried, the program tries to apply every rule indicated to every cover being processed.

Starting with the first cover, METQA tries to apply each rule in turn. For each p-rule, the program steps through the cover from the beginning, segment by segment, looking for a path with the class memberships needed to satisfy some option of the class list for the first position slot of the particular p-rule.

If the first slot's requirements are satisfied, then METQA pauses here and checks the immediately following segments to see if they satisfy the requirements of the succeeding slots of the p-rule. If the succeeding slots are not satisfied, then METQA resumes stepping through the cover, still trying to match the same rule.

A successful match for all the slots, however, means that METQA must perform the indicated permutation before continuing with the same cover. This is accomplished by moving the actual paths in the cover around to the positions stated in the p-rule. A p-rule can apply more than once to a single cover, so after performing the permutation, METQA continues down the cover still trying to apply the same p-rule. Each suggested p-rule is tested for applicability to the cover; then METQA moves on to the next cover and tries to apply each rule again.

In order to clarify the permutation process, let us consider the simple example in Figure 7-3. Assume that METQA still knows the p-rule Rule1, which was developed in Figure 7-1 and modified in Figure 7-2. Also assume that all the words involved are previously known, with intuitively-named class memberships as indicated.

Suppose METQA is asked to translate the input "ADOGLIKESTHE HAPPYBOY". Since all the words are known, METQA recognizes all the segments, forms the paths shown, and builds a complete cover CVR.

The paths go through the cycle of procedures, transforming node by node through memory. No permutation rules are implied or suggested until Path2 reaches the "chien" node and Path5 reaches

Figure 7-3. Application of permutation rule.

Permuation rule:

```
Rule1:    order(2,1)/[C(French).and.C(adjective)](2),
                     [C(French).and.C(noun)](1)
```

Input: ADOGLIKESTHEHAPPYBOY

Paths	Path Strings	P-Rules Suggested
Path1	A → UN	
Path2	DOG → CHIEN	Rule1
Path3	LIKES → AIME	
Path4	THE → LE	
Path5	HAPPY → HEUREUX	Rule1
Path6	BOY → GARCON	

Class memberships assumed:

un ∈ C(article), C(masculine)
le ∈ C(article), C(masculine)
chien ∈ C(noun, C(French), C(animal)
aime ∈ C(French), C(verb), C(goodthings)
heureux ∈ C(adjective), C(French), C(goodthings)
garcon ∈ C(French), C(masculine), C(noun)

Intermediate cover during first cycles:

CVR: Path1 -Path2 - Path3 - Path4 - Path5 - Path6

or, externally: UN CHIEN AIME LE HEUREUX CARCON

Application of permutation rule:

METQA tries to apply Rule1 because suggested by nodes "chien" and "heureux".

Rule match at Path5 - Path6; permute to order (2,1).

Final cover after successful application of Rule1:

CVR: Path1 - Path2 - Path3 - Path4 - Path6 - Path5

Response: UN CHIEN AIME LE CARCON HEUREUX the "heureux" node. Since each node has previously been involved with the permutation rule Rulel, each suggests that Rulel might be applicable to this cover. Consequently, during the next phase of that cycle, METQA tries to apply Rulel to CVR.

The first position slot of Rulel specifies matching a path whose node(s) have membership in classes C(French) and C(adjective). Starting from the beginning of the CVR, METQA looks for such a path. Note that although the "chien" node suggested Rulel, the needed adjective is not there before it and so Rulel does not apply at that position. When at Path5 a French adjective is found, METQA pauses to try to match the succeeding p-rule slots. The "garcon" node of Path6 satisfies the second and final slot's requirements for membership in classes C(French) and C(noun), so METQA decides that Rulel should be applied to the Path5 - Path6 area of the cover.

After the cover paths are permuted to the prescribed order 2,1, then METQA continues to try more rules. There are no more rules and no more covers, however, so the cycle is finished. Since all the paths have reached target terminal level, METQA has finished the translation and outputs the permuted Response: UN CHIEN AIME LE GARCON HEUREUX.

4. The Role of the Human Trainer

In this chapter we have seen how METQA learns permutation rules at the final stage of transformation learning, and how these rules are used in the transformation process. The exact form of the resulting set of p-rules is a function of the experience received from the human trainer. Given any instance of segmental permutation, METQA will construct new classes if necessary and will try to learn a pair of p-rules governing that particular permutation. If the trainer is unaware of the internal class structure that METQA has learned, this can lead to the learning of too many rules or "special case" rules.

For example, consider Figure 7-4, and suppose that no class memberships are initially known for "yellow", "green", "chair" or "book". Then if METQA receives Experience 2 immediately after Experience 1, METQA will build the two separate rules shown to govern (what we would view as) the same type of permutation. METQA is not currently programmed to combine two existing p-rules, so even if both Rule2 and Rule3 are later correctly generalized, METQA's permanent memory is cluttered with two rules which govern the same permutation.

Thus we see how METQA can benefit from a carefully chosen input sequence. To the extent that the trainer understands METQA's procedures, observes METQA's behavior, and infers what METQA has already learned, then he can present permutations in a sequence which will produce more general and intuitively pleasing p-rules and classes, and <u>fewer</u> of them, than if permutations are learned from a haphazard input sequence.

<u>Experience 1</u>

Input1:	YELLOWBOOK
Response1:	JAUNE LIVRE
Feedback1:	LIVREJAUNE

<u>Resulting new class construction:</u>

C1	contains	"jaune"
C2	contains	"livre"

<u>New rule:</u>

Rule 2: order(2,1)/[C1](2), [C2](1)

<u>Experience 2</u>

Input2:	GREENCUP
Response2:	VERTE TASSE
Feedback2:	TASSEVERTE

<u>Resulting new class construction:</u>

C3	contains	"verte"
C4	contains	"tasse"

<u>New rule:</u>

Rule 3: order(2,1)/[C3](2), [C4](1)

Figure 7-4. Construction of two rules. (Note that only half the constructions are shown; complementary rules and their classes are omitted.)

CHAPTER 8

THE LEARNING OF INFORMATION

1. Conditions for Learning

When learning a new memory, METQA's early experience must be devoted to acquiring a working vocabulary. Only after all the words of an input have been learned can METQA consider that input as a piece of information, as well as a string to be transformed.

There are three modes under which the program can operate. First, inputs from the human trainer can be considered simply as strings to be transformed to some target form for output. In this mode, METQA learns new words and their different transforms, contexts, and classes in order to translate input strings.

When the trainer feels that METQA has an adequate vocabulary, he can set a flag so that future inputs are considered as potential information. That is, METQA performs the usual recognition and transformation of the input string and outputs a translation. Feedback strings are processed as usual. However, if the input was completely understood (i.e. recognized), METQA also represents the input internally as a fact and adds this fact to the memory's store of information.

This information learning mode can also operate without the translation. That is, completely understood input strings can be simply accepted and incorporated into memory without the need for either translation or answer.

In the third mode of operation, the input received from the trainer is a _question_ to be answered using the information METQA has learned. In this mode, METQA consults the facts in its memory in an attempt to produce the best answer possible, given the current state of its knowledge.

2. How a Fact is Learned

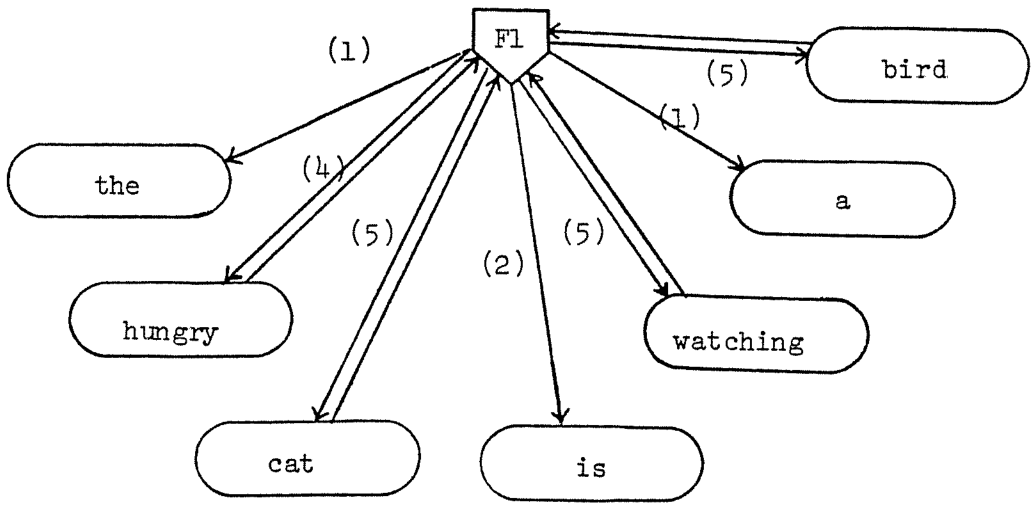
Any input string which is to be learned as information will be called a _fact_. A _fact_, then, is not necessarily a fact in the usual sense of the word; it is simply any input which is learned as a basic piece of information. Any fact is assumed to be true until repeated bad responses reduce (the weight of) its credibility.

Some information processing systems have different classifications of information, based perhaps on whether the item is a piece of specific data (e.g., "Tom is six feet tall") or a more general, universal truth (e.g., "A foot is 12 inches"). Examples are the STUDENT system with its "global" and "local" information [Bobrow, 1968] and the CONVERSE system with its "general" and "specific" facts [Kellogg, 1968, 1971]. For METQA, however, all factual information is incorporated in the same way into the semantic memory net which serves METQA as the permanent memory for all its tasks. This enables the use of very general routines for both the learning and the use of information in the net.

Just as it used character symbols as the basic units which combined to form the words in the terminal nodes of memory, METQA uses recognized words as the basic combining units with which to form facts. An input is learned as a fact by attaching a link to each component word in the input from a single node which functions in memory as the "fact node". In accordance with the basic conditions of the program (cf. Chapter 2, Section 1), the facts are unstructured strings whose meaning is the combination of their relationships in the memory net. No linguistic or logical structure is ascribed to any fact.

The fact node has ordered description links to the idea nodes of all the fact's component words. The description links are weighted according to the relative importance of the particular word to the fact. METQA determines the importance of each word by maintaining a frequency count of its occurrence throughout METQA's experience. In general, the very common or "function" words which have mostly a grammatical function (e.g. articles, conjunctions, prepositions) are given lower weights when building a fact, while less frequent "content" words having higher information content are given greater weight. For further discussion of this idea, see the paper by Simmons, Klein and McConlogue [1964].

Of all the fact's component words, only those words with the relatively high information contents are in turn linked <u>back</u> to the fact node. Thus for each content word, a combination link connects its idea node to the fact node; the weight of that link indicates the word's relative importance in the fact. There are no links from the common, function words to the facts which contain those words, because the words are not meaningful enough to be good indicators of the information in the facts. In addition, there would simply be too many facts indicated by each particular function word.



Links from F1: description links

Links to F1: combination links

Figure 8-1. Sample fact node.

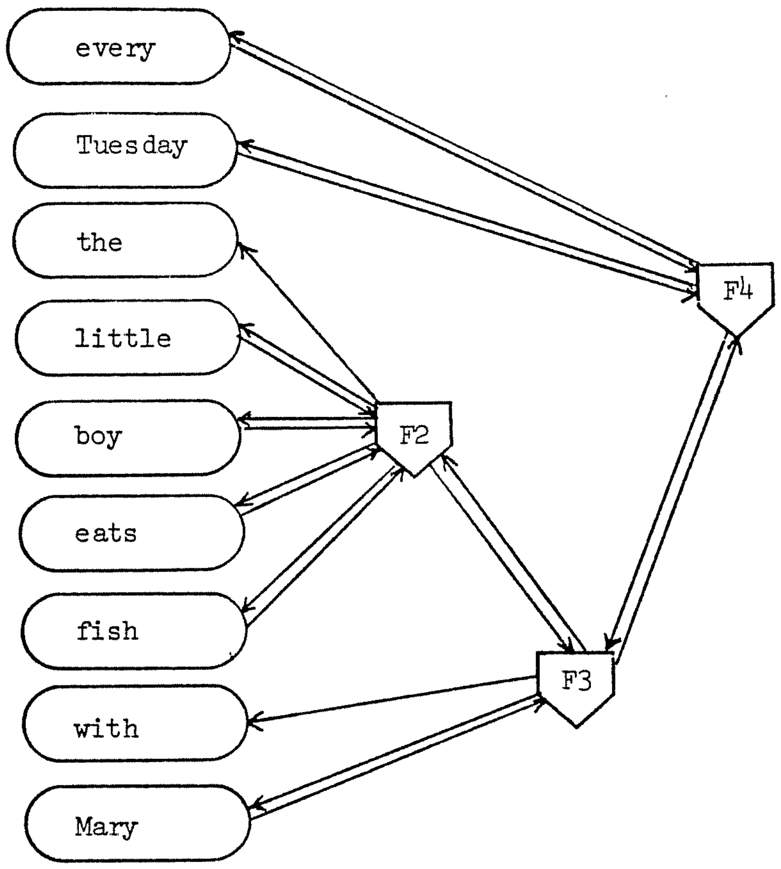
For an example of the links and weights involved in a fact node, see Figure 8-1. The word nodes shown are assumed to be idea nodes. Each description link has an importance weight shown in parentheses; combination links are here assumed to have the same weights as their partner description links.

3. Combination of Facts

In order to make efficient use of memory space and to relate to and make use of previously learned information, METQA combines new facts with old ones whenever possible. For this reason, fact node structures can be nested or can have class or group nodes as their component nodes. That is, a component node pointed to by a fact node's description link may be one of three different types: 1) a simple idea node for some word, 2) a class or group node with two or more member nodes, or 3) another fact node.

When there is a fact to be learned, METQA first searches memory to see if there is a similar fact to combine with. If there is an existing fact (EF) which matches some sufficiently big (typically, half or more of the total "word-weight"), connected subpart of the new fact (NF) to be learned, then the program will build a _nested fact_. That is, EF will still exist as a separate fact, but it will also combine with other nodes to form the larger new fact NF. The fact NF will have description links down to all of its component nodes, one of which will be the complete, self-contained fact EF. This nesting can be arbitrarily deep, on any node.

An example of a nested fact is shown in Figure 8-2. Assume the nodes labeled with words are the idea nodes of those words. The nesting is easiest to learn if the facts appear in the order F2, F3, F4, but it is also possible to nest downward.



