A MEMORY-PROCESS MODEL OF SYMBOLIC ASSIMILATION

William C. Mann

Carnegie-Mellon University

Prepared for:

Defense Advanced Research Projects Agency
Air Force Office of Scientific Research

April 1974

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD/A004331

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 75 - 0132	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A MEMORY-PROCESS MODEL OF SYMBOLIC ASSIMILATION		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) William C. Mann		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D A0-2466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE April 1974
		13. NUMBER OF PAGES ~~289~~
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research (NM) 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number) We describe research on problems of using knowledge to make available information useful, which we call "assimilation" problems. The resulting theory contributes to psychology as a model of human short term memory, and to information science as an effective collection of new general methods. The vehicle for study is a computer program, called the Slate system, which manipulates knowledge and experience represented as labeled directed graphs. We seek to understand how people isolate, identify and remember complex objects, given information which is noisy, loosely structured, incomplete and which may represent several objects rather than one. People somehow organize

abstract (continued)

information into meaningful units and hold organized information for immediate use in a way that depends on its organization. The Slate system represents a particular selection from an infinity of possible models for such activities. It embodies a memory management method, a notion of a meaningful unit of information, and processes which relate synthetic knowledge to synthetic experience. We compare the system's performance on a broad range of experimental tasks, so that there is a genuine issue of whether its methods will work at all. The tasks include a synthetic noisy speech task in which response to syntactic and semantic structure is sought, a digit-encoding ordered recall task and a free recall chunking task. Both the qualitative features of the range of human performance and the detailed memory capacity behavior are studied. Representing those psychological experiments does not stretch the system to its limits. Another group of tasks having significant non-sequential structure is used to increase the diversity of assimilation problems studied. Discovering control structure constructs in compiled instruction streams, completing partial control structure descriptions, and interpreting the connectivity of the Necker cube are performed by the same system. These tasks are used to investigate the power of its graph-processing methods. A single constuctive partial mathc method is used in performing all of the tasks.

A MEMORY-PROCESS MODEL OF SYMBOLIC ASSIMILATION

William C. Mann

APRIL 1974

Ph. D. Thesis

Computer Science Department

Carnegie-Mellon University

Pittsburgh, Pennsylvania

This research was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract F44620-73-c-0074.

ABSTRACT

We describe research on problems of using knowledge to make available information useful, which we call "assimilation" problems. The resulting theory contributes to psychology as a model of human short term memory, and to information science as an effective collection of new general methods. The vehicle for study is a computer program, called the Slate system, which manipulates knowledge and experience represented as labeled directed graphs.

We seek to understand how people isolate, identify and remember complex objects, given information which is noisy, loosely structured, incomplete and which may represent several objects rather than one. People somehow organize information into meaningful units and hold organized information for immediate use in a way that depends on its organization. The Slate system represents a particular selection from an infinity of possible models for such activities. It embodies a memory management method, a notion of a meaningful unit of information, and processes which relate synthetic knowledge to synthetic experience.

We compare the system's performance against human performance on a broad range of experimental tasks, so that there is a genuine issue of whether its methods will work at all. The tasks include a synthetic noisy speech task in which response to syntactic and semantic structure is sought, a digit-encoding ordered recall task and a free recall chunking task. Both the qualitative features of the range of human performance and the detailed memory capacity behavior are studied.

Representing those psychological experiments does not stretch the system to its limits. Another group of tasks having significant non-sequential structure is used to increase the diversity of assimilation problems studied. Discovering control structure constructs in compiled instruction streams, completing partial control structure descriptions, and interpreting the connectivity of the Necker cube are performed by the same system. These tasks are used to investigate the power of its graph-processing methods. A single constructive partial match method is used in performing all of the tasks.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The fear of the Lord is the beginning of knowledge.

Proverbs 1:7

Special thanks are due to my advisor, Allen Newell, for generous gifts of time and patience, and to the many people who provided the generous computational support which made this kind of exploration possible. I am grateful to the other members of my thesis committee, Raj Reddy, William Chase and especially Herbert Simon, for their repeated attention and guidance. The work has been supported in major part by the Advanced Research Projects Agency, and also in part by the Alcoa Foundation and IBM Corporation.

The early encouragement of Albert Schild is gratefully acknowledged, and special thanks go to my family, who have been helpful in hundreds of ways.

LIST OF FIGURES

GENERAL INTRODUCTION

CHAPTER 1-1

The overall concern which forms the context for this research is a desire to create a comprehensive theory of intelligence. We would like to know how intelligent systems may and may not be organized, how existing intelligences manage to behave intelligently, how to characterize tasks which such intelligences might perform, how one intelligent system may imitate another, and so forth. Such a theory would (as a subpart) explain how humans think, the nature of meaning in language and a host of other currently obscure questions. At present, there is no such comprehensive theory, nor is there an approach (including mine) which seems about to yield such a theory. There are not even component theories to cover some of the major parts of the domain.

TASK

In this thesis we focus on a small but vital portion of the activity of intelligent systems, dealing with knowledge about the world in which the system is embedded. This world consists of a diversity of phenomena which are not under the control of the system. The system's access to information about the phenomena is limited in a variety of ways. Such factors as the physical limitations of sensors, the unavailability of the past, its limitation to particular locations and times and the presence of noise are unavoidable to the system.

A significant part of the intelligence of humans and some other systems lies in their collections of goals and methods which may be applied in order to pursue goals. We can observe that the particular limitations of access to the world at any moment are largely irrelevant to the methods and goals of the moment. So, for example, an experimental subject solving a cryptarithmetic problem must cope with gaps in written letters on the blackboard and with street noises which obliterate parts of the experimenter's comments, but the particular gaps and noise have little to do with the progress of his solution.

We use the term "assimilation" to denote the act of converting available information on phenomena outside of a system into a form directly useable by problem solving methods of the system. Figure 1.1 shows the relationships of information access between a world, a problem solver, access paths and an assimilator.

Some of the features of assimilation that make it a challenging problem are:

1. Incompleteness - Only partial information is available about the phenomena, and there is a large amount of variability in the ways in which the information may be incomplete.

2. Compositeness - Information about multiple phenomena may be available only in mixed form.

3. Distributedness - Information about a single phenomenon may be distributed rather than available in a single region of the available information, so that identification of the parts and joining them into wholes is difficult.

4. Bulkiness - Available information may be voluminous relative to the amount of information needed to characterize the phenomena. The bulk may arise both from redundancy of relevant information and from the presence of information irrelevant to the methods of the problem solver.

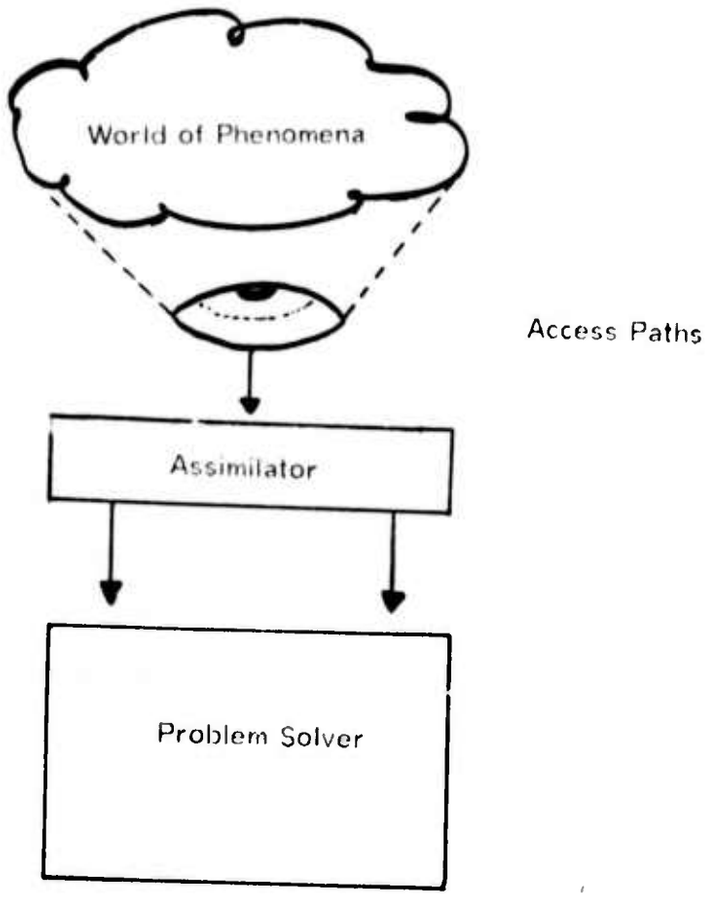


Figure 1.1- Assimilating Phenomena for a Problem Solver

5. Uniqueness - The particular ensemble of available information may always be unique in the experience of the system, thereby limiting the methods useful for assimilating it.

6. Abstractness - The facts of interest to the problem solver may be expressible only by multiple steps of abstraction from the available information, being effectively in a different language.

7. Complexity of Judgment - There are many kinds of evidence that may bear on the acceptability of a characterization of available

information. The possibility of making good characterizations may depend on being able to respond to a diversity of kinds of evidence in a way which tends to select highly evidenced characterizations.

Information processes which assimilate given information are of direct interest for a theory of general intelligence. We would like to have explicit means for performing many tasks which seem so trivially easy to people and yet which elude the methods which we understand. Creating methods which can respond to the kinds of information to which people respond turns out to be difficult. One difficulty is that representing the problem to be solved often seems to provide the answer as well as the question. The selection and assimilation of the relevant data make the remaining steps of problem solving seem quite obvious. The processes which perform the problem preparation need to be included in order to capture the entire intellectual act. For example, in the well-known "monkey and bananas" problem, which has been used as a task for a variety of problem solvers, the room is typically represented to the solver as three "locations," one where the monkey is, one where the box is, and one under the bananas. The walls, the door, the details of the box are not represented because they are irrelevant in the particular problem. If the monkey were physical rather than symbolic, the perception of the box as an object separable from the floor would be part of the problem. It is not included because the assimilation has been performed by the experimenter.

The task of an assimilator is to present to the problem solver a highly plausible characterization of the information which it receives. The characterization should be in a language or form useable by the problem solver, protecting the problem solver from the bulk and incompleteness of the received information. Since we are seeking a theory

of intelligence we are particularly interested in symbolic activity, the discrete actions and symbol processing actions rather than continuous actions and motor actions.

SCOPE OF STUDY

This study of assimilation seeks broadly applicable, general solutions to assimilation problems. They are studied with relatively little coupling to important parts of their context, namely specific sources of information, specific goals and tasks for the intelligent system, and specific methods which must utilize assimilated information. The sources of information used are all rather synthetic, and the applications of the resulting information all rather primitive. Both the generality of scope and the related separation from process context contrast with much of the related work in both psychology and computer science. So we should examine the choice. What kinds of results and benefits depend directly on choosing such a general scope?

The potential benefits are of two kinds. The broad approach yields knowledge of the nature of assimilation tasks. We really have very little information on whether the differences in assimilation tasks are superficial or not, on how they differ, on how to characterize particular tasks. This thesis provides evidence that the similarities between assimilation tasks are substantial, that broadly applicable methods exist and that development of those methods yields immediate benefits for the spectrum of assimilation problems rather than being restricted to particular sensory modalities or problem solvers.

Another benefit is that the practical range of accomplishments of computational methods may be expanded.

We need to look for the evidence that methods developed in such circumstances are likely to be useful. Why do we suppose that when our methods are required to deliver information to particular problem solvers, pursuing particular non-trivial goals, that such methods will remain relevant and effective? Surely there will be remaining unsolved problems, some of which can be glimpsed in this thesis. Surely there are ineffective combinations of assimilator, problem solver and task. We need evidence that assimilation can be effective when the eventual uses of the information are unknown.

The feasibility of assimilating information in a manner independent of its particular use is easily established. People often encounter information long before taking up a goal for which it is relevant. What they see and hear is independently characterized and remembered, with substantial reconstruction from that characterization. [BF71],[N67]

Natural language functions as a representation for the results of many kinds of assimilation. Our common experience is that the language is generally adequate for describing to others what we see and hear, that it can represent new and unique experiences without much change in the representation schema itself. So, to the extent that human assimilation tends toward describable outcomes, it appears as a general facility which is independent of goal and methods, and which spans several sensory modalities. In the Whorfian view, its forms limit the range of possible assimilations in a fundamental way. [W56]

It is less clear to what degree assimilation methods can be developed without reference to the characteristics of particular information sources. However, tasks which require joint interpretation of information from sensory modalities do not seem particularly difficult for humans. This suggests that use of information from more than one modality in assimilation may occur at very primitive levels. At any rate, there seems to be little evidence that the nature of the problem varies greatly from one modality to another. [M73]

NATURE OF RESULTS

For any particular computer program which does something we can evaluate it in two different ways, for its psychological aspects or for its information processing aspects. We can ask:

> How does this program contribute to our understanding of human intelligence as a model of human processing?
>
> How do the methods used in this program contribute to knowledge of general intelligence?

In this thesis we pursue both of these questions. The psychological question is approached by comparison of a computer program (called the Slate system) with data on human performance from tasks posed in experimental psychology. The question of the contribution of the methods is answered by evaluating the same program in the context of the field of artificial intelligence. The remainder of the thesis is divided on this basis,

chapters 2 to 7 being focused on the psychological question and chapters 8 to 18 on the question on methods.

INTRODUCTION TO THE SLATE SYSTEM

The Slate system performs assimilation tasks by processing directed graphs.* Its major information flows are shown in Figure 1.2 . One memory, called the Slate, holds a graph which represents the state of knowledge of the system about its given input information. All of the input information is deposited in this graph, and each addition to the knowledge of the input is an expansion of this graph. The Slate is the system's representation for human short-term memory.

A second memory (called bulk memory) holds many small directed graphs (called chunks.) This memory is the system's representation for human long-term memory. Under the general direction of a task control process for the particular task at hand, five kinds of processes affect the content of the Slate.

1. Input processes, which insert information about the world of phenomena external to the system.

2. A search process, which seeks chunks in bulk memory which resemble parts of the Slate content.

---

* We presume that the reader is familiar with the notion of a directed graph. For examples, see some of the "arrow diagrams" which are numerous in this thesis.

3. A match process, which investigates the resemblance between the chunks located by the search process and the Slate content. It judges whether, according to a combination of several rules of evidence, some part of the Slate content can be treated as a subpart of the chunk, (i.e. whether the concept represented by the chunk is plausibly also represented in the Slate.) If so, the entire chunk is used to expand the Slate content by a kind of copying, completing the match operation, and a new search and match attempt is begun.

4. A Slate space management process, which limits the content of the Slate to a fixed number of chunks. A chunk in the Slate is a named set of arcs. Each arc may be in one or more chunks. When the match process puts a new chunk in the Slate, all the arcs of the other chunks which overlap it are included in the new chunk. When the number of chunks in the Slate exceeds the limit, the space management process selects one for removal. A chunk is removed from the Slate by changing the status of each arc in the Slate that was in the chunk, and by removing from the Slate any arch that was in only the chunk being removed. Selection of a chunk for removal is random under certain constraints.

5. Output processes in some tasks scan the Slate, producing output (corresponding to response by experimental subjects) and record in the Slate the fact that the output was produced.

Most of the interesting features of the system arise from the character of the match process and the space management process. Two of these features are illustrated in the example below: the role of conflict in directing assimilation, and the relationship between successful matching and use of memory space.

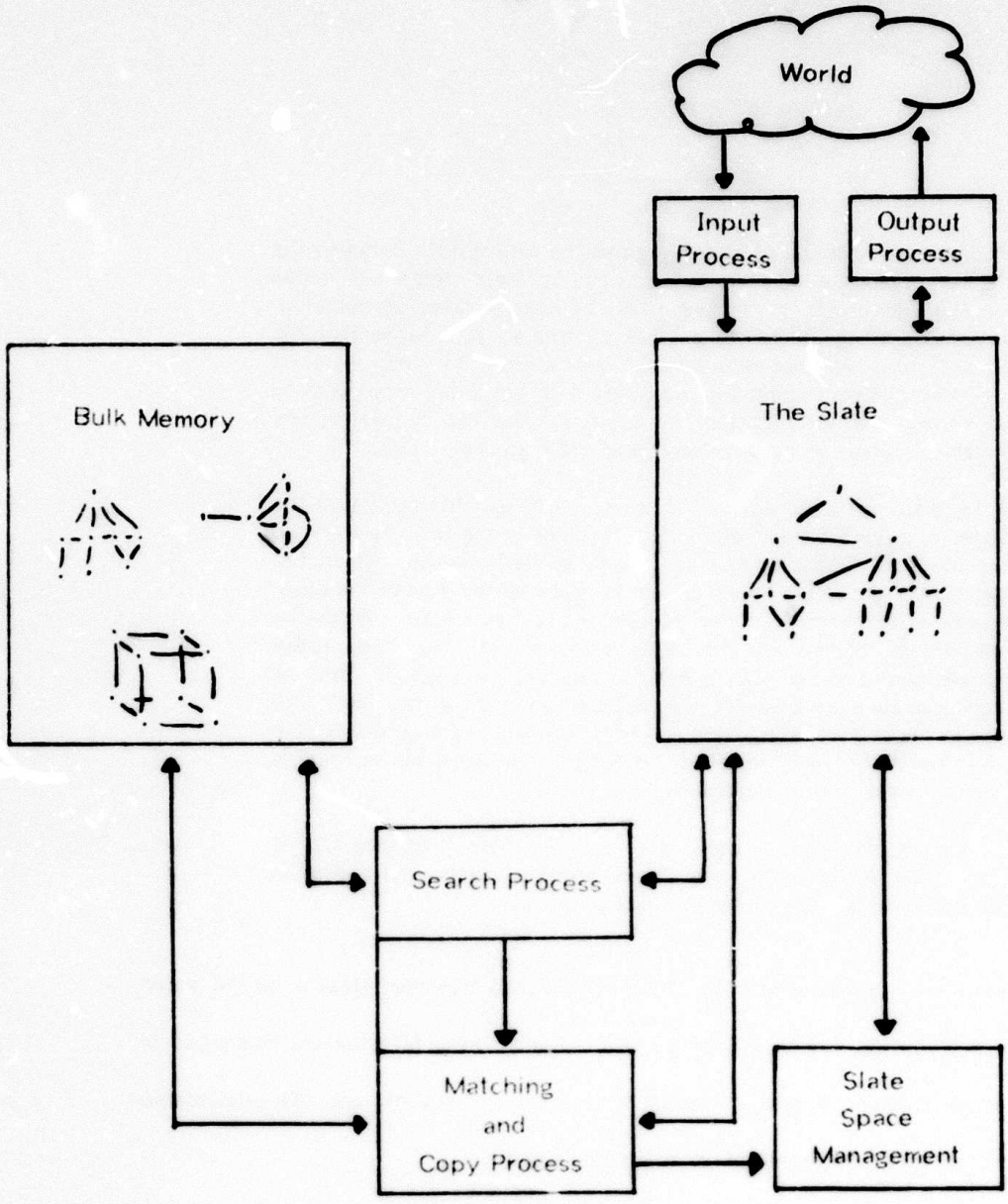


Figure I.2- Major Information Flows In The Slate System

EXAMPLE

The Slate system has received as input the three letter sequence C, A, T. It attempts to relate this sequence to stored knowledge, first trying a chunk for the word ACT and then for the word CAT.

The graph created by the input process to represent the sequence is shown in Figure 1.3 .* It consists of three chunks, indicated here by the

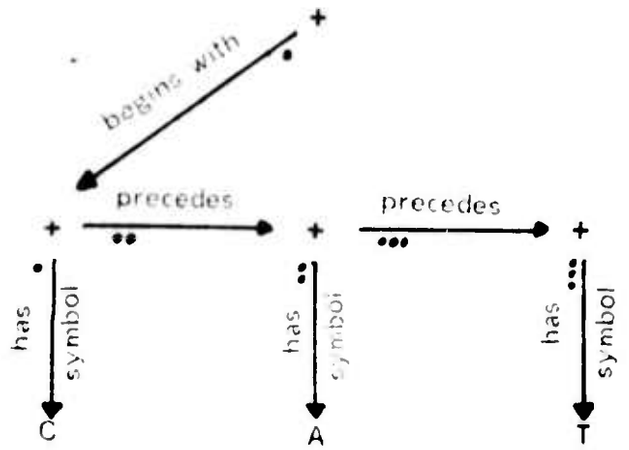


Figure 1.3- Sequence C, A, T Has Been Received

groups of dots on the arcs.

The chunk for the word ACT is shown in figure 1.4 . The system tries to reconcile the two by finding good correspondences of some of the tokens, represented in the figures by +'s.

There are several ways in which the correspondences might be made. The simplest is to let the tokens correspond in the left-to-right order shown in the figures. The result, after copying, would be the graph in Figure 1.5 . It refuses to form this graph because it contains tokens for events having more than one symbol. The system has enough knowledge of the "has symbol" relation to require that there be at most one such arc from any

---

* A unique name is given to each vertex in the Slate system. For simplicity, names of tokens which do not appear in this text are replaced by + signs in most figures.

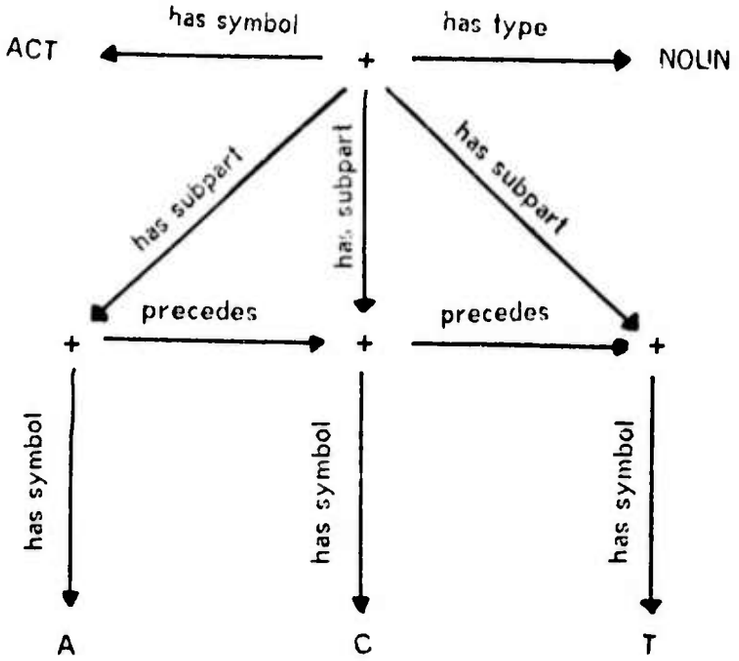


Figure 1.4- Chunk For The Word "ACT"

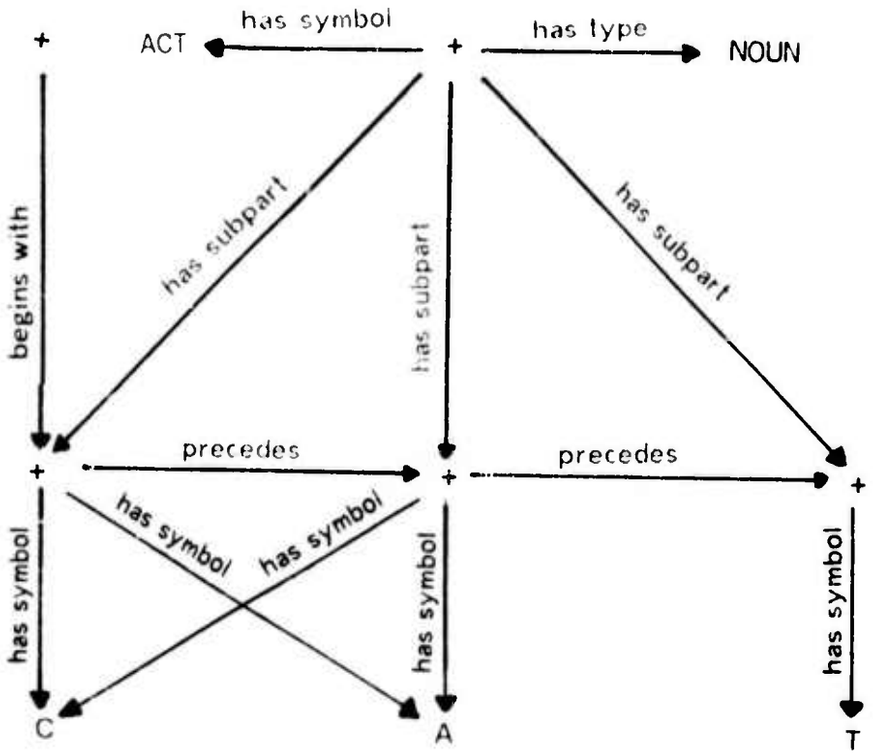


Figure 1.5 - Attempt To Treat C, A, T as ACT

particular vertex in any of its graphs (There are several other possible correspondences which the system rejects based on its knowledge of the "precedes" relation as a total order relation.)

There are no acceptable ways to assimilate the sequence C, A, T as the word ACT, and so the effort to use the chunk for ACT fails and has no effect on the Slate content. The chunk for ACT is rejected despite the positive evidence (letters, length, ending) that it might be correct. On the other hand, the chunk for the word CAT, shown in figure 1.6, can be used without violations of any of these restrictions. The resulting Slate content is shown in Figure 1.7, in which four new arcs have been copied into the Slate.

All of the arcs in the Slate are in one new chunk. The three chunks which it has overlapped are marked as available for deletion. The way that the three have been combined into one chunk illustrates the system's capacity to recognize and treat as single units configurations which correspond to chunks in its bulk memory.*

The example above represents the basic actions of the match process and Slate space management.

---

* For simplicity one detail has been left out of the chunks shown in the figures. An arc on the relation "is the source of" relates the vertex representing the entity (in this case the word) to a unique constant vertex for each chunk. See chapter 12.

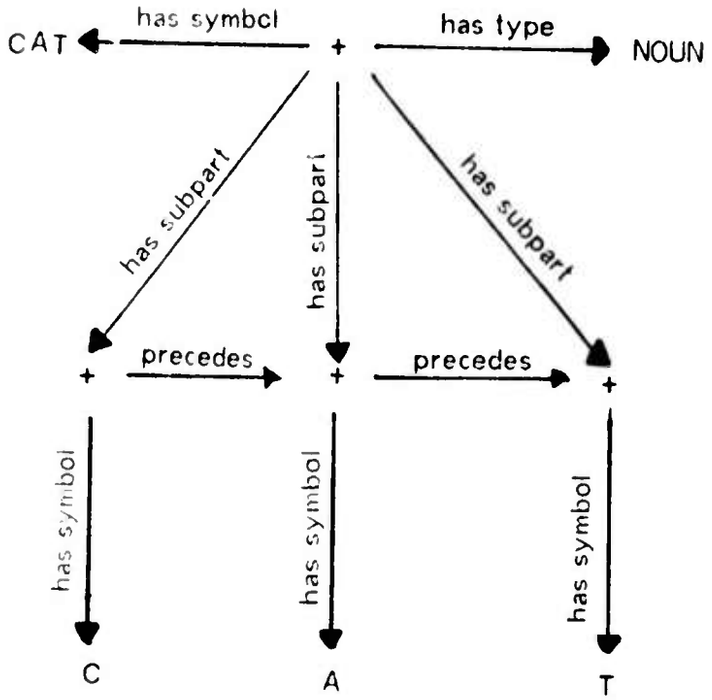


Figure 1.6 - Chunk for the Word "CAT"

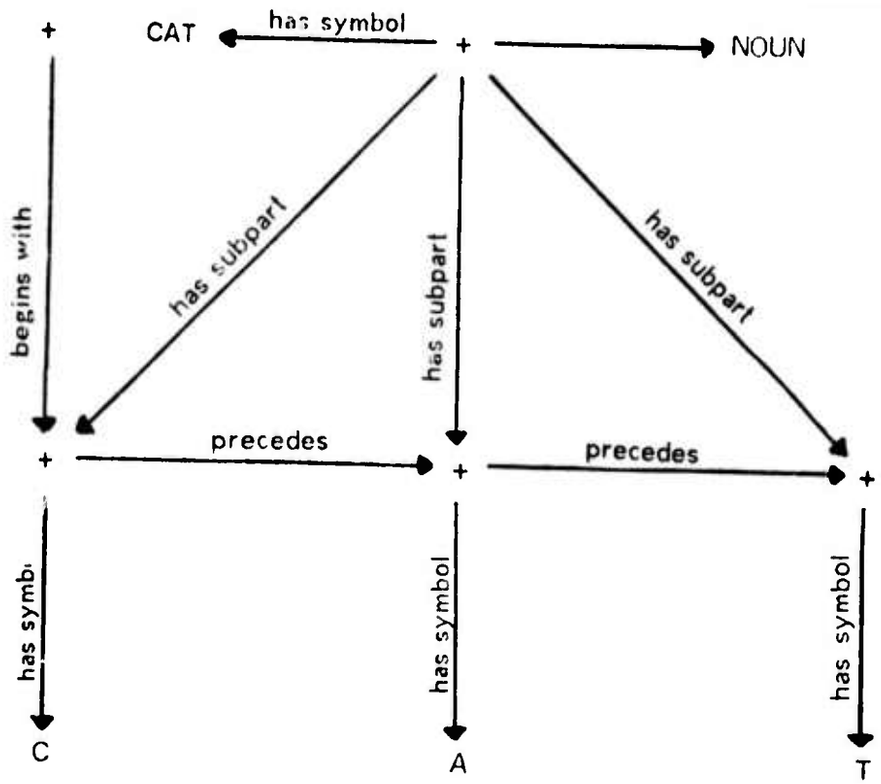


Figure 1.7 - Successful Attempt to Treat C, A, T as CAT

STUDY OF SHORT TERM MEMORY

CHAPTER 2 - I

SCOPE

In this thesis we seek to develop a model of how certain functions of human intelligence are accomplished. The functions arise in "short term memory" experiments in experimental psychology, where they are grouped under the term "chunking."

The usual experimental arrangement, which is followed in all of the specific experiments treated below, is this: One or more subjects is given a repetitive memory task. Each cycle of the task consists of a presentation of information, its withdrawal, and a reproduction of some of the information by the subject. Each cycle typically lasts less than a minute. The joint variations of reproduction performance and presentation conditions over large numbers of presentations are characterized in the results.

Several features found frequently in interpretations of such experiments can be taken as a sketch of the idea of short term memory (STM):

1. The subject has an approximately constant capacity of items, called "chunks."
2. A variable amount of symbolic content is in a chunk, determined primarily by past experience with closely related symbols.
3. The subject has relatively fast access to his "long term memory" for purposes of retrieving a chunk.
4. He has much slower access to long term memory for purposes of depositing a chunk.

Many experimental conditions lead to speech-related error patterns or to active rehearsal of items by the subject during the delay period.

The organization of presented information into chunks by the subject is usually designed to be either dominated by correspondence with past experience (as in grouping a sequence of letters or sounds to form a familiar word) or dominated by formation of new, unfamiliar groupings (as in grouping the letter sequence JRSTCAMLEJSPXCT, which might be presumed unfamiliar.) Of the two kinds, we model only the phenomena of familiar arrangements in this thesis.

One of the paradoxes of chunking is that a subject may regularly treat a particular set of presented elements as one chunk, part of a chunk or several chunks depending on the arrangement of elements, so that the same elements occupy varying amounts of memory capacity. So, for example, the letters

ERAMDLO

could be treated as up to seven chunks,

MAREOLD

as two, and

OLDMARE

as one, even though the same set of letters is involved in each case. The elements which are units to the experimenter are not treated consistently by subjects, who tend to group "unrelated" objects in ingenious, personally meaningful ways. Tulving, [T68] in a review paper, noted that

> "One of the important problems facing experimenters and theorists interested in free-recall -- as well as those concerned with other memory tasks -- lies in the specification of the functional units of material that are

> remembered and recalled. We may be ignorant as to the exact S(ubject's)-units in any given situation, and we may have to temporize by counting such easily identifiable units as trigrams or words, but sooner or later we have to come to grips with the S-units in an objective fashion. In the long run, nothing will be gained by pretending that gaps between sequences of printed letters or between sequences of spoken phonemes define the units of information processed by the human memory system.