Links to left: description links

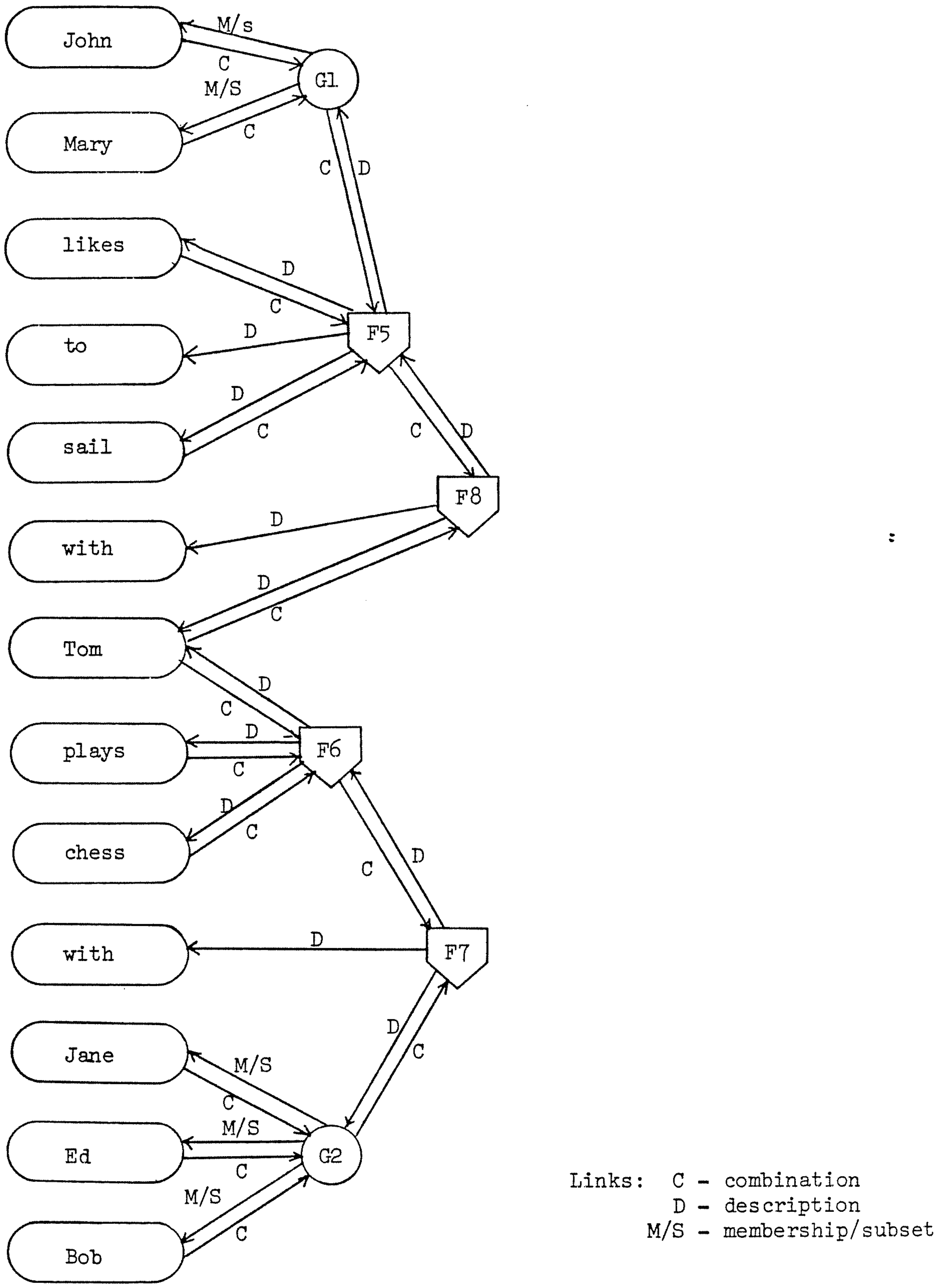
Links to right: combination links

Figure 8-2. Example of a nested fact.

If METQA finds two facts which are identical except for one (set of) node(s) in the same position of each, then the two facts can be combined, with a group node at that point of difference. A group node is structured like a class node in memory, with pointers down to all its member nodes. In contrast to the class links connecting members to a class node, however, here we have combination links connecting members to the group node. Figure 8-3 shows examples of "grouped" facts. Again, the nodes labeled with words are assumed to be idea nodes.

There is not necessarily a unique sequence of input facts to result in a particular fact node structure. In Figure 8-3, fact F5 may have first been learned as "Mary likes to sail". If so, the next development with F5 could have been the formation of either the group node G1 (by learning that "John likes to sail") or the fact node F8 (by learning next that "Mary likes to sail with Tom"). The order could have been entirely different, but the resulting structure holds the same information.

When a group node is formed in a fact structure, the members are assumed to have an inclusive-or relationship; the stated group fact is true for the whole group of nodes, but perhaps not necessarily for all of the members at the same time (or as the same fact). Hence in Figure 8-3, fact F5 can be read three ways: 1) "John likes to sail", 2) "Mary likes to sail", and 3) "John and/or Mary likes to sail". For this reason, a fact is not allowed to include more than one group node, since erroneous combinations of group options could



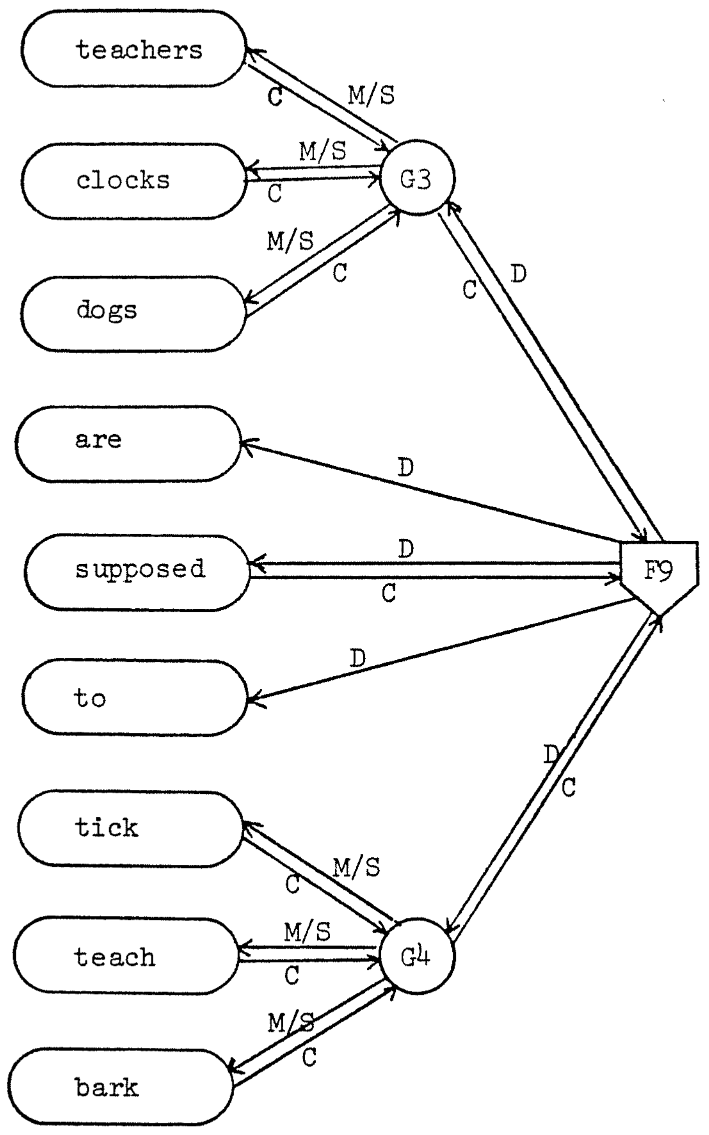
Note: Both pointers to "with" point to the <u>same</u> node.

Figure 8-3. Sample facts with group nodes.

certainly occur. Consider the fact node F9 in Figure 8-4. Although this complex fact may have been learned with very reasonable statements, METQA could now conclude that "Teachers are supposed to tick" and "Clocks are supposed to bark".

As we have seen, METQA's goal is to combine facts as much as possible, but with caution in order to maintain information consistent with the truth. Currently there are strong constraints on the combination possibilities during a single fact learning operation. Two facts can be combined only if one can be matched by a connected subpart of the other, or if the sentences are identical except for one set of nodes in the same position of each. Even with these constraints, more complex fact structures can be built up over time if the sentences match closely, as shown in Figure 8-3.

METQA's main approach is conservative and cautious, however. The program can always easily enter a whole new fact into memory, but it is more difficult to recover from the bad information resulting from too liberal a combination of facts. The unlearning of badly combined information will probably require "forgetting" the whole combination of facts after they have been downweighted enough times for causing bad behavior. Then METQA would have to relearn these facts separately when it sees them again.



Links: C - combination
D - description
M/S - membership/subset

Figure 8-4. Fact node with too many group nodes, yielding erroneous conclusions.

CHAPTER 9

ANSWERING QUESTIONS

1. Question Answering--An Interruption of the Transformation Process

After the program has learned and stored some information into its memory net, any input received may be a question to be answered, rather than a string to be translated. In the question answering mode (QA mode), METQA must consult the facts it has learned and try to produce the best answer available from its net. The reader should recall that the goal is to build in as little information as possible (see Chapter 2, Section 1); hence there can be no built-in linguistic information or logical procedures to aid METQA in this task. There is only the memory net and the class structures which METQA has learned.

The question answering procedure is, in a sense, an interruption of the regular translation procedure with a replacement of object strings. The question string is recognized and transformed as usual, but only to the level of the idea nodes which have access to the information portion of the net. At that point, the question answering procedure takes control and tries to produce the best answer possible, given only the information which METQA has learned. Then after an answer is produced, the program returns it to the usual transformation routines, where it is "translated" to external form for output.

2. Initiation of Answer Procedure

When the program is operating in QA mode, recognition and transformation of input segments proceed as usual. The main difference is in the path status (its location in the net) sought for termination of the cycle of transformation procedures explained in Chapter 5, Section 7.

Whereas in translation each path should reach terminal level in order to stop cycling, in the QA mode, each path should reach the level of its idea node. Only from the idea nodes of words can the facts in the net be reached. When each path in the question cover(s) reaches this desired level, then the transformation cycling terminates and control is passed to the question answering routines.

If more than one cover of the question has survived to this point, METQA arbitrarily chooses any one which contains no garbage characters. Just as when learning information, METQA must already know all the words involved when answering a question; therefore it will not attempt to answer a question when there is no complete cover.

3. Finding Content Words

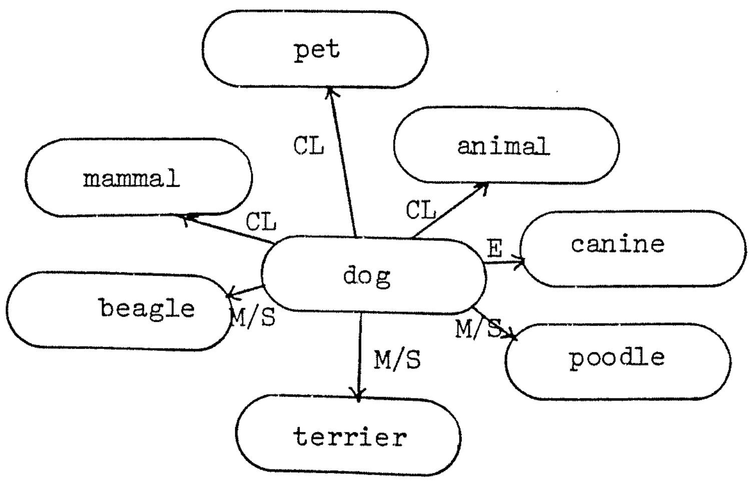
In order to produce an answer, METQA must first pick out from the question the important words with which to retrieve information. Using the assumption that the significance of a word varies inversely with its frequency of occurrence (as explained in Chapter 8, Section

2), METQA ignores the function words and considers only the content words important enough to choose relevant information. Hence all the content words of the question are put into a list of relevant words; this list will be used later to extract information from the net.

4. Use of Extended Meanings

As the reader may recall, in Chapter 3 we discussed the "extended meaning" of a word in METQA's memory as a means for extracting a richer selection of information relevant to that word than the word alone would yield. The extended meaning of some word W includes not only the node for W itself, but also those word nodes connected to W by relationships such as class membership or set inclusion (superclasses and members or subsets) and usage equivalence. Thus if we have a question including the word "dog", for example, METQA can retrieve more information by using the extended meaning, which might include all the idea nodes shown in Figure 9-1. The new nodes found by extension are added to the list of relevant words which METQA will use to gather information. This extension operation thus gives a wider selection of relevant information from which to select and assemble an answer.

The use of extended meanings is an optional feature (set by interactive command), which the trainer may or may not wish METQA to use at a given time. We will assume for this discussion that the flag has been set so that METQA will use extended meanings of



Links used:	CL -	class membership in
	M/S -	member/subset of this class
	E -	equivalence; can be replaced by

Figure 9-1. Sample extended meaning.

words whenever possible.

5. Finding Relevant Facts

Using all the relevant words extracted from the question and its extension words, the program next starts to gather information for use in assembling an answer to the question. To do this, METQA uses a technique very similar to the coordinate indexing of the SYNTHEX system of Simmons, Klein and McConlogue [1964]. One by one, each relevant word is consulted for a list of facts in memory which are relevant to that word. That is, which facts actually use this word node in their description? All the facts so indicated are gathered into one list of relevant fact nodes. This list comprises all the first level information known by METQA which might possibly be relevant and useful in answering the question. Given these fact nodes selected by the question as a data base, METQA will pursue every possibility which could lead to an answer.

6. Building Fact Chains

In order to develop related facts into a logically connected answer, METQA assembles structures called fact chains. A _fact chain_ is a list of fact nodes of which the first fact is indicated by the words in the question (including their extended meanings), and in which each succeeding node is _linked_ to by words (or their extensions) from the preceding fact. A _link word_ is a word shared by two adjacent facts in a chain, either directly or by a node of its extended meaning. A link from fact node N1 to N2 in some chain CHN would be any word (or words) which appeared in the chain for the first time in fact N1; N2 is some other fact which also uses that word (or words). That is, when some new word NW appears in the realm of discourse, say in fact node N1, then METQA notices it and sees if this newly relevant word can introduce new and useful information. The new word NW is therefore consulted for a list of the facts relevant to it, and the new facts (such as node N2) are used to extend this fact chain CHN and are added to METQA's list of relevant fact nodes for this question.