Since chunking is included in such a broad spectrum of behavior, a theory which describes how chunking is performed can be expected to account for parts of many experiments and for the entirety of a few.

This thesis presents an account of several phenomena related to chunking, including memory-capacity/ coding effects, some interference effects, and some joint effects of noise, syntax and meaning on reproduction of sequences of word elements. The method of investigation is to study an explicit model of chunking processes, represented as a computer program. The experimental conditions for the subjects of several published experiments are presented to the program. Its behavior corresponds in significant ways to the experimental results. The properties of the program which create or prevent particular correspondences are theoretically significant. The model of STM is intentionally a partial one, so that for example the processes by which existing chunks are accepted or rejected as immediately relevant are represented, but the creation of new chunks from novel experiences is not.

The presentation of the computer program is reserved to the non-psychological chapters. There it is distributed, with various parts and attributes being introduced where they become instrumental. A complete presentation of the program is in chapter 13.

PROBLEMS

The psychological questions which this work addresses are:

1. How is chunking possible at all? What operations on the information given to the subject would be sufficient to produce the behavior in question?

2. How are particular chunks selected for use by subjects? Why are other similar chunks not used?

3. How can chunks be selected based on information which is incomplete in a-priori-unpredictable ways?

4. How is measured STM capacity (digit span or similar) related to underlying storage capacity and processes?

5. Do different kinds of chunks require different chunking processes?

6. What parts of the chunking task must be performed serially, and what parts are suitable for parallel performance?

METHODS

The methods used to deal with the problems cited above all center around the creation and study of a computer program in correspondence to prior STM experiments conducted with human subjects. In each case the program acts as a scrutable subject, performing in ways which correspond to the performance of the human subjects.

This approach is one of two commonly called "simulation." In the other, a program is put into part-by-part correspondence with a known system, such as a telephone exchange, so that the program performance may be studied. The aim is to answer questions about the performance of the simulated system. Here we put program performance in correspondence with human performance with the aim of answering questions about the nature of the human system. A principal advantage of using a program as a theoretical vehicle is that the consequences of assumptions about how the data could have arisen can be made observable. Done correctly, this eliminates from consideration theories which can not possibly be made explicit, and theories which do not in fact predict the performance which they are alleged to cover.

We use one program to cover all of the experiments studied rather than providing a separate version for each. This contrasts with a common practice of developing a separate model or set of assertions for each experiment or concurrent series, without verifying that the model covers prior experiments as well. There is a substantial risk, near certainty, that the models so developed will be inconsistent in ways that are not apparent from their descriptions. We would rather have a consistent model, or at least a set of models having known conflicts, hoping that as the issues of contrast are made known and resolved a single adequate model and theory will eventually emerge.

EXPERIMENTS

The human performance data for this study of STM all come from reports of published experiments. They are selected as representative of current views of the nature of STM and the phenomena which every STM model must account for. We deal with three topics below, some represented by more than one experiment. They are arranged roughly in order of increasing complexity.

1. Interference - represented by an experiment by Gordon Bower comparing interference effects of single words and multi-word cliches.

2. Coding - represented by Smith's work on his own remembering of streams of ones and zeroes, reported by George Miller; also by a 1965 group experiment in binary encoding by Pollack and Johnson.

3. Chunking based on incomplete information; chunking and syntax; chunking and meaning - all represented by Miller and Isard's 1963 experiment on recognition and memory for "sentences" heard with noise.

In all of our psychological tasks, the information is available serially, and order of events plays a significant role. This is not true of assimilation problems in general. Visual information is multidimensional, and auditory information can be usefully treated as multidimensional. There is evidence that our knowledge of the sequence in which information is received is often hazy, even for auditory information.

The assimilation methods which we propose should not be restricted to dealing with serial, tightly ordered information. The limitations of method which we accept into psychological models should reflect limitations observed in people's performance. When people's performance reflects their responding to serial order in an experiment, a model of their activity should provide some explicit representation of such response, since not all tasks provide such serial structure for the subject.

There is an example of Slate system action, described in detail in chapter 11, in which the given information describes the connectivity of the Necker Cube. (Figure 2.1 .) The system chunks the connections as a cube with a particular front face, corresponding to attention directions from the system user. Attention may be shifted so that the cube "reverses," that is, it is chunked with the other four corners on the

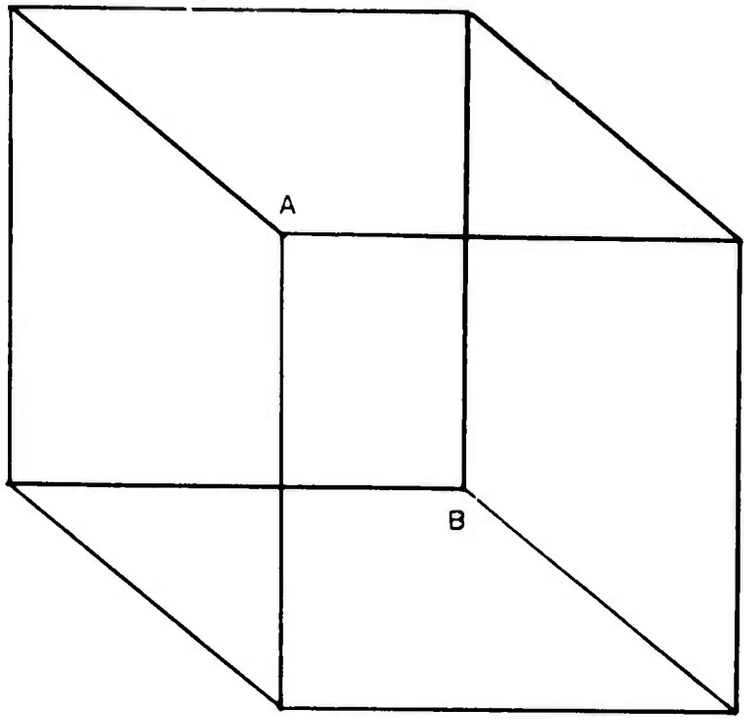


Figure 2.1 - Necker Cube

front face. Repeated reversal follows the directed shifts of attention. The Slate content is never jointly chunked with any corner on both a front face and a back face. The reversing property arises directly from the conflict-handling mechanisms which are basic to the system, and from the processes developed to deal with mis-anticipation on Miller and Isard's task.

The point to be made here is that the psychological scope of assimilation includes such tasks, and that building models which exclude them is at best a temporary expedient. They are part of the activity of reconciling our knowledge about the world with current information.

The treatment of the Necker Cube in this thesis is not a realistic simulation of human performance, although that seems feasible. Rather it is a means of demonstrating the scope of the problem, some particular properties of the Slate memories and the effectiveness of a general approach.

AN INTERFERENCE EXPERIMENT WITH CHUNKING

CHAPTER 3 - 1

The first and simplest experiment we address is on interference in free recall, by Gordon Bower. [B69]

> "This experiment is concerned with the "chunking" hypothesis applied to free verbal recall (cf. Miller, 1956; Tulving, 1968). For present purposes, a chunk of material may be identified as a highly integrated group of words, indexed by a strong tendency for Ss to recall the words together as a unit. The chunking hypothesis asserts that free recall is limited by the number of chunks that can be produced from memory without the aid of some specific retrieval scheme or cuing system such as the pegword mnemonic (cf. Wood, 1967). According to this hypothesis, recall improves over practice because the words become more strongly bonded into subjective chunks, and several chunks may be coalesced into a single chunk by a recursive (hierarchical) process of grouping subgroups. The number of chunks recalled is approximately constant, while the number of experimental items per chunk presumably increases with practice.
>
> ...
>
> METHOD
>
> Design
>
> The design involved a within-group comparison of three free recall lists of 24 units. Each 24-unit list consisted of a set of 12 "critical" units (words) plus 12 "filler" units. The filler units were varied over the three lists, and the hypothesis expects concomitant variation in recallability of the 12 critical units in the list. For the one-word list, the fillers were simply 12 nouns presented singly; for the three-word list, the fillers were 12 triplets of unrelated nouns presented as units; for the cliche list, the fillers were 12 familiar, three-word cliches presented as units. The critical words and filler units of a list were mixed for random presentation, and S freely recalled all he could. The chunking hypothesis expects recall of the 12 critical words to be about the same for the one-word and the cliche fillers, but significantly poorer with the three-word fillers.
>
> Materials and Procedure
>
> The three sets of 12 critical words were unrelated nouns selected for high concreteness from the norms of Paivio, Yuille, and Madigan (1968). The one-word fillers were similarly selected by the same criteria. The 12 three-word fillers were composed of unrelated nouns. The 12 cliches were

noun phrases of high frequency (intuitive estimates) in our Ss' linguistic community. For eight of the cliches, all three words had a noun form, although in the cliche context the first two functioned as adjectives. These eight cliches were: ball-point pen, mail-order catalog, Rose Bowl parade, birth control pill, ice cream cone, Bay Area transit, tick-tack-toe, and turtle neck sweater. The other four cliches contained some adjectives and were: Happy New Year, fair-weather friend, Great Salt Lake, and good old days. Most of the cliches have a single semantic referent, whereas the unrelated three-word fillers (e.g., couch, flag, sun) have three separate referents.

The three free recall lists were composed by pairing the three critical-word sets with the three filler sets, using the six possible pairing about equally often over Ss. The 12 critical words and 12 filler units were typed in capital letters on 24 flash cards, shuffled thoroughly, and shown to S at a rate of one card for 3 sec, with a 2-sec intercard interval. There was one study trial, then an immediate recall trial on each of the three lists. The Ss recalled in writing, giving as many words as they could in any order. Time allowed for recall was 96 sec for the one-word filler list, and 192 sec for the cliche and three-word filler lists; this is calculated at 4 sec recall time per word on the input list. This was always more than enough time for Ss to recall all they could.

The order of the three treatment lists within the session was counterbalanced over Ss. The Ss were 18 undergraduates fulfilling a service requirement for their introductory psychology course.

RESULTS

An initial observation is that the three-word cliches were recalled in perfect all-or-none fashion, either completely or not at all. On the other hand, the unrelated triplets were not recalled all-or-none, Ss frequently recalled some but not all of such triplets... Therefore, in the following, we adopt the convention of treating a three-word cliche as a single recall unit (i.e. 12 filler units) but unrelated triplets as three separate recall units (i.e., 36 filler units).

In these terms, Table 1 shows the average recall of critical words and of filler units for the three lists."

The principal conclusions are drawn from various analyses of Table 1, which appears as Table 3.4 in this chapter.

"These results support the implication of the chunking hypothesis: recall of the critical words was about the same whether the other 12 units were one-word or three-word (cliche) units, but recall was reduced if the other words comprised more (than 12) chunks.

...

> Recall of the filler units also supported the chunking hypothesis (line 2 of Table 1). Consistent with hypothesis, the cliches and one-word fillers were equally well recalled."

To model how the subjects performed the experiment, we must model their actions of reading cards, their means for retaining what was read, their means for reporting what they can, and some selective means for determining what is reported and what is not.

PROGRAM AND TASK

The Slate system performs this task by applying its operator called "RECALL WORDS." The action of this operator is to take a sequence of symbols one at a time from the input, storing them in an internal form in the Slate, until an end-of-sequence indication is given. The symbols of the sequence which are retained in the Slate are written as program output. This gross action thus corresponds to the action of an instructed subject who reads and reports one list. The chunks in bulk memory correspond to the cliches attributed to the subjects. One chunk would represent the fact that occurrence of the three symbols ROSE BOWL PARADE in an input sequence may be treated as one familiar unit. Appendix I gives complete examples of these chunks.

We have sketched enough of the Slate system to identify the principal correspondences between the interference experiment and the Slate system model of it. (Table 3.1)

ORIGINAL EXPERIMENT	SLATE SYSTEM MODEL
words	recognizable symbols
cliches	ordered sets of symbols which correspond to chunks in bulk memory
unrelated triples	ordered sets of symbols which do not correspond to chunks in bulk memory
subjects	the Slate system
instruction of the subjects in the task	use of the "RECALL WORDS" operator
reading a word	taking a symbol from the input
terminating a list (by running out of cards)	taking a special sequence-termination symbol from the input
reporting the words of a list	examining the Slate content for symbols

Table 3.1- Correspondences of Bower's Experiment and Program Replicate

The sequence of program steps is indicated in Figure 3.1.

For a list of 12 critical words and 12 three-word cliches, the cycle would be traversed 48 times, with one new successful match on each of 12 of the traverses.

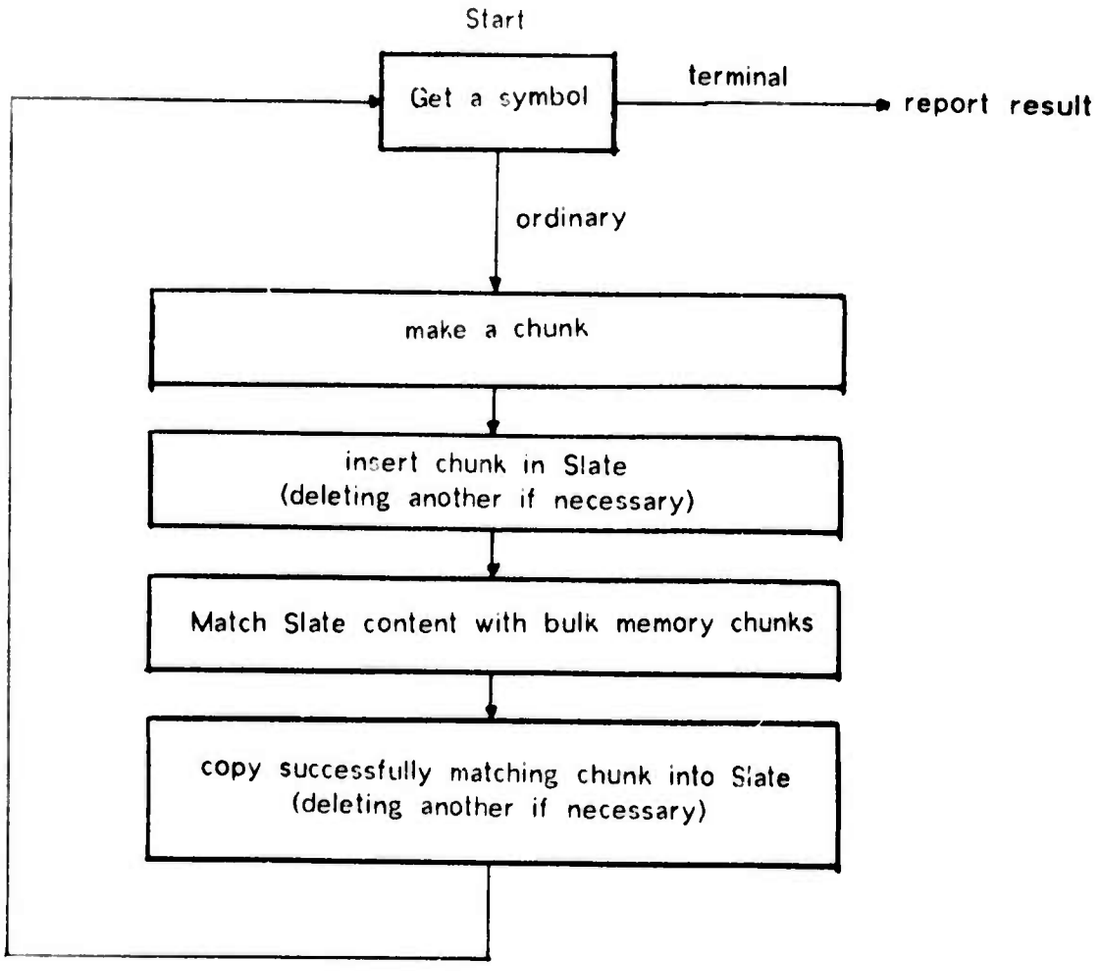


Figure 3.1 - Major Program Steps for Word Recall

RUNNING THE CORRELATE EXPERIMENT

The Slate system does not have any fixed symbol vocabularies or bulk memory chunks. These must be defined by suitable data before the experiment is run. For

convenience, we use the words of the cliches given in the article for building cliche chunks, and the words ACE, KING, QUEEN, JACK, RUFF, DUCK, FINESSE, TRUMP, HEARTS, SPADES, CLUBS, DIAMONDS as the unrelated critical words and filler words. Twelve chunks corresponding to the 12 cliches are entered into bulk memory. We have an advantage in that we can verify that words such as KING and QUEEN are unrelated in the system, and that TICK, TACK, and TOE are really functioning as separate words.

Lists of words corresponding to the original 48 lists can be prepared as input. However, for the two conditions in which the bulk memory chunks do not affect the outcome, the average number of words of each category which will remain in the Slate can be calculated on the basis of their equiprobable random selection and the capacity of the Slate. We calculate rather than use sampling procedures to derive results where applicable. We use 16 lists containing cliche words to verify the correct detailed functioning of the model, and a simpler equivalent version of the system to simulate 400 subjects. Each has a random permutation of 24 items, each of which is either a three-word cliche or a critical word. A sample list is shown in Table 3.2 below, together with the list termination symbol.

KING
RUFF
FAIR WEATHER FRIEND
MAIL ORDER CATALOG
TURTLE NECK SWEATER
TICK TACK TOE
DIAMONDS
GOOD OLD DAYS
SPADES
ICE CREAM CONE
FINESSE
TRUMP
BIRTH CONTROL PILL
GREAT SALT LAKE
CLUBS
BAY AREA TRANSIT
BALL POINT PEN
HAPPY NEW YEAR
ROSE BOWL PARADE
QUEEN
ACE
JACK
HEARTS
DUCK
QUIT

Table 3.2- Sample System Input List

The experiment with the cliche lists is performed by initializing the system and then performing a cycle of clearing the Slate, reading a list like the one above, and printing the symbols in the Slate, once for each list. Slate capacity for the series is set equal to the single word recall capacity, 13 chunks.

EVALUATION OF THE CORRELATE EXPERIMENT

The all-or-none effect for the recall of cliches was observed in the program output just as it was in the original experiment. We use the same filler unit construct in analyzing the results. Table 3.3 shows the results of this action for the 400 lists, and the calculated averages described above.

SLATE SYSTEM RECALL OF CRITICAL NOUNS
AND FILLER UNITS FOR THE THREE TYPES OF LISTS

	Fillers		
Type	One-word	Cliches	Three-word
Critical nouns	6.5	5.55	3.25
Filler units	6.5	6.13	9.75
Total units	13.0	11.68	13.0

Table 3.3- Slate System Recall Averages

Comparison of this table with Table 1 of the original experiment shows that the desired gross correspondence has been achieved.

RECALL OF CRITICAL NOUNS AND FILLER UNITS
FOR THE THREE TYPES OF LISTS

	Fillers		
Type	One-word	Cliches	Three-word
Critical nouns	6.3	5.5	3.7
Filler units	6.7	6.6	9.4
Total units	13.0	12.1	13.1

Table 3.4- Bower's Table 1 : Recall Averages

1.
The recall of critical words is the same in the presence
of either cliche or single word fillers,
but is lower in the presence of unrelated triples of filler words.

2.
Recall of cliche and single word fillers is about equal.

The presence of the appropriate chunks in bulk memory causes the Slate system to organize its record of the input word sequence in such a way that the interference unit is a chunk rather than a word.

Beyond these gross features, which arise from the fact that cliches are stored as single units, the Slate model also predicts the following:

1.
Recall will be slightly lower when cliches are used as fillers
than when single words are used.

2.
Part of this loss will be in recall of cliches, and
part will be in recall of critical words.

3.
The part which represents critical words will be larger.

We see that each of these is fulfilled in Bower's data and that the Slate model predicts the levels of recall rather accurately.

SHARED WORKSPACE EFFECT

The basis for these predictions, which we call the shared workspace effect, is as follows:

For single words, no chunking occurs, so that memory is filled by the presentation of the early part of the list and remains full for the rest of the presentation; recall equals the number of STM spaces available.* For lists containing cliches, STM may not be full at the end of the presentation of the list. If the list ends with a three-word cliche, then three spaces are used to receive the words, they are then grouped into a single chunk, and two spaces are left available. Items from the first part of the list are already lost, and so cannot be used to fill these spaces. Recall is therefore reduced for these cases. The workspace, STM, has been shared by word chunks (inputs to the cliche-chunking process) and cliche chunks (outputs from the cliche-chunking process.)

If the list ends with a cliche followed by a critical word, one space is left available. Otherwise, the list ends with two critical words, and all spaces are filled at the end.

---

* We recognize that this number may represent an effective value produced by composition of several process and memory resources.

Considering the probabilities of those events, the predicted reduction in recall is 1.26 items, comparable to 0.9 observed and 1.32 for the 400 simulated subjects. Since Bower's experiment was not controlled for the actual frequencies of various ending patterns, the 0.9 observation may in part reflect a lower than average rate of final and next-to-final cliches.

Because this recall loss is conditional on having cliches near the ends of lists, the critical words on the list have more opportunities to be deleted from memory, while the final cliches are protected by their recency. Thus more of the loss is expected to appear in the critical words, as is observed.

The model makes the following further predictions that are not verifiable in the data:

1. The magnitude of the reduction in recall with cliches relative to the single word case depends on the length (in words) of the cliches used, and on the ending configuration of the list, according to the following relationship:

> Let the average number of recalled single words be S, the length of the final cliche item in a list be C, and the position of the final cliche, counting from the end, be P. (P=1 is the last position.) Then the expected number of items to be recalled will be:
>
> | | |
> |---|---|
> | S | if no cliche occurs in the last C-1 list positions. |
> | S+P-C | otherwise. |

2. The magnitude of reduction in recall with cliches is constant over individual differences in single word recall.

3. The magnitude of reduction is also constant over changes in the number of other cliches in the list for any particular last-cliche position.

These predictions should be easy to verify in a variety of experiments. They indicate that chunking experiments should include control for the way that the given information ends.

These predictions and the fit to Bower's data both depend on the assumption that memory spaces are used for all of the subelements of a cliche. In terms of discussions elsewhere in this thesis, this is an assumption that the subject does not use "anticipation" to save memory space. We will see in the next chapter that the same assumptions allow us to predict deliberate coding behavior accurately. They are predictions of differences which do not depend on assumptions made about the size of STM, and so are not affected by treating STM size as a free parameter.

SUMMARY

The principal result of replicating this experiment is that the shared workspace effect predicts much more of the detail of the subjects' memory capacity variation than the chunking hypothesis alone.

DELIBERATE CODING AND SHORT TERM MEMORY

CHAPTER 4-1

The subject of deliberate coding has a central place among the phenomena associated with STM, especially since it received a prominent role in Miller's famous 1956 paper.[M56] Neisser [N66] says, in explaining chunking,

> "Often the 'chunks' which the subject stores and recalls are not those which were presented, perceived or originally stored. For this reason, we must assume that there is a verbal memory which is not simply echoic. The most elegant demonstration of this kind of recoding - and at the same time the clearest indication of the need for some such concept as the "chunk" - is S. L. Smith's experiment, reported in Miller's ... paper. Smith tested his own memory span for "binary digits," i.e., strings of zeroes and ones such as 0110010111010001. Having established his span (about 12), he deliberately memorized various methods of reading binary digits into other number systems. When he had learned "octal" numbering (001 = 1, etc. ...), his memory span for binary digits rose to nearly 36! In effect he was translating every triad of zeroes and ones into a single octal digit and then storing 12 of those. Similar results have been obtained by Pollack and Johnson(1965)."

In view of the conceptual prominence of coding it is necessary for our theory (or any other) of chunking to represent the relevance of deliberate coding experiments.

Coding experiments differ from most STM experiments in that fact that there is a deliberate voluntary re-representation which can be engaged in or avoided by the subject, and also by the small vocabulary and periodic structure of the codes studied and the use of the same vocabulary in more than one code. In the literature since 1956, deliberate coding has been invoked far more often than it has been studied.*

---

* Induced chunking has been more frequently studied. [BW69],[M70]. Nevertheless it is worthwhile to reexamine some of this literature in order to see how a knowledge of coding might contribute to a theory of chunking.

SMITH'S BINARY DIGIT EXPERIMENT

We cannot review the published report of Smith's experiment because the report is unpublished.* The closest representation of it seems to be Miller's description. The only point in studying it here is that it is an influential piece which is a source of data which we would hope to compare to a model of coding processes.

All of the quantitative results are given in Miller's Figure 9, (Figure 4.1 below,) which relates "memory span for binary digits" to "recoding ratio" by means of two curves, marked "observed" and "predicted from span for octal digits."* There are inconsistencies between the graph and the text which indicate that the figure is erroneous, at least for the "predicted" curve. The text indicates that Smith could recall 12 octal digits; the number assumed for 1:1 recoding ratio was 15 rather than 12, making the predicted and observed curves agree. The caption speaks of bases 2,4,8 and 10 whereas the graph

---

* During an attempt to locate a report corresponding to Pollack and Johnson's reference (which turns out to be erroneous), Dr. Smith told us that the report was not published, and that he had no copy.

---

* The original title is: "The span of immediate memory for binary digits is plotted as a function of the recoding procedure used. the predicted function is obtained by multiplying the span for octals by 2, 3, and 3.3 for recoding into base 4, base 8, and base 10, respectively." Thus the title and figure do not correspond.

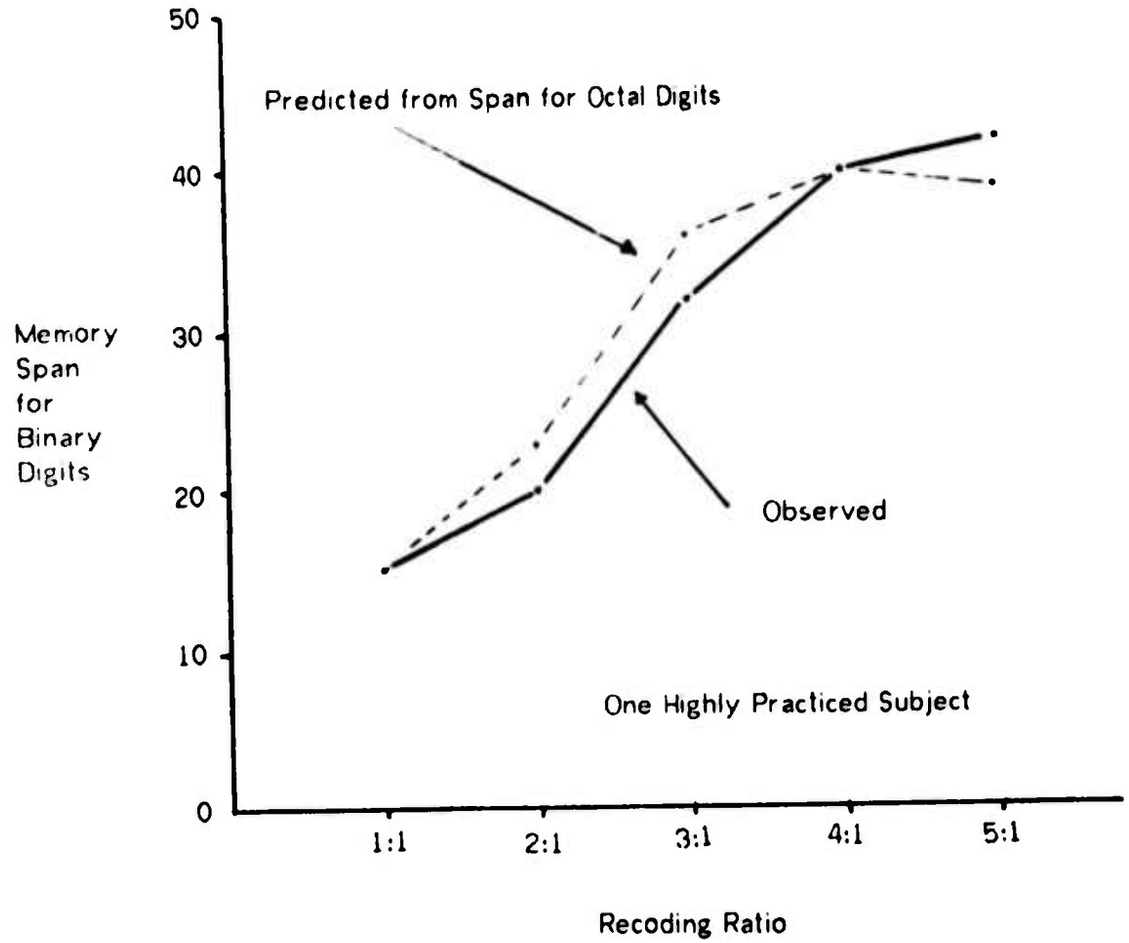


Figure 4.1 - Miller's Graph of Smith's Memory Span Data*

and remaining text deal with five recoding ratios.

It is hard to imagine the basis for the "predicted" curve, particularly for its departure from linearity. Several simple hypotheses which we have tried fail to fit the curve.

There are no obvious troubles with the "observed" curve. It would have been helpful to know what the criteria for judging "Memory Span for Binary Digits" were.

Another serious deficiency in the available description is that we are not told what the stimulus modality or the reporting modality were. This makes it difficult to judge its relationship to other experiments, especially attempted replications.

POLLACK AND JOHNSON'S CODING EXPERIMENTS

Pollack and Johnson [PJ65] describe their experiment as follows:

> "In 1952, Smith demonstrated that the memory-span for binary digits may be substantially increased following instructions about efficient coding procedures. The present study is a direct extension of Smith's study...
>
> METHOD AND PROCEDURE
>
> Subjects. A group of 6 university students was tested 1.5 hr. per day on 28 successive testing days with both verbal and tachistoscopic materials. They were then taught the decimal equivalent of the 4-digit binary code for 12 days, and, finally, were retested for 10 additional days.
>
> Memory-span materials. Verbal messages, consisting of randomly selected scramblings of 'ones' and 'zeroes,' were recorded. The 'zero' was read as 'oh.' Half of the messages were read without interruption; half of the messages, called "split messages," were read with a unit-pause after every four digits. The rate of reading was so adjusted that the average

rate of presentation was constant at 1.4 binary digits per sec. S's task was to reproduce the message upon termination. The termination was indicated by a tonal signal. The answer sheet was subdivided into groups of four units. The length and 'split' of the message was announced before each presentation. Ten to 15 different messages of each message-length were recorded to reduce learning of specific messages.

Tachistoscopic materials. The tachistoscopic messages were 8 circles arranged in a horizontal line, exposed for 0.04 sec., subtending a visual angle of 11 degrees at a seating distance of 16 ft. Each circle was either filled or unfilled. A pool of over 100 displays was scrambled on successive days. In half of the slides, a dividing line split the display into two groups of four units each. Twenty-four 'split' and 24 unbroken displays were presented each testing day.

Coding instruction. Instruction about coding was accomplished by explaining the principle of binary numerals. the following code was employed: 0000 to 0; 0001 to 1; ..., and 1111 to 15."

The experiment measured two statistics of performance: error rate and memory span. For the tachistoscopic materials, there was no span increase after learning coding.

We need not attribute this failure to the stimulus modality. The tachistoscopic task was at least a double-encoding task, representing the stimulus successively as "filled circles" or "unfilled circles," as corresponding "ones" and "ohs," and then by decimal numbers such as 3 and 12. The difficulty for the subjects may well have arisen from the complexity of controlling such a task.

For both kinds of materials, there was a trend to decreased error rates over the duration of the experiment, with the effect of practise and the effect of learning coding confounded. The question of whether learning coding affects the final, stable error rate (if there is such) is open because of the failure of the control half of the experiment.

> "The initial experimental design called for two groups: an experimental group and a control group. ... the control group ...defeated their purpose by learning the code clandestinely."

The natural point of comparison of this work to Smith's work is in the memory span statistics. Pollack and Johnson's results contrast with Smith's in that the fractional increases in memory span are relatively small; the average span before coding was 11 , the average after coding was 17.3, so that the span ratio increased by about 56%.* (The corresponding increase indicated by Smith is 113%.)

Even this modest increase is somewhat of an exaggeration because of the optimistic scoring method used. Credit was given for a presumed rate of partial correctness in incorrectly reported decimal digits.

> SCORING. An error in decimal notation was treated as equivalent to a response of four binary digits. Such an error will be termed a "sectional error." Reconversion to binary errors yields 2.13 binary errors per error in decimal notation on the assumption that all errors in decimal were equally-likely. This method was conservative; sample comparisons between errors in decim the original binary messages yielded 1.9-2.0 binary errors per erro. mal notation."

Thus for example, if a subject, when reporting 8 decimal numbers representing a sequence of 32 binary numbers, got all 8 incorrect, he would be credited with a span of 14.96 digits for that trial, thus exceeding the group's best uncoded performance of about 13 digits. The experimenters have failed to discount properly for guessing. The scoring method made coding improvement for the longer sequences absolutely inevitable because scores as low as the performance without coding were unachievable in principle. The unavoidable minimum turns out to be the major component of the subjects' "substantial gains."

---

* From Figure 3 in [PJ65].

Furthermore, the advantages of coding in span were confounded with practice effects.

> "Because of the large initial changes in performance, only the messages presented during the last seven days prior to coding are considered."

The stability of the reference pre-coding performance was thus not established. The coding advantage includes unknown amounts of continuation of improvement with practise on this class of tasks.

The most serious criticism of the experiment is that the task was changed at the point of learning coding. Subjects now responded in decimal notation, so that the portion of the task involved in serially producing the binary sequence was no longer required. Since the decoding step is eliminated, we would expect the task to be easier than Smith's to some unknown degree. This task is not Smith's task, and Pollack and Johnson's experiment in no way constitutes a replication of Smith's experiment or confirmation of his results.

In view of the optimistic treatment of the task definition and scoring methods, the weakness of the effect on memory span and the several uncontrolled factors which might account for this effect, it seems reasonable to consider this experiment a non-confirmation of S. i.n's result to the extent that quantitative comparison is possible.

Possibly the contrast is understandable in terms of individual differences between subjects. Pollack and Johnson note that

> "The better S's were often able to receive the split 40-unit messages without error."

which is performance far above the cited averages for the 6 subjects.

OVERVIEW

The work on deliberate coding is surprisingly sketchy in view of the place that it holds in current views of STM. Coding phenomena have not been systematically or thoroughly explored, nor have the widely accepted phenomena been well verified experimentally. The best established phenomenon is the fact (but not the amount) of increased digit span when a code is being employed.

Some of the relatively unexplained phenomena include:

1. The capacity to use codes in non-overlapping ways. (Digits included in one chunk are somehow prevented from being included in others.)
2. The capacity to restrict the encoding to be periodic. (Persons who can group by 2's or by 3's can select to do just one of these, even though thay are trained for both.)
3. The capacity to use different codes, and to change codes. (The digit "one" is a subunit in several different codes.)
4. The tendency to report a stimulus only once.
5. The capacity to perform correct encoding and decoding after being instructed in the task. (The instructions are assimilated into chunking methods.)
6. Improvement of coding performance with experience.

Coding experiments provide a particularly good opportunity to observe overlap, periodicity and code selection effects, in contrast to the interference experiment above, where stimulus elements were not reused, words in the stimulus lists did not form

periodic groups, and all the words were meaningful only in a single code (English).

The phenomena of instructability of subjects and single reporting are taken for granted, but they are given no theoretical status. The subject's receipt of instructions is ordinarily treated as part of the experiment, but somehow not a part that must be accounted for. The resulting accounts are necessarily partial at best. Because the subjects' performance depends intimately on what he is instructed to do, it is limited by his capacities for interpretation and conversion to action. If the instructions and their interpretation are not given theoretical status, we have no way of representing what features of his performance reflect such limitations.

GOALS FOR INVESTIGATION OF CODING

We desire a model of human coding processes which is sufficient to account for occurrence of certain kinds of performance observed in the experiments above. Since our model is to be organized and verified by means of a computer program, we want the program to exhibit at least the following:

1. It receives and repeats in order sequences of symbols.
2. Given a code, it encodes and decodes sequences using the code.
3. It has a stable symbol span for any particular experimental arrangement.
4. The span is responsive to the presence and structure of the code in a way which is compatible with the data on Smith's performance.
5. The instructions to the subject are explicitly represented.

Having achieved a model which is sufficient in this sense, we can go on to learn how the realism of the model's performance varies with variations in its content, namely the forms of chunks and chunk manipulation processes.

BASIC CODING PERFORMANCE OF THE SLATE SYSTEM

The principal difference between the free recall instructions given for the previous chapter's task and the instructions for coding tasks such as Smith's is that order of reporting is specified in the coding task. Because of this order requirement, the operator RECALL WORDS which was used to perform the interference task is not adequate for coding tasks.

The Slate system has another operator, called REPEAT SEQUENCE which acts similarly but gives an ordered report after termination of the sequence. Since the sequence may be incomplete, it reports by first finding the beginning of the sequence (if it is present) in STM and reporting successive symbols. If this succession does not terminate at the end of the sequence, it then locates the end and reports (in forward order) the successive symbols of the fragment of the sequence found at the end. Any other remaining symbols in STM must be in isolated middle fragments of the sequence. They are not reported. It inserts "-UH-" into the sequence presented to the user under 4 conditions:

1. It cannot find the sequence beginning.
2. It cannot find the next symbol at a non-final point in the sequence.
3. It cannot find the symbol for a token which is part of the sequence of tokens having symbols.
4. It cannot find the sequence end.

OVERLAPPING ENCODINGS:

Binary digit streams can be encoded into any particular higher base code in more than one way, with differing results. For example, 101001110 can be encoded conventionally as 516 in octal code, or in another way as 5241376, using the same code but recognizing every adjacent triple of binary digits. A covert rule of the conventional kind of encoding requires that the spans of the encoded symbols be disjoint, so that 516 is correct and 5241376 is not. Somehow the subject avoids using the code in the inappropriate ways. His restriction of the encoding act to just those places where it is appropriate is one of the encoding phenomena to be accounted for. The possibility of overlap raises several questions:

1. Can subparts be shared among chunks?
2. If so, how is inappropriate sharing avoided?
3. Can chunking apply to the outcomes of chunking?

An inadequate explanation would be to say that chunk domains may not overlap, that they must simply partition the field of available information.

This is an improper assumption for some problems, such as the problem of describing a chess position, where one piece may be involved in more than one significant structure, and therefore should be included in more than one chunk. It also makes it difficult to model hierarchies of chunks in STM, where the entities in some chunks are not input entities (e.g. the phrase entities of the noisy speech task.)

When a particular digit is made part of an encoded group in the Slate, an assertion is made to the effect that:

<THE DIGIT> is part of <THE GROUP>.

Any later assertion that:

<THE DIGIT> is part of <SOME OTHER GROUP>

is rejected as inconsistent. The selective use of assertions which can combine in consistent or inconsistent ways provides a basis for establishing hierarchic relationships where that is appropriate and yet rejecting overlapping codes and other inappropriate combinations.

The criteria for accepting a matched chunk as relevant in the Slate system do not require that a complete match of chunk to Slate content be achieved. As soon as an adequate partial match is achieved, the relevance of a chunk can be established. As a direct consequence, the system is able to anticipate the remainder of a partially presented chunk; for our experiments anticipation may occur after two letters of a word or two digits of a code chunk are present. Anticipation does not occur on the quaternary (base 4) coding task, since the quaternary code is a two-digit code. The general effects of anticipation are discussed elsewhere below, and its effect on octal coding performance is shown in the figures below. However, it should be noted that we can explain Smith's performance in detail only on the assumption that anticipation does not occur in the model.

SHARED WORKSPACE EFFECTS ON CODING

The efficiency of encoding of short term memory contents depends on the structure of the code and the discipline used to insert elements into the memory. We would like to know how code structure relates to memory capacity.

Consider the simplest codes in which any sequence of N input elements can be encoded into one grouped element. Let memory capacity be fixed at C elements. Let the insertion process be such that the coded group is recognized and then stored in STM. This simple scheme would fill memory with C encoded groups of N input elements each, giving C*N total capacity. The relationship between the length of the longest sequence that the subject could hold and the length of each code element would therefore be linear.

We contrast this with the Slate model in which uncoded and coded digits both appear in STM. The encoding process acts as follows:

1. Sequence elements are received serially.
2. Each sequence element is placed in an empty memory space.
3. Whenever N sequence elements are present in memory, an encoding operation removes them and inserts one corresponding element representing the group.

Under these assumptions, encoding may proceed without loss until there is no empty memory space for the next sequence element. This condition defines the maximum encoded capacity of the memory for that code.

A simple geometric analogy makes it easy to visualize the limiting process. Consider the problem of filling a box C units deep with blocks through a hole in the top. Each

block must be placed directly under the hole. Any vertical stack of N blocks may be turned on its side to be one block high. For C=6 and N=3, Figure 4.2 shows the box at the point of fullness. Four groups of 3 blocks and two additional blocks are in the box.

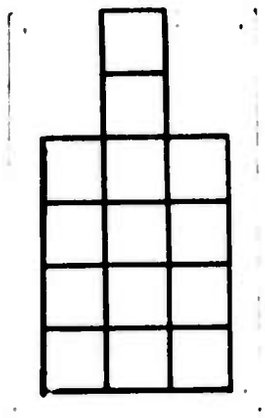


Figure 4.2 - Box-of-Blocks Analogy to STM Capacity

No more blocks can be added, and there is no group of 3 which can be laid down. In general, N*(C+1-N) blocks are in groups and C-1 blocks are ungrouped at the point of fullness.

Table 4.1 shows the capacity at fullness for various sizes of memory and code lengths.

MAXIMUM MEMORY CAPACITY FOR A SIMPLE CODED SEQUENCE

	LENGTH OF A CODE GROUP						
	2	3	4	5	6	7	8
4	7	8*	7				
5	9	11*	11*	9			
6	11	14	15*	14	11		
7	13	17	19*	19*	17	13	
8	15	20	23	24*	23	20	15
9	17	23	27	29*	29*	27	23
10	19	26	31	34	35*	34	31
11	21	29	35	39	41*	41*	39
12	23	32	39	44	47	48*	47
13	25	35	43	49	53	55*	55*
14	27	38	47	54	59	62	63*

Table 4.1- Memory Capacity with Simple Code

The best codes for each capacity are marked. Note that the optimal group length is always about one half of C.

It is interesting that large changes in memory capacity produce relatively large changes in maximum encoded sequence length but only small changes in optimal code length. Small reductions in code length from the optimum carry only small disadvantages in achievable sequence length. We can also see that the simple linear formula C*N is generally a poor estimator of maximum sequence length. This formula would apply if every element of memory capacity could hold an encoded chunk.

We might consider a more complex code, in which two levels of encoding occur rather than one, so that grouping N groups of N elements into one element is also allowed. Table 4.2 shows the capacities achievable for several codes and memory sizes. Note that again the optimal code length is a slowly increasing function of memory size.

MAXIMUM MEMORY CAPACITY FOR A SIMPLE CODED SEQUENCE

	LENGTH OF A CODE GROUP							
	2	3	4	5	6	7	8	
3	7	5						
4	11	8	7					
5	15	17	11	9				
6	19	26	15	14	11			
7	23	35	31	19	17	13		
8	27	44	47	24	23	20	15	
9	31	53	63	49	29	27	23	17
10	35	62	79	74	35	34	31	26
11	39	71	95	99	71	41	39	35
12	43	80	111	124	107	48	47	44
13	47	89	127	149	143	97	55	53

Table 4.2- Memory Capacities using a Two Level Code.

The Slate system could use either of these codes by having the appropriate code chunks. The ones which were given cause it to use the single level code, so that its peak performances correspond to Table 4.1.

SLATE SYSTEM PERFORMANCE

The Slate system performance on binary encoding/decoding tasks turns out to be very simple and strongly patterned despite the complexity of the mechanism which performs the chunking. Little of the diversity of the system is seen on this task.

We will examine four variations of the task:

1. Performance without coding knowledge
2. Quaternary coding
3. Octal coding without anticipation
4. Octal coding without restrictions.

In all variations, fo each elementary experimental sequence, the system input consists of a command to repeat a sequence, followed by a random sequence of ones and zeroes, followed by QUiT, a sequence terminating symbol. On QUIT, the system recovers what it can of the sequence from the Slate and presents it. As described elsewhere, there are two ways in which digits can be lost between input and output. The chunks representing a digit can be deleted from the Slate in order to make room for some incoming chunk; an isolated middle subsequence of digits may not be locatable from either end of the sequence. There are no mechanisms for guessing. For the case in which the system has no code knowledge, Figure 4.3 shows representative relationships of the average number of digits in the output sequence to the number in the input sequence.

The form of the relationship in Figure 4.3 arises as follows: For any number of digits up to the chunk capacity, no digits are lost and sequence repetition is perfect. Thereafter performance is limited to the chunk capacity. Losses increase with sequence length because of the increased incedence of isolated middle fragments.

For the case of quaternary coding, the system does not anticipate because the length of a code segment (2 digits) provides the minimum number of symbols required for accepting a chunk. Therefore, up to the point at which the Slate is full, the sequence is encoded perfectly and repetition of a number of digits up to twice the chunk capacity of the Slate can be achieved. Beyond this, the two mechanisms described previously cause declining numbers of output digits with increasing sequence length, as shown in Figure 4.4.

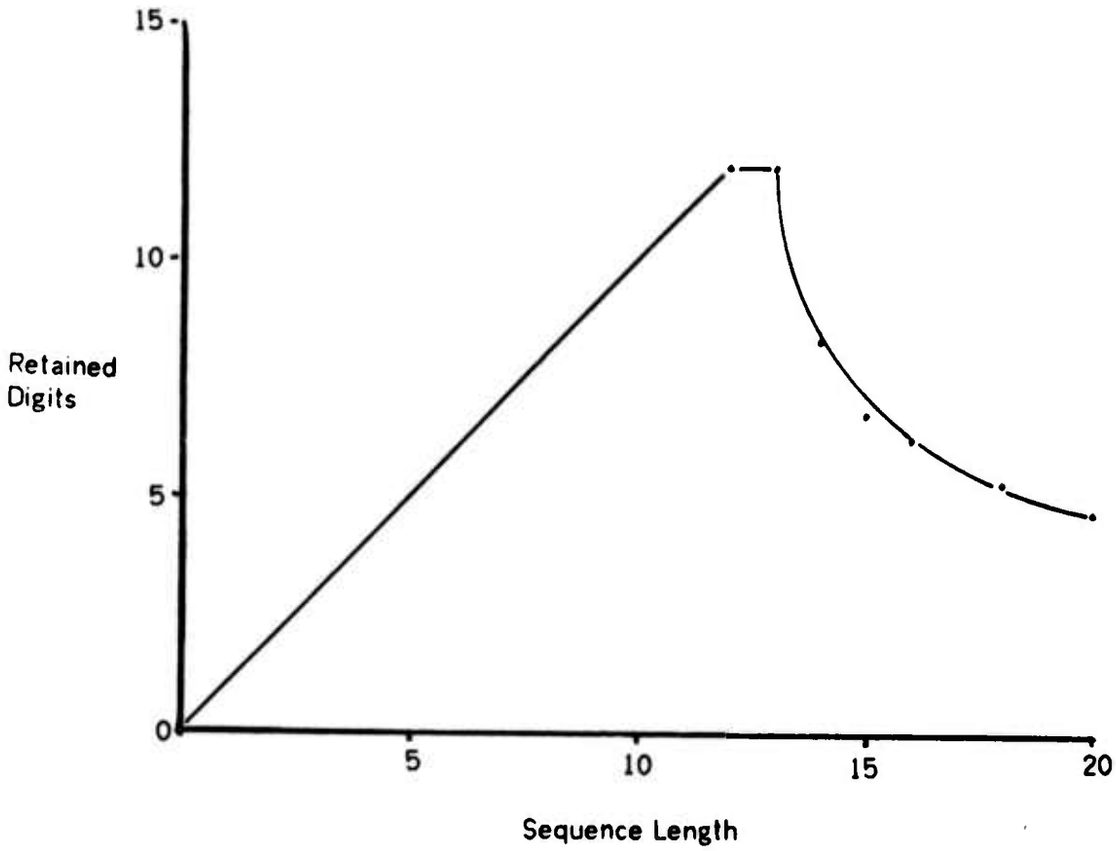


Figure 4.3 - Uncoded Digit Retention Performance

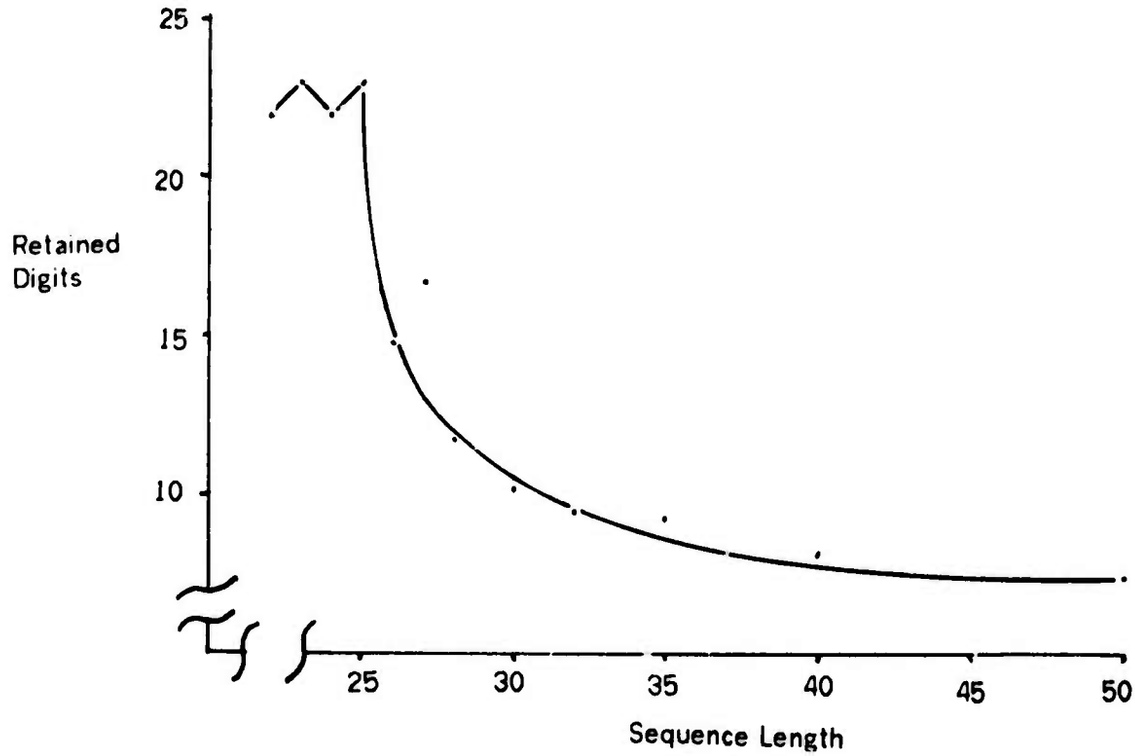


Figure 4.4 - Effect of Quaternary Coding on Digit Retention

For the third case, chunks for the binary triples of octal code are in bulk memory, and anticipation has been prevented. The results, in Figure 4.5, are similar to those for quaternary coding, with perfect retention of a number of digits up to three times the chunk capacity of the Slate. The analogous result will hold for larger spans as well.

Performance in octal coding is more complicated in the unrestricted case, where anticipation occurs as described above. Some binary digits do not get encoded at all, so the number of digits which can be retained is shorter. Figure 4.6 shows the effect.

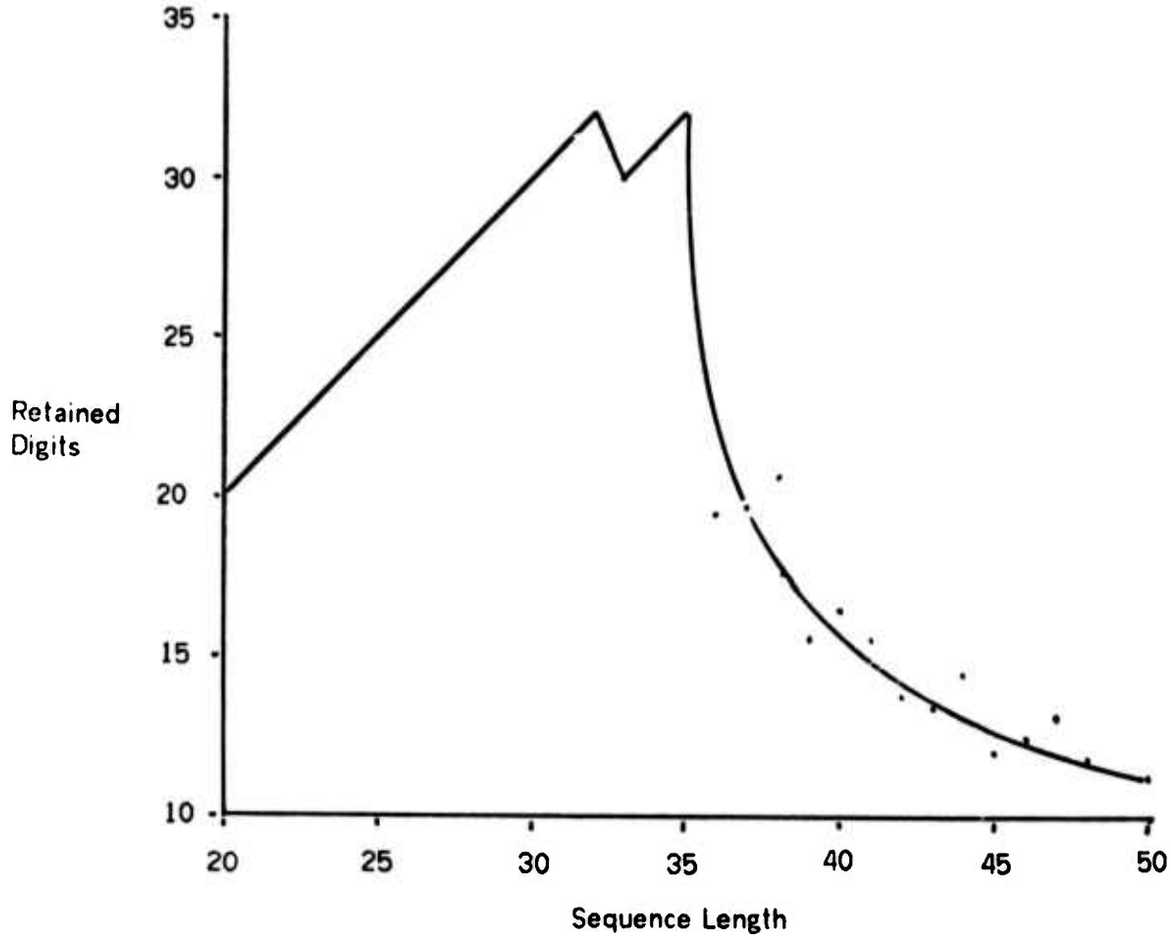


Figure 4.5 - Effect of Octal Coding on Digit Retention

EVALUATION

We evaluate the coding performance of the Slate system in two ways, for its effectiveness in coding and for its similarity to human performance.

The system is successful in all of the basic operations of the coding task. It receives digits, encodes using the appropriate chunks, and repeats the sequences given. It exhibits memory capacity limits and stable memory spans. It performs coding without overlap because the code chunk structure prohibits overlap. In other words, the system provides a sufficient explanation for performance of acts of sequence repetition on coding tasks.

We have not represented the aspects of the task involving initial acquisition of the task from given instructions and improvement of performance with practise. The ability to hold several codes while using one is also not represented, although there are straightforward ways to do so in the Slate framework.

We can compare Smith's performance with that of the Slate system using the shared workspace concept. Figure 4.7 shows Smith's curve along with the full workspace curve for 12 chunks. The fit is extremely good. We can appreciate how good it is by comparing the data with the predictions of other related hypotheses. In Figure 4.8 the straight line represents the simple linear model introduced above, assuming that there may be 12 fully encoded chunks in STM. The other curves are the shared workspace curves for 10 and 14 chunks. Each of these three alternatives departs seriously from the data. While the shapes of the latter two are appropriate for these other Slate sizes, the fit to the data is tight only at 12 chun. The curve of Figure 4.7 fully

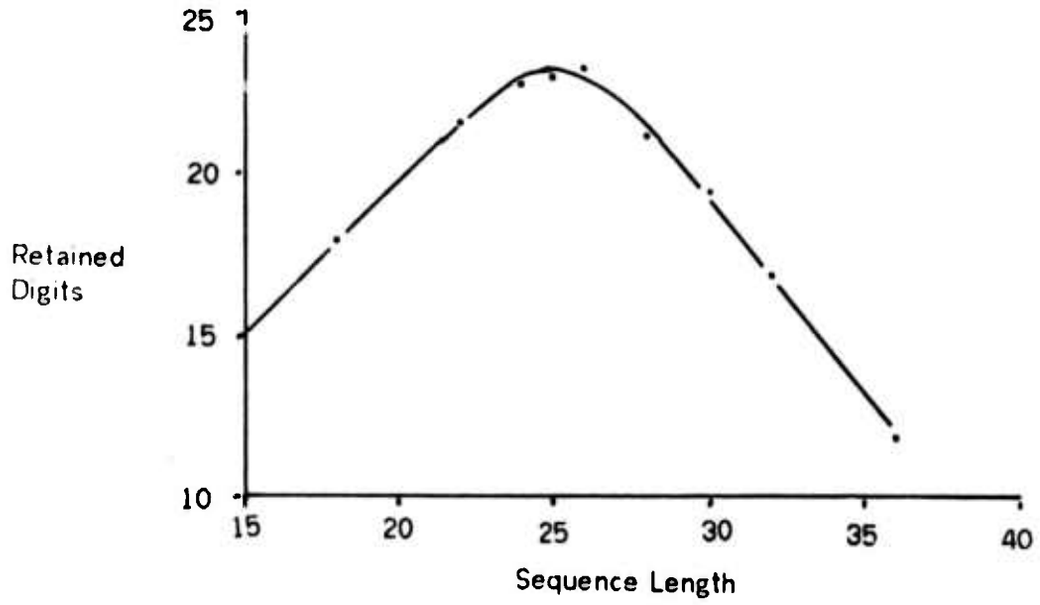


Figure 4.6 - Effect of Octal Coding with Anticipation

accounts for the description of Smith's experiment. It is important to note that it was necessary to use the shared workspace assumption and the assumption of no anticipation to achieve the fit. These are the assumptions used to achieve the fit to Bower's interference data as well.

The data of Pollack and Johnson resemble the Slate system as it performs octal coding with anticipation. Although our improvement in span of 55% corresponds to their improvement of 56%, for the reasons discussed above this seems to be mostly coincidence.

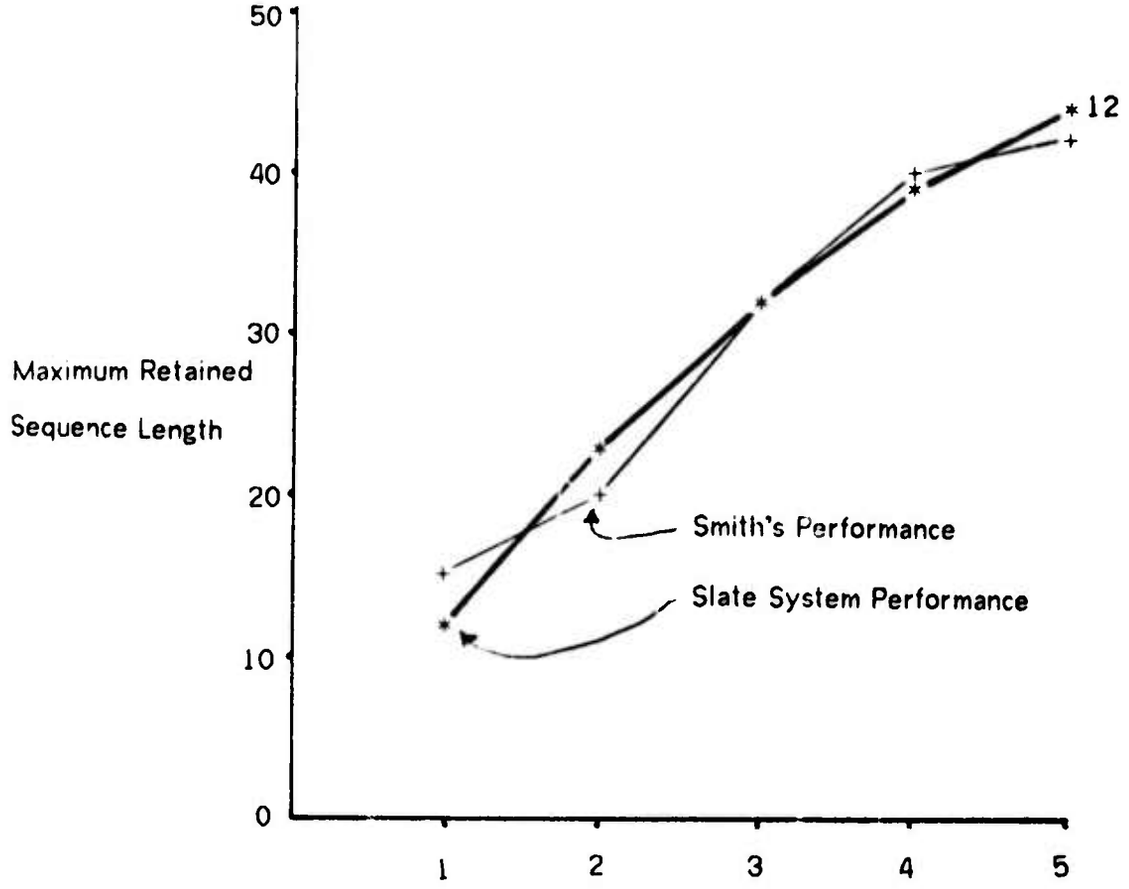


Figure 4.7 - Fit of Smith's Data by a Shared Workspace Model

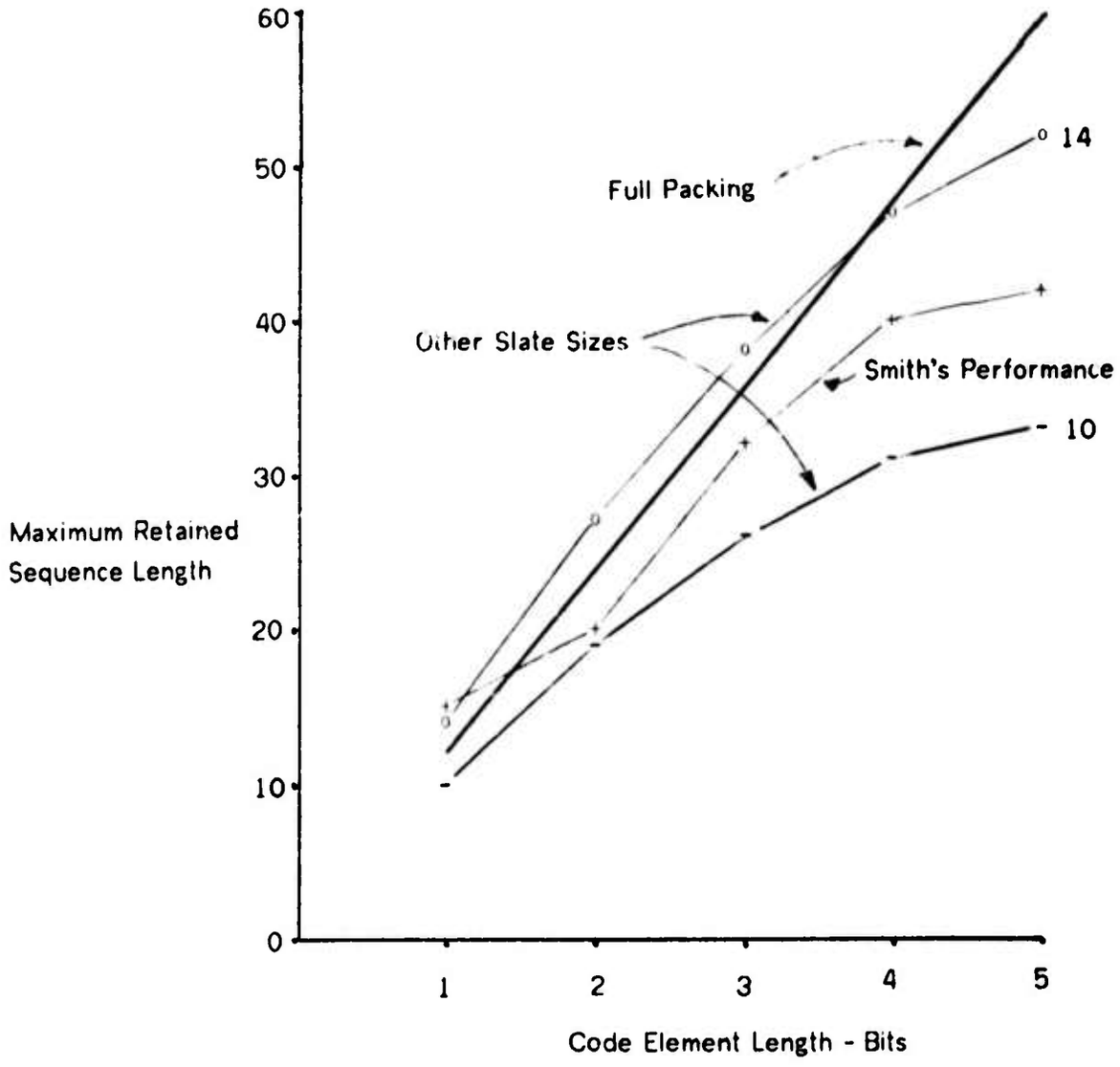


Figure 4.8 - Failure of Alternative Models to Account for Smith's Data

There may be a deeper basis for the qualitative correspondence which is present. The performance of relatively untrained or unmotivated subjects probably arises in large part from their normal linguistic skills, in particular from abilities to decode sequences with internal redundancy. It is the presence of redundancy that gives a significant saving of effort to the person who can accurately anticipate what is coming. The anticipation features of the Slate system were developed to deal effectively with letter strings representing unseparated words, where the same kind of redundancy prevails. A salient feature of the coding task is that such redundancy is absent. An inexperienced coding subject may be disturbing his own capacity to encode by anticipating rather that waiting for complete code groups to be presented.