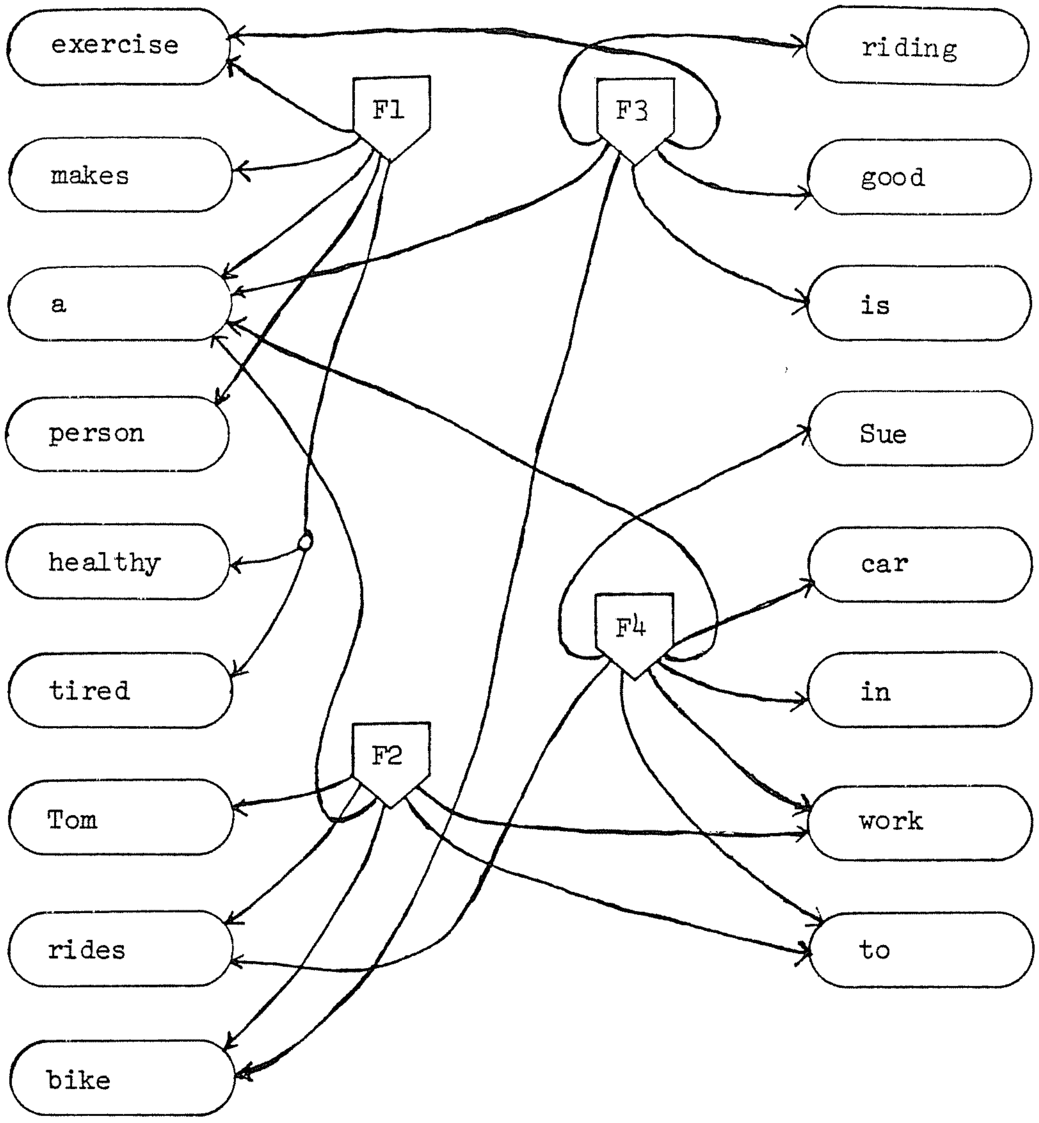
Perhaps an illustration will make the chaining process clearer. Suppose the four facts shown in Figure 9-2 have been learned by the program, and that METQA receives the question "Is Tom healthy?". Starting from the only fact suggested by the content word "Tom", the chaining process shown yields two fact chains. Linking words are shown in brackets; if a new word can find no new fact, it is followed by a "#". Of the two resulting fact chains, the one which METQA would use to produce an answer (we will see why shortly) is CH1: F2(Tom rides a bike to work) → (bike) → F3(Riding a bike is good exercise) → (exercise) → F1(Exercies makes a person (healthy/tired)).

As indicated in Figure 9-2, fact chains are formed to pursue all branches offered by their link words. Thus when two facts (F3 and F4) can be linked to from fact F2, METQA builds two chains so that each possibility can be explored.

7. The Coverage of a Chain

In order to judge the relevance of a fact chain and the completeness of its information in providing an answer, the program keeps a record for each chain of the parts of the question which are accounted for by that chain. We use the term coverage of a chain for this measure of the question segments "covered", or accounted for, by the chain. A content word of the question is said to be covered by a fact chain if that word (or a node of its extended meaning) has been used by a fact in the chain. In its current stage, METQA automatically assumes any function word of the question to be covered.

Figure 9-2. Building fact chains.



Facts in memory (content words underlined):

F1: <u>Exercise</u> <u>makes</u> a <u>person</u> (<u>healthy</u>/<u>tired</u>).

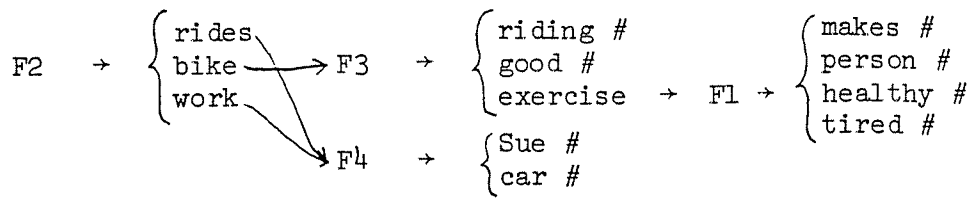
F2: <u>Tom</u> <u>rides</u> a <u>bike</u> to <u>work</u>.

F3: <u>Riding</u> a <u>bike</u> is <u>good</u> <u>exercise</u>.

F4: <u>Sue</u> <u>rides</u> to <u>work</u> in a <u>car</u>.

Question: Is <u>Tom</u> <u>healthy</u>?

Chaining from F2 (# means no new facts from here):



Sample chains starting from node F2 (links shown in parentheses):

CH1: F2 → (bike) → F3 → (exercise) → F1

CH2: F2 → (rides,work) → F4

Figure 9-2. (Continued)

The goal of each fact chain is to cover all of the question. If each content word in the question has been somehow accounted for by the facts in a chain, then METQA hypothesizes that these facts will include the answer to the question.

8. Fact Chain Intersections

In order to hasten the complete covering of the question, as each chain is extended from fact to fact it is constantly looking for a way to link with or intersect other chains. Two fact chains A and B are said to intersect if some fact node occurs in each chain, or if the link words from a fact FA in one chain, say A, connect it to some fact FB, which is present in the other chain B.

The intersection of two chains provides a set of connected facts, or we might think of it as a loop chain, such that the facts on either end contain words from the actual question, and the facts in between are associated with each other by means of sharing some of the same content words. This combining of chains and their respective coverage indicators is desirable because it usually results in a more complete coverage of the question. The goal is of course to find an intersection which yields a complete coverage of all the question.

For an example of how fact chains intersect, let us again consider Figure 9-2. Using the content words of the question "Is Tom healthy?", METQA would initiate two fact chains, say CHF2 from fact F2 ("Tom rides a bike to work") and CHF1 from fact F1 ("Exercise makes a person (healthy/tired)"). (Link words are underlined.) Neither chain covers the question completely, so the chains will be extended to other facts in order to gather more information.

Using the new words found in fact F2 ("rides", "bike", "work"), chain CHF2 can extend to either fact F3 ("Riding a bike is good exercise") or fact F4 ("Sue rides to work in a car"). Consequently, METQA follows both branches, extending CHF2 to node F3 and building a new chain CHF2A to extend to F4. Thus the two F2 chains can be represented as follows:

CHF2: F2 → (bike) → F3

CHF2A: F2 → (rides,work) → F4

Next, chain CHF1 must be extended. Using the new words for this chain found in fact F1 ("exercise", "makes", "person", "tired"), the only fact which can be linked to is F3 ("Riding a bike is good exercise"). Always watching for fact chain intersections, METQA discovers that node F3 is shared by both CHF2 and CHF1. Thus the chains are linked as follows:

$$\begin{tikzcd}
\text{CHF2:} & \text{F2} \arrow[r] & \text{(bike)} \arrow[r] & \text{F3} \arrow[dr, equal, bend left] & \\
\text{CHF1:} & \text{F1} \arrow[r] & \text{(exercise)} \arrow[rr] & & \text{F3}
\end{tikzcd}$$

Neither CHF2 nor CHF1 completely covered the question by itself, but now METQA checks to see if the two intersected chains together provide a complete cover of the question "Is Tom healthy?". In this case, both content words of the question are accounted for, so METQA concludes that CHF2-CHF1 contains an answer, and the chaining of facts is terminated.

9. Two Special Word Classes

To facilitate the use of interrogative words (such as "who" and "what") and negation words (such as "not" and "never") in questions, the human trainer must help METQA build up the classes of question words (which we will refer to as q-words) and negation words.

The reader should recall from previous discussions of class formation that METQA builds classes of words (i.e., of their nodes) automatically as a means of categorizing different input situations in order to choose from among alternate behavior possibilities. For example, a word class would be learned in an attempt to categorize the cases where "small" transforms to "petite" instead of "petit". These automatically formed classes can be used in a very general manner by the program, but they have the disadvantage of being anonymous; METQA has no way to know the nature of the contents of any class, and there is no name by which it can refer to such word classes as "feminine", "human", "plural", etc. (Certain planned remedies for this limitation will be discussed in Chapter 10, Section 4.)

However, in the question answering routines it seemed necessary to refer explicitly to the classes of question words and negatives as special types requiring special treatment. METQA might actually form these classes of words on its own; but since the groups would be anonymous, the program could not make special use of them. Thus it was necessary to make an exception to the rule that METQA must learn everything alone: two class names (for question words and negation words) are explicitly built into the program, and METQA can enter any words into these classes as instructed by the trainer. The use of negation words will be discussed later in Chapter 10, Section 9.2).

10. Processing Question Words

Suppose that METQA has learned some members of the q-word class. For every member of this class, METQA is taught by the trainer or can learn on its own some classes to which this q-word should belong. These class memberships are then used to decide which words in a fact might cover or account for any q-word which appears in a question.

Suppose the word "who" is known to be a member of the classes C(q-word) and C(person). Then whenever "who" appears in a question, in order to cover that word METQA must find in one of the relevant facts a word which is also a member of C(person). Similarly, if "when" is a member of C(q-word), C(time), and C(day), then some member of C(time) or C(day) must be found relevant in order to cover "when" in a question.

To illustrate, suppose the following classes have at least the members shown:

C(q-word) = (who,what,when,where,qui,quand,...)

C(person) = (Mary,who,sister,man,Tom,qui,...)

C(time) = (when,night,soon,early,noon,quand,...)

C(day) = (Tuesday,when,holiday,Sunday,...)

C(thing) = (what,table,tree,hand,...)

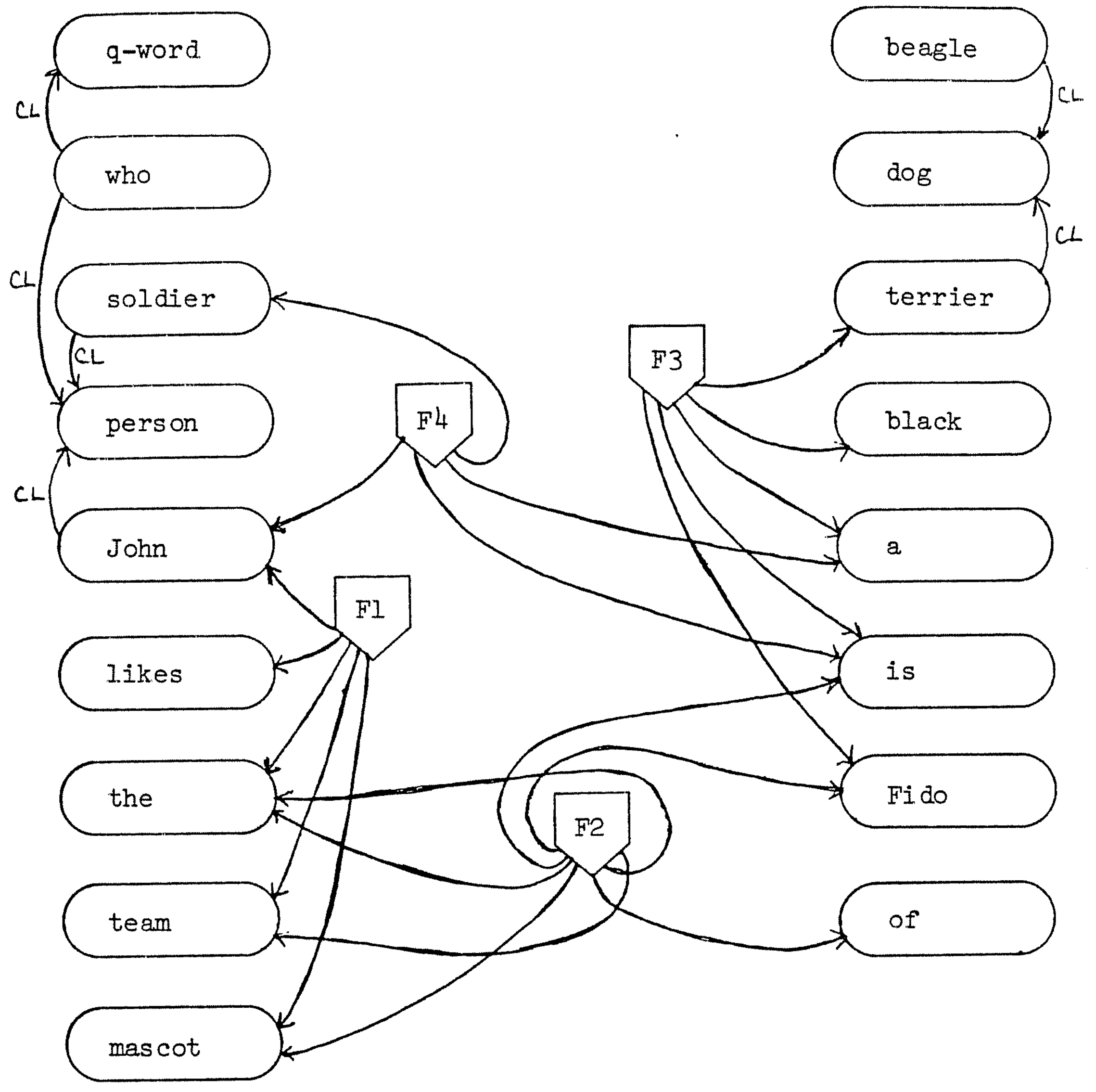
Given these class memberships, METQA would allow "sister" to account for "who" in a question, but would not allow "table" to. "Tuesday" could cover "when" in a question, but "tree" could not. Thus the question word procedures make strong use of all the class memberships METQA has learned.