This kind of analysis supports the interpretation of Pollack and Johnson's results as representing conditions with major individual differences, unstable performance and in particular unstable control of the moment of encoding.

The commonly used assumption that anything in STM can be brought out appropriately is questionable. The Slate system could not do so because order information was being lost by interference.

The 12 chunks available to Smith for his highly practised performance may be more typical than the 7 plus or minus 2 often assumed. The difference may well be a general inability to use all of the memory resource because of weakness of the methods used to perform the task. It will be necessary to pay close attention to subjects' methods of task performance in order to analyze their actions well enough to find out.

SUMMARY

The simple view that STM is a repository for things that have been grouped does not explain the observed relationship between capacity to repeat digit sequences and the codes used by Smith. In contrast, the Slate system models this relationship well. The assumption that STM is a workspace shared by coded and uncoded digits is instrumental in explaining the relationship.

CHUNKING AND AUDIO PERCEPTION

CHAPTER 5.1

In 1963 Miller and Isard published the results of an experiment in speech perception, relating the ability to hear (and repeat) word sequences presented with audio noise to the structure of those word sequences. [MI63] They used three different methods for preparing sequences, each resulting in a different kind of underlying structure. Samples of the three are:

Complete sentence:	The sticky humid weather frayed tempers.
Syntactic sentence:	The sticky young rhythm ate wonders.
Disordered sequence:	Tempers young rhythm ate secret the.

Their basic result was that sequences which were complete sentences were heard significantly better than syntactic sentences, which in turn were heard significantly better than disordered sequences.

Miller and Isard describe their experiment as follows:

> "...the output of a noise generator was mixed with the speech signal before it was amplified and led to the S's earphones. The spectrum of the noise was tailored to match the long-term spectrum of speech; this spectrum provides a very effective masking signal. Five different noise intensities were used, ranging from +15 to -5 db relative to the level of the speech. ... Two groups were run, totalling six S's altogether."

The principal results are shown in Figure 5.1, from the paper.

> "Sentence intelligibility scores are shown graphically in Fig. 1, where the per cent of the sentences repeated exactly is plotted as a function of the speech-to-noise ratio for each type of sentence. The results from ... unmasked speech are also plotted in Fig. 1."

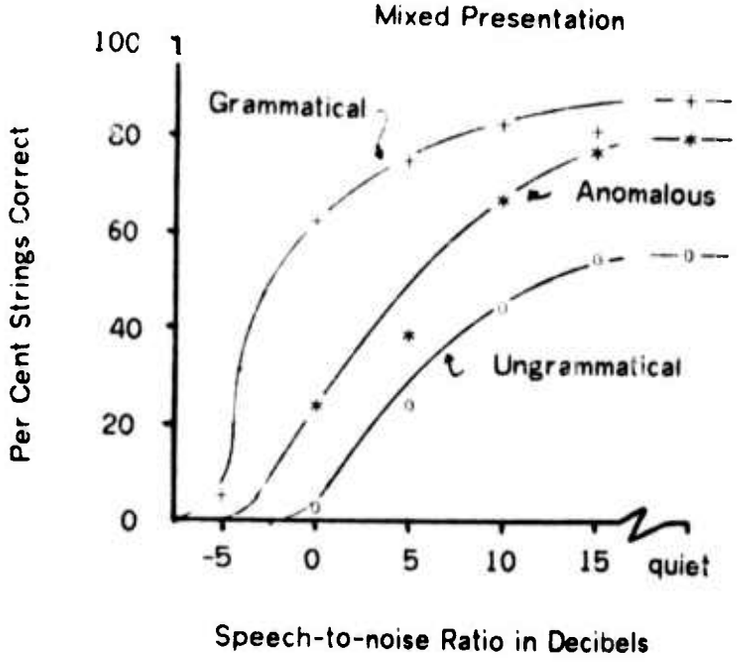


Figure 5.1 - Per cent strings heard correctly as a function of speech-to-noise ratio when the three types of test strings are presented in mixed order.

Miller and Isard conclude:

"The general conclusion that these experiments appear to support is that linguistic rules of a non-phonological sort do indeed have measurable effects on our ability to hear and repeat sentences. Moreover, both syntactic and semantic rules are effective, since grammatically acceptable but semantically anomalous sentences are intermediate in difficulty, falling below the normal sentences but above the ungrammatical strings in terms of the measures of perceptual accuracy that we have employed. A complete psycholinguistic description of speech perception, therefore, must take into account the syntactic and semantic rules of the language - and perhaps a number of non-linguistic pragmatic rules as well."*

GOALS FOR THIS EXPERIMENT

This experiment is much more complex than the previous two, with interactions between noise, grammar, words, meaning and memory capacity dominating the results. Each of these is known to have its own non-trivial complexities. We are primarily interested in how such interactions could be modeled at all. What sort of a model will allow words, grammar, noise, memory capacity and semantic effects to be represented together and to interact meaningfully? Other explicit models of STM (discussed in Chapter 6) do not represent the interactions needed to approach Miller and Isard's experiment, primarily because the representations for the content of STM exclude or make inaccessible some necessary information.

Our goals are therefore qualitative ones, examining feasibility rather than accuracy. Because of the difficulty of comparing the Slate's representation of noise with audio noise, we cannot form precise criteria of quantitative correspondence of error rates. However it is possible to rank relative word loss rates for the three kinds of strings.

The principal results which we derive in detail below are:

1. The Slate system is able to perform word segmentation and grammatical analysis in the presence of noise.

2. The system performance responds correctly to the presence and absence of noise and the presence and absence of grammatical

---

* Miller and Isard also dealt with subjects' set, which is outside the scope of our experiment.

structure in the given information.

3. Grammatical analysis can be treated (at least in part) as a chunking activity.

4. The system fails to model the observed relationship between meaningfulness of a sentence and noise.

In order to apply the Slate system to Miller and Isard's task usefully, we need a way of representing the stimuli and criteria for judging the correspondence between the Slate system action and their results. The criteria are chosen to establish that the performances are in fact comparable and to show how the particular methods used by the Slate system lead to results that agree with or differ from Miller and Isard's results.

The simplest criteria concern the capacity of the Slate system to perform the task at all.

1. Ability to identify words in an input stream
2. Ability to identify phrases in an input stream
3. Ability to respond differentially to grammatical and ungrammatical stimuli.
4. Ability to respond differentially to semantically coherent and incoherent input.
5. Ability to function in the presence of noise.

In addition to these, the following properties would indicate correspondence of performance:

1. Performance under low noise is nearly perfect.
2. Syntactically structured strings are more likely to be repeated correctly than non-structured strings.
3. Semantically coherent strings are more likely to be repeated correctly than strings with only syntactic structure.
4. Increased noise leads to decreased likelihood of correct repetition under all conditions.
5. Losses are expressible in terms of words right and wrong.

CHUNKING AND SYNTAX

The notions of chunking and parsing have generally received separate treatment in the psychological literature. There are several historical reasons for the separation. Chunking has a short technical history in which it has appeared most often in simple phenomena of recurring events. Parsing has a long technical history and now appears as a family of complex theories of forms, including transformational grammars and various semantic systems. Linguists commonly exclude claims that there is any correspondence between steps in their grammatical processes and psychological events. *

Effective hearing of language certainly involves psychological events, and these events make the linguistic distinctions possible simply because they lead to verbal behavior. Understanding a spoken sentence involves relating it to learned constructs, as does the chunking that invades so many psychological experiments. It is of psychological interest, therefore, to discover what part of the processes of linguistic understanding can be accounted for by the same processes used to account for chunking. The account below is a modest step in that direction.

---

* Chomsky's remark below is representative:

> "To avoid what has been a continuing misunderstanding, it is perhaps worth while to reiterate that a generative grammar is not a model for a speaker or a hearer. It attempts to characterize in the most neutral possible terms the knowledge of the language that provides the basis for actual usage by a speaker-hearer." [C65]

PARSING WITHOUT NOISE

The task given to the Slate system to investigate parsing without noise is to identify words and phrases in sequences of unseparated letters, e.g.

BIGTREEANDLITTLEDOG

Part of the problem for the subject is to segment the given input stream into units. Elimination of spaces makes the segmentation problem non-trivial for the Slate system.* Two kinds of chunks are provided, word chunks and chunks representing syntactic rules. Examples of word chunks, and all of the syntactic chunks, are given in appendix CHUNK CATALOG. The composite stimulus problem may appear at several levels of hierarchy, and each instance may require evidence from several levels of hierarchy for its resolution. Although we deal with only two levels (word and phrase) in this experiment, the system is not committed to particular levels or numbers of levels. The techniques may be applied at other levels as well. For example, voice formant recognition and phoneme recognition might be performed. It should be noted that the grammar used here is not a linguistically serious grammar; similarly the word chunks might better be replaced by "morpheme chunks," the letters by "phoneme chunks," and so forth as one's theory requires.* Letters are presented serially exactly as binary digits are presented in the coding experiments already described. The result is taken to be the sequence as repeated by the system.

---

* The segmentation problem is a special case of the need to represent theoretically how it happens that adjacent parts are not regarded as shared in the result. This input is a non-overlapping composite stimulus.

---

* The grammar is a minor extension of one originally used to represent the noun-phrase grammar of the book **Go, Dog, Go**, a first reader for children. [E61]

Figure 5.2 is a schematic representation of the derivation of the Slate content at the end of a sequence.

The interpretation of the notation is as follows: Each of the boxes in the bottom row

Given:

(1) BIGTREEANDDOG

Derivation:

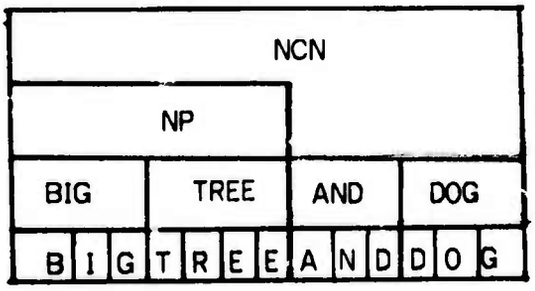


Figure 5.2 - Slate State Derivation Diagram

represents a chunk formed by receiving input information. Each box above the bottom row contains the (bulk memory) name of a chunk which was matched to the Slate content and included the chunks named immediately below it. Since there is one box in the top row, there is one chunk in the Slate which includes the entire sequence. Some of the subordinate chunks also retain their identity in the Slate, and others have been lost.

(This particular sequence can be completely chunked as shown maintaining only 3 independent chunks in the Slate.)

We see that the action of the Slate system is entirely successful in interpreting the given sequence in terms of the available chunks.

Figure 5.3 shows several other successfully interpreted sequences.
We see from items (5) and (7) that repeated use of a chunk is successful. In item (5) we see that recursive use of a chunk is successful, since this example includes an outer NOUN produced by the NP chunk and which includes an inner NOUN produced by the NP chunk as a subpart. Item (4) includes a use of the ARTN chunk, which has a letter and a word as subparts.

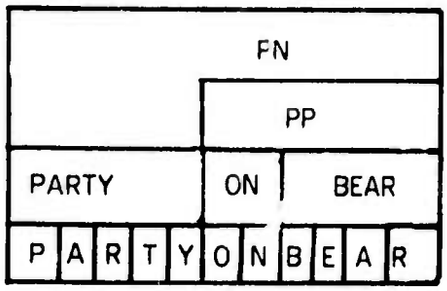
Figure 5.4 shows several sequences for which the available supply of independent chunk memory spaces was fully depleted during processing. For comparability the sequences are selected from the ones above, but the number of available chunk spaces has been reduced from 5 to 4.

Note that although the input is received letter by letter, losses tend to be in terms of whole words and completed phrases. This is a general property which is a direct consequence of the way the system chunks. Units lost are the independent aggregates which are resident in the Slate at the moment of interference.

Given:

(2) PARTYONBEAR

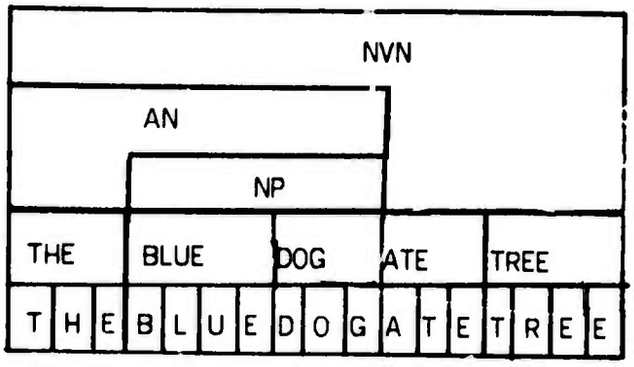
Derivation:



Given:

(3) THEBLUEDOGATETREE

Derivation:



Given:

(4) ONATREE

Derivation:

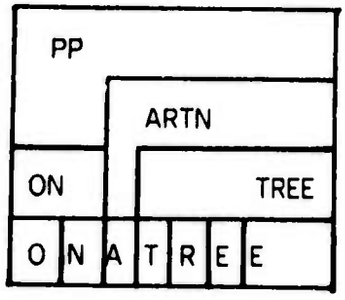


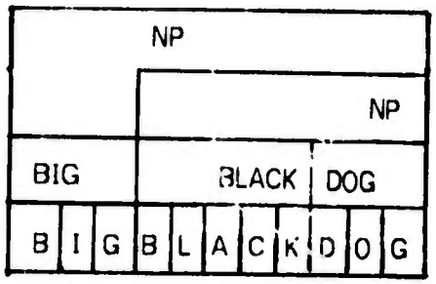
Figure 5.3 - Other Successful Derivations

Part 1

Given:

(5) BIGBLACKDOG

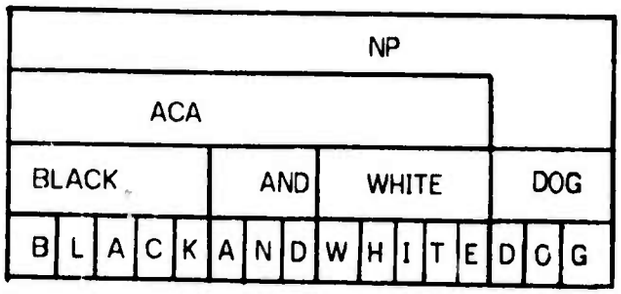
Derivation:



Given:

(6) BLACKANDWHITEDOG

Derivation:



Given:

(7) TREEANDTREE

Derivation:

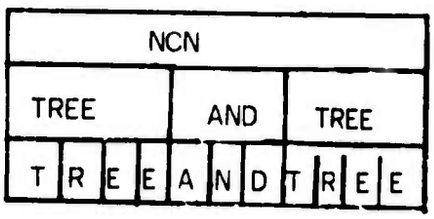


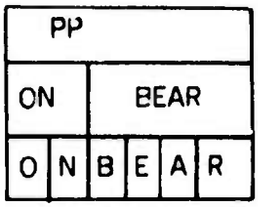
Figure 5.3 - Other Successful Derivations

Part 2

Given:

PARTYONBEAR

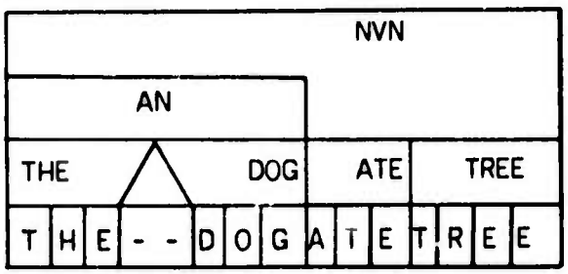
Final state:



Given:

THEBLUEDOGATETREE

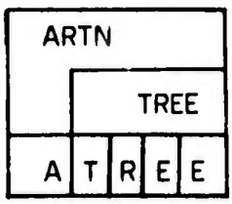
Final state:



Given:

UNATREE

Final state:



Given:

BIGBLACKDOG

Final state:

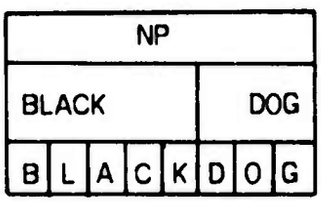


Figure 5.4 - Performance Degradation by Exhaustion of Memory

PARSING WITH NOISE

The Slate system has a provision under the Repeat Sequence operator which allows a special symbol for "noise" to be substituted for an input symbol of the sequence. When such a symbol is received, its occurrence in the sequence is recorded in the Slate, but no symbol is recorded corresponding to that event.

Figure 5.5 shows some sequences containing noise(*), together with the corresponding Slate content schematic representation.

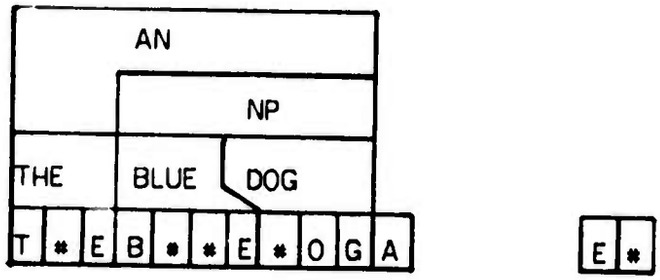
Each of these sequences is chunked correctly with 5 available chunk spaces. All missing letters are filled in, and the output sequence is unbroken. The figure shows the effects of having only 4 spaces. The first sequence which is "the blue dog ate tree," retains "the blue dog" in a single correct chunk, and the final letter. The second, which is "big tree and dog," has a substitution of "the" for "tree." The final state also contains an unfulfilled anticipation of another noun ($ in the figure) to be first in the noun-conjunction-noun group. The chunk for np "big tree" was lost partway through the sequence input. The third sequence, which is "party on bear," does not chunk on letter N because a single letter is inadequate evidence for a word; short words are thus generally more vulnerable under uniform noise.

Source Sequence: THEBLUEDOGATETREE

Noisy Version Given: T*EB**E*OGA*E*RE*

System Output Sequence: THEBLUEDOG -UH- E

Final Structure Derivation:

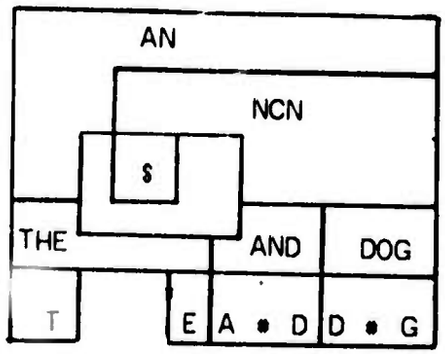


Source Sequence: BIGTREEANDDOG

Noisy Version Given: BIGT***A*DD*G

System Output Sequence: -UH- THEANDDOG

Final Structure Derivation:



Source Sequence: PARTYONBEAR

Noisy Version Given: P**TY*NBE**

System Output Sequence: -UH- NBEAR

Final Structure Derivation:

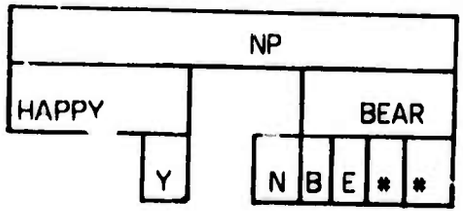


Figure 5.5 - Derivations on Noisy Input Sequences

RESPONDING TO SEMANTIC CONTINUITY*

One way of describing the differences between the normal sentences and the anomalous ones is to say that there are semantic discontinuities in the anomalous ones; certain words set up expectations which are fulfilled in the normal sentences and not in the anomalous ones.

We make this operational by including in some of the word chunks references to the vertex constants PHYSICAL, EVENT and ANIMATE. These constants are held in common by appropriate pairs of adjectives and nouns, so that they are available as retrieval indexes into bulk memory. The chunks are arranged so that tokens attached to these constants need not match, but occurrence of a match acts as evidence for the correct mapping of a chunk. Semantic agreement between an adjective and a noun is signified in the Slate by having a single token which has positively incident arcs for the relation "expects" from the adjective and "fulfills" from the noun. This occurs whenever they have semantic constants in common. Lack of agreement is signified by having two different tokens for these arcs. Figure 5.6 shows fragments of the chunks for the three words BIG, SUDDEN, TREE. Figure 5.7 shows a part of the graph which results from

---

* Up to this point the discussion has been in terms which required little knowledge of the programs or the chunk representation of the Slate system. It is necessary to use such knowledge here in order to make explicit the result of studying Miller and Isard's task. The representation and system description chapters are 12 through 14.

In representation-free terms, what is told below is as follows:

The Slate system achieves nearly complete qualitative replication of the experiment. It is unable to achieve sufficient differences in performance between normal and anomalous (syntactic) sentences. This is because the Slate system does not respond to certain kinds of knowledge about the chunks already found. A more sophisticated approach would overcome the problem by responding to some particular kinds of distinctions which are not represented in the present system. These changes would also allow more concise chunk definitions and more efficient processing.

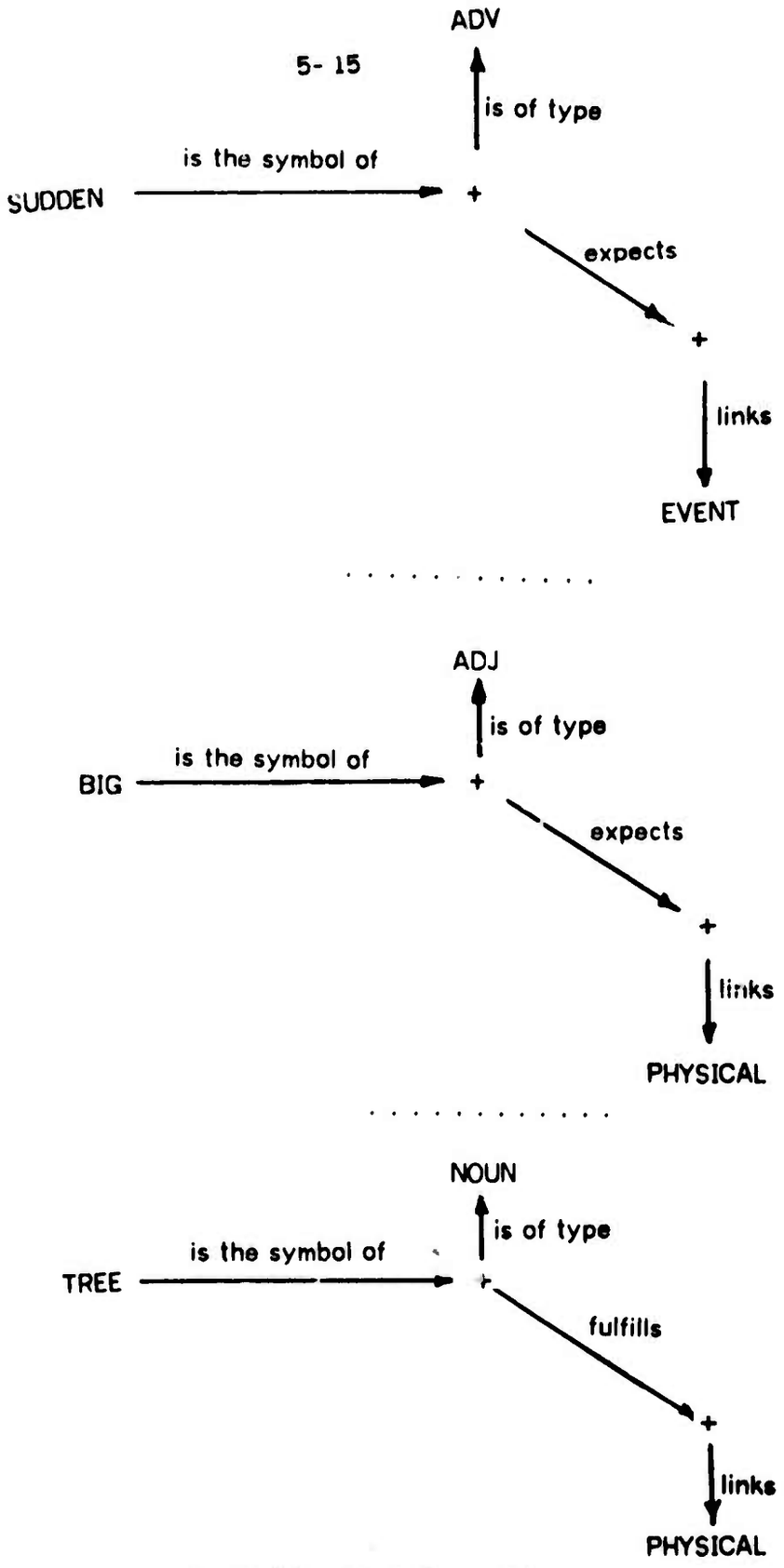


Figure 5.6 - Chunk Fragments

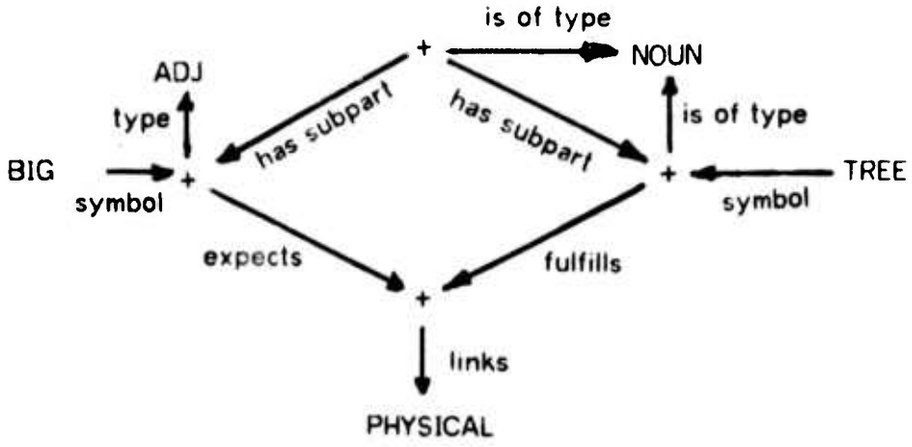


Figure 5.7 - Semantic Agreement Condition for BIGTREE

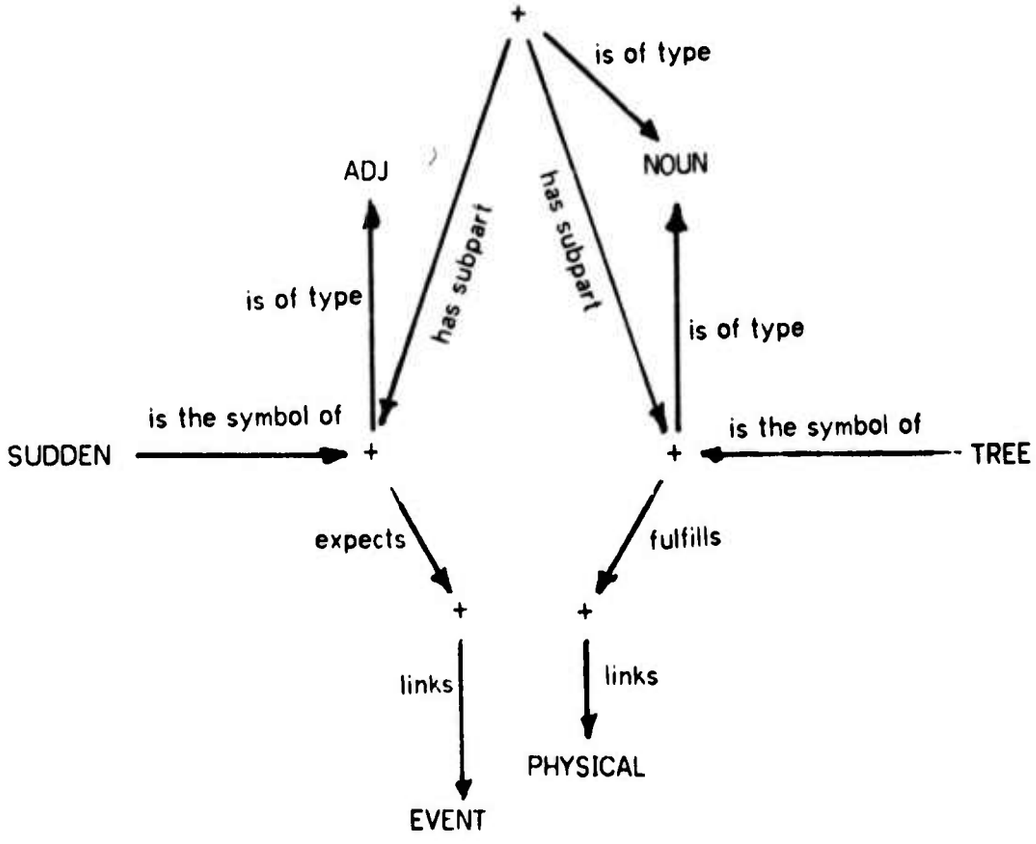


Figure 5.8 - Semantic Disagreement Condition for SUDDENTREE

processing the input sequence BIGTREE, and Figure 5.8 shows the comparable fragment from the sequence SUDDENTREE. The fact that the arc to the vertex PHYSICAL is a region of overlap between the two word chunks influences the system's interpretation of evidence. The intent was that it would lead to relatively early examination of the chunk for TREE in those cases where a PHYSICAL chunk was expected, for example, and that this could be made to exhibit a bias toward semantic coherence.

If all of the retrieved chunks are tried whenever an interpretation cycle (of operator REGARD) is performed, then all syntactically proper combinations will be found, regardless of the presence or absence of semantic fit. Since the loss mechanism of the Slate system is interference, an excess of uncovered chunks, there will be little difference between the loss rates in normal sentences and those of anomalous ones.* However, if the system is somehow "trying the wrong things" from a limited set, then much larger differences should appear. We use three devices to get this latter effect:

1. Rather than trying bulk memory chunks in the order of their discovery, their names are held until retrieval is complete, and they are ranked in order of a prospective degree-of-fit evaluation function. The evaluation score for a particular chunk is the number of different constants held in common by the chunk and the "hot vertices" of the Slate.*

---

* Chapter 13 discusses chunk covering as an element of Slate space management.

2. The chunks are tried in rank order, best fit first.

3. A cutoff level is established, so that only the first N chunks are tried. This simulates limited time to respond in the experiment.

If these methods are effective then we should be able to choose a cutoff level for which the proper syntactic combinations are not formed when the semantic discontinuities occur. This would leave more independent words in the Slate, leading to early interference losses relative to normal sentences.

What happens is that the method fails because an appropriate cutoff level cannot be established. There are several reasons for this, the principal one being a direct consequence of the way that candidate chunks are retrieved. We seek chunks which overlap the Slate content, especially in their use of constants. The chunks that overlap best with animate physical adjectives are those for other animate physical adjectives, not the nouns which they modify. As long as they all specify their expectations in the same way, this kind of similarity overlap will exceed that of the complimentary expected item, whatever it might be. No sensible cutoff threshold can be established, since the number of similar chunks which rank ahead of the correct chunks depends on the specific vocabulary and category involved.

---

* Chapter 13 discusses "hot vertices" as an element of query formation for LTM retrieval.

This is a serious representation problem that shows up as a source of inefficiency as well as an inability to meet the quantitative data of Miller and Isard. The fact that people's listening speed does not drop off strongly as their vocabularies increase indicates that they have other means for dealing with this problem.

There are some attractive methods which might be tried for overcoming the difficulty in the context of the Slate system. It is intimately involved with some major inefficiencies of the present system. (See the section Faster Methods in Chapter 15.)

SUMMARY OF PERFORMANCE ON MILLER AND ISARD'S TASK

We have observed the following features of performance common to the Slate system and Miller and Isard's subjects:

Chunking:

Word segmentation and recognition are performed.

Syntactic assembly is performed.

Appropriate semantic combinations are recognized.

Chunking takes place in the presence of noise.

Loss phenomena:

The units of loss are among the largest independent chunked units, (words and phrases), rather than subunits of these.

Small amounts of noise cause no loss.

Increasing noise results in increasing losses and increasing

misperception rates.*

Unstructured strings have greater loss rates than structured strings.

The importance of these qualitative features is simply that they show the model to be capable of approximately replicating the experiment, in contrast to models of STM which have no means for dealing with these experimental phenomena at all. We know of no other STM model which responds to syntax, performs in the presence of noise and deals with composite stimuli. Many models either fail to address the task at all, or lack enough detail to make a determination of their suitability for modeling these phenomena possible.

---

* Miller and Isard do not specifically mention misperception, but we may presume that it is one of their error phenomena, since it is a common experience in similar non-experimental contexts.

COMPARISON TO OTHER MODELS OF STM

CHAPTER 6-1

The simple idea of STM as a limited store, with the idea that the items in that store are constructed by grouping given information, is adequate to explain some of the gross phenomena of short term memory experiments. The fact that effective capacity for ordered binary digits increases under deliberate coding is one such phenomenon. The approximately equivalent effects of words and cliches as interference units is another.*

The Slate system represents these phenomena much more accurately, in that it represents the sharing of STM as a workspace before and after chunking, and identifies the storage rule which determines how much STM capacity a group being formed will take up.

There are a number of other models of STM which are relevant to understanding the contribution of the Slate system. Each goes beyond the simple notions of STM and chunking in various ways. The ones which are particularly relevant are those which are explicit enough to show how phenomena might arise, which represent the form in which immediately accessible information could be stored, and which represent how knowledge from previous experience is brought to bear on the present task. Three of these models are compared to the Slate system below.

---

* Chapters 3 and 4.

FRAN, by John Anderson, is of interest because of the resemblance between the representation it uses and that of the Slate. [A68]

John Morton's model, remembered for its logogens, is of particular interest in that it includes an explicit model of word recognition under noise. [M69]

The EPAM model, also manifested as MAPP, includes both an explicit recognition method and a representation of the nature of STM content. [F59], [FS62], [GS67], [SF61], [SF64], [SG73].

FRAN

The FRAN (Free Recall in an Associative Net) model is embodied in a computer program which acts as a subject in standard free recall experiments.

> "By 'standard free recall' we mean to reference those experiments where the list to be studied is composed of common nouns chosen in a fairly random fashion. The words are presented one at a time at a constant rate. If it is a multi-trial experiment, study and recall phases are alternated."

The model is built to represent the ability of subjects to learn lists with practise and to show several effects which relate subjects' performance to linguistic relationships between list items.

FRAN is a processor having three memories. The largest, LTS, is a network in which words are linked together by relations, similar to Quillian's. [Q64], [Q69] The second, STS, is a set of pointers to words in LTS, with a first-in first-out discipline imposed on a fixed-length list. Another set of items, called ENTRYSET, is effectively a small memory which functions under a different discipline than STS.

FRAN functions only as a free recall subject, alternately receiving and reporting word lists. When receiving a list, FRAN is marking its LTS in a probabilistic manner.

Marks are cumulative. Reporting a list involves searching for words in marked regions, starting from STS and ENTRYSET. The identities of the relations which hold LTS together have no effect on either the marking process or the search process.

FRAN contrasts with the Slate system in a number of important ways. First, there are no chunks in LTS. The system of words and links has no natural boundaries which divide it into parts. Similarly the STS unit is always the word, rather than any sort of familiar group. There is no encoding for word recognition or search for structure in the incoming word sequence; BOWL PARADE ROSE and ROSE BOWL PARADE must be given approximately the same response. All of LTS is descriptive of the language of the subject; there are no obvious means for storing information of other kinds.

These features of FRAN make it unable to deal with a broad range of phenomena because it is unable to represent the structure of the given information. The parts of FRAN have clearly been included because of their potential involvement in free recall simulated-subject behavior. The system reveals structure in the task and data on subjects, much as a grammar reveals structure in speech; it tells relatively little about how a general intelligence might accomplish free recall as one of the tasks in its range. In contrast, the Slate system is much broader in the range of tasks which it can accomplish, but it has not been developed to the point of producing a diversity of free recall phenomena.

LOGOGEN MODEL

Morton's model differs from FRAN in that it is more elaborate and addresses more than one experimental paradigm. It includes attempts to provide detailed models of specific acoustic and visual phenomena which are beyond the scope of the Slate. We can compare their word recognition methods and information representations directly.

The model has a visual word-feature recognizer and an auditory word-feature recognizer that present results to the logogens. There is one logogen for each word in the vocabulary. A logogen is a kind of accumulator of feature scores. Words are assumed to arrive at one of the recognizers independently, one at a time. As the feature recognizer identifies features which are present, each logogen sums scores corresponding to those features which are parts of its own word, using a logarithmic formula. One logogen is assumed to be the first to exceed its threshold, with the result that the corresponding word is made available for output, and responses from other logogens are suppressed. If noise is present, then it may be that no logogen reaches its threshold (because some feature identifications were absent.) If so, all thresholds are jointly reduced until some one logogen reaches its threshold. That word is then made available for output, and others are suppressed as before.

Use of context is effected as follows:

> "The effect of a context on the production of a response is identical in principle to the effect of a stimulus. The context, which can be of any kind, causes what is here called the Cognitive System to send semantic information to the Logogen System. Attributes such as <noun>, <animate>, <male> might be produced by the sentence "He was a drunken ---J---" and all logogens whose semantic sets contained these items would be incremented accordingly. These increments would effectively reduce the amount of sensory information required to produce the response, the attribute counts from the two sources simply adding without regard to source."

The Logogen and Slate models are similar in that they have explicit word representations which are used for recognition, and explicit processes applied to the input characterizations. Both deal with noise as a kind of incompleteness of input. The logogen system relies principally on arithmetic methods for judging evidence, whereas the Slate system relies on more qualitative and logical matching methods.

The assumption that words arrive one at a time turns out to be essential to the Logogen model, since the recognition process selects one word for each group of features presented. The model is inappropriate for connected (ordinary) speech and for other composite stimuli in general. This is a real weakness, since there is evidence that there are no consistent acoustic features which correspond to word boundaries. [N73] Composite stimuli also occur in a number of classical experimental paradigms, including monaural shadowing.

The capacity of the Slate system to deal with composite stimuli is most clearly represented in its chunking of computer instruction streams, presented in chapter 9.

Most words in spoken or written English are redundant, so that more independent features of a word can be found than are needed to identify it. In the absence of the complete set of features, some subset can be used to identify the word correctly in its logogen. (Some such capacity is necessary in any system which is to recognize in the presence of noise.) Identification can occur in spite of the presence of features which indicate that the word does not have the correct form. Thus if the word REDACT is unknown it may be identified as REACT, rather than as an unfamiliar combination. What is missing is the ability to respond to contrary evidence, to respond to those features

from all of speech which should not be present in the given word.

Word recognition depends on a complex variety of related kinds of evidence, requiring in the model of the recognizer a corresponding capacity to deal with complexity. The Slate system deals with a greater variety of evidence than the Logogen model, since it does respond to contrary evidence in the form of conflicts of relational properties. These conflicts occur in a variety of ways, so that the Slate system incorporates several different methods for responding to contrary evidence.

The model as stated has a flaw which could prevent it from functioning at all. The context mechanism described above does not distinguish between a word in the context and a word in the stimulus when collecting evidence in the logogens. The first word of a sequence which was recognized would enter the context (in the Cognitive System) with all of its features, keeping its own logogen above threshold and thereby suppressing all further recognition. Any of several minor modifications would eliminate this problem. It is a non-trivial problem, however, because it is necessary to provide some means of recognizing and representing the second occurrence of a word in a context. (On this basis alone, we recognize that Morton did not test this model.)

Two related problems are more serious. First, the assumptions about an effective way to derive a contextual bias toward certain words are equivalent to those we used to attempt to get the same effect in an experiment above. (Chapter 5.) They failed for us for reasons that seem to apply equally to Morton's model. The role of a word in context generating expectations differs from the role of a word being recognized, so that the recognition features are unsuitable as context cues.

A second problem concerns the possibility of an excess of evidence from the context. It is possible to match enough context to recognize a word with no evidence whatever from the stimulus. If context is thereby expanded, the process may not stop. During the development of the Slate system, some versions had a tendency to enter such fantasies freely. The problem would seem to be worse with Morton's system because of the lack of response to contrary evidence.

The form of this model differs significantly from that of the Slate system in that separate facilities are used for deriving features, recognizing words and using words. Since we have found STM to be a shared workspace, where grouped and ungrouped elements reside together, this feature is questionable.

The knowledge of words in the logogens is committed to the single purpose of word recognition in the Logogen system, whereas in the Slate it is available for either recognition or generation. (Chapter 10.)

MAPP AND EPAM

These two systems are represented togother here since they are variants on one theme. EPAM (as EPAM III) performed various verbal learning tasks, and more recently MAPP has been developed to perform a chessboard observation and reconstruction task. [SG73] The systems use a common form of LTM to store the relevant task knowledge. One half of LTM is a discrimination net, a binary tree of tests on available information. There is a single entry point, and at each of the terminals of the tree is the name of an

identified chunk (stored in the other half of LTM) which has been located by the set of tests on the path from the top of the tree to the terminal node.* (This chunking action makes it directly comparable to the Slate system.) The STM is an unordered set of these LTM chunk names.

Since MAPP must process one set of given information (a chessboard) through one tree of tests and yet obtain more than one resulting chunk, it needs a means for varying the path through the tree. The mechanism which does this is a chessboard scanner which is part of the task supervisor. It selects a varying starting square based in part on the most recent tests of the board. After a fixed number of chunks have been identified, STM is full and observation processing stops; nothing leaves STM.

Board reconstruction is then performed by reading the contents of the chunks named in STM, each chunk being a set of pieces with their locations specified. EPAM is able to deal with incomplete stimuli if the missing parts are all in places in the stimulus which are not subject to tests in the discrimination net.

> "Images may also be evoked upon presentation of partial stimulus objects (as C-T may evoke CAT)." [SF62]

MAPP and EPAM both perform exact matches in the tree; all elements tested for must be present. For redundant stimuli, this converts the recognition problem into a smaller recognition problem on a non-redundant stimulus. The redundancy is therefore

---

* Part of the importance of EPAM is that it embodies an explicit theory of the acquisition of the knowledge of single stimuli; the Slate system lacks such a theory. Neither Slate nor any of the EPAM family performs acquisition of LTM chunks based on composite stimuli.

unavailable to combat noise during recognition. In MAPP the tree always contains an exhaustive set of tests for each chunk. This is appropriate for their task, since the chessboard is entirely visible during the time in which it is being observed.

We have tried to find a straightforward way to augment the EPAM/MAPP techniques in order to let them handle single incomplete stimuli, but there are difficulties. One approach is to let the tree be ternary, with a possible indeterminate outcome on each test. This leads to combinatorial expansions of the tree. To recognize a 10-letter word by any 5 or more letters present requires at least 252 different pre-terminal test nodes for that one word. There is no corresponding expansion of the Slate system because the set of tests to be applied for a given chunk is generated from the chunk at match time.

Since terminal nodes in the tree correspond to single identified stimuli, the system, like the Logogen model, is not suitable for assimilating composite stimuli. In the case of MAPP's treatment of the chessboard, which is a composite stimulus, the problem is avoided by use of its chessboard scanner and by performing a complete match to detect any chunk.

If incomplete and composite stimuli are to be assimilated, then the tests performed must be somehow constrained so that the elements of one chunkable configuration are never taken as parts of another. So, given

THESUN

we must not take the final N as evidence for the word THEN. The means available in the Slate system for getting these tests to come out right are clear because the chunk

graphs are available and the tests are all embodied as graph match acceptance tests. For the EPAM family the case is less clear because the set of allowable tests and the representation of the stimulus are not similarly constrained.

Because STM is an unordered set of pointers to chunks, the things which can be known in STM are limited in strange ways. There is no place in the EPAM memory to store information from the stimulus which is variable from one occurrence of a chunk to another, rather than being fixed as part of the chunk. An example is the inability of MAPP to hold a single chunk embodying the concept of "two knights in mutual defense" which was discussed in a previous chapter. Since the Slate retains arcs which express the assertions about the stimulus which do not match the chunks, this is not a difficulty. The problem of representing variability also makes it difficult to adapt EPAM or MAPP to respond to syntactic structure, where the combinations of allowable words in phrases should not be enumerated in the tree.

Because the stimulus is distinct from STM, and testing in the tree is applied only to the stimulus, the EPAM family does not exhibit a shared workspace effect. The "workspace" of EPAM is the locus of control in the tree.

Each element of knowledge embedded in a test in the tree must be independently re-represented in the chunk at the terminal node so that the knowledge becomes accessible from the name of the chunk. This double representation is avoided in the Slate. The difference is mostly an aesthetic matter for computer models, but becomes more serious if the physical embodiment of the knowledge is subject to error, since the two may drift apart without detection. Whether double representations are empirically

justified (by data on aphasics, for example) is unclear.

COMPARATIVE SUMMARY

Several of the points of comparison which appear repeatedly above are summarized in Table 6.1 below.

	SYSTEMS		
COMPARISON	SLATE	LOGOGEN	EPAM/MAPP
Encodes on incomplete input	yes	partial	yes
Recognizer accepts composite stimuli	yes	no	no
Shared workspace effect	yes	no	no
Variable information in STM	yes	no	no

Table 6.1- Comparisons of Systems

Each difference has some direct effect on the range of experimental tasks and data which can be accounted for.

PSYCHOLOGICAL SUMMARY

CHAPTER 7-1

The purpose of this section is twofold: to discuss some psychological topics which span the experiments and theory at hand, and to summarize our psychological results by presenting them as a coherent theory. The topics to be discussed are:

Interpreting the Shared Workspace Effect
Seriality
Variability and novelty
The role of redundancy
The role of conflict
Efficient control of well-practised tasks
Opportunities created by this work.

INTERPRETING THE SHARED WORKSPACE EFFECT

We have seen in the studies of interference and coding that capacity for holding symbols before chunking them into groups and capacity for holding the resulting group chunks are subject to a common upper bound on their sum. The upper bound is applied continuously, so that at each point in the reception of a list, chunked and unchunked symbols may compete for space.

complete chunks	+	fragments of incomplete chunks	≤	total capacity

In both of the experiments in which the shared workspace effect appeared, the fragments were words (one, zero, rose, bowl.) An understanding of the diversity of kinds of fragments which share this (or other) workspace would be particularly interesting

PSYCHOLOGICAL SUMMARY

CHAPTER 7-1

The purpose of this section is twofold: to discuss some psychological topics which span the experiments and theory at hand, and to summarize our psychological results by presenting them as a coherent theory. The topics to be discussed are:

Interpreting the Shared Workspace Effect
Seriality
Variability and novelty
The role of redundancy
The role of conflict
Efficient control of well-practised tasks
Opportunities created by this work.

INTERPRETING THE SHARED WORKSPACE EFFECT

We have seen in the studies of interference and coding that capacity for holding symbols before chunking them into groups and capacity for holding the resulting group chunks are subject to a common upper bound on their sum. The upper bound is applied continuously, so that at each point in the reception of a list, chunked and unchunked symbols may compete for space.

complete chunks	+	fragments of incomplete chunks	≤	total capacity

In both of the experiments in which the shared workspace effect appeared, the fragments were words (one, zero, rose, bowl.) An understanding of the diversity of kinds of fragments which share this (or other) workspace would be particularly interesting

because it would exhibit some of the relatively inaccessible structure of STM.

The fact that the effect is a bounded sum of memory capacities has consequences for the admissibility of STM models in general. (None of the 3 alternate models discussed above exhibits a shared workspace effect.) The only sorts of structures which appear to be suitable for accounting for the effect are various kinds of memory resources shared by multiple processes or information sources. Every insertion in such a memory must obey its capacity limit formula. Our experiments deal with two kinds of insertions:

1. An item is received prior to chunking.
2. A complete group of items is consolidated (chunked) into a unit.

Figure 7.1 illustrates the information flows of these two kinds of insertions, based on the assumption that the knowledge which is the basis of consolidation is stored in LTM.

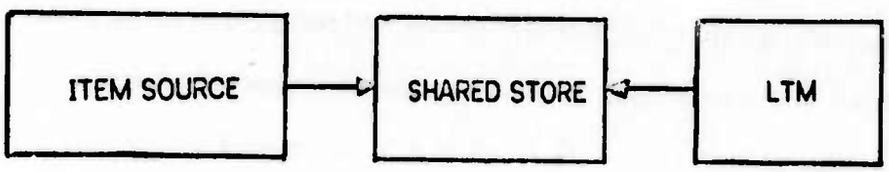


Figure 7.1- Shared Store Information Flows

Since we believe that information comes from LTM on the basis of some kind of search or association of content, we must add a flow from the shared store to LTM, thus forcing a loop. (Figure 7.2.) No matter how we complicate the model, some manifestation of a loop will remain. (Note that this loop is not the long term storage and later use

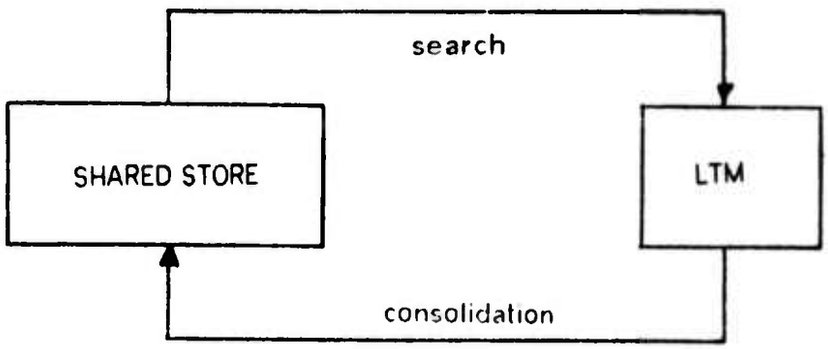


Figure 7.2- Inherent Loop in Workspace Model

loop, but rather a loop entailed by single acts of chunking in recognition.) On this basis we may rule out all models whose information flows do not contain a loop through the shared store and LTM. All "pipeline" models, in which information transformations form a chain (or a lattice without loops) from stimulus receipt to response delivery, are ruled out on the assumptions of a shared store and LTM search. Since such models are rather numerous, this provides an interesting means of discrimination.

SERIALITY, PROCESSES AND SHORT TERM MEMORY

In seeking to understand human "short term memory behavior," a great variety of descriptions have been formulated. They exist in a diversity of representations, including verbal statements, statistical hypotheses, computer programs, flow charts, and so forth. One point of concensus is that there is an aspect of seriality about STM-related activity, that at some level of description it is accurate to say that things happen one at a time.

The seriality can be attributed to the retrieval processes which access long term memory, the processes which accommodate the results of retrieval, control processes which use STM as a resource, a requirement that certain operations on STM use as input the results of other such operations, or the nature of STM itself.

At first glance it would seem that the Slate system could not provide any evidence on the nature of this seriality, since all parts of the system operate on a serial computer. However, we can dismiss some of this seriality as inessential and show that:

1. The Slate system must behave in a generally serial fashion.
2. Its essential seriality is extremely local.
3. The essential seriality arises from the properties of the Slate memory itself rather than the retrieval or control processes which use it.
4. The need for those properties arises from the nature of the chunking task.

In chapter 16 it is shown that there are no serial constraints on the process of locating chunks in bulk memory which are candidates for entry into the Slate. It is also shown that there are no order constraints on the transfer of elements of a chunk from bulk memory into the Slate. Either each element of the chunk is successfully transfered, or one of the essential conflicts between the chunk and the Slate content is found. Any one may be found with the same effect: rejection of the bulk memory chunk. Since both the identity of the particular conflict which is found and the order of events leading to a conflict are irrelevant to the result, and since all of the possible ways of succeeding are equally acceptable, the transfer may be developed in a parallel fashion.

In contrast, the conflict-producing mechanism of the Slate is essential, since it is the only means for avoiding chunking acts which result in incoherent combinations of assertions. The Slate is necessarily a one-hypothesis memory. This is illustrated in the example of the Necker Cube in chapter 11, where the one-hypothesis-at-a-time property of the Slate leads directly to a reversal phenomenon.

The one-hypothesis property was not sought in the design of the system. Neither was a particular localization of its seriality deliberately designed in, since it was developed as a wholly serial system. The need for restricting the set of assertions which jointly reside in the Slate comes from the desire to disallow certain combinations. We desire that sets of assertions which do not correspond to acceptable external interpretations be disallowed, and yet that all sets which yield acceptable interpretations be allowed. (The logical properties enforced over relations, as described in chapter 12, are the specific means for accomplishing this.) The fact that the memory itself is the essentially serial element thus arises from the constraints of the chunking task, along with the choice of a representation in which the choice of a representation in which the functional requirement for coherence of memory content is satisfied by the memory itself.

Note that this is a kind of seriality which is relative to the particular memory content. Conflicting hypotheses must be considered one at a time, but several independent hypotheses may be considered at the same time, and so may be chunked in parallel. The assumption of this locus of seriality thus has consequences in time predictions.

VARIABILITY AND NOVELTY

The representation of a chunk in the Slate system allows for the chunk to specify essential attributes and to fail to specify attributes which may vary freely over the span of application of the chunk. This allows familiar chunks to appear in novel combinations, and provides a homogeneous medium for representing familiar and novel aspects. Consider for example chunking of piece configurations on a chessboard, where we wish to be able to treat the notion of "two knights in mutual defense" as a recurrent configuration represented by a single chunk. In the representation described in chapter 12, we might use the graph in Figure 7.3 as the chunk. The relationships of defense and type must correspond to the board for the chunk to be applicable. In contrast, the specification of the particular piece locations is variable from one board position to another.

It is essential that a theory of STM provide for the occurrence of both types of information. It would be unreasonable to require specifications of board locations in the chunk (the approach taken in MAPP), since this would limit its application in unproductive ways. It would likewise be unreasonable to prohibit representation of the piece locations in STM because they were not included in the chunk.

The alternative of providing (in one's theory) two different sub-memories of STM, one for chunk recurrences and one for novel information, does not seem to be justified in its complexity by our present knowledge of STM phenomena. It also leaves the problem of explaining how things in the "novelty representation" submemory can be made to eventually appear in the "familiar chunk representation" submemory.

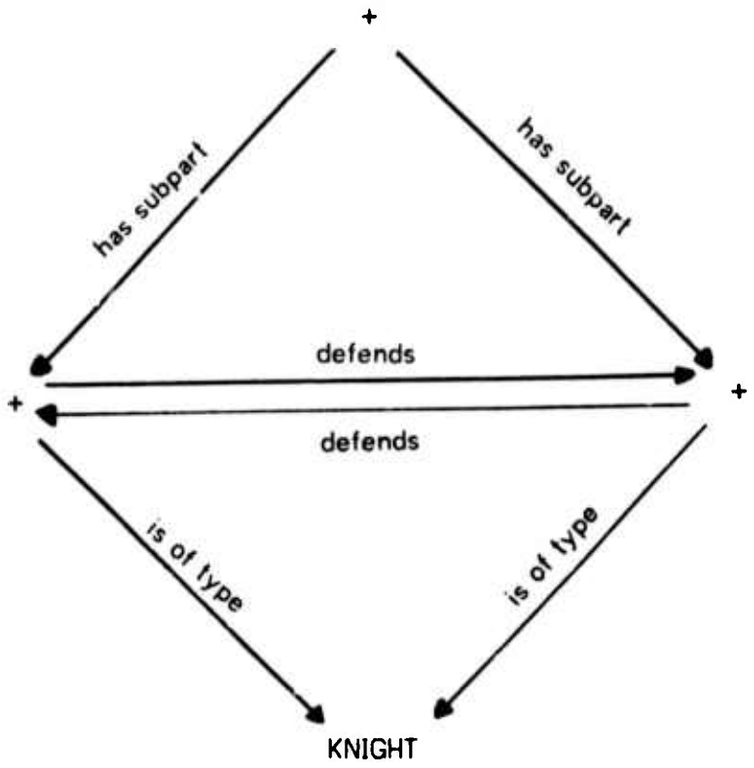


Figure 7.3- Chunk for Two Knights in Mutual Defense

The Slate system provides for the occurrence of both kinds of information by providing a homogeneous medium in which chunks can be further annotated by assertions which are peculiar to the occurrence at hand.

REDUNDANCY

It is necessary for a theory of human information processing to represent the fact that people can make effective use of the redundancies in information presented to them. Identifying words in the presence of noise is one instance of this capacity, which extends across sensory modalities. Presented information can be incomplete in numerous unpredictable ways without disturbing the subject's ability to interpret it; the information which remains is used to identify and complete the whole. The difficulty for an information processing theory is that the methods for interpretation must cope with the whole range of variation, in which the information needed to make particular elementary decisions during interpretation may not be present. Sufficient combinations of parts must be detected in order to identify objects. One way to deal with this need is to enumerate all the sufficient combinations and seek each in turn. The enumeration might be done either at the point of need or a-priori, but it is subject to combinatorial explosion of the number of cases either way. Some theories would require an a-priori enumeration, vastly complicating acts of interpretation.*

Since a recognition act which leads to identifying an object must be able to deal with the entire object, it is natural to include the use of redundancy as part of the chunking process. The Slate system uses the redundancy of given information without engaging

---

* See the discussion of Mapp below.

in any enumeration of the possible incomplete manifestations of possible objects. The given information must agree with the corresponding chunk for recognition, and the missing parts are simply filled in from the chunk. Identification depends on the amount of positive evidence, not on its configuration. Thus it is reasonable to consider the Slate system as a model of human chunking in complex redundant environments, since it avoids engaging in a combinatorially large number of elementary steps.

THE ROLE OF CONFLICT IN CHUNKING

People are selective in their chunking behavior, so that they do not chunk given information in ways that are "obviously wrong" in the given symbolic context. The rules which define what is "obviously wrong" are task dependent. For letter strings we have used the rule that at most one letter follows a given letter in a totally ordered sequence. It has turned out to be vital to be able to reject certain chunks even though they fit well in part, by use of such rules of conflict. The continuous application of such rules keeps the Slate content coherent.

We might imagine that the set of such rules grows as fast as the set of tasks that we attempt, but such is not the case. All of the rules of conflict have been codified into a small number of classes which are used on all of our tasks. The current Slate system uses 9 of them. Increasing the fluency of the representation might cause this figure to rise, but we expect that the set would remain fairly small. The discovery that such a small set is adequate to support chunking processes is one of the interesting psychological results of the work.

It is interesting to consider what would be the effect of adding these rule forms one at a time to a chunking system, seeking to determine its range of application at each step. There are so few forms, so broadly used, that we can anticipate large qualitative changes in system capabilities with each addition. Operations which were impossible might suddenly be trivial, and information which required long times to organize might be handled in a glance.

Similarly we can ask about the development of the same fundamental forms in humans. Do they exhibit a developmental sequence? Is it the same for all individuals? Are the sequences which occur limited only by formal necessity or are there other limits?

The role of these forms in the Slate system suggests that they might generate a developmental sequence of qualitatively different intellectual capabilities. Comparison to the work of Piaget is therefore of direct interest.

CONTROL OF WELL-PRACTISED TASKS

This research has not focused on the form of control of tasks which include chunking. The capacity to chunk appears as a separable resource which may be utilized and directed by goal oriented processes. We have evidence in the tasks examined of people's methods for using the chunking resource effectively.

From Smith's coding task, it appears that he was not chunking except at the ends of code groups. Since the groups have no internal redundancy, this is the first point at which there is more than a 50% chance of correct chunking. In the symbol string tasks

we have found a significant efficiency advantage in anticipation, chunking before all of the evidence is in. The advantage is lost if many of the anticipations are wrong, leading to wasted reformation of memory content.

The kind of control of chunking which we expect on a well-practised task balances the effort savings of correct anticipation and the effort loss from bad anticipation, so that chunking is initiated when there is enough evidence on hand so that there is a high probability that the result will survive.

We also expect that control of well practised tasks will devote a maximum amount of STM space to achieving task performance, eliminating extraneous chunks. Smith's performance is particularly stable in his use of his 12 chunks.

For less practised tasks, control of the initiation of chunking is less effective, so that the apparent number of available chunks is lower and performance rates are lower as well. Improvement in performance with practise can thus arise either from improvement of the control process or development of more suitable chunks.

The scope of control of chunking may also encompass the selection of parts of STM content as interesting and the decision to abandon or retain chunks; the Slate system does not provide for these kinds of control in its present form. The evidence from our tasks suggests that there is task control over the moment of epplication of the chunking process, and that that control is most effective when there is a high probability of survival of the result whenever chunking is initiated.

OPPORTUNITIES CREATED AND FACILITATED BY THIS WORK

This section presents several short sketches of psychologically motivated developments which might be pursued with this work as a starting point. I view them as research projects relatively closely coupled to the present work but for the most part loosely related to one another. Another group of opportunities with non-psychological motivations appears elsewhere in the thesis. The opportunities discussed below are:

1. Learning Models
2. Single Modality models
3. Second-generation Study of Noisy Speech
4. Representation Development
5. Structure of STM

1. In building explicit models of learning followed by performance, it pays to be sure that achieving the learning will be sufficient to permit the desired performance. Having such knowledge, we can separate the model-building problem into two subproblems:

> 1. Find an adequate representation and content for the learned knowledge, where adequacy is demonstrated by using the knowledge to achieve the desired performance.
>
> 2. Find a way to get the system/organism to change from its beginning state to one which incorporates that specific representation and knowledge.