Every question is checked to see if any of its words are members of the q-word class, and any q-words are flagged. Suppose a question contains some q-word. Then as each fact chain progresses to another fact node, METQA checks for the appearance of some new word whose class memberships qualify it to substitute for the q-word in the question. If it can, then the q-word is covered by that chain. In this manner, METQA can process any word which the trainer presents as a question word.

11. Processing a Sample Question

To illustrate the question answering procedures discussed so far, let us consider the example in Figure 9-3. Suppose that the memory net contains at least the facts and class memberships shown, and that METQA receives the question "Who likes a black dog?". The

Figure 9-3. Sample question process.



Facts (content words are underlined):

F1: <u>John</u> <u>likes</u> the <u>team</u> <u>mascot</u>.

F2: <u>Fido</u> is the <u>mascot</u> of the <u>team</u>.

F3: <u>Fido</u> is a <u>black</u> <u>terrier</u>.

F4: <u>John</u> is a <u>soldier</u>.

Figure 9-3. (continued)

Relevant class members/subsets:

C(q-word) = (who,...)

C(person) = (John,who,soldier,...)

C(dog) = (terrier,beagle,...)

C(animal) = (dog,...)

Question: Who likes a black dog?

Content words and their extensions: likes,black,dog,terrier,beagle, animal

Q-word to cover: who - C(person)

Chaining from question:

$$\begin{tikzcd}
 & \text{likes} \arrow[r] & \text{F1} \arrow[r] & \text{John} \arrow[r] & \text{F4} \\
 & \text{black} \arrow[ddddr] & & \text{person} & \\
 & \text{dog} & & \text{team} \arrow[dr] & \\
\text{Q} \arrow[r] & \text{terrier} \arrow[ddr] & & \text{mascot} \arrow[r] & \text{F2} \arrow[dd, equal, bend left] \\
 & \text{beagle} & & & \\
 & \text{animal} & \text{F3} \arrow[r] & \{\text{Fido} \arrow[r] & \text{F2}
\end{tikzcd}$$

Fact chains (* means q-word covered; links underlined are by extension):

CH1: (likes) $\rightarrow$ F1 $\rightarrow$ (John) $\rightarrow$ F4 *

CH2: (likes) $\rightarrow$ F1 $\rightarrow$ (mascot,team) * $\rightarrow$ F2

CH3: (black,_terrier_) $\rightarrow$ F3 $\rightarrow$ (Fido) $\rightarrow$ F2

Answer chains: CH2-CH3

Answer output:

		Q content words covered:
F1:	John likes the team mascot.	who,likes
F2:	Fido is the mascot of the team.	
F3:	Fido is a black terrier.	black
	Dog has as member-or-subset terrier.	dog

question string is processed to idea node level; then the question answering routine takes over.

By inspection of the question, the program determines that the content words to be accounted for are "who" (a q-word), "likes", "black", and "dog"; the extended meaning of "dog" adds "terrier", "beagle", and "animal" to the list of relevant words. "Who", the question pronoun, must be accounted for by a member of the class C(person). The other relevant words are used to select facts with which to start fact chains.

Two facts, F1 and F3, are indicated relevant by the link words for the question. Neither fact alone covers all the content words of the question, so METQA will extend the chains. The link words from F1 ("John", "person", "team", "mascot") indicate both F4 and F2, so METQA extends a copy of the chain to each fact. The only new word in fact F3 is "Fido"; it provides a link to fact F2.

As yet, no chain covers all the question, but METQA now discovers an intersection between chains CH2 and CH3 at F2, so the coverage of the combined chains is checked. "Who" can be covered by "John" of F1; "likes" is covered in F1, and "black" in F3. Finally, "dog" is covered by extension with "terrier" of F3, so the entire question has been accounted for. Deciding that the set of connected facts in CH2-CH3 contains the best answer available for the question, METQA then transforms the facts down to external level for output.

Whenever sentences in a chain are linked by some means other than direct use of the same word in the two sentences, then the nature of the link is output in an explanation sentence built by the program. Thus in the example of Figure 9-3, the fourth sentence of the output answer is an explanation of the link between "dog" of the question and "terrier" of fact F3.

12. A Troublesome Problem

As the reader might suspect, METQA's lack of knowledge of the syntactic relationships among words in a sentence can lead it far astray when producing an answer. As an example, let us consider Figure 9-3 again. Instead of the facts F1, F2, and F3, suppose that METQA knows the following set of facts (content words underlined):

F1: *John*'s *cousin* *likes* the *mascot* of the *team*.

F2: *Fido* and the *team* *mascot* *became* *lost*.

F3: *Fido* *bit* a *black* *terrier*.

Unfortunately, using the same technique as before and forming exactly the same chains with even the same link words, METQA will decide that these three facts (plus the knowledge that a terrier is a dog) contain a complete answer to the question "Who likes a black dog?". Obviously, the needed coverage of the question is there; the problem is, there are too many *other* words there, too. METQA needs a way to avoid using facts in which the desired word is indeed used, but only as a small part of a large and perhaps unrelated event or concept.

A heuristic which may help alleviate this problem is to not necessarily follow all possible branches of fact chains, and to choose facts in which the extraneous content is minimized. That is, given a choice of two facts which a) account for the same number of question segments and/or b) contain the same number of link words from a given chain fact, METQA should give priority to the fact which contains the fewer words besides those involved in a) or b) above. Specifically, METQA would do this by using the weights with which words inidcate their relative importance to the facts they imply (see Chapter 8, Section 2). Each word of each fact has its own separate relative-importance weight. To minimize extraneous content, METQA would try to maximize the ratio of the sum of word weights currently indicating the fact to the total word-weight of the entire fact. The goal is to minimize extraneous content, and to maximize the relative importance in the fact of the words for which we are choosing the fact.

For example, suppose METQA is given the link words "mascot" and "team" from F1, and suppose there are two facts available:

F2: Fido is the mascot of the team.

F2A: Fido and the team mascot became lost.

Then METQA should choose F2 over F2A, because F2 has fewer extraneous words and therefore the word-weight ratio is higher. This constraint should keep METQA's chain of facts more "to the point".

13. Defense of the Question Answering Algorithm

The process of forming sets of facts connected only by use of the same word or semantically similar words cannot be said to be a hard, logical technique. It is similar in certain respects to a human train of thought, however.

On hearing a question, an initial set of information comes to mind--that information which follows directly from the terms used in the statement of the question. If the initial information is not sufficient to account for all the features of the question, then one searches about in his mind, as if asking "What do I know about that, or about that, which might be relevant and useful together with the information I already have?"

As a person thus flits from thought to thought, perhaps there is no firm logical connection among the facts he calls to mind. We are all at times subject to over-generalization or non sequiturs in our arguments, and METQA has the same defect. It does seem that interesting and effective answers can result from this technique, however, just as there will also at times be inadequate, nonsensical collections of facts in a candidate response. Probably what is missing is a routine that assesses the (linguistic and logical) relevance of the arrived-at facts to each other and to the question; unfortunately, the constraints in METQA's design (cf. Chapter 2, Section 1) disallow the use of linguistic information and logical routines.

As this investigation continues, we hope to learn more about the role of semantic associations in answering questions, and about how much "intelligence" of behavior can result from a system guided by the principles we have described.

14. Summary of the Algorithm

To summarize the question answering procedure, METQA begins by collecting and forming chains with all the facts which follow immediately from the content words of the question. From there on, and at each level, the program operates on each fact chain by asking in turn:

1) what new words have been introduced (either directly or by extension) by the latest fact to the realm of discourse for this chain?
2) what new information (facts) follows immediately from these words?
3) do these new words or facts provide an intersection connection with some other fact chain?

At each stage, METQA updates its record of each chain's coverage of the original question. Chain intersections are sought in an effort to combine coverage of the question and form a complete answer. Additional coverage can also result from finding a suitable substitute for any q-word (question pronoun) which appears in the question. At any point where METQA discovers complete coverage of the question by a single fact chain or a combination of chains, then the question answering procedure is terminated, and the answer chain(s) are processed for output (as explained in Section 15).

Even if METQA does not discover a complete answer to the question, fact chain processing must stop when the chains reach a certain maximum number of fact nodes set for the run. METQA is given this fact chain limit not only to limit the question processing to some reasonable amount, but also because it seems likely that the information would tend to get too far-fetched and unrelated to the question if the chains were allowed to wander out very far through the information net. Thus the reason for the constant search for chain intersections is not only to achieve more coverage of the question, but is also a good means of always steering toward the most relevant information.

If fact chain processing is terminated before complete coverage of the question is attained, then METQA investigates the coverage resulting from various combinations of the existing fact chains. The combination of fact chains which results in the most extensive coverage of the question will be used as the answer.

15. Transformation and Output of the Answer

Once the program has decided on some small set of fact chains as the best answer available for the question, then the facts used must be put into some form suitable for output.

Recall that the question answering procedures form a sort of interruption of the regular transformation process. When all covers of the input question were in the interior of the net, then METQA stopped to produce an answer to the question. Now the answer facts are still in the interior of the net; it is time to resume the transformation process, but this time on the answer, not the question.

The internal fact nodes can reach their external representations by following transform links from the component idea nodes of the facts out to the surface terminal nodes. The usual transformation procedures are followed, with each fact being treated like a separate cover of the input. In this case, however METQA is careful not to let any of the facts be destroyed or discarded because of any inability to satisfy some context requirement.

After the answer facts have reached terminal level, they are output in sequence as they appear in the fact chains. As indicated in Section 11, the links between adjacent facts are detailed in a separate explanation sentence if necessary.

CHAPTER 10

DISCUSSION OF ASSORTED PROBLEMS

1. Introduction

This chapter includes sections that discuss and clarify techniques used by METQA and explain some special problems and possible remedies. In addition, we will discuss several future extensions of METQA's ability which are under consideration.

2. The Learning Technique

As the reader is probably aware, there are a great number of evaluation weights in operation in the memory net at all times. There are weights literally all over the net, on every link and every possible restriction of a link. As METQA learns, the evaluation weights soon number in the thousands, so there is no hope that the human trainer can monitor them. Instead, we must depend on the basically simple idea of self-adjustment (via reweighting of links in the net) according to behavior to allow the program to learn intelligent behavior without too much flailing about.

One might question the validity of METQA's learning. There is no real assurance that the program will make the correct hypothesis at a given time, any more than there is for a human. All METQA can do is hypothesize relationships on the basis of its current perspective and evaluate their validity by using these hypotheses to direct future behavior. Taking advantage of the ability to make multiple, parallel hypotheses to explain a certain behavior, METQA tries each time to learn all the most likely hypothetical relationships, so that the desired correct hypothesis will not fail to be included. Then according to its behavior, a given hypothesis will be reweighted, modified or even discarded. Thus, hopefully, the erroneous hypotheses will fail, the correct ones will prevail, and the program will emerge with intelligent behavior.

3. The Role of the Human Trainer

Again we should emphasize the importance of the role of the human trainer in METQA's learning. By the choice of which words to teach and in which order, the trainer has much control over the organizational structure of METQA's permanent memory. He can vary this internal organization from test to test, in order to study METQA's behavior with different "outlooks".

For example, he can teach regular plurals and feminine forms as "merges" (requiring merged paths; see Chapter 6, Section 5) of the stem plus affixes, or if he wishes, they can be taught as units by presenting the longer forms first. Very general or very specific permutation rules can be built up if the trainer plans the input sequence carefully (see Chapter 11, Section 1.2 and Chapter 7, Section 3). Similarly, with attention to the statement of learned facts, the trainer can influence the resulting structure (such as nesting of facts) in the factual information portion of METQA's permanent memory.

The trainer should start with simple strings to be learned as transformations, and later, with simple sentences to be learned as facts. As he observes METQA's behavior and infers what it has learned, the trainer can guide the direction of the learning and retrain METQA if he is dissatisfied with its progress.

Thus, just as a child is molded by his environment and benefits from wise teaching, the program METQA is strongly influenced by its human teacher.

4. Metalanguage--To Use It or Not

Throughout the development of METQA we have been concerned about the use of metalanguage in communication with the program. By *metalanguage*, we mean specially marked (by initial "==") instructions to the program on how it should operate, as opposed to input strings on which to operate. In a sense, by using metalanguage, or *metacommands*, we *talk* to the program rather than giving it experiences to which it can respond and from which it can learn.

A certain amount of metalanguage seems desirable for any large program. Metacommands are used to set flags determining the amount of intermediate output given by METQA and to switch on or off the learning of information. Various internal parameters can be adjusted by use of metacommands. Among other things, these include limits on a) the number of facts allowed in a question answering chain and b) the number of link "layers" METQA is allowed to investigate outward from a relevant word or fact, as well as c) the initial weight to be given to any new class member or subset link and d) the maximum "wobble" width allowed for matching garbled character patterns.

The above uses of metalanguage seem completely legitimate and are not questioned. The uses which we do question are those metacommands which would modify the contents of the memory itself. Such metacommands might inform the program that "'waitress' is 'feminine'" or "'giraffe' is member/subclass of 'animal'" or "'big' is equivalent of 'large'".

As we have stressed throughout this paper, METQA is a learning program and hence should build and modify its own memory. Thus each proposed metacommand must be closely scrutinized to ensure that we are not somehow "cheating" or going beyond the constraints that were initially set for the program.

Suppose the program is learning a certain class structure on its own, but this is requiring more time and processing than the trainer is willing to spend. In such cases, it is expedient to allow the trainer to quickly build the desired class(es) using metacommands. To facilitate this class building, we could add the convenience of naming an existing class or a class under construction by means of a metacommand (e.g., "Name 'feminine' the class containing 'femme', 'fille', 'soeur'").

In order to test METQA's question answering ability, it is desirable to have a fairly large and interrelated set of vocabulary and facts in memory. To hasten the acquisition of such a memory, the trainer can use metacommands to build links and classes (e.g., "Let 'giraffe', 'dog', 'bird' be member/subclass of 'animal'" or "Let 'small' be equivalent of 'little'"). The trainer can thus more easily observe and assess METQA's behavior with different class structures.

A similar issue, but one which does not require a metacommand, is that of having METQA check each input string for the appearance of some special keyword in a certain frame and respond with certain behavior. For example, if a preprogrammed test found the pattern "..... is ....." in the input string "(A) giraffe is (an) animal", METQA could connect the 'giraffe' and 'animal' nodes with class links before going on with the normal processing of the string. This is just another instance of the familiar "pattern → operation" type rule so often used in artificial intelligence. However, this keyword procedure does not seem desirable for this program, since it singles out certain strings to signal special behavior and those strings thus cease to be treated anonymously and generally in the memory structure. Rather, the program should somehow be taught to <u>learn</u> that certain words can serve special purposes. There is no facility for this now, however.

5. Working With Multiple Languages

As indicated in Chapter 2, Section 9, most of the examples of METQA's operations have shown use with only two languages at a time, usually English and French. Although METQA's memory and procedures can handle any number of languages, it tends to get confused on the target direction when working with more than two languages. It is probable that, given enough time, METQA would learn the context classes necessary to decide which words always occur together (and are therefore the elements of one language). These few, wide-ranging context classes would contain most of the words in METQA's memory. Then the words of an input would contextually guide each other out to a single target language.

METQA has not yet been tested on very long runs, however, and it is possible that the adaption to any change of desired target language would be slow, erratic and time-consuming. Therefore, it seems desirable to introduce (with only a small amount of programming necessary) the idea of a higher-level context--we might call it a "supercontext".

That is, the supercontext indicating the particular source language and target language desired would become a sort of overall, higher-level context which is there no matter what specific words are in a sentence. Thus, if there is a choice of links at some ambiguous node, and the disambiguation is not obvious from the immediate context of the surrounding words, then METQA would try again to choose, this time on the basis of the target language specified by the supercontext. Thus the supercontext would serve somewhat as a compass to guide paths through the memory net, providing direction whenever it is needed.

The target language specified by the supercontext could be determined in various ways. For example, it could be set by meta-command (e.g., "==Translate into French:...", or it could be deduced or hypothesized from the words seen in recent feedback strings (perhaps by a simple majority vote). Either of these methods, and in general any use of target and source supercontext, assumes that the necessary class memberships exist or are being learned in some regular frequency and widespread manner.

Perhaps one simple way to explicitly indicate the desired target direction in multilingual memory and to concurrently learn the class memberships necessary for navigation of the memory would be to use input pairs such as those shown in Figure 10-1. For this technique, the use of supercontext is not necessary, although it would be convenient. The desired target language is indicated at the beginning of the input to be translated (or is understood to continue unchanged). METQA will learn and remember the target language class indicator and use it to guide paths through memory toward the (hopefully) correct terminal string. Then when a feedback string is given, METQA checks to make sure each segment recognized in the feedback is indeed a member of the remembered target class; if it is not, the necessary class membership is learned.

```
F)BOY
=GARCON
WOMAN
=FEMME
E)FILLE
=GIRL
G)BOY
=KNABE
```

Figure 10-1. Multilanguage input sequence.

Thus for the first input string "F)BOY" in Figure 10-1, METQA looks for a transform of "BOY" which, if possible, is a member of the class named "F)". When the feedback string "=GARCON" is received, it is processed back through memory as indicated in Chapter 6, but in addition, METQA makes sure that "garcon" is a member of the language class "F)". Since the second input string "WOMAN" has no explicit target indicator, METQA will continue to use the target direction already in effect, and will make sure the second feedback's segment "femme" is also a member of the language class "F)".

6. Same-language or Same-string Transformations

Transformation of a segment between two forms which are expressed as the same string (in the same or a different language) presents a different range of problems from those discussed in the preceding section. The primary rule ensuring METQA's progress through the memory net is that a path cannot "backtrack" to the node it just came from, at least not for the same usage of that node.

Sometimes the links to and from a node indicate that it can be used for more than one contextual situation. In many cases, this will cause no confusion, where the meanings of the string are quite different (e.g., "mange" in French and English, "welt" in German and English). However, when the multiple uses involved still have the same meaning (e.g., "table" in French and English, "tasse" in French and German) then the need to backtrack can raise problems.

The procedure followed when a path traverses a link from some node with multiple uses (i.e. with several contextual options restricting traversal) is to disallow returning to that node (backtracking) by means of the same context which allowed the first traversal. The return link <u>can</u> be traversed if some other context option is satisfied, however.

As an example, suppose the links to and from the "table" node are restricted to contexts of a) English-noun or b) French-feminine-noun. Then given an input of "HEBOUGHTTHETABLE", the path for "TABLE" traverses from the terminal node to the idea node by means of context a), since it appears among members of the English language class. Now in order to return to the same terminal node for "TABLE", the transformed input paths must satisfy context b). If the needed (return) context is not there, it must be learned when feedback shows that the correct response should have been the same string.

The problem can be more difficult when METQA is trying to do a same-language transformation, say from active to passive in English (see Figure 10-2). Many of the words will have the same usage in both source and target strings. It is ridiculously inefficient to

```
E)P)MARYHITSJOHN
=JOHNISHITBYMARY
E)A)THEFISHISEATENBYTOM
=TOMEATSTHEFISH
```

Figure 10-2. Possible active (A) and passive (P) input training sequence.

try to learn matching active and passive contexts for every word in memory. Hence it seems more reasonable, when the source language is the same as the target language, to just suspend the backtracking check (at least for those words which have not been put into some learned class containing words which change between active and passive). There are additional difficult problems to solve in treating active-passive transformations, but the above heuristics seem workable in general for same-string transformations.

7. Treatment of Pronouns

In Chapter 9, Section 10 we described the heuristic used to handle question pronouns. METQA does not yet attempt to handle anaphoric reference, but we feel that much the same heuristic can be used to determine the referent of any pronoun. That is, the learned class memberships of a pronoun can be used as indicators of characteristics the pronoun's referent must have. Thus, if "she" is a member of the "person" and "feminine" classes, then the referent of "she" must also have those memberships. Hence "aunt" and "Mary" might be referred to by "she", but not "table" or "Tom". This heuristic would require explicitly putting all pronouns into a third special class (like the question word and negation word classes discussed in Chapter 9, Section 9) which can be directly referenced by the program.

8. Anaphoric Reference

The "supercontext" idea of Section 5, which was discussed with reference to the problems of a multiple-language memory (i.e. one which contains more than two languages), should be implemented for other reasons, too. Besides the function of guiding METQA to the correct target language in transformation, the supercontext could also serve to maintain a "realm of discourse" context which might permit METQA to handle anaphoric references--at least for simple pronouns in the specially learned pronoun class (see the preceding section and Chapter 9, Section 9). A reference to some person(s) or thing(s) is "anaphoric" if its full interpretation depends on the identifying association with an "antecedent" in the current or in some previous sentence. With supercontext, METQA could use the learned characteristic class memberships for a pronoun to determine the pronoun's referent not only in the current sentence, but in the past few sentences (or however much the supercontext would include). Thus, given the appropriate class memberships and the input sequence

> Harvey has a new sweater.
>
> Mary bought it in Madison.
>
> She goes to school at the university.

it is conceivable that METQA could determine that "sweater" is the referent of "it" and "Mary" is referred to by "she". The problems of handling connected sentence discourse were not considered in setting the goals and constraints for METQA, but the use of supercontext may provide a means of gathering more information than that in the immediate sentence.

9. Possible Future Extensions to Handle Special Types of Questions

Although METQA is not currently programmed to deal with special types of questions such as true-false and "discussion" questions, some thought has been given to the algorithms to be used. As experimentation with METQA's question answering capabilities progresses, these routines will be introduced with minimal programming effort.

9.1 Discussing a topic

Perhaps the simplest addition will be a type of "discussion" question, in which METQA is asked to "Discuss (node)", or "What about (node)". For this type of query, METQA will simply follow all links out from the node asked about, using the extended meaning nodes, facts known about all these nodes, and perhaps even the different external representations (translations) of the original node. To keep the output within some reasonable bounds, a limit can be imposed on the number of levels which METQA travels outward on each link from the original node. With this type of question, a questioner can ask for all the information available about some topic, and the trainer can spot-check the status of METQA's memory.

9.2 True-false questions

In order to deal with true-false questions, METQA will try to build fact chains and account for all the words of the question, much as described earlier in the chapter. Here however, special attention will be given to the "extra" words in the facts, those words which were not used for coverage of the question.

Recall from Chapter 9, Section 9 that negation words are put into a specially named class by the trainer, so that METQA has access to them explicitly as negation words, not just as members of anonymously learned classes. The ability to recognize a word as a negative is very important in deciding whether a statement is true or false. Hence, when trying to answer a true-false question, METQA searches all through the fact chains for words which are members of the special negative class. Any such negation words can indicate to METQA that the information at hand may actually be a counterexample to the statement in the question.

Another expected addition to the program is the use of a relation to indicate "oppositeness" or reverse meanings between word nodes. This information, when present for the words of a statement, can be used in much the same way as negation words. Thus "happy" and "sad" may be linked by the opposite relation, as well as "healthy" and "sick". Hence when a true-false question statement includes a word which is discovered to have an opposite link, then METQA will check the extra words of the fact chain(s) for the opposite word(s) as well as for negation words.

Briefly, the true-false algorithm which will be tried for METQA is as follows (refinements will doubtless be needed). Suppose a fact chain was found which accounted for all the words in the true-false question statement. If the extra words contain no negatives or opposites, then METQA reports that the statement is true and outputs all the facts used in that chain. If there are negatives or opposites among the extra words, then METQA reports that the statement appears to be false and outputs the facts which led to this conclusion. If no fact chain was found which could account for all the parts of the question then METQA must report that the statement is false for lack of sufficient information, but any information which was found would then be output.

9.3 Enumeration questions

Another type of question which could be handled after moderate programming changes would be requests such as "Name the people that go to school", or "List the animals that eat grass", or "What are some cities", where the underlined segments vary according to the information desired. These enumeration questions will be handled by chaining facts to cover phrases and/or simple listing of class members (if available) for a single node request.

For example, to answer "List the animals that eat grass", METQA would try to find facts which account for the segments "eat" and "grass". Then those same facts would be searched for segments which account for (by class memberships in the desired node, for example) the node "animals". Any such segments found would be presented as the answer. Here again, METQA will probably follow up its answer with all the facts used to produce the answer.

These and perhaps other question types will be attempted as experimentation with METQA continues. The goal is to explore the limits of METQA's intelligence, given the basic and very general organization we have described, rather than to add special tricks to extend particular capabilities.

10. Thoughts on QA Feedback

Currently there is no provision for accepting feedback for an answer to a question (QA feedback); new or corrected information can enter the memory net only independently in the information learning mode, not from an analysis of feedback. Although separate entry or learning of facts is vital for building up a store of facts, it seems clear that a QA feedback capability is a necessary addition, so that METQA can more easily unlearn incorrect facts or reinforce productive fact connections.

Different methods of feedback have been considered. The simplest type of QA feedback would be just an indication of whether the answer was right or wrong. If the answer was wrong for any reason, then all the facts used and all the links followed between facts would be downweighted to reflect their participation in undesirable behavior. If a fact node or a link is downweighted to some minimum (perhaps a zero weight), then that node or link is removed from the memory.

There are several disadvantages to this type of feedback learning. First, it is not at all selective--an answer is either right or wrong, with no levels in between. So even if it was a minor part of an answer which made it unacceptable, every link followed by the answer would be penalized. In addition, if bad answers always cause downweighting, then it follows that all good answers must be reinforced. That is, if the answer was correct, then every node and every link used must be upweighted to reflect its contribution to desirable behavior. Otherwise, a link could be repeatedly downweighted and finally discarded, even though the link had also been used to produce other good answers. This constant upweighting and downweighting could possibly cause troublesome instability, and would certainly use a lot of processing time.

An alternate feedback method would have an optional feedback string given by the trainer. This feedback string will contain the answer which the trainer thinks _should_ have been given to the question (this is the kind of feedback given when in the transform mode). In order to learn from this feedback string, METQA must first try to account for or cover all the feedback segments with connected fact chains, just as it did for the question string (and much as is done in transformation when using feedback to find intersecting paths; see Chapter 6, Section 2).

Let us call the resulting feedback fact chain FFC, and let the answer fact chain be called AFC. The program compares the two fact chains, searching especially for the use of the same fact nodes by both chains. These points of intersection mark the parts of the answer which were correct. The nodes which appear in AFC (the answer), but not in FFC (the feedback), are either themselves erroneous or were incorrectly used; therefore METQA will downweight those facts, thus reducing their credibility, and will downweight the links followed to use those nodes. On the other hand, the nodes which appear in the feedback chain, but not in the answer chain, are nodes which <u>should</u> have been used. The credibility weights of these facts will be increased, and the links followed to the nodes will also be upweighted. The nodes which appear in both chains are operating satisfactorily; METQA can either leave them alone or perhaps give them a small increase in weight.

As an example, suppose METQA receives a question and outputs the answer fact chain AFC = A - B - C - D. Then the trainer responds with feedback, a better answer. Suppose METQA finds complete coverage of the feedback in the feedback fact chain FFC = D - E - F - B. Comparing the fact chains, METQA discovers that nodes B and D appear in both chains and were therefore correctly used. Facts A and C, appearing only in AFC, are downweighted, along with the appropriate links. Then the facts E and F, which should have been used and were not, are upweighted along with their links.

There are several advantages to this method of learning from QA feedback. Unlike the first method discussed, feedback now would be optional, necessary only in order to correct a wrong answer. Hence there is a great saving of computing effort. Furthermore, the feedback and attendant learning are much more specific. By analyzing the better answer given in a feedback string, METQA can determine which fact nodes were wrongly used and which ones should have been used, and specifically readjust their weights.