The advantage of this organization is that any failure of performance can be identified as either a failure to reach the desired knowledge state or a failure to use it properly. The site needing improvement is therefore known at each failure. For most kinds of human performance we lack the knowledge of an adequate representation.

Some progress is evident in models of game playing, and good representations are known for many numeric and symbolic mathematical tasks.

The opportunity provided by this work is that it offers an explicit representation whose capabilities are understood and demonstrable. The development of a learning model can focus effort on the questions:

1. How can chunks like those of the Slate system be acquired from experience?

2. How can appropriate control processes be acquired?

The learning of chunks is particularly interesting since, unlike control processes, they are not identified with particular tasks. Chunks might be learned as follows:

Our representations for knowledge and experience are homogeneous, so that it is possible to store experiences in LTM, chunked to whatever degree occurs, as well as abstract knowledge. We can discover correspondences between stored and current experiences by using matching methods like those which we have already used to relate chunks to current experiences. Regions of stored experience which repeatedly correspond to new experiences can be abstracted, named and stored as chunks.

2. Single Modality Models - Our uses of the Slate system have not addressed the details of any particular sensory modality. There are a number of specific phenomena related to chunking activity which need to be accounted for. An example would be the various categories of errors in auditory memory experiments. In order to model them with a Slate-like system, initial requirements would include a substantially faster system

and chunks which reflect the fine structure of the input information to be organized. The Slate system can be examined as a model of knowledge use for a variety of time scales of human performance, from the several- millisecond level to the multiple- minute level. Visual tasks can be addressed if the handling of positional ordinality is improved.

3. Second-generation Study of Noisy Speech - Our study of noisy speech has pointed out an inadequacy of the Slate representation for expectations. The inadequacy is not identified with particular task domains or sensory modalities. A new approach to characterizing expectation could be developed along the lines of the learning model described above. It would be possible to ask of the stored experience: What normally follows this chunk? or What regularities are there in what normally occurs in the neighborhood of this chunk? Alternately an abstract expectation part of a chunk might be developed.

4. Representation Development - The power of the representation in the Slate system is too weak for direct application to semantic investigations. Specific deficiencies are discussed in chapter 17. Development of the representation would make it possible to study issues of how subjects understand and misunderstand the experimenter's instructions, what is the possible range of subject's task representations for a given set of instructions, and how instructions affect performance. The fact that subjects regularly do what they are told to do by experimenters is one of the most frequent and least modeled experimental phenomena.

5. Structure of STM - There is evidence that between the most primitive sensation of external energy and the workspace for chunking of words into phrases there are

other forms of memory. We can push examination of the shared workspace effect in the direction of more elementary constructs, and perhaps discover a boundary of sharing below which workspace is not shared with words and phrases.

THEORY OF CHUNKING

This section summarizes the psychological results of the thesis by presenting a theory of chunking derived from work with the Slate system. The theory makes explicit what parts and features of the system we regard as theoretically relevant.

The theory is an explanation of human acts of information organization which put knowledge of given stimuli in correspondence with previously acquired knowledge. The organizing behavior is not identified with any particular sensory modality, but rather spans several, including hearing and vision. We do not identify organizing behavior with sensory activity; rather any source of information is potentially subject to organization.

The theory is not specific on time spans. The elementary presentations of the experiments studied take from several hundred milliseconds to a half minute or so. The chunking is rapid enough so that it does not limit the rates of most human activity.

The results of organization are discrete symbolic augmentations of the given information.

ELEMENTS OF THE ORGANIZING PROCESSOR

Organization is performed by an information processor which consists of:

1. A long-term memory
2. A short-term memory
3. Means of access to information sources
4. A chunking process
5. A control process

1. The long-term memory (LTM) is an associative memory, i.e. it can be accessed only by partial specification of the content which is of interest. It contains an unlimited number of non-overlapping chunks. Each chunk is a set of assertions. Each assertion is an ordered n-tuple of symbols, one of which (the relation name) is subject to constraints of occurrence. There is a small fixed set of forms which specify how occurrences of relation names are constrained. Constraints apply to the entire content of the memory. The number of symbols which may occur is unlimited.

2. Short-term memory (STM) is an associative memory which is able to hold assertions of the same kind as LTM, using the same symbols and constraints as LTM. Each assertion in STM is part of one or more identified sets of assertions (chunks) in STM. Chunks in STM thus may have assertions in common, in contrast to those of LTM in the sense that the constraints of occurrence must hold for the entire STM content, STM is a single- hypothesis memory. There is a small fixed maximum number of STM chunks.

3. Each access path to an information source is a means by which chunks are inserted into STM. Each sensory modality which affects STM has one or more such access paths.

4. The chunking process operates in discrete steps, each of which is an attempt to partially match current STM content to LTM chunks and copy the LTM chunk into STM under the guidance of the match. Each step has one of two outcomes:

1. The content of STM is unchanged, or
2. One chunk from LTM is used as a pattern to construct a chunk in STM.

5. The chunking process may access any number of assertions in LTM at one time. LTM content is unaffected by the chunking process.

6. The only way in which one step of the chunking process affects another is by alteration of the content of STM.

7. The chunking process may have no effect on a particular step because the achievable overlap of the LTM chunk to current STM content is too small. (Overlap is the set of assertions in common.) Among the feasible ways of copying an LTM chunk into STM, the chunking process tends to pick one with large overlap.

8. When a chunk is copied from LTM, a new chunk is formed in STM which contains all of the copied assertions and all of the assertions in chunks overlapped by the copy.

9. The control process is capable of initiating the chunking process. The control process selects the particular chunk to be removed whenever the fixed limit of chunks in STM would otherwise be exceeded, and may select chunks for removal under other conditions. The control process may create and insert chunks into STM.

10. Chunks in LTM will not be copied into STM if their assertions span serial information access which is not yet complete.

11. If there is a chunk whose assertions have been included in another chunk in STM, such a chunk will be selected for removal in preference to a chunk whose assertions are not so included. (This part and parts 2, 3, 8 and 10 above embody the shared workspace effect.)

* * *

We can see that the above theory is partial in the sense described in the introduction, that it covers organization and representation of current events in accordance with prior knowledge, but not the creation of the chunks of prior knowledge which constitute LTM content.

STUDY OF ASSIMILATION METHODS

CHAPTER 8-1

From this point on our attention is on methods for achieving assimilation. We are interested in knowing how assimilation problems may be solved, what makes them difficult and how various methods differ in their capacity to assimilate.

The difficulty of the problem is somewhat obscured by the simplicity of the tasks taken up in previous chapters. They suggest that assimilation might be performable by a simple string match of some sort.

In order to evaluate this suggestion, we might view string matching as a simple primitive process with unspecified control. Alternatively we might examine sophisticated string matching approaches with highly developed control mechanisms to see what they might contribute. We choose to do the latter, using SNOBOL4 as the representative body of methods. Chapter 18 includes a reexamination of a task introduced in Chapter 9, showing how Snobol4's string-match control methods fail to respond to some of the most important features of assimilation.

Three new assimilation tasks are introduced in chapters 9 to 11. They each serve to demonstrate diversities of Slate system performance that are required for some assimilation tasks but not for the psychological experiments taken up thus far. The directed graph representation is presented in Chapter 12. The structure of the Slate system is presented and its heuristic methods are analyzed in various ways in Chapters 12 to 14. Its use of time is presented in Chapter 15, with some suggestions for significant improvements.

The remaining chapters are devoted to interpretation of the system. It can be seen as a highly parallel production processor which is serially implemented. (Chapter 16). The generality of the system and the power of its representation are evaluated in chapter 17, and it is contrasted to other relatively sophisticated match-based systems in Chapter 18.

DISCOVERING CONTROL STRUCTURE IN MACHINE CODE

CHAPTER 9-1

THE PROBLEM OF DISCOVERING PROGRAM CONTROL STRUCTURE

The problem is one of assimilating computer programs written in machine code according to knowledge of the control structures of a source language. Given the machine-language results of a compilation, we would like to be able to recover from the program a complete account of the routines, loops, case statements and so forth found in the uncompiled program, by examination of the compiled instructions alone. The problem as stated is less than complete "decompilation" in that it does not aim at reconstructing a full program text or describing the non-control parts of the program. (It appears that the latter is a straightforward extension of the methods used below.)

EXTERNAL PROBLEM REPRESENTATION: A compiler produces a machine language program from a source language program, representing the result in a listing. The machine language descriptions in this listing are the external representation of the given information.

INTERNAL PROBLEM REPRESENTATION: A "problem graph" containing vertices which correspond to the machine language instructions listed in the listing, their operation codes and memory references. The source language program is not represented.

SOLUTION REPRESENTATION: The problem graph is augmented with vertices and arcs which describe the control structures attributed to the source language program.

CRITERIA FOR JUDGING SOLUTION:

1. Any control structure known to the Slate system and present in the actual source program must be identified in the machine code.

2. The beginning and end of the scope of each control structure must be correctly identified.

3. Mutual reference (such as occurs between routine calls and routines) among control structures must be identified.

4. Any control structure not present in the actual source program must not be identified in the machine code.

5. Parts of identified control structures which are represented in the system's knowledge must also be represented in the result.

6. All of the assertions of the solution must be mutually consistent and consistent with the assertions of the problem representation.

The compiler used is the Bliss compiler for the PDP-10 computer. Bliss is a language with an Algol-like block structure. Since our examples will use only a portion of the language, we will present only those constructs actually used. The extension to all of the Bliss control constructs is discussed later.

Figure 9.1 shows a short program in Bliss.

```
MODULE XX=(                                          (1)
ROUTINE INTSUM(X)=(.X * (.X + 1))/2;                 (2)
GLOBAL          NUM,SUM;                             (3)
                SUM ←                                (4)
                CASE (.NUM GTR 0) OF                 (5)
SET                                                  (6)
                -INTSUM(-.NUM);                      (7)
                INTSUM(.NUM)                         (8)
TES                                                  (9)
) ELUDOM                                             (10)
```

Figure 9.1 - A Short Program in Bliss Language

It computes the sum of the integers in the inclusive interval from the number in location NUM to 0.

Line (2) specifies calculation of the sum by a method which is valid for positive numbers. The occurrences of "." in the program all denote taking the content of the memory location named by the variable which follows.

Line (4) specifies that the value of the following expression shall be stored in variable "SUM". The case expression in lines (5) to (9) performs one of two computations. Choice of which computation to use is made in the expression (.num gtr 0) in line (5), which tests the value of variable "NUM." If the value is positive, the value of the expression is 1, and 0 otherwise. For the expression value of 0, case 0 is computed (line (7)). For the expression value of 1, case 1 (line (8)) is computed. The result of the chosen computation is stored in "SUM." Lines (1) and (10) define the boundaries of the program.

The Bliss compiler translates the program to PDP-10 assembly code, also producing a listing of the result, Figure 9.2 .

```
00100   0001   MODULE XX=(
00200   0002   ROUTINE INTSUM(X)=(.X * (.X + 1))/2;

                0000            JSP     12,.ENT.0 (11)
        0003    0001            MOVE    $V,-2($F) (12)
                0002            MOVE    04,3      .
                0003            AOJ     04,0      .
                0004            IMUL    04,3      .
                0005            ASH     04,-1
                0006            MOVE    $V,4
                0007            JRST    $S,.EXT.0 (18)

00300   0003   GLOBAL  NUM,SUM;
00400   0004           SUM ←
00500   0005           CASE (.NUM GTR 0) OF
00600   0006   SET
00700   0007           -INTSUM(-.NUM);
00800   0010           INTSUM(.NUM)
00900   0011   TES
01000   0012   ) ELUDOM

        0004    0000            MOVEI   05,1      (19)
                0001            SKIPG   04,XX.G+0
                0002            SETZ    05,0
                0003            XCT     $S,L1404(05)
                0004            JRST    $S,L1352
                0005    L1404:  JRST    $S,L1366
                0006            JRST    $S,L1370 (115)
        0007    0007    L1366:  MOVN    06,XX.G+0
                0010            PUSH    $S,6
                0011            PUSHJ   $S,INTSUM
                0012            SUB     $S,[000001,,000001]
        0010    0013            MOVN    $V,3      (120)
                0014            JRST    $S,L1352
        0011    0015    L1370:  PUSH    $S,4
                0016            PUSHJ   $S,INTSUM
                0017            SUB     $S,[000001,,000001]
        0012    0020    L1352:  MOVEM   $V,XX.G+1 (125)
        0013    0021            CALLI   $S,12     (126)
```

FIGURE 9.2 - Compiler Output

The notation is generally that of PDP-10 machine language, MACRO-10. Line (i5), for example, is an instruction to integer-multiply the content of address 4 with the content of address 3.

Each instruction has several fields which are treated separately in its interpretation. Those of particular interest here:

1. The operation code field (Opcode), which selects the symbolic operation to be performed.

2. The address field, which names an address in memory relevant to the operation.

The address field is used in different ways under different opcodes.

CONVERTING TO GRAPH REPRESENTATION OF THE PROBLEM

In order to represent the salient information about instructions we give the Slate system definitions for the following relations:

precedes
is the opcode of
is the address in
appears before

We define the following to be constants in the system:

XCT	PUSH	PUSHJ	JSP
POPJ	JRST	CAME	AOS
AOJ	SETOM	SETZ	ADD
MOVEM	MOVNI	MOVEI	MOVE
CALLI	SUB	SKIPGE	SKIPG
IMUL	SETO	AOJA	ADDB
MOVN	ASH	DPB	ADDI
CAMLE	CAILE		

All of the constants above are opcode names.

We establish the convention that every instruction and every item which appears on an address field will be represented by a token. Most of the tokens which we use will have three-letter names, since such tokens can be generated automatically in the Slate system. The names are arbitrary, but hopefully pronounceable. *

We could represent the instruction on line (i5) of Figure 9.2 by the graph in Figure 9.3 .

In Figure 9.3, RIH is the token representing the instruction. NEJ, JER and DAD are also tokens. All of those tokens represent locations in memory used by the program being represented.

The graph representing the machine code shown in Figure 9.2 is developed systematically in two steps. First, a copy of the compiler listing is prepared which has every instruction labeled with a token name and a token name for every other memory

---

* In figures, tokens which are not named in the text are often represented by + signs. Vertical unlabeled arcs represent the "precedes" relation. Converse names are used freely.

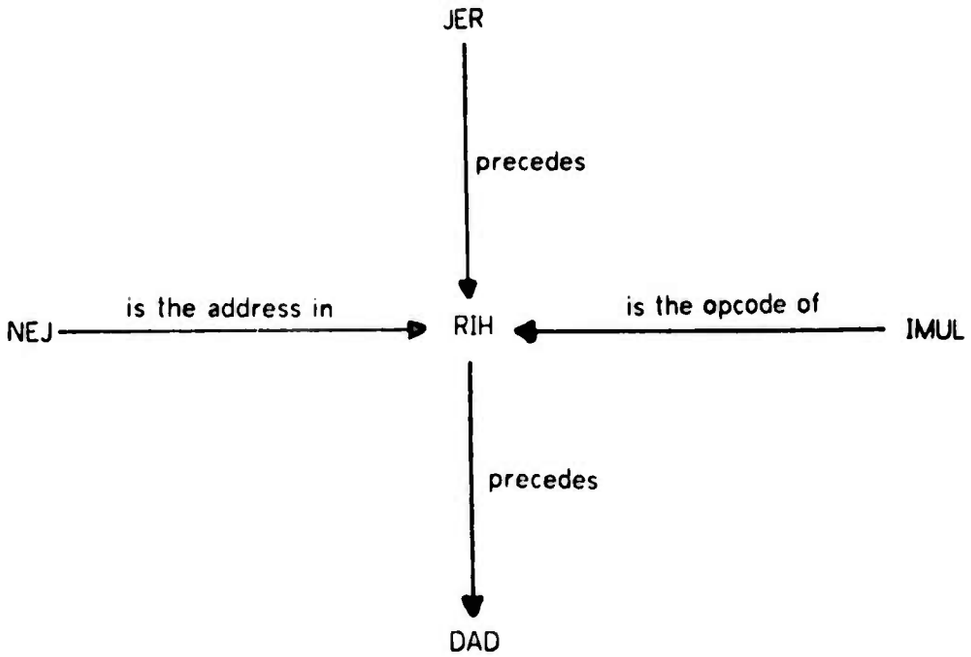


Figure 9.3 - One Instruction Graph

location, as illustrated in Figure 9.4 .

```
00100   0001   MODULE XX=(
00200   0002   ROUTINE INTSUM(X)=(.X * (.X + 1))/2;
GEM             0000            JSP     12,HAW
WAJ     0003    0001            MOVE    $V,X
REK             0002            MOVE    04,NEJ
JER             0003            AOJ     04,PIK
RIH             0004            IMUL    04,NEJ
DAD             0005            ASH     04,TIR
FAF             0006            MOVE    $V,BOH
LOL             0007            JRST    $S,EXITLOC
00300   0003   GLOBAL  NUM,SUM;
00400   0004           SUM
00500   0005           CASE (.NUM GTR 0) OF
00600   0006   SET
00700   0007           -INTSUM(-.NUM);
00800   0010           INTSUM(.NUM)
00900   0011   TES
01000   0012   ) ELUDOM
BIB     0004    0000            MOVEI   05,1
KIT             0001            SKIPG   04,NUM
WEV             0002            SETZ    05,PIK
BAG             0003            XCT     $S,KIN(05)
VIS             0004            JRST    $S,WEB
KIN             0005    KIN:    JRST    $S,BEW
SAP             0006            JRST    $S,TID
BEW     0007    0007    BEW:    MOVN    06,NUM
JOM             0010            PUSH    $S,6
GIJ             0011            PUSHJ   $S,INTSUM
NOK             0012            SUB     $S,MOP
JOR     0010    0013            MOVN    $V,NEJ
KAH             0014            JRST    $S,WEB
TID     0011    0015    TID:    PUSH    $S,BOH
MIF             0016            PUSHJ   $S,INTSUM
BEL             0017            SUB     $S,MOP
WEB     0012    0020    WEB:    MOVEM   $V,SUM
HET     0013    0021            CALLI   $S,12
```

Figure 9.4 - Compiler Output with Tokens

The seventh line of the figure is the instruction described above.

In order to complete the representation of the program we need to define the relations to be used in the graph. For each relation name, we need to know all of the subgraphs which are to be treated as unacceptable.

The result of this process, which is described in detail in Chapter 12, is that "precedes" is defined as a total-order relation, "contains address" and "contains opcode" are defined as single-valued so that entities may contain only one opcode or address, and "appears before" is defined as a transitive partial order relation.

This completes the definitions necessary to represent the program. The resulting graph is shown in Figure 9.5 .

Figure 9.5 represents the sequence, address and opcode aspects of the program. We shall see that for purposes of control structure attribution, the other parts of the PDP-10 instruction are redundant, although they are vital to computation.*

---

* Those parts are the index register and accumulator fields and the indirect bit. We can surely formulate descriptive tasks based on machine code for which these values would be required. However we know of no reason that a straightforward reuse of the style described below would not serve such tasks as well.

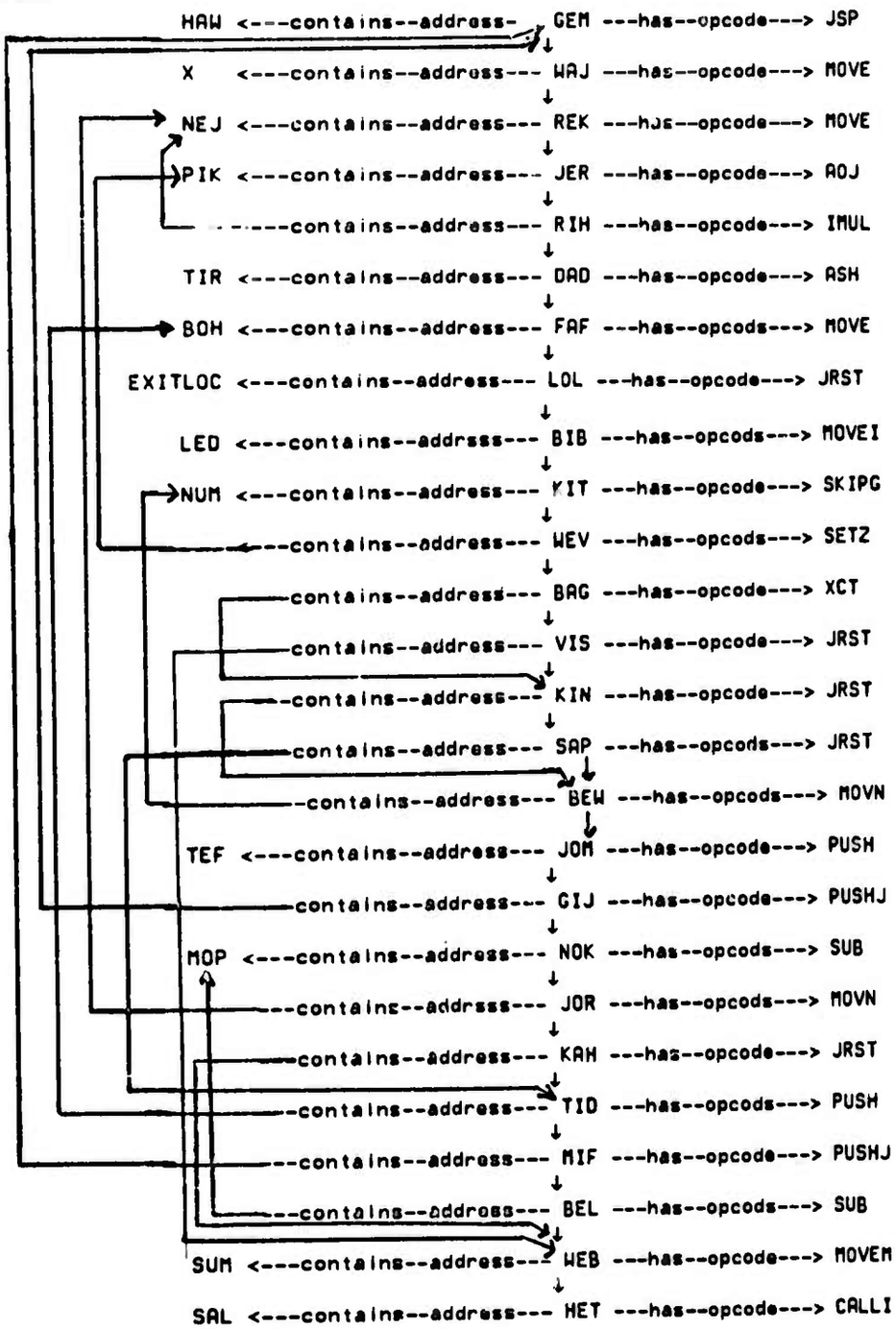


Figure 9.5 - Program Represented as a Graph.

REPRESENTATION OF THE KNOWLEDGE OF ROUTINES

This section presents the definition process for the knowledge of Bliss-produced machine code.

These assertions constitute the general knowledge of routines for our purposes.

In order to define the knowledge of the system, the following additional relations are defined.

is the value of
is the content of
is the function of
starts with
finishes with
has subpart

Table 9.1 - Relations Used in Knowledge graphs

The following constants are defined for this task and later related tasks:

CASE HEAD	CASE SECTION	VREG
DISTRIBUTOR	COLLECTOR	CASE STATEMENT
CASE EXIT	ROUTINE NAME	EXITLOC
LOOP	EXPR	TAIL
TEST	TEST SETUP	INCREMENT SETUP
E1	E2	E3
SELECTION TEST	SELECT STATEMENT	SELECTION VALUE
SETUP SELECT	CONTROL SELECT	TESN
VREG	PARAMETER SET	ROUTINE CALL
	CALL SEQUENCE	

There are configurations of instructions which occur somewhere in the machine code for every routine, routine call and parameter citation in the source program. Routines

always start with an instruction having opcode JSP, and end with an instruction having opcode JRST with one of a few particular locations in their address field. Such a location is called EXITLOC here. The JSP executes before the JRST. The routine name used by the rest of the program is the name of the JSP instruction.

The above assertions are represented by the graph in Figure 9.6 . RT1, RT2 AND RT3 are tokens.

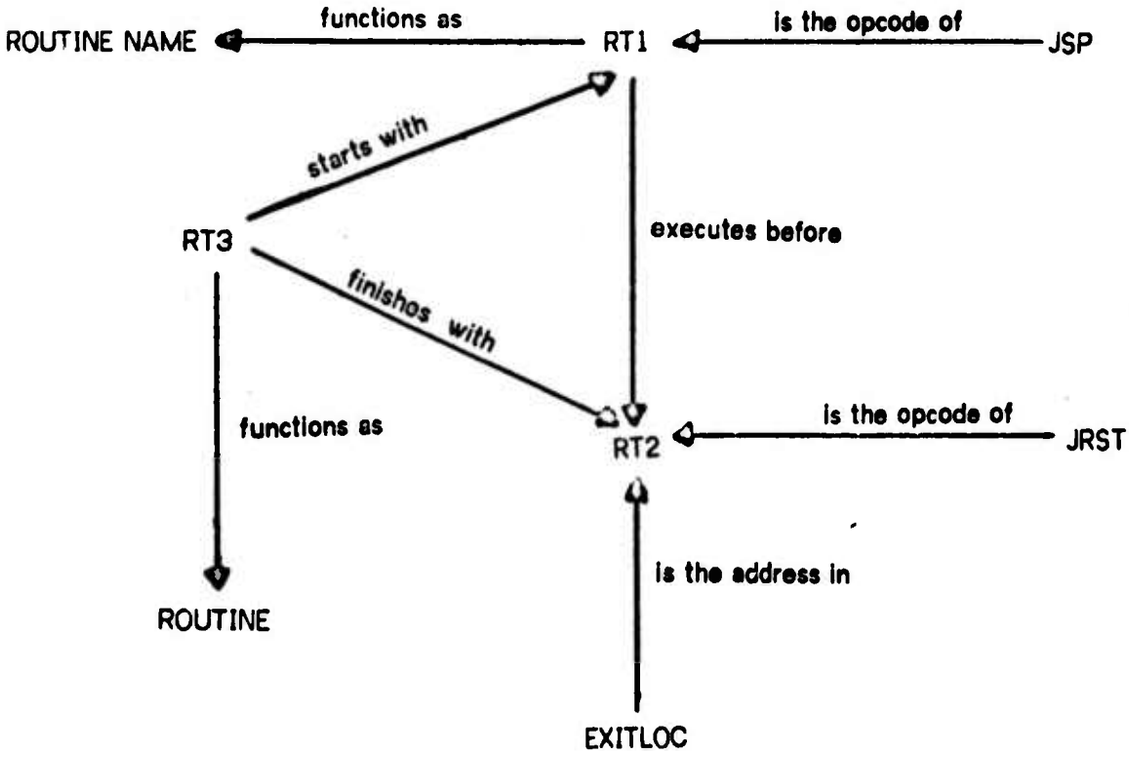


Figure 9.6 - Chunk For identifying the Code of a Routine

The set of arcs in figure 9.6 is used as one chunk in performing this assimilation task. There are several things that we should note about this chunk. First, it corrasponds to part of the graph which we have at hand (in Figure 9.5). If we were to substitute GEM for RT1 and LOL for RT2 in the chunk, then three of the resulting arcs:

JSP is the opcode of GEM
JRST is the opcode of LOL
EXITLOC is the address in LOL

would be identical to arcs of the input graph.

Second, the arcs of the chunk which result from that particular substitution could be added to the input graph without creating any conflict. This means that the knowledge of a Routine and the input assertions about GEM and LOL are fully consistent.

Third, there is no instruction other than GEM in the input graph for which such a non-contradictory substitution could be made for RT1. Therefore, for GEM, the opcode must be JSP. For any other instruction, say WAJ, the substitution would result in a subgraph such as Figure 9.7 .

Figure 9.7 - Opcode Conflict

This subgraph is defined to be contradictory, violating the definitional constraints on "is the opcode of," which make it single-valued.

There is a third token in the chunk, RT3, which represents the entire routine. There is no token in the program graph (Figure 9.5) which corresponds to it. In order to have arcs in an extension of the input graph which correspond to these arcs:

RT3 starts with RT1
RT3 finishes with RT2
RT3 functions as ROUTINE,

a new entity must be posited in the input graph. Assertions about this entity can be used in further extending the knowledge of the input.

USING THE KNOWLEDGE OF THE INPUT

Figure 9.8 shows the result of mapping the chunk into the input graph.

By adding the vertex representing the routine to the graph, we are abstracting on the input. The Slate system treats the new abstract entity in the same way that the input graph is treated, so that the process of abstraction is not limited to the input graph per se.

The mapping of this chunk onto the input graph adds five new arcs to the graph. Two of them are assertions about the functions of parts, two are declaring the substructure of the routine, and one asserts an order relation about the execution paths through the routine. Of these three kinds, the ones which declare functions are the most useful for further abstraction.

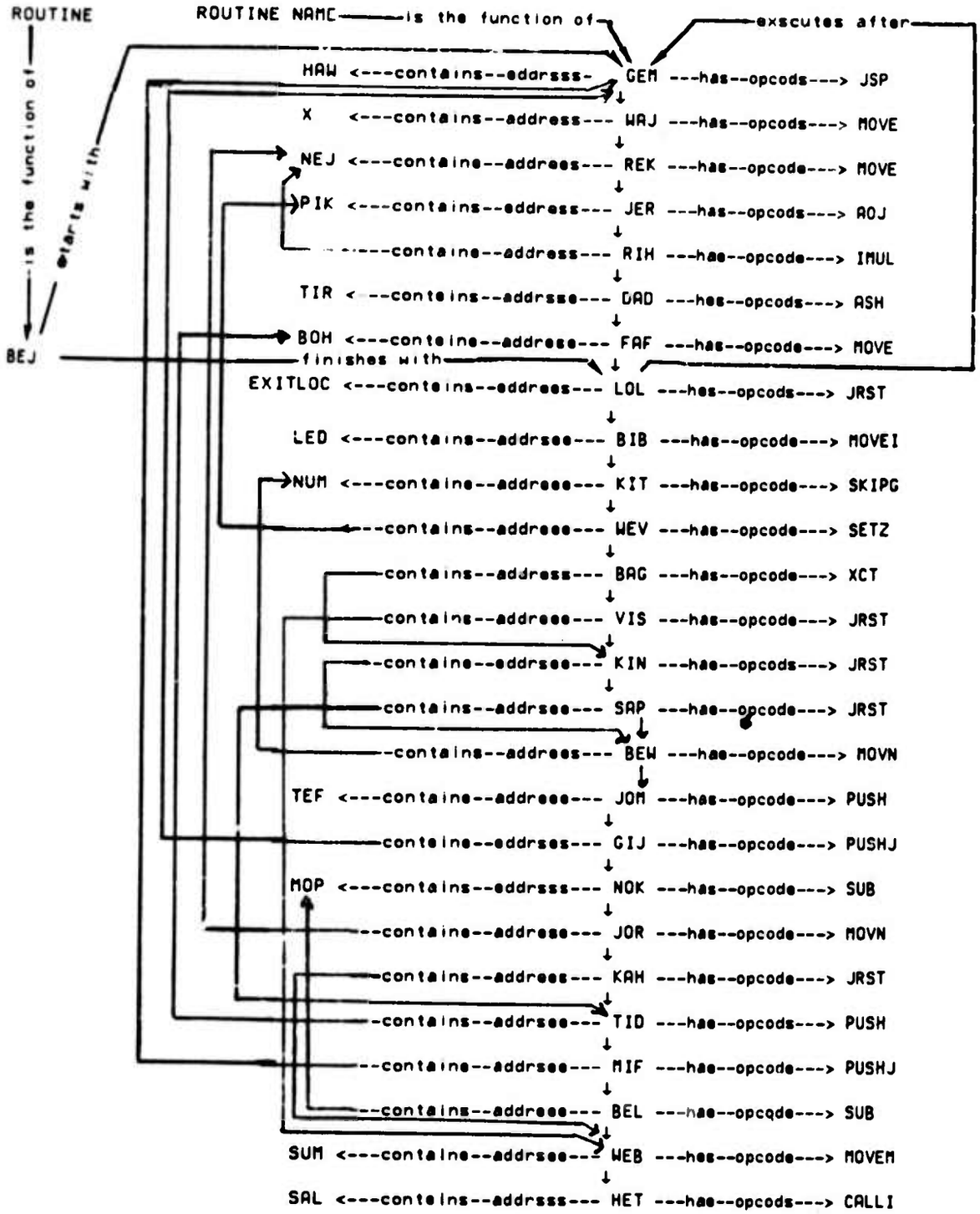


Figure 9.8 - The State After Mapping One Chunk

There are three other chunks used to represent the knowledge of routines and calls on routines. They are shown in figures 9.9, 9.10, and 9.11 .