Note the similarity between this feedback learning method and the learning of transformations from input and feedback string pairs as discussed in Chapter 6. In each case, METQA is given a model of good behavior and must analyze it and decide how to improve the memory net. By reweighting different parts of its own structure, METQA is able to use its experience and adapt its behavior to approach the "ideal" offered by the trainer.

CHAPTER 11

RESULTS AND CONCLUSIONS

1. Running the Program

As indicated in Chapter 2, METQA is a 7000 statement Fortran V program, which is completely coded and code-checked. The transformation routines have been extensively debugged, but are still not completely tested out. The question answering routines have not been tested. The program runs very fast on the Univac 1108 computer. Response time is immediate. Computer time is roughly 6 seconds per 25 input-feedback pairs (including learning).

Throughout the program description (Chapters 3 through 9) we have looked at examples of METQA's performance. In this section, we will examine more examples of METQA's behavior, and illustrate how the different mechanisms fit together.

1.1 Sample run showing class generalization

In order to orient the reader, the first example is not a teletype listing but an actual input sequence listed in a table (see Figure 11-1). The input-feedback string pairs are numbered for easy reference, and METQA's typed response to the trainer are shown in the second column. The internal relationships METQA learns are shown in the third column.

In experience pairs 1 through 7, the trainer is teaching METQA a simple vocabulary. In the response to the eighth input, "LA GARCON",

	Trainer	METQA's Response	Learned Relationships

1.	GIRL
	=FILLE
		U(GIRL)
		U(FILLE)
			GIRL ↔ FILLE

2.	WOMAN
	=FEMME
		U(WOMAN)
		U(FEMME)
			WOMAN ↔ FEMME

3.	CHIEN
	=DOG
		U(CHIEN)
		U(DOG)
			CHIEN ↔ DOG

4.	BOY
	=GARCON
		U(BOY)
		U(GARCON)
			BOY ↔ GARCON

5.	PETITE
	=SMALL
		U(PETITE)
		U(SMALL)
			PETITE ↔ SMALL

6.	THEWOMAN =LAFEMME	U(THE) FEMME U(LA) TMINTS*(FEMME)	THE ↔ LA
7.	THEGIRL	LA FILLE	
8.	THEBOY =LEGARCON	LA GARCON U(LE) TMINTS(GARCON)	$\left\{\begin{array}{l}\text{GARCON} \in \text{C1}\\ \text{THE} \dashv\!(\text{C1})\!\longrightarrow \text{LE}\\ \text{LE} \rightarrow \text{THE}\end{array}\right.$
9.	SMALLGIRL	PETITE FILLE	
10.	SMALLBOY =PETITGARCON	PETITE GARCON U(PETIT) TMINTS(GARCON)	SMALL $\dashv\!(\text{C1})\!\longrightarrow$ PETIT PETIT → SMALL
11.	SMALLDOG =PETITCHIEN	PETITE CHIEN INTSEC*(PETIT) TMINTS(CHIEN)	CHIEN $\in$ C1
12.	THEDOG	LE CHIEN	

* METQA uses the abbreviations "TMINTS" (for terminal intersection) and "INTSEC" (for nonterminal intersection) to indicate to the trainer that a path intersection has been found between the input and feedback strings. The internal intersection addresses which always follow the abbreviations have here been omitted; instead, we insert the feedback word whose path intersected the input path.

Note: The termination symbol ("<") has been omitted from all inputs from the trainer.

Figure 11-1. Input sequence showing class generalization

METQA makes a mistake for the first time. The feedback string indicates that the new and unknown string "LE" should have appeared in this case. Consequently, according to the procedures described in detail in Chapter 6, METQA forms a new class (which we will call Cl) with the "garcon" node as its first member. Then the new form "le" is learned as the transform of "the" to be used when the string context includes a member of class Cl.

In experience pair 10, METQA must learn that the new form "PETIT" occurs with "GARCON" instead of "PETITE". Since "garcon" already is a member of a class, METQA generalizes the use of that class to this situation. Thus it learns that "SMALL" transforms to "PETIT" in the context of class Cl.

The next input pair teaches METQA that, in the context of "CHIEN", "SMALL" should transform to "PETIT". METQA already knew the form "PETIT", but did not know to use it in this situation. Since the form "PETIT" has been learned to occur in the context of members of class Cl, METQA hypothesizes (according to the heuristics detailed in Chapter 6) that "chien" must be another member of class Cl.

Finally, in experience 12, METQA uses the class generalizations it has learned to correctly transform "THEDOG" to "LE CHIEN". Although "chien" has not occurred before with "le", which has restricted usage, METQA has learned that "chien" belongs to the class of words which elicit the form "le".

1.2 A longer teletype run

Figure 11-2 shows a complete teletype run in which the trainer is teaching METQA the adjective-noun permutation and feminine forms of French adjectives. The listing may require some preliminary explanation. METQA prompts the trainer for input by outputting the string "...". Input strings from the human trainer are distinguished by the termination symbol ('<'). Lines beginning with "==" (and the small integer below) are flag-setting metacommands of the type discussed in Chapter 10, Section 2. The strings "TMINTS" and "INTSEC" are explained in Figure 11-1. The numbers output by METQA are internal addresses and can be ignored by the reader. For convenience of reference, the input pairs are numbered in the right column, and some brief comments have been inserted.

In the example run of Figure 11-2, METQA starts out with a memory which is empty except for the names of the basic link types (see Chapter 3, Section 1). The trainer teaches METQA several vocabulary transformations, including two different transforms for "the". The transform to "le" is unrestricted because it was learned first; when the second form "la" is learned, it is restricted to contexts involving the usage class which contains "femme".

With the seventh input pair ("green dog"-"chien vert"), the trainer starts teaching METQA a permutation rule. METQA creates classes for the words involved and permutes the string correctly when it is received again as Input 9.

After teaching METQA a little more vocabulary, the trainer presents more permutations. With Input 12, METQA learns that "brun" can also be involved as the first word (from the source direction,

```
UNIVAC 1108 TIME/SHARING EXEC  VERS MACC25.33.17 TTY UWT109

@RUN JORDAN,2279,2553,1M,30
@RUN Y05316,2279,2553,1M,30
DATE: 090371        TIME: 120930
BEGEETRD PLEASE
CONTINUE
@ASG,A SJPROG.,F2///500
READY
@ASG,A SJMEMA.,F2
READY
@ASG,A SJMEMB.,F2
READY
@ASG,A 6.
READY
@USE 10,SJMEMB.
READY
@USE 11,SJMEMA.
READY
@XQT SJPROG.ABS
TAPE   1
 ...
==SET TRIM
  MTACMD  ==SET :  TRIM
  1
 ...
==SET TTY
  1
 ...
THE<                                 1.  Vocabulary training starts
    2995   U(THE)
 ...
=LE<
    2987   U(LE)
 ...
THEDOG<
    2983  LE  U(DOG)                 2.
 ...
=LECHIEN<
    2963  TMINTS((PTH 2975, CEL 33 +PTH 2991, CEL 2981))
  U(CHIEN)
```

Figure 11-2a. Sample teletype run.

```
 ...
FEMME<                            3.
    2995   U(FEMME)
 ...
=WOMAN<
    2987   U(WOMAN)
 ...
THEWOMAN<                         4.  Alternate form "le" learned for
    2969  LE FEMME                    "the", New class C1 gets "femme".
 ...
=LAFEMME<
    2949   U(LA) TMINTS((PTH 2965, CEL 87 +PTH 2983, CEL 2
  973))
 ...
VERT<                             5.
    2995   U(VERT)
 ...
=GREEN<
    2987   U(GREEN)
 ...
THEWOMAN<                         6.
    2969  LA FEMME
 ...
GREENDOG<                         7.  Start teaching permutation.  Class
    2969   VERT CHIEN                 C2 gets "vert", C3 gets "chien".
 ...                                  Rulel:  C2(2), C3(1)
=CHIENVERT<
    2939  TMINTS((PTH 2965, CEL 63 +PTH 2983, CEL 2973)) T
  MINTS((PTH 2953, CEL 141 +PTH 2991, CEL 2977))
 ...
THEDOGISGREEN<                    8.  New word "is"
    2941  LE CHIEN  U(IS) VERT
 ...
=LECHIENESTVERT<
    2869  TMINTS((PTH 2909, CEL 33 +PTH 2991, CEL 2929)) T
  MINTS((PTH 2921, CEL 63 +PTH 2981, CEL 2951))  U(EST)
  TMINTS((PTH 2897, CEL 141 +PTH 2969, CEL 2939))
 ...
GREENDOG<                         9.  Successful permutation of input
    2961  CHIEN VERT                  seen before.
```

Figure 11-2b.

```
 • • •
BROWN<                                      10.  More vocabulary.
    2995   U(BROWN)
 • • •
=BRUN<
    2987   U(BRUN)
 • • •
GARCON<                                     11.
    2995   U(GARCON)
 • • •
=BOY<
    2987   U(BOY)
 • • •
BROWNDOG<                                   12.  Generalize use of p-rule, so  C2
    2965  BRUN CHIEN                             contains "vert", "brun".
 • • •                                           Rule1 still C2(2), C3(1)
=CHIENBRUN<
    2931   TMINTS((PTH 2959, CEL 63 +PTH 2981, CEL 2963)) T
  MINTS((PTH 2945, CEL 277 +PTH 2991, CEL 2973))
 • • •
GREENBOY<                                   13.  C3 contains "chien", "garcon"
    2965  VERT GARCON
 • • •
=GARCONVERT<
    2931   TMINTS((PTH 2957, CEL 301 +PTH 2979, CEL 2963))
  TMINTS((PTH 2947, CEL 141 +PTH 2989, CEL 2971))
 • • •
BROWNBOY<                                   14.  Generalized across classes (C2
    2961   GARCON BRUN                           and C3) to permute form not seen
 • • •                                           before.
FILLE<                                      15.
    2979   U(FIL) THE
 • • •
=GIRL<
    2961   U(GIRL)
 • • •
THEGIRL<                                    16.  Teach feminine form of article
    2969  LE FILLE                                for "fille". C1 contains "femme",
 • • •                                            "fille"
=LAFILLE<
    2915   INTSEC((PTH 2951, CEL 41 +PTH 2991, CEL 2993)) T
  MINTS((PTH 2937, CEL 237 +PTH 2983, CEL 2967))
```

Figure 11-2c.

```
 ...
ISTHEWOMANGREEN<                17.  Teach alternate form of adjective.
    2925  EST LA FEMME VERT
 ...
=ESTLAFEMMEVERTE<
    2849  TMINTS((PTH 2907, CEL 247 +PTH 2991, CEL 2955))
  TMINTS((PTH 2915, CEL 119 +PTH 2983, CEL 2923)) TMINTS
  ((PTH 2901, CEL 87 +PTH 2975, CEL 2937)) TMINTS((PTH 2
  891, CEL 141 +PTH 2965, CEL 2935))  U(E)
 ...
·GREENWOMAN<                    18.  Teach new permutation.
    2947  VERT E FEMME
 ...
=FEMMEVERTE<
    2761  TMINTS((PTH 2915, CEL 87 +PTH 2961, CEL 2925)) T
  MINTS((PTH 2875, CEL 141 +PTH 2953, CEL 2937)) TMINTS(
  (PTH 2821, CEL 443 +PTH 2951, CEL 2931))
    2777  TMINTS((PTH 2915, CEL 87 +PTH 2961, CEL 2925)) T
  MINTS((PTH 2771, CEL 141 +PTH 2953, CEL 2937)) TMINTS(
  (PTH 2773, CEL 443 +PTH 2951, CEL 2931))
 ...
THEGREENWOMAN<                  19.  Successful permutation
    2913  LA FEMME VERT E
 ...
GREENGIRL<                      20.
    2943  VERT E FILLE
 ...
=FILLEVERTE<
    2745  TMINTS((PTH 2907, CEL 237 +PTH 2957, CEL 2919))
  TMINTS((PTH 2873, CEL 141 +PTH 2949, CEL 2933)) TMINTS
  ((PTH 2809, CEL 443 +PTH 2947, CEL 2925))
    2761  TMINTS((PTH 2907, CEL 237 +PTH 2957, CEL 2919))
  TMINTS((PTH 2755, CEL 141 +PTH 2949, CEL 2933)) TMINTS
  ((PTH 2757, CEL 443 +PTH 2947, CEL 2925))
 ...
THEGREENGIRL<                   21.  Successful permutation
    2913  LA FILLE VERT E
 ...
LIVRE<                          22.
    2985   U(LIVRE)
 ...
=BOOK<
    2977   U(BOOK)
```

Figure 11-2d.

```
 ...
THEBOOK<                            23.  Deliberately teach a wrong form
    2959  LE LIVRE                       of "the" for use with "livre"
 ...
=LOLIVRE<
    2915   U(LO) TMINTS((PTH 2941, CEL 501 +PTH 2973, CEL
  2957))
 ...
THEBOOK<                            24.  METQA has learned wrong form;
    2959  LO LIVRE                       correct it.
 ....
=LELIVRE<
    2887  INTSEC((PTH 2939, CEL 41 +PTH 2991, CEL 2993)) T
  MINTS((PTH 2911, CEL 501 +PTH 2973, CEL 2953))
 ...
THEBOOK<                            25.  METQA has unlearned wrong form.
    2959  LE LIVRE
 ...
NOMORE
 MEMORY SAVED  1
 MTQA STOP

DATA IGNORED - IN CONTROL MODE
@FIN,E


    RUNID:  Y05316   PROJECT: 2279                USER: 2553

  ITEM                                  AMOUNT       COST(DOLLARS)

CPU TIME                          00:00:05.727        $1.27
I/O REQUESTS                                91        $0.33
I/O-WORDS TRANSFERRED                    96634        $0.17
EXCESS CORE USAGE                                     $0.06
CARDS IN                                    61        $0.07
PAGES PRINTED                                7        $0.70

TOTAL COST                                            $2.61

    THE ABOVE DOLLAR AMOUNTS ARE APPROXIMATE AND ARE BASED ON
      PRIME SHIFT RATES
```

Figure 11-2e.

at least) in the same permutation. Hence, class C2 gains a new member, but the p-rule stays the same. Then with Input 13 METQA learns that "garcon" can behave as a member of class C3 in the p-rule. As the classes of the p-rule gain members, the combination possibilities and generality of the p-rule increase. Thus, Input 14, which METQA has not seen before, is correctly transformed and permuted from "brown boy" to "garcon brun".

Now the trainer turns to some feminine words. Notice that when the unknown string "FILLE" is received, METQA recognizes the string "LE" and transforms it to "THE". The "LE" path is not intersected by any path from the feedback string, however, so METQA lumps the unrecognized "FIL" and the unintersected path for "LE" together into one hunk (see Chapter 6, Section 4) and learns the new word "FILLE".

A similar case occurs with input pair 17. METQA transforms "GREEN" to "VERT", but the feedback string contains "VERTE". METQA recognizes the first four letters of "VERTE" and immediately finds a terminal intersection with the original path from "GREEN". This leaves the final "E" unrecognized as an unclaimed unknown hunk (see Chapter 6, Section 5, Figure 6-5). In this case, since an intersection occurred, METQA will learn the "vert" + "e" word as a merging of two paths (hence, the space in the later responses of "VERT E") which occurs in the context of "femme".

Since neither "verte" nor "femme" is a member of the classes (C2 and C3) describing the first permuation rule, the permutation presented in input pair 18 is learned as a new rule. The first rule could easily have been generalized by means of inputs such as "REDBOY" - "=GARCONROUGE" and "REDWOMAN" - "FEMMEROUGE", which would allow METQA to bridge the "gender gap" and use the same rule.

In the final sequence of inputs, the trainer deliberately teaches METQA a misspelled form of "the" and then changes it to the correct form. Since METQA can unlearn the incorrect form, it is not a catastrophe when the trainer really does make a mistake.

As shown, the entire teletype run took only 5.7 seconds of CPU time (and about ten minutes of clock time). The inputs were admittedly quite simple and the memory small, but only when the learned vocabulary is much larger (causing many attempted pattern matches) and the input sentences quite ambiguous (yielding many covers or parses) should the processing time increase appreciably.

2. Summary of Results

The long run in the preceding section shows how METQA learns a language when interacting with a human trainer. The program learns the basic words and their transforms. It also learns to compound interacting words into idioms. The program learns that certain words transform to alternate forms according to the context in which they appear; these context words are put into classes based on their similar behavior. By generalizing from one member of a context class to the other (word) members which have (at some time in the past) exhibited similar behavior, METQA can respond correctly to input situations which it has never seen before.

Certain transformations require a rearrangement of the order of the string segments. In order to produce the correct response, for each situation METQA learns a permutation rule which directs the rearrangement of groups of segments which satisfy the class requirements of the rule. New permutation rules are combined with old ones (by modification of the old rule) if possible, in order to build as small and general a set of rules as possible.

METQA learns discontinuous morphemes by flagging certain nodes so that they can transform simultaneously to two other nodes in the network; for the case of coming from the direction of the other language, the two nodes are flagged so that their transform links merge together at one node. In the example run of the previous section, METQA learned a merge path ("vert" + "e") which was not discontinuous and did not require permutation.

In previous chapters we examined how METQA learns by building relationships (labeled links) between nodes of the memory net and associating with each link an evaluation weight which is readjusted up or down for good or bad behavior. If a link is downweighted enough times because of bad behavior, the weight is reduced to some minimum and the learned relationship is erased or "forgotten". Unfortunately, due to a minor bug in the system, we are unable at this time to show a <u>gradual</u> case of forgetting a relationship; the forgetting of "lo" shown in the last sequence of the long example looks rather abrupt.

There are still a few bugs in the transformation routines. When they are cleaned up, we expect METQA to show the gradual unlearning mentioned above, plus permutation of discontinuous morphemes, which currently causes METQA to stumble and fail. In addition, we plan to remedy the same-string problem (see Chapter 10, Section 6), which will allow testing on transformations such as active to passive or change of verb tense.

3. Concluding Comments

It seems hopeless to try to pre-program all the information and contingency plans needed to deal with a great variety of tasks in undetermined and probably changing situations. A system must be able to structure and adapt itself to its environment, even if it also uses sophisticated techniques and information files produced outside itself. Although the use of auxiliary information from the outside world (such as logic and linguistics) is to be recommended for most systems, we have tried to avoid it in developing METQA. Instead, we seek to determine the extent of the intelligent manipulation of language which can be learned without such aids.

The purpose of this research and of using translation and question answering as the tasks is a) to study METQA's ability to learn, on its own, the words, idioms, transforms, classes, permutations, and the interrelationships of natural language units, and b) to assess the adequacy of the learning mechanisms, the learned memory net structure, and the simple and general heuristics being used.

4. Concluding Summary

With the research reported in this thesis we are exploring the learning capabilities of METQA, a program which uses a memory net structure to store what it learns from processing input strings of unstructured natural language (received from a human trainer) and outputting a response string. The tasks with which METQA demonstrates its learning are transformation between languages or between forms of the same language, and question answering based on information it has learned.

Instead of building in linguistic information for the language(s) which METQA might see, and the logical information to aid its question answering, METQA is allowed to organize its own knowledge of the language(s) by means of simple classification heuristics (categorization into classes according to similar behavior) for learning of words, compounds, contexts and permutations. The memory is modified when errors are discovered and, in general, is adapted to suit the changing "view of the world" which it receives from a human trainer.

Chapters 3 through 9 provide a description of the memory net and its contents, as well as detailed explanations of the heuristics which guide the program's behavior and learning.

The program has three modes of operation, in which it a) learns and transforms strings, b) learns information "facts", or c) answers questions using those facts. The human trainer presents strings of natural language which are specially marked to indicate which mode(s) should be active. If he wishes, the trainer can follow an input string (and METQA's response to it) with a feedback string showing what the correct response should be. METQA decides what to learn by comparing its own response with the feedback string given by the trainer. The program tries to determine which string segments it processed incorrectly and modifies its memory by reweighting (up or down) the links which contribute to good or bad behavior.

In order to translate an input source string, METQA uses pattern recognition to enter the memory net and, for each segment, traverses links from node to node (as permitted by the context of the string) until the target string is reached.

When answering a question, METQA uses the content words of the question to choose relevant facts and, starting from this first-level information, builds up chains of related facts which try to cover all the information requested by the question.

In pursuing this research, we are experimenting with a memory structure and set of heuristics which produce an interesting kind of learning. We would like to determine the extent to which such a learning program can be self-sufficient in producing intelligent manipulation of language. By analyzing the behavior of this program with its strengths and weaknesses, we should be better able to determine which types of (hopefully minimal) built-in background information will yield the greatest increase in performance on multiple tasks with natural languages.

APPENDIX

The SMALL System

1. Introduction

SMALL (String MAtching List Language) is a list processing system with string matching capabilities. The system is written entirely in FORTRAN V and is currently running on a Univac 1108 computer. The SMALL system was designed to be embedded in a large FORTRAN V program and acts as a high level language for manipulating list structures.

Although it was modeled after the SAC-1 system of Collins [1971] and is rather similar to SAC-1, the SMALL system differs in cell structure and the available space setup, and dispenses with the reference count feature. These differences were necessitated by the need for several more item fields to hold information in a cell than SAC-1 cells provide. In addition, routines for simple string matching were added. The SMALL routines will be described in later sections.

2. Representation of Lists

In SMALL, a list is represented in computer memory by a collection of cells. A cell consists of two memory locations, specifically, two adjacent elements of the available space array, SPACE.

Each cell contains nine fields, overlapping for various usage situations. The field configuration of each cell and list is left entirely to the programmer, both in construction and later use. The field which is universal to all cells and which ensures uniform traversal of lists is the link field, which contains the address of the next cell in the list (or zero, to indicate the end of a list).

The fields of the cells can be represented by rectangular drawings such as those in Figure A-1. In any cell configuration, the LINK field is a half-word field containing the address of the next cell in the list. In cell (a), the second memory word is assumed to contain a character string; in cell (b), the contents of the second word are assumed to be two integer values or addresses (pointing to other cells or lists). The programmer should check for field values consistent with the cell configuration he expects.

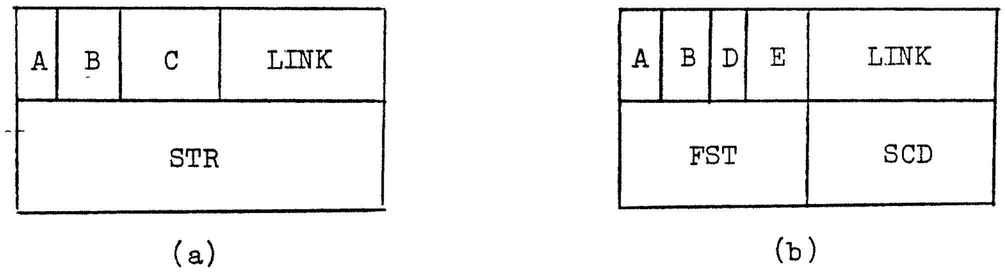


Figure A-1. Two example SMALL cell representations showing the nine fields A-E,LINK,FST,SCD, and STR.

Half the "top" word of each cell is divided into three or four fields (depending on usage), which in the current implementation vary in size from three to nine bits each. The following list shows the names by which the SMALL routines refer to the fields labeled A through E in Figure A-1:

A -	TYPE	(3 bits)
B -	TCW	(6 bits)
C -	WT9	(9 bits)
D -	TYP	(3 bits)
E -	WGT	(6 bits).

Originally, the field names were suggestive of the use METQA made of each field; the field usage has since expanded much beyond that suggested by the names, however. These fields are used to contain the various flags, switches, tallies and weights needed for a program such as METQA.

Different cell configurations may appear in a single list; the complete list can be quite branchy if cell configurations such as (b) above are used. The programmer (or the drive program) is expected to keep track of the fields used in a particular type of list, but in any case, progress through the list is effected by use of the link address field.

3. Erasure and the Available Space List

As with other list processing systems, SMALL uses an available space list to hold cells which are not currently being used in the representation of some list. The available space array for SMALL contains a doubly-linked cell list of order one (i.e., it contains no sublists), on which two available space mechanisms operate in opposite directions, one from each end of the array SPACE. Conceptually, the two mechanisms operate as two separate available space lists. However, since the two lists can collide somewhere in the middle of the SPACE array as space runs out, the two available space lists (MAVAIL and WAVAIL) are not independent.

At the lower end of the SPACE array is the top of the MAVAIL (<u>m</u>emory) available space list. Lists built with cells from MAVAIL are protected from automatic erasure and are used to form METQA's permanent memory.

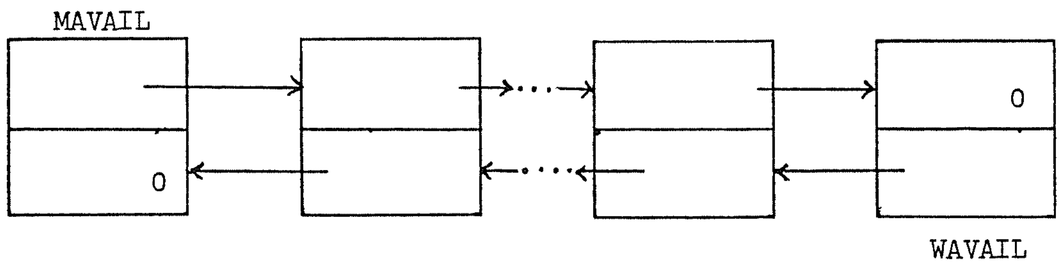


Figure A-2. Illustration of SMALL's doubly linked available space list.

At the upper end of the SPACE array is the top of the WAVAIL (<u>w</u>ork space) available space list. Lists built with cells from WAVAIL are used by METQA as work space for its behavior and learning operations and "housekeeping". For each available space list, an extent variable is maintained (MAXMAV and MINWAV); its value equals the farthest extent that the associated available space list has ranged toward the middle (or other end) of the SPACE array. When the two variables attain the same value, then the memory and work space have collided, and an appropriate output message warns that available space is exhausted.

Lists from either MAVAIL or WAVAIL can be erased by specific request (call of subroutine ERLIST), but in addition, the entire work space (i.e., all the SPACE array which is not involved in permanent list representation) is automatically erased and reordered periodically by calling subroutine RESET. For the program METQA, RESET is called after each input-feedback pair. In this manner, any stray cells which were not specifically erased are gathered back into the available space list.

The initial available space list is created by the subroutine SETUP, in which a common block holds the SPACE array and a variable indicating the size of the array. SETUP creates from the given array space the doubly-linked list of two-word cells and initializes the values of MAVAIL, WAVAIL, MAXMAV and MINWAV.

As indicated earlier, the available work space is reorganized periodically by a call of the subroutine RESET. RESET erases and relinks all the cells in the work space, then resets the values of the space list address WAVAIL and the extent variable MINWAV.

The only system subprogram which removes cells from the available space list is the function Z=GETCEL(WORK). A cell is taken from the work space list if the parameter WORK equals one; otherwise a cell is removed from the permanent memory space list MAVAIL. The address of the new cell is returned as the value Z.

Cells are returned to the available space by means of two subprograms, ERCEL(X) and ERLIST(X). Depending on the address X, the cells are added to either MAVAIL or WAVAIL. ERLIST erases only a single level list; the programmer should explicitly erase any sublists which he longer needs, especially if he is concerned about space. It was not advisable to erase sublists in the SMALL system, since a list may be pointed to by several other lists, and the reference count feature of SAC-1 is not available in SMALL. Since the available space is erased and reset periodically, stranded cells have not presented a problem.

4. The Primary List Processing Routines

We shall refer to the subprograms which operate explicitly on elements of the SPACE array as the primary routines of the SMALL system. All other (secondary) SMALL subprograms use these primary subprograms to manipulate cells and lists. We have already described the primary routines SETUP, RESET, GETCEL, ERCEL and ERLIST. In addition to these, the only other primary subprograms are those which store and retrieve data in the cells.

The STORE routines for the nine fields (STYPE, STCW, STYP, SWGT, SWT9, SLINK, SSTR, SFST, SSCD) have two arguments, X and Y. In each case, the argument Y is the location of some cell and the argument X is an integer value (except in the case of SSTR, where X can be a character string) which is to be stored in the appropriate field of the cell at Y. Two further subroutines SETW1(T,C, W,L,Y) and SETCEL(T,C,W,L,S,Y) allow storage of several field values at once.