When the Slate system uses these three chunks on the input graph, they map in in the order·

ROUTINE CODE
ROUTINE CALL
PARAMETER SET

although the chunks are not ordered in storage. This particular order arises from the fact that each of these chunks builds on the assertions deposited by the previous one. The chunk for routine code deposits an arc

GEM -------functions as------->ROUTINE NAME

The chunk for routine calls maps arc

RC2 ----------functions as-------->ROUTINE NAME

onto this arc. It in turn deposits an arc

GIJ ----------functions as-------> CALL

The PARAMETER SET chunk maps arc

PC2 ----------functions as----->CALL

onto this arc. Each of these deposited arcs is treated as evidence for mapping in the succeeding chunk. In this particular sequence, the evidence is decisive at each step. By this we mean that if the arc in question is removed, the Slate system will find no further mappings which qualify according to its rules of evidence.

The graph which results after these three chunks map in is shown in Figure 9.12 . At the point shown in the figure, the ROUTINE CODE chunk has mapped in once, the

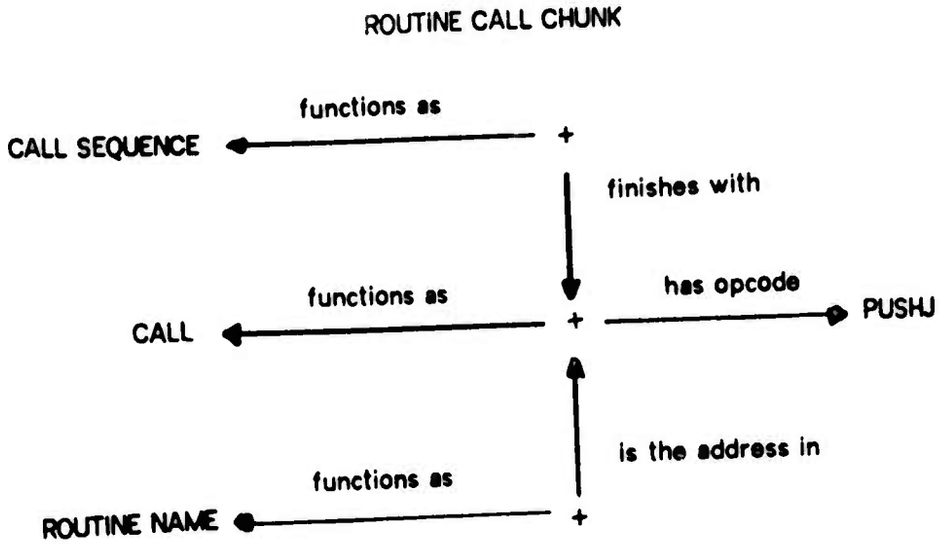


Figure 9.9 - Chunk for Routine Call

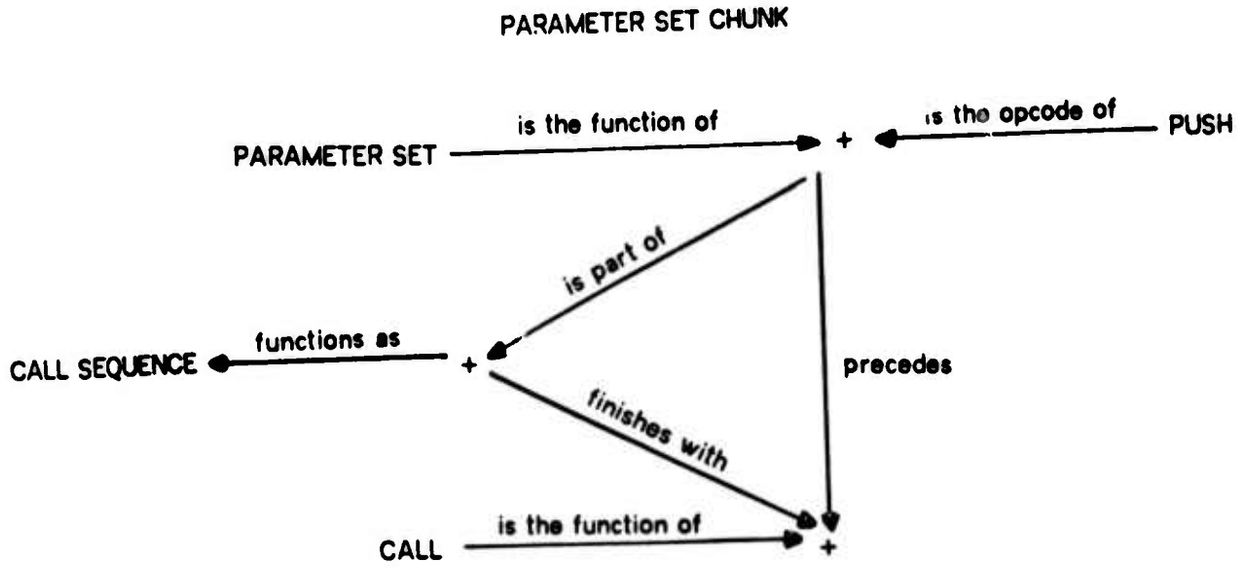


Figure 9.10 - Chunk for (Optional) Parameter Group of Routine

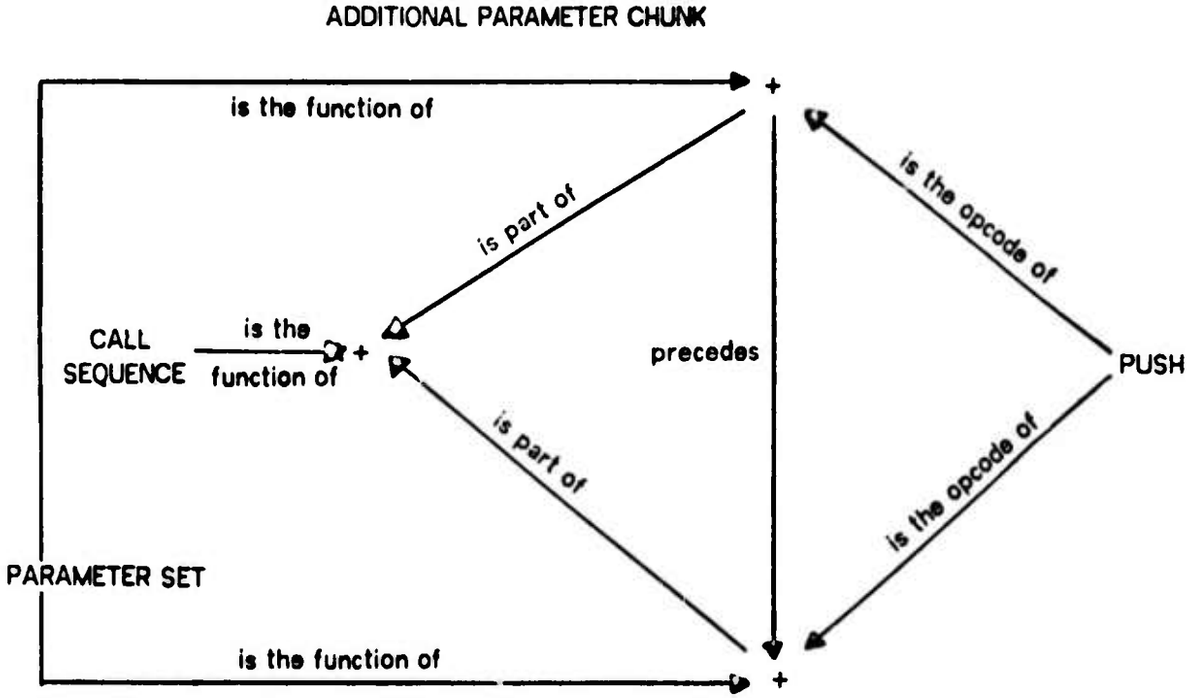


Figure 9.11 - Chunk for (Optional) Multiple-Parameter Calls

ROUTINE CALL and PARAMETER SET chunks twice each (once each on each of the two places where routine calls occur.)

We note that all of the uses of routines and routine parameters and the boundaries of the routine have all been correctly identified.

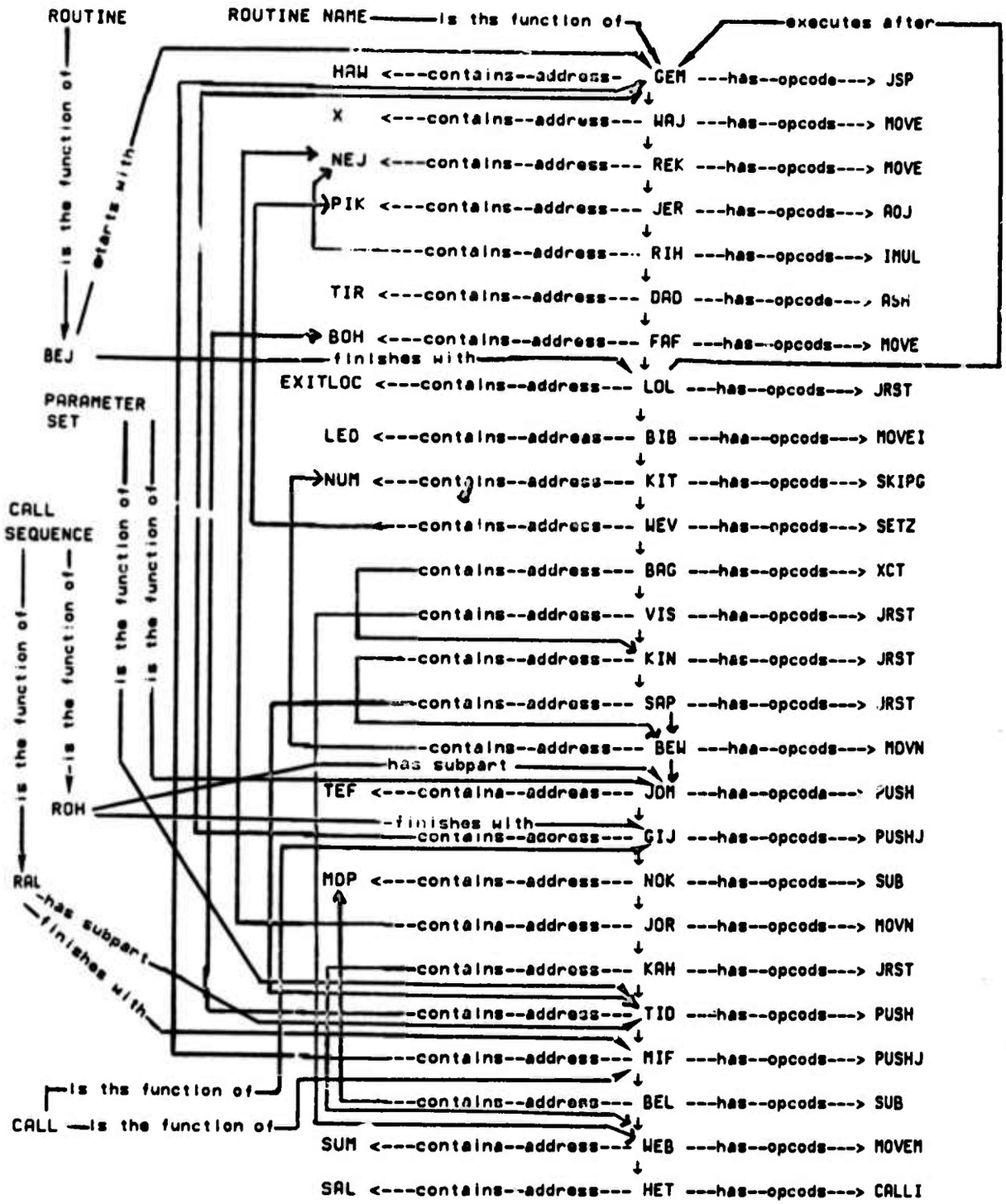


Figure 9.12 - Program Graph After Mapping the Chunks for Routines

REPRESENTATION OF THE KNOWLEDGE OF CASE STATEMENTS

We now go on (more briefly) to treat the knowledge of case statements and the use of that knowledge in this example. The style is very similar to that used for routines. There are three chunks used to represent the required knowledge. They are shown in Figures 9.13 to 9.15 .

The syntax of a case statement allows any number of sections. We expect the chunk for case head (for the beginning of the statement) to map in once, and the chunk for case section to map in once for each section actually present. The case statement chunk also maps in several times, once for each section which the statement includes.

The new portion of the result graph, after the case head chunk maps in once and the other two each map in twice, is shown in Figure 9.16 .*

---

* The graph shown in Figure 9.16 is the result of a hand-simulated completion which was performed after computer runs were complete. The method presented so far is defective in that the last case section, which does not end with a JRST instruction but rather with an instruction which directly precedes the case exit, cannot be recognized by the CASE SECTION chunk above, which requires the JRST. We fix this by adding a FINAL CASE SECTION chunk, which can perform the correct recognition of such sections. This chunk must match the last distributor, which is the only distributor directly preceding a section beginning. Construction of the chunk is straightforward.

The notion of the "appears before" relation in these chunks is that it is the transitive closure of "precedes," and we could have defined it so by an interaction definition. However it was much more efficient to prestore it only where it was needed instead.

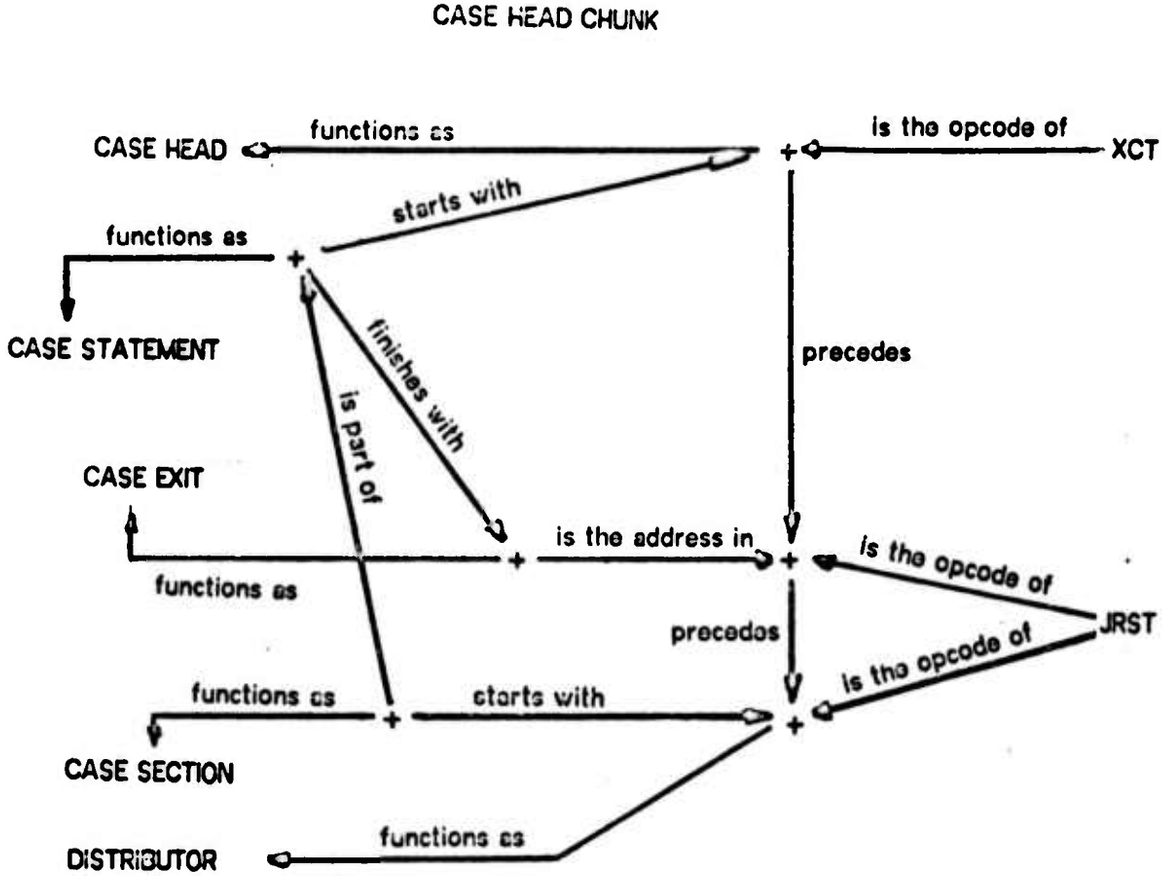


Figure 9.13 - Case Head Chunk

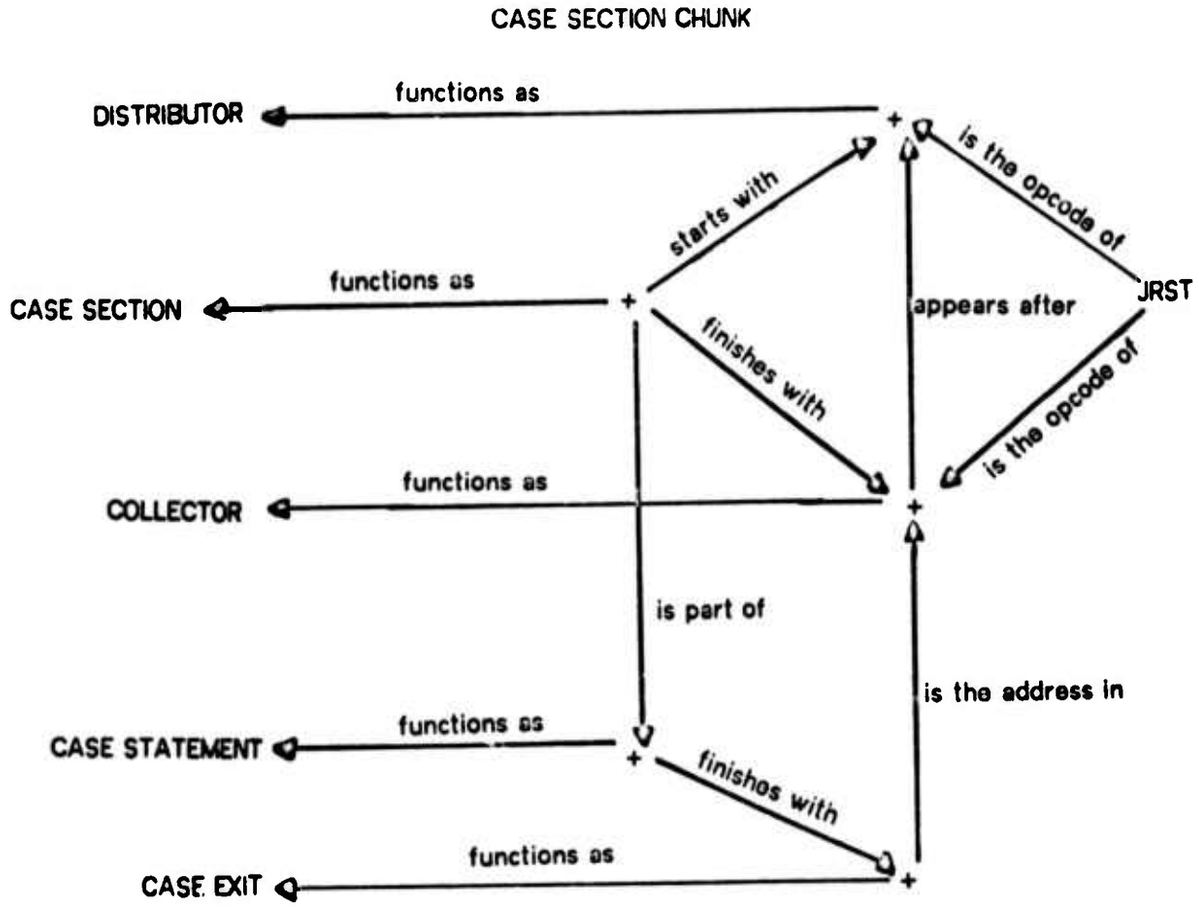


Figure 9.14 - Case Section Chunk

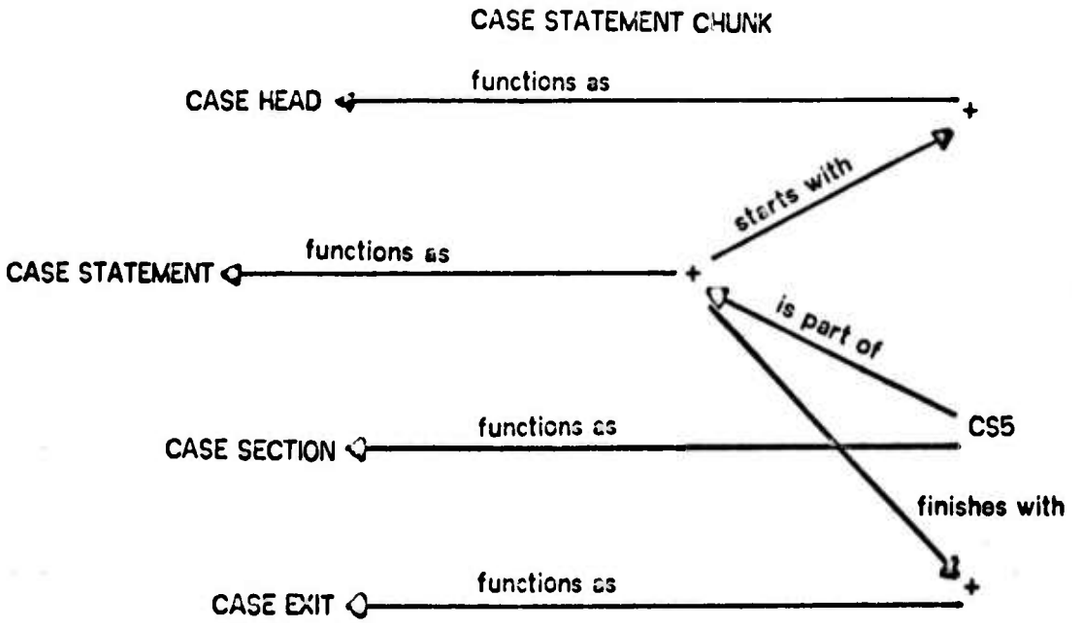


Figure 9.15 - Chunk for Gross Structure of a Case Statement

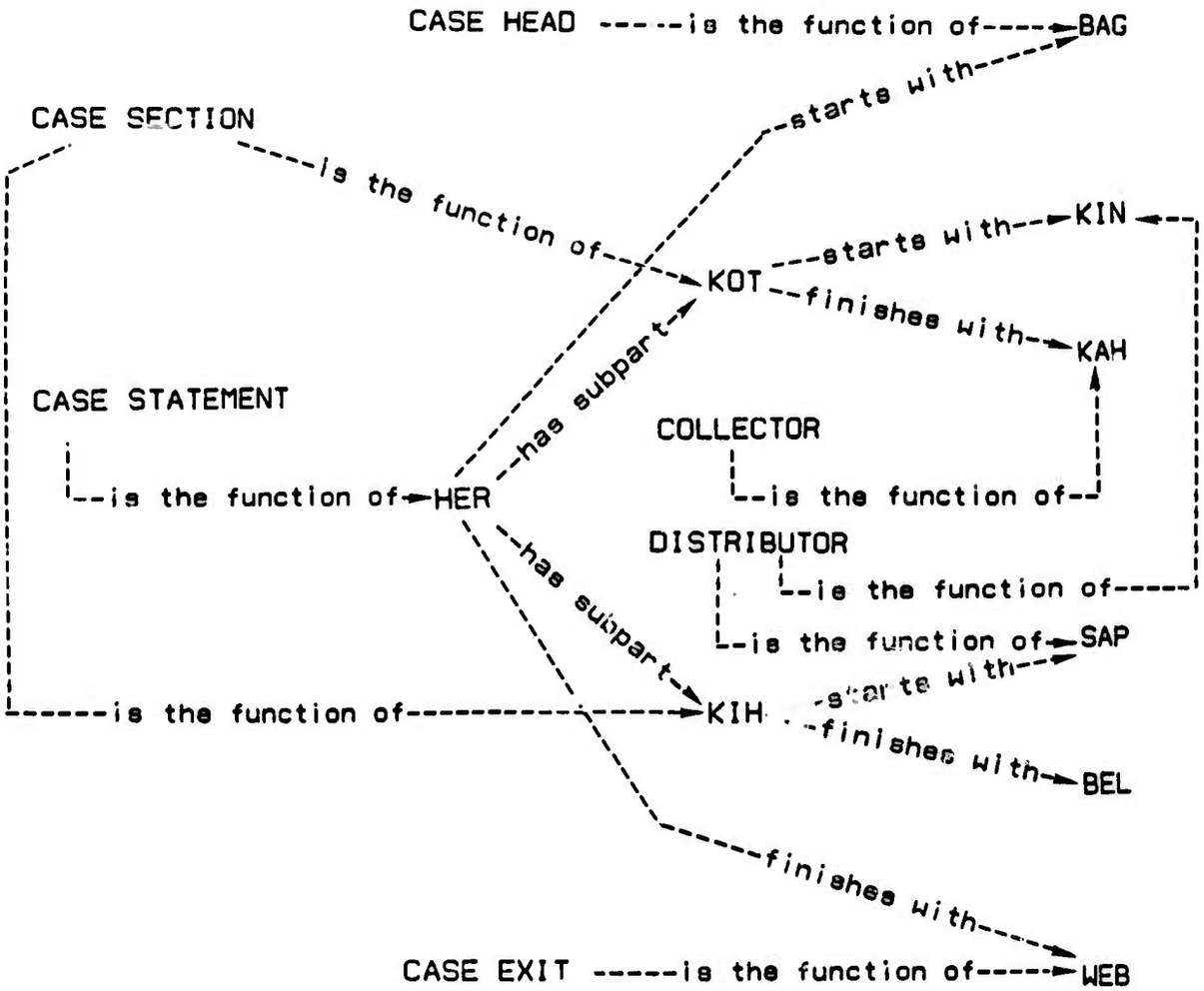


Figure 9.16 - Final Additions to Slate Content

We can now judge figures 9.12 and 9.16 above, which together display the problem solution, against the initial criteria.

1. All control structures present have been identified correctly by means of new vertices representing the structures as a whole, and by arcs on the relation "functions as" indicating the particular structure identified.

2. The scopes of all control structures are correctly identified by arcs on relations "starts with" and "finishes with."

3. Mutual reference or routine calls is indicated by correctly identifying the call sites in arcs of the form (XX functions as ROUTINE CALL), where XX is any vertex.

4. Control structures not found in the source program were not attributed to the machine code.

5. Substructure of all control structures is correctly indicated by arcs on relation "functions as."

6. Since the solution is a graph which includes the input graph, the added arcs and input graph are self consistent and mutually consistent.

In summary, the goals for this example have been completely achieved.

OTHER PROGRAMS

The integer [illegible] program is one of about a dozen programs which have been used in the control-structure discovery task. Some small programs were used to develop the ability to deal with single control structures. Several other composite examples were used as well, including the one in Figure 9.17 below.

This program has 54 instructions when compiled, and yields a graph having 168 arcs before chunking and about 250 arcs after chunking.

The fact that the presence of some of these chunks does not interfere with the use of others was established both by the composite examples and by the general practise of having all of the chunks for the task in place whenever any of the chunks were being

```
MODULE MYTIME(MLIST)=
BEGIN
GLOBAL A,B,C,D,E,F,G;
LOCAL M,N,P,Q;
ROUTINE GF(T,U)=@T2*.U;
        INCR X FROM .A TO .B BY .C DO
BEGIN
        SELECT .D OF
NSET
        .M:     INCR Y FROM 2 TO 7 DO
                M  .M - .E;
        .P:     CASE .F OF
SET
                GF(D,4);
                Q.P;
                P.Q
TES
TESN
END
END
ELUDOM
```

Figure 9.17 - Another Bliss Program

tested. (The same practise was followed on all tasks, with the same result.)

One very large program was attempted. It was designed to have enough control structure so that every chunk would map into the Slate at least twice. Some therefore necessariiy had to map in oftener. The effort was abandoned because of its cost in computer resources. However, examination of partial states of compietion indicated that progress was satisfactory and that no qualitatively new problems were affecting the course of events.

INTERPRETING THE RESULT GRAPH

In finishing the exampie of the integer sum program, we have left to the reader one vital step in the soiution of the original probiem. We have not transformed the result graph into any other notation before evaiuating it. Rather we have found the assertions of function and inclusion of the result to be correct "by inspection."

This direct interpretation has been possible because the arcs can be read like sentences. The language of reiationai triples is not English, but it is like English in important ways. In this oxampie we can see that we can read the result graph as a set of sentences, with the tacit understanding that the token names are like proper names or place hoiders. For example, the arc JSP is the opcode of GEM is a relationai triple which resulted from the mapping of the chunk for identifying routine code above. Reading it as a sentence, we understand that GEM is the name of a particular instruction, and that JSP is the unique opcode of that instruction. The uniqueness of the opcode

value is suggested by "the" in the relation name, and is enforced by the relation definition for the relation name "is the opcode of."

The results of such reading are not deceiving. They accurately represent the results which the system has derived. The use of the right relation definitions has allowed the system to keep its triples in correspondence of meaning with the English sentences which they form.

In developing the Slate system, the fact that the sets of triples derived could be read off easily in English was a strong aid in understanding what was going on. The triples-language acts as a convenient meeting ground between the formal operations of the program and my own informal operations.

Achieving a correspondence between triples and sentences clearly depends strongly on being able to define relations having the right properties. The system of inferences and constraints must enforce the meaning imputed by the name. Each relation name is a phrase which must fit properly in infix position. The class of relations which we used is adequate for a wide variety of tasks, as the remainder of the thesis illustrates. Sometimes the requirement that every relation have a name for both the relation and its converse has led to an unwieldy converse name, but at least one convenient name for a relation has always been easy to find.

Watt has discussed a common problem which he calls the "habitability" problem, which arises with computational systems which accept natural language input from people.[W68] Since the present art cannot deal with unrestricted natural language input,

each system accepts only its preferred subset of the language. The principles of subsetting which are used turn out to be extremely difficult for people to acquire and use. Thus the subsets are not "habitable" for people.

We have finessed the habitability problem for the Slate system graphs, because people understand them as a formalism, but with sentential semantics. Since each arc can be read as a sentence, and since the constraints on joint occurrence of arcs correspond to the expectations for joint truth of the corresponding sentences, people have little difficulty in understanding or remaining within the formalism. The REL family of languages takes major advantage of this correspondence. [BGST73]

The difficulties arise from inability to express some ideas in the formalism; these are recognized and so do not lead to improper system input when graphs are being built.

COMPLETING CONTROL STRUCTURE DESCRIPTIONS

CHAPTER 10-1

PROBLEM PRESENTATION

This section presents another task defined on the same subject matter, program structure. Its purpose is to demonstrate an aspect of the diversity of kinds of problems which the Slate system can handle. In the previous chapter, chunks were applied to detailed knowledge of code in order to impute knowledge of source-language constructs. In this chapter we do the opposite, applying the same chunks to knowledge of source-language constructs in order to impute knowledge of machine code, according to the criteria stated below. We want to show that the same chunks work either way.