There are nine field retrieval (GFLD) function subprograms; TYPE, TCW, TYP, WGT, WT9, LINK, STR, FST, and SCD. Each function has a single argument Y, which is the location of some cell. The appropriate field of cell Y is extracted and returned as the value of the function. The additional subprogram GVALS(Y,T,C,W,S) retrieves field values covering the entire cell Y (Y becomes LINK(Y)).

5. The String-matching Routines

The basic tools for string matching in the SMALL system are the two subroutines FIND and MATCH. The subprogram MATCH(CELL,AR,ARLOC) is a logical function which seeks an exact string match at a specific location in an array of characters. MATCH returns the value .TRUE. if and only if the character string beginning in cell CELL matches the string in array AR, starting after ARLOC characters.

The function FIND(CELL,AR,ARLOC,MASK), which calls the function MATCH, looks for a string match within a range of locations in an array of characters. Starting after ARLOC characters in array AR, FIND searches along some number of locations determined by the argument MASK for a match of the string beginning in cell CELL. According to the value of MASK, the matching search can begin only at the location ARLOC, or can extend along some MASK characters from the location ARLOC, or can extend to the end of the array for a match anywhere after ARLOC characters. The function value returned by FIND is N, if the string beginning in CELL was matched after N characters of array AR, or else -1 is returned if no match occurred.

The author's intent, in the design of FIND and MATCH, was to provide an elementary basis for string matching which could be used to build various types of sophisticated string recognition programs. Thus these two basic routines provide adequate capabilities for the recursive (because of indirect referencing; see Chapter 4, Section 6) n-tuple string-matching needed for METQA's operation.

6. The Remaining Secondary List Processing Routines

Besides the primary and string-matching routines described in the preceding sections, there are some twenty-odd other routines which evolved into the current version of SMALL. A list and brief description of these routines will be given below. Function subprograms will be indicated by a prototype of the form Z=F(X1,X2,..., XN), and subroutines will be indicated by a prototype of the form S(X1,X2,...,XN).

The several routines which use GETCEL to acquire new cells from available space include LOC as one of their arguments. If LOC=1, the new cell comes from WAVAIL (temporary work space). If LOC=0, the new cell comes from MAVAIL (permanent memory space). The reader should note that all variable names and all numerical values used by the SMALL system are of FORTRAN type <u>integer</u>.

Z=AR2LST(AR,LOC). Transfers the character string packed in array AR into a newly created list. Z is the new list address.

Z=ATTCEL(ST,LST). Looks in STR (string) field of each cell in list LST for name matching the string ST. On success, Z is the address of the cell whose string matched ST; otherwise, Z is -1.

Z=ATTVAL(ATT,NODE). Looks in each cell of list NODE for FST field containing the address ATT (cell ATT contains the string name of some attribute). On success, Z is the address in the SCD field of the matching cell; otherwise, Z is -1.

Z=CELBEF(X,Y). Z is the address of the cell before (pointing to) cell X in list Y. If Y=0 or X=Y, Z is zero. If X=0, Z is the address of the last cell in list Y. If cell X is not found in list Y, Z is -1.

Z=CELCNT(LST). Z is the number of cells in the top level of list LST.

Z=CELLN(N,LST). Z is the address of the Nth cell of list LST. If N=0, Z is zero. If list LST has fewer than N cells, Z is -1.

Z=COPY(X,LOC). Produces a copy of list X. Z is the address of the new list, or else zero, if X is zero or negative.

Z=DELETE(CEL,LST,PRECEL). Deletes cell CEL from list LST and returns CEL to available space. To speed processing, PRECEL is the address of the cell preceding (pointing to) CEL, if known; if not known, PRECEL is zero. Z is the address of the cell following CEL, or is -1 if CEL was not found in LST.

Z=GETSTR(DLST,LOC,LAST). Z is a new list containing the character string represented in the (description) list DLST. DLST may have marked in it nested indirect references to other lists; GETSTR retrieves all the lists' strings recursively. LAST is the address of the last cell in the new list Z.

Z=INSERT(CEL,LST,PRECEL). Inserts cell CEL into list LST after cell PRECEL. If PRECEL=0, CEL is placed at the beginning of LST. Z is the (possibly new) address of list LST.

Z=INSLST(NULST,LST,PRECEL,NULAST). Inserts list NULST into list LST after cell PRECEL. If PRECEL=0, NULST is placed at the beginning of LST. NULAST is the address of the last cell in NULST, or else zero if last cell is unknown. Z is the (perhaps new) address of LST.

LST2AR(LST,AR,ARCNT). Packs the character string (STR) fields of list LST into array AR. ARCNT is the number of array words required.

Z=MERGE(NULST,OLDLST,PERCEL). Merges items in list NULST into list OLDLST. Each list has PERCEL (1 or 2) items per cell. All those items in NULST which do not already appear in OLDLST are prefixed to OLDLST. Z is the address of the resulting total list.

Z=NUMSTR(X,NC). Transforms the positive integer X into its representative alphanumeric character string, which is returned as Z. NC is the number of characters required for the number's character string representation in Z.

Z=NXNCHA(N,ALOC,AR,LAST). Retrieves from array AR the next N characters starting after the first ALOC characters. Z is the address of the new list built to contain the N characters. LAST is the address of the last cell in list Z.

Z=NXTCH(J). Z is the character J is pointing at in some list. J contains a list cell address and character position, and is reset to point to the next character.

Z=NXTCHA(J,AR). Z is the character J is pointing at in array AR. J contains the array index and character position, and is reset to point to the next character.

NXTNCH(NCH,LST,AR,ARCNT). Packs the next NCH characters from list LST into array AR. ARCNT is the number of array words required.

Z=NXTNCL(NC,X,LOC). Produces a copy of the next NC cells from cell X (inclusive). Z is the address of the new list containing the copied cells, or else zero if X or NC is zero or negative.

Z=ORDER(LST,FD,ORD). Rearranges the cells in list LST according to the values in the FD field of the cells. Cells are put in increasing order if ORD=1, in decreasing order if ORD=-1. In case of duplicate values, cells are ordered just as found. Z is the address of the ordered list.

Z=ORDINS(CEL,LST,PERCEL). Inserts item in SCD field of cell CEL into list LST, maintaining descending numerical order of the items in LST. LST contains PERCEL (1 and 2) items and associated weights per cell. Z is the (possibly new) address of LST.

Z=ORDMRG(NULST,LST,PERCEL,LOC). Ordered merge. LST and NULST are lists whose items are in descending numerical order, with PERCEL (1 or 2) items and associated weights per cell. ORDMRG inserts the NULST items into LST, maintaining correct order and combining weights of duplicate items. NULST is not altered. Z is the (possibly new) address of LST.

Z-PAKLST(LST,LOC,LAST). LST is a list whose cells contain character strings, up to 6 characters per cell. PAKLST makes a copy of LST in which the strings are packed, 6 characters per cell, with trailing blank fill in the last cell. Z is the address of the new, packed list. LAST is the address of the last cell in Z.

Z=PFL(PL,LST). Prefixes (concatenates) list PL to list LST. Z is the address of the resulting list.

Z=REPLAC(SA,SB,LST). Replaces the first occurrence of sublist SA by sublist SB in list LST, where the second sublist is considered null if SB=0. Z is the address of the SA match in LST, or else is -1 if SA was not matched. A sublist "match" means word 2 of each cell matched. If SA=SB, REPLAC just locates sublist SA, but does not change LST.

Z=SEARCH(ITEM,IWT,LST). Searches through list LST for a half-word data field containing ITEM. If the value in the weight field associated with the matching half-word equals or exceeds IWT, then Z is the address of the successful cell. Z is -1 if the ITEM was not matched or IWT was not satisfied.

STAK(STACK,A,B,ADA,ADB). A cell containing the values of the last four arguments is pushed down onto the STACK list.

UNSTAK(STACK,A,B,ADA,ADB). Removes the top cell from the push-down STACK list, and retrieves from the cell's fields two integers A and B and two integer addresses ADA and ADB.

7. Recursion in SMALL

Since FORTRAN allows no recursive calls of subprograms, recursion is simulated in SMALL by a process similar to that used in SAC-1 [Collins, 1971]. The process is based on the use of a pushdown stack list and the subroutines STAK and UNSTAK (described in the preceding section). When use of a recursive procedure is necessary, the essential quantities describing the status of the current execution of the procedure are placed on the pushdown stack. Then program control is transferred to a point at which the procedure is initiated for another execution.

When the execution of the procedure is completed, the next (top) cell of the pushdown stack is popped off. This cell contains the essential values to restore the status of the previous procedure execution, which instigated this most recent "recursive call" of the procedure. When the pushdown stack is empty, the "top level" execution of the recursive procedure has been completed.

Examples of METQA's use of this technique for recursion are the building of "covers" of input strings (see Chapter 5, Section 3) and the matching of character string patterns which may contain (possibly nested) indirect references to other patterns (see Chapter 4, Section 6).

REFERENCES

Bledsoe, W. W. and Browning, I. (1966). Pattern recognition and reading by machine. In [Uhr, 1966], pp. 301-316.

Bobrow, Daniel G. (1968). Natural language input for a computer problem-solving system. In [Minsky, 1968], pp. 146-226.

Chomsky, Noam (1964). A transformational approach to syntax. In [Fodor and Katz, 1964], pp. 211-245.

_____________ (1965). Aspects of the Theory of Syntax. MIT Press, Cambridge, Mass.

Collins, George E. (1971). The SAC-1 list processing system. University of Wisconsin Computing Center Technical Report No. 24, Madison, Wis., July 1971.

Craig, James A., et al. (1966). DEACON: DIrect English And CONtrol. Proc. AFIPS 1966 Fall Jt. Comp. Conf., Vol. 29, Spartan Books, New York, pp. 365-380.

Feigenbaum, Edward A. and Feldman, Julian (1963). Computers and Thought. McGraw-Hill, New York.

Fodor, Jerry A. and Katz, Jerrold J. (1964). The Structure of Language. Prentice-Hall, Inc., Englewood Cliffs, N. J.

Green, Bert F., et al. (1963). Baseball: an automatic question answerer. In [Feigenbaum and Feldman, 1963], pp. 207-216.

Harris, Zellig S. (1964). Co-occurrence and transformation in linguistic structure. In [Fodor and Katz, 1964], pp. 155-210.

Kellogg, Charles H. (1968). A natural language compiler for on-line data management. Proceedings AFIPS 1968 Fall Joint Computer Conference, Vol. 33, Thompson Book Co., Washington, D. C., pp. 473-493.

Kellogg, Charles H., et al. (1971). CONVERSE natural language data management system: current status and plans. In Proceedings of the Symposium on Information Storage and Retrieval (Jack Minker and Sam Rosenfeld, eds.), Conference Division, University of Maryland, 1971.

Klein, Sheldon (1965). Automatic paraphrasing in essay format. Mechanical Translation, Vol. 8, No. 3 and 4, June and October, 1965, pp. 68-83.

Klein, Sheldon, et al. (1968). The Autoling system. Computer Sciences Tech. Rept. No. 43, Univ. of Wis., Madison, Wis., Sept. 1968.

Lamb, Sydney M. (1969). Linguistic and cognitive networks. Paper prepared for Symposium on Cognitive Studies and Artificial Intelligence Research, March 1969, Univ. of Chicago Center for Continuing Education, Wenner-Gren Foundation for Anthropological Research.

Lindsay, Robert K. (1963). Inferential memory as the basis of machines which understand natural language. In [Feigenbaum and Feldman, 1963], pp. 217-233.

Luhn, H. P. (1958). The automatic creation of literature abstracts. <u>IBM Journal of Research and Development</u>, Vol. 2, No. 4, pp. 159-165.

Maron, M. E. and Kuhns, J. L. (1960). On relevance, probabilistic indexing and information retrieval. <u>Jrnl. ACM</u>, Vol. 7, No. 3, July 1960, pp. 216-244.

Meadow, Charles T. (1967). <u>The Analysis of Information Systems</u>. Wiley and Sons, New York.

Minsky, Marvin, ed. (1968). <u>Semantic Information Processing</u>. MIT Press, Cambridge, Mass.

Quillian, M. Ross (1968). Semantic memory. In [Minsky, 1968], pp. 227-270.

________________ (1969). The teachable language comprehender: a simulation program and theory of language. <u>Comm. ACM</u>, Vol. 12, No. 8, Aug. 1969, pp. 459-476.

Raphael, Bertram (1964). A computer program which "understands". <u>Proc. AFIPS 1964 Fall Jt. Comp. Conf.</u>, Vol. 26, Pt. 1, Spartan Books, New York, pp. 577-589.

Selfridge, Oliver G. and Neisser, Ulric (1963). Pattern recognition by machine. In [Feigenbaum and Feldman, 1963], pp. 237-250.

Shapiro, Stuart C. and Woodmansee, George H. (1969). A net structure based relational question answerer: description and examples. In [Walker and Norton, 1969], pp. 325-345.

Siklossy, Laurent (1968). <u>Natural Language Learning by Computer</u>. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn.

Simmons, R. F., Burger, J. F. and Schwarcz, R. M. (1968). A computational model of verbal understanding. Proc. AFIPS 1968 Fall Jt. Comp. Conf., Vol. 33, Thompson Book Co., Washington, D. C., pp. 441-456.

Simmons, R. F., Klein, S. and McConlogue, K. (1964). Indexing and dependency logic for answering English questions. American Documentation, Vol. 15, No. 3, pp. 196-204.

Uhr, Leonard (1964). Pattern-string learning programs. Behavioral Science, Vol. 9, No. 3, July 1964, pp. 258-270.

____________ ed. (1966). Pattern Recognition. Wiley and Sons, New York.

____________ and Jordan, Sara (1969). The learning of parameters for generating compound characterizers for pattern recognition. In [Walker and Norton, 1969], pp. 381-415.

____________ and Vossler, Charles (1963). A pattern-recognition program that generates, evaluates, and adjusts its own operators. In [Feigenbaum and Feldman, 1963], pp. 251-268.

Walker, Donald E. and Norton, Lewis M., eds. (1969). Proceedings of the International Joint Conference on Artificial Intelligence, Washington, D. C.

Woods, W. A. (1967). Semantics for a question-answering system. Mathematical Linguistics and Automatic Translation Report No. NSF-19 to the National Science Foundation. The Aiken Computation Laboratory, Harvard U., Cambridge, Mass., Sept. 1967.

____________ (1968). Procedural semantics for a question-answering machine. Proc. AFIPS 1968 Fall Jt. Comp. Conf., Vol. 33, Thompson Book Co., Washington, D. C., pp. 457-471.