It often happens that information to be assimilated includes some abstract knowledge as well as (or instead of) elementary detail, while the problem-solver requires both kinds of knowledge. If the structure of the assimilator is such that it simply recognizes structure in detail, then the abstract knowledge cannot be assimilated, even though the relevant knowledge is in memory. For example, discrimination-net representation of knowledge involves a commitment to only one of these two kinds of imputation. (See chapters 6 and 18.)

The problems in this chapter are problems of assimilating abstract program control structure descriptions. They are synthetic demonstrations, in that we do not identify them with the context in which this problem arises.

EXTERNAL PROBLEM REPRESENTATION: English language statements describe a particular program by naming its control structures.

INTERNAL PROBLEM REPRESENTATION: A graph contains vertices which correspond to the control structures named, their subparts if they are specified, and the relations between them.

SOLUTION REPRESENTATION: The problem graph is augmented with vertices and arcs which describe the additional control structures which necessarily accompany those mentioned and the machine code which must be present to implement the control structures.

CRITERIA FOR JUDGING SOLUTION:

1. Any machine code known to be necessary to the given control structures must be attributed to them.

2. The beginning and end of the scope of each control structure must be correctly identified.

3. Any control structure not required by the given statements must not be posited.

4. All of the assertions of the solution must be mutually consistent and consistent with the assertions of the problem representation.

The first example is that of a select statement having three sections. The external representation of the given information might be: "A certain Bliss select statement has three select sections." We present the Slate system with the ten-arc graph in Figure 10.1 .

The knowledge of program structures is all the same knowledge that was used in the previous task. The same chunks are used to represent it.

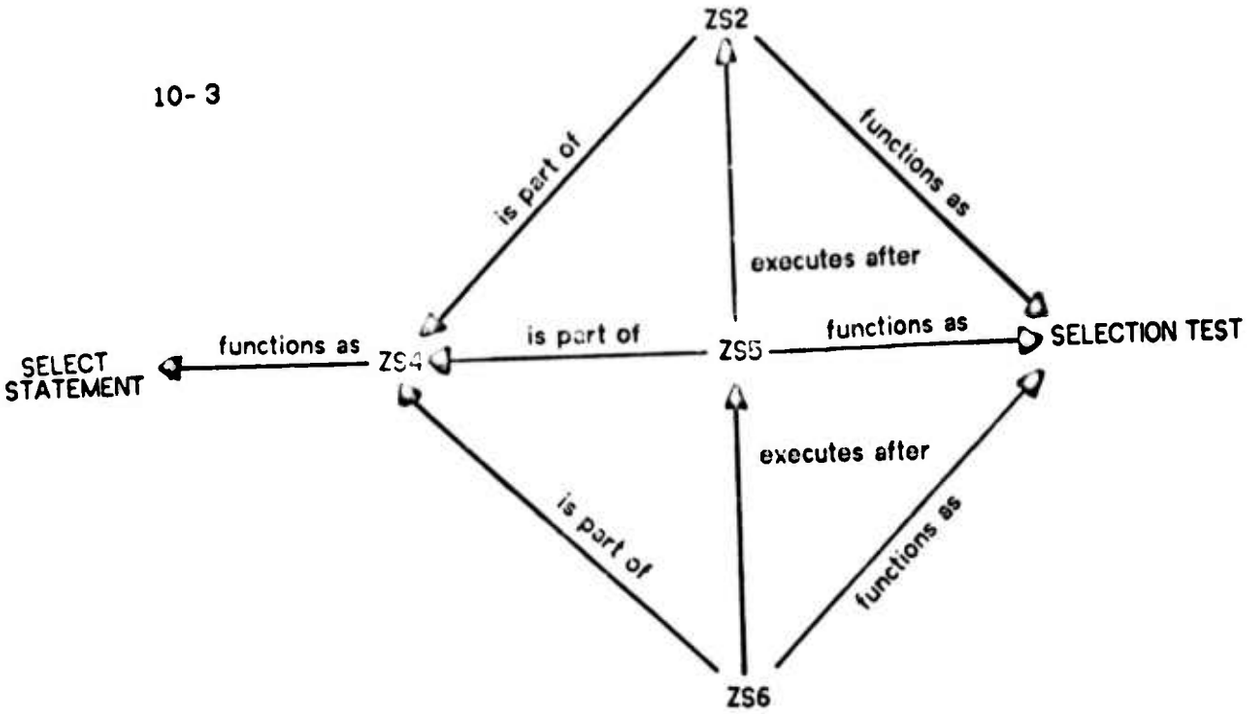


Figure 10.1 - Three Section Select Statement

One goal of this example is to demonstrate that the knowledge itself is not task-specific. Rather it can be used in a variety of tasks which involve the subject of the knowledge.

Another goal of the example is to demonstrate that the knowledge interpretation scheme which we are using has no inherent commitment to abstraction, completion or elaboration. It operates effectively on any of these, and can be expected to handle mixtures as well.

PROBLEM RESULT GRAPH

The mapping sequence which occurs is as follows: The sequence of mapped chunks is: three mappings of the SELECT STATEMENT chunk, the TESN chunk, the SETUP SELECT chunk and three mappings of the SELECT SECTION chunk. The final state at which no more acceptable mappings can be found is shown in Figure 10.2 .*

It is interesting to note what is missing from the graph. The knowledge of the order of the sections was provided, and has been completed. However, there have been no interactions between "executes after" and "precedes," so the result has less explicit sequence information than it could.

The knowledge that was available has been fully applied by the system. There are no further assertions which are resident in the chunks that could be applied to the final graph. The missing parts of the statement (head section and exit section) have been supplied, and the two select sections that were mentioned in the input have been fully elaborated in the result graph.

We can compare the result to our original criteria:

1. All machine code known to be necessary has been properly attributed.

---

* For simplicity, the complete transitive closure of the "executes after" relation has not been shown, although it is maintained.

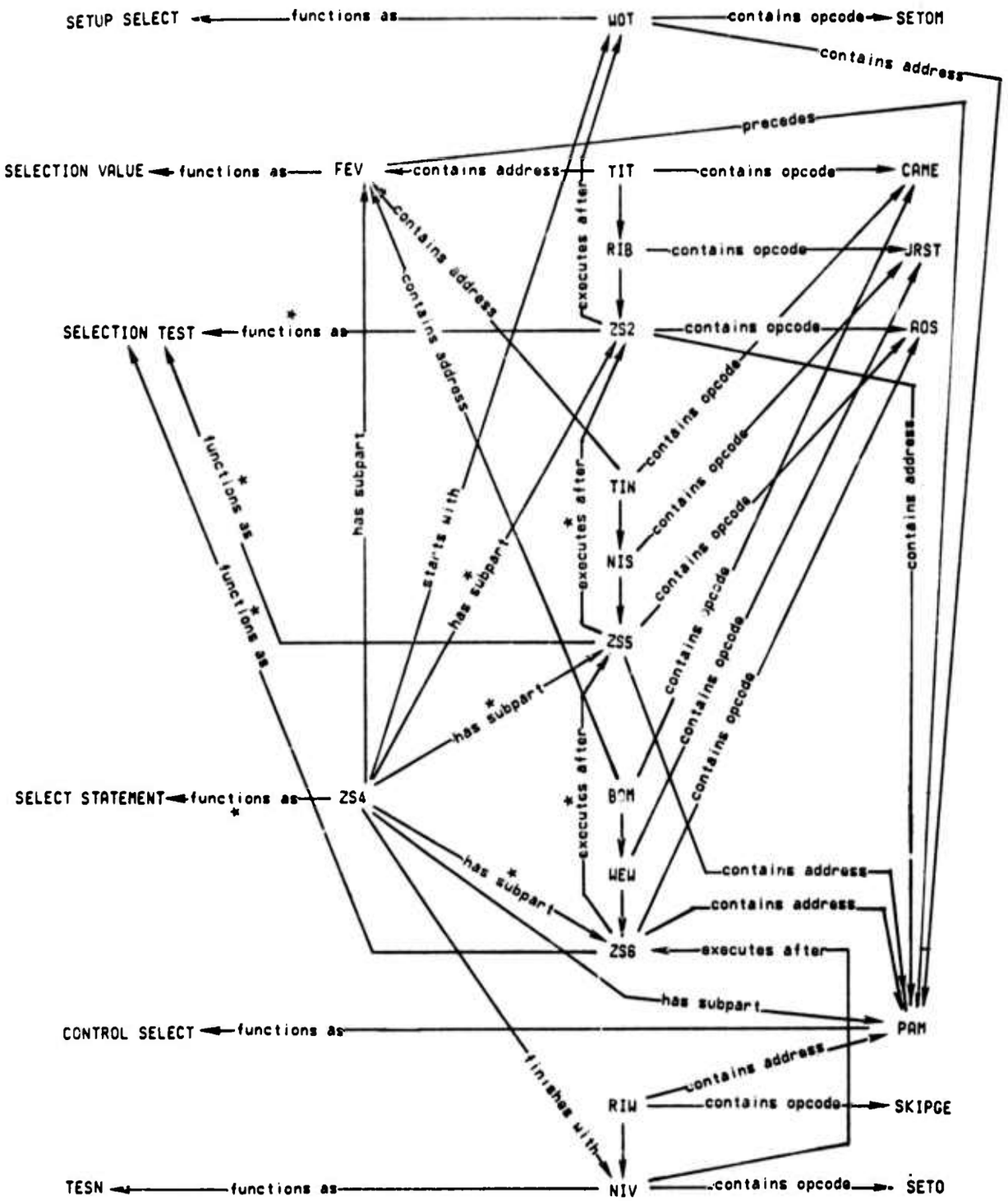


Figure 10.2 - Result of Explicating a Three-section Select Statement

2. The scopes of the given control structures in the machine code have been correctly delimited by "starts with" and "finishes with" arcs.

3. No control structures not present have been attributed.

4. All assertions are consistent as before.

In summary, the system has been completely successful in elaborating the input graph on the basis of its knowledge.

The sections below present solutions of additional examples of the same kind of problem.

EXPLICATION OF A CASE STATEMENT

This example is analogous to the previous one, except using chunks for the Bliss case statement. The external reprosentation of the problem is: "A certain Bliss select statement has three sections." The graph given to the system is shown in Figure 10.3. CASE STATEMENT maps in twice, then CASE HEAD maps in once, and finally CASE SECTION maps in twice.

The result is a large graph which is qualitatively comparable to that of Figure 10.2 . In evaluating with the four criteria, we have the same completely successful result as on the previous example. Since section order and adjacency were not given or inherent in the chunks, they are unspecified in the result.

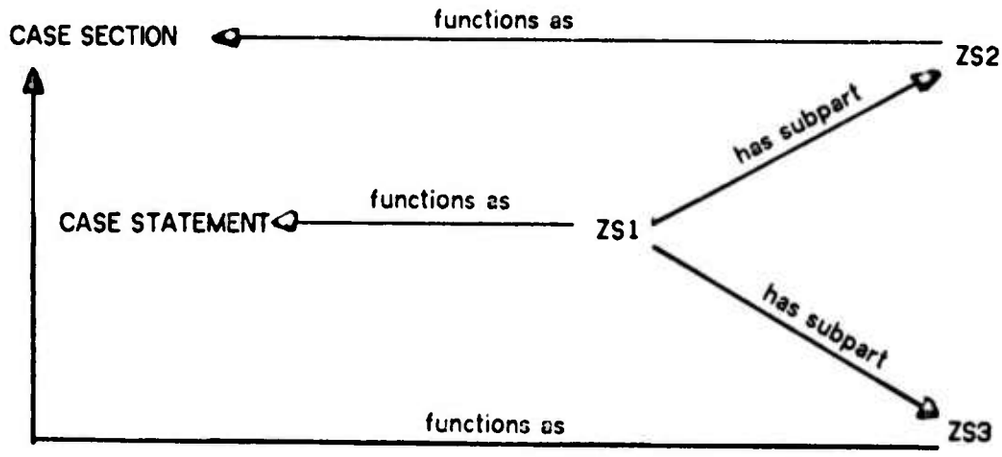


Figure 10.3 - Input Graph Describing a Two-section Case Statement

EXPLICATION OF THE INTEGER SUM PROGRAM

This example corresponds to the first assimilation problem of chapter 9. The external statement is "A certain Bliss program contains a case statement which has two case sections and a routine which is called by two one-parameter calls." The corresponding internal statement is the graph shown in Figure 10.4 . Note that the graph is not connected.

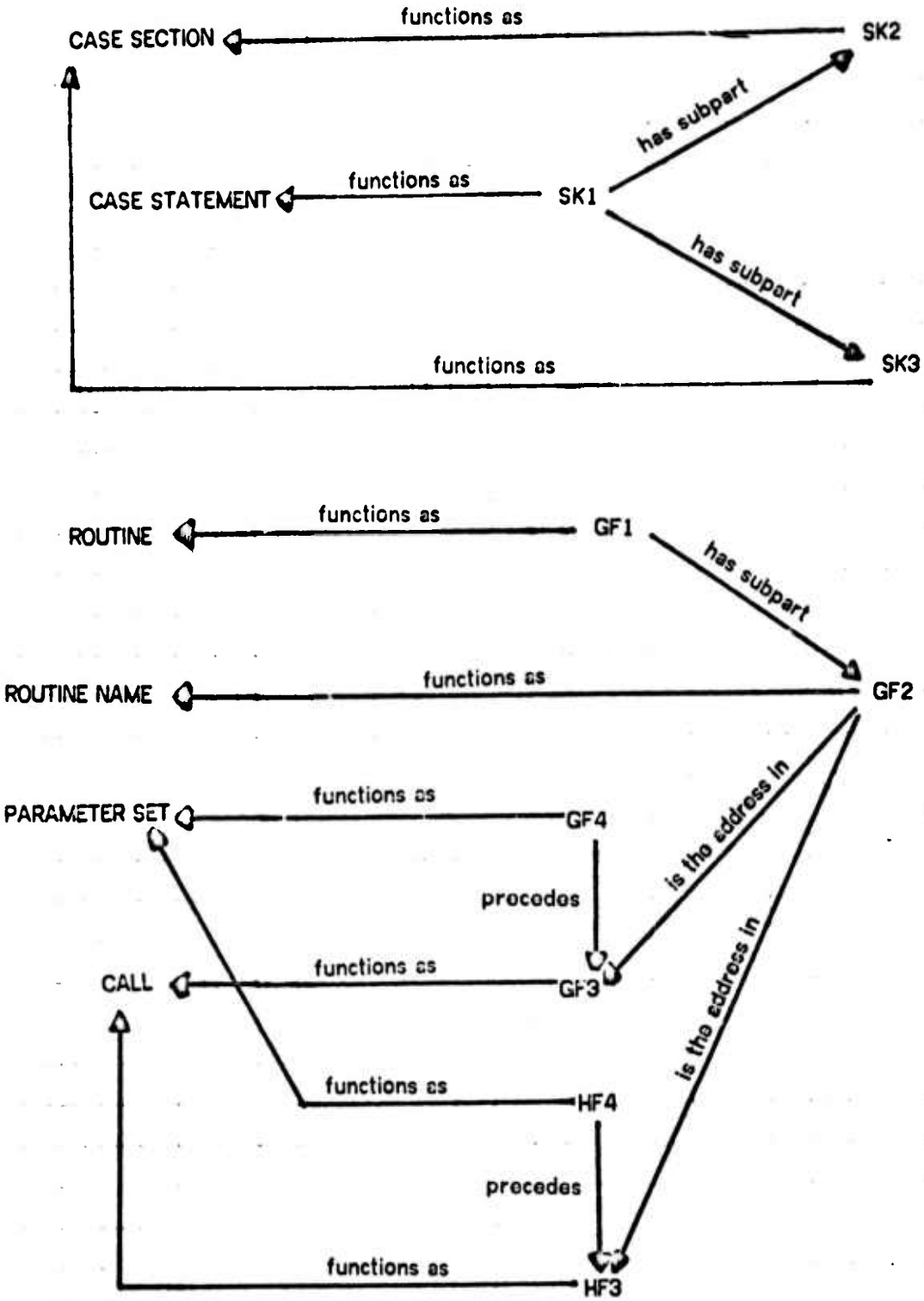


Figure 10.4 - Integer Sum Program Control Description

There are several problems which arise with this graph, which exhibit some weaknesses of the methods used in some cases, and faults in the particular relation and chunk definitions used in others. If we apply the REGARD2 operator as in the other examples, the PARAMETER SET chunk maps twice, which is proper, and then the ADDITIONAL PARAMETERS chunk maps an unlimited number of times. On a graph of machine code, the ADDITIONAL PARAMETERS chunk will map a limited number of times provided that instructions bearing opcodes other than PUSH precede each routine call, which is the usual case. Here, the sequence of instructions is absent, and there is formally sufficient evidence for the mapping of ADDITIONAL PARAMETERS, in which one prior instruction is invented per cycle.* A solution to this problem is described in chapter 17.

Deleting chunk ADDITIONAL PARAMETERS, the experiment runs to a stable, almost correct result. The chunking sequence consists of 10 mappings: the PARAMETER SET chunk maps twice, the CALL chunk twice, the CASE STATEMENT chunk twice (once for each section incorporated), the CASE HEAD chunk once, the CASE SECTION chunk twice, and the ROUTINE chunk once. This is exactly the right set of mappings, and the order does not violate information dependencies. The process adds 12 new descriptive constants to the State, 8 new tokens, and over 40 arcs.

---

* About 10 other examples of various sizes have been run, including some which exhibit the same effect.

All of these correspond directly to the actual source and machine code, with two exceptions:

1. Because of the shortcut taken with respect to the interaction between "appears before" and "precedes," a contradictory pair of arcs got into the Slate. This is directly preventable by use of a correct relation interaction definition.

2. On the last chunking (for ROUTINE,) the system found an instruction in the Case Section portion which could be identified as the final JRST. It used it rather than creating a new token, and the graph became connected. This sort of effect has been prevented in most cases by defining the fucntions of parts in such a way that one part can not have two independant functional descriptions. Providing such definitions (using "functions as") in the CASE SECTION and ROUTINE chunks would prevent this problem. The problem occurs only in explication; in recognition, on complete instructions, a conflict of address contents would have prevented the effect.

The system has not developed the knowledge of the nesting of subparts in the Slate, and it appears that making it do so would considerably complicate the task.

Also, it is not dealing at a conceptual level which permits easy generalization to cover variations which preserve program equivalence, either in explication or in recognition. The variations introduced by local optimization create massive difficulties because the regularities which they depend on include control flow equivalences, which are not represented in our graphs at all.

Of the 26 instructions of the program, 10 have been provided by this explication of its control structure.

EXPLICATION OF LOOPS

The graph in Figure 10.5 represents the least specific representation of a loop which the system will treat as non-trivial.

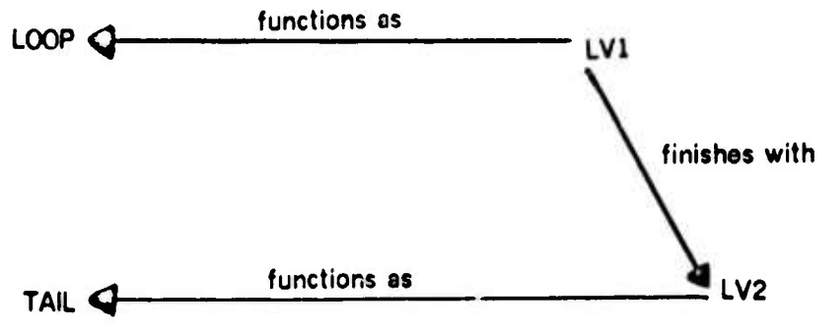


Figure 10.5 - Minimal Loop Specification

Unlike the case statement and select statement examples, the result for this one is not uniquely determined in our system.*

Compiled instruction sequences for loop expressions can vary in several ways. Incrementing by 1 is a special case which utilizes PDP-10 opcodes which alter numbers by 1. Constants (known at compilation time) are treated differently than variable expressions, resulting in use of different opcodes in many cases.

---

* In a more complete representation of the language the others would also have alternatives.

We have therefore provided a variety of chunks for representing fragments of loop constructs. When no basis for preferring one construct over another is in the Slate, then loops may be explicated in a variety of ways. For the graph in Figure 10.5, it assumed in effect that for a source language statement:

"INCR X FROM E1 TO E2 BY E3"

E1, E2 and E3 were all expressions computed at run time. The result was successful by the previous criteria.

To show that other results are equally feasible, we deleted the three chunks that were used from bulk memory and ran the experiment again.

This time a successful completion was reached by mapping two other chunks. The system in effect assumed that E3 was the constant 1, E2 a computed expression and E1 unknown. Because the order of accession of chunks is not controlled in the Slate system, any kind of minor change could yield the opposite order of results.

SUMMARY

We have established in this chapter that the Slate system is capable of explication as well as recognition, and that its knowledge representation does not carry a committment to only one of these uses. The same chunks may be useful for both kinds of operations. In the case of incompletely specified explication tasks, the system develops an arbitrarily chosen alternative.

SEEING THE NECKER CUBE

CHAPTER 11 - 1

To facilitate judging the generality and flexibility of our methods, we investigate an assimilation task which differs in important ways from all of the previous tasks. In 1832, Louis Albert Necker reported that that edge drawings of certain crystals can be seen in two different ways, depending on the point of attention. Figure 2.1 illustrates one version of the phenomenon in which, according to Necker's description, visual fixation on corner A will cause it to be seen as nearer than B, and fixation on B will cause it to be seen as nearer than A.

The problem which we deal with is as follows:

> Given assertions of the vertical connections, horizontal connections and other connections of the corners of the cube drawing of Figure 2.1 shown above, and the identities of the two rectangular faces and the corners which they include, (but no depth assertions,) and having a knowledge of cube drawings which includes depth assertions, assimilate the givens so that they are augmented by additional assertions attributing depth to the givens. We require that the assertions made in assimilation be consistent with Necker's hypothesis, extended so that attention to any corner causes it to be seen (asserted) as part of the closer face.*

This problem differs from the previous ones in several respects:

1. The topology of part-to-part connections of the cube is unlike the topologies of previous tasks.

2. The task involves geometric constraints.

---

* We should point out that this is not a psychologically serious attempt to represent reversing figure phenomena or the Necker cube phenomenon in particular. [A71] As such, it would clearly be inadequate although it does establish the feasibility of explicit models built along the lines of the Slate system. This task is an exploration of the methods and a demonstration of particular qualities of the Slate system memories.

3. There are multiple symmetries in the given information and in the knowledge of the cube.

4. The attention or focus of the subject is an explicit part of the problem.

Each of these differences turns out to show features of the Slate's methods which do not appear on the other tasks.

One way to represent the cube drawing is shown in Figure 11.1, where by convention an undirected arc in the diagram represents a pair of oppositely directed arcs on the same relation.* The previously used relations are defined as in the other tasks. The relation "connects to" is defined to be symmetric but otherwise unconstrained. Relation "is the side of" analogous to "is the type of". There are two relation interactions defined:

AA has subpart BB
and
AA is on side CC
yield
BB is on side CC

AA has subpart BB
and
BB is on side CC
yield
AA is on side CC

These cause the faces to be asserted to be on the same side as the corners which they

---

* For clarity on all of the succeeding cube figures, eight arcs of the form <token> is of type CORNER have not been shown, even though they were present in processing.

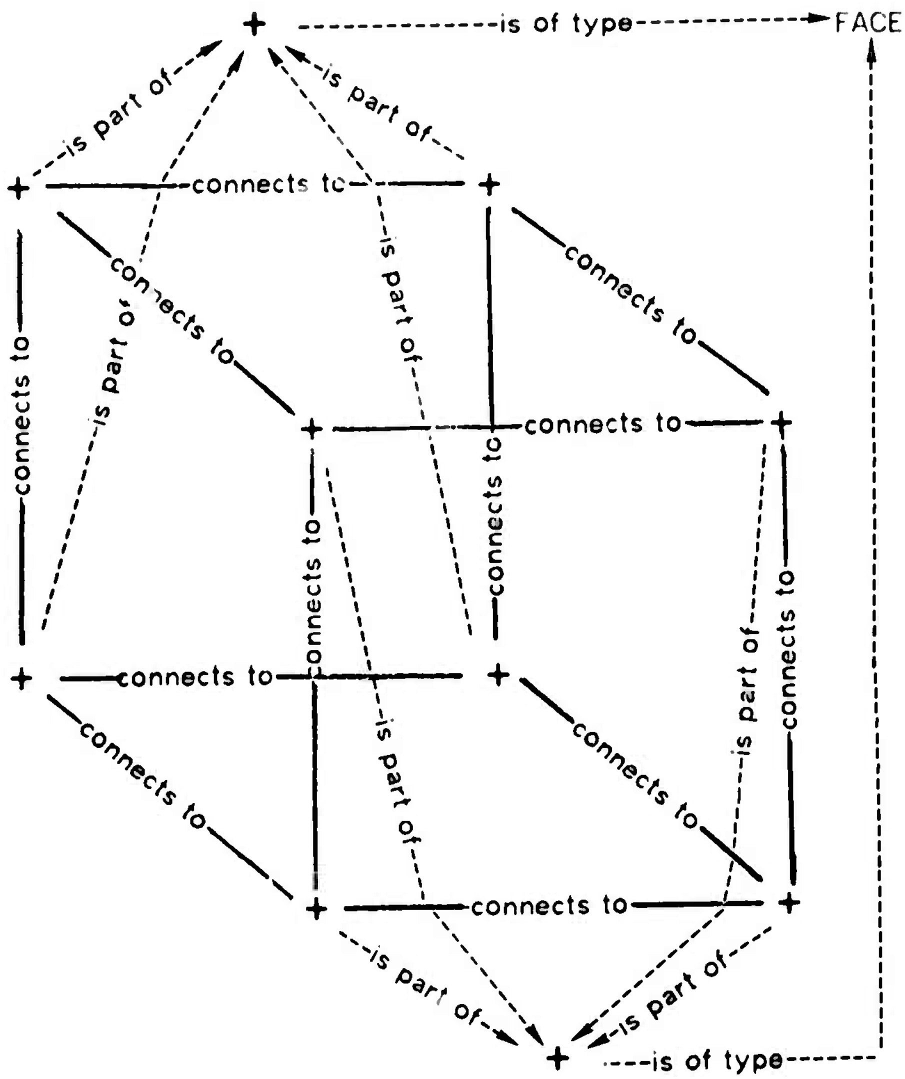


Figure 11.1 - Graph Representing Necker Cube

include, and cause all of the corners of a face to be asserted on the same side.

As an interpretation chunk for this graph consider the graph in Figure 11.2.

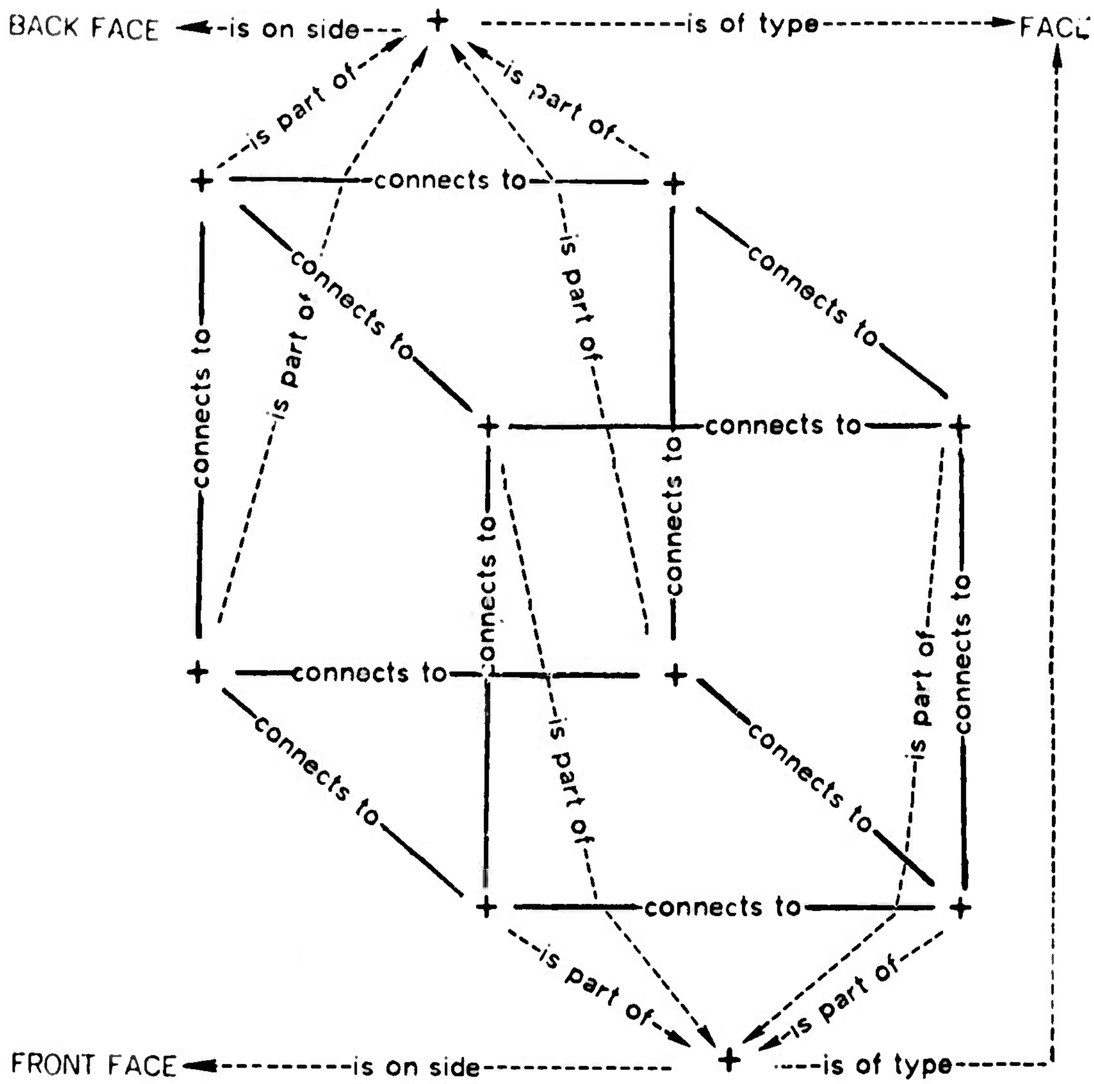


Figure 11.2 - Chunk Representing Necker Cube

The chunk can be mapped so that either face of the graph is identified as the front face, giving a kind of two fold ambiguity. Because we have not identified directions, there is an additional eight fold ambiguity associated with rotations and reflections for each of the face identifications, all without adding any "connects to" relations. In order to remove some of the orientation ambiguities, the graph and chunk shown in Figures 11.3 and 11.4 were used instead of the ones shown above.*

Relations "is left of" and "is over" are defined as total order relations analogous to "precedes."

Demonstration of the cube assimilation and reversal phenomena are done as follows: Input information acquisition is performed by a set of operators directly analogous to the operators that process letter sequence input. With a given graph in place, the system prompts the user: "Attending to what corner? :" The user replies with the name of a corner vertex, say C1S. This results in storage of an arc: "C1S is part of FRONT FACE." Bulk memory is addressed with a query, which finds the cube chunk and maps it in with face identification controlled by the input arc. The chunk is made to include the arc stored at the point of the user's reply. After chunking is complete, the user is again prompted: "Attending to what corner? :" and an arc is stored in the graph using his reply as before. If he names a corner on the current back face, then the arc will conflict with the chunk just acquired, and that chunk will be removed. The same chunk in bulk memory then maps in with the other face being the front face, and the user is again

---

* As with the machine-code tasks, the availability of the given information in external memory is simulated by locking it into the Slate. The chunk organization of the resulting Slate content shows that all of the given information was assimilated.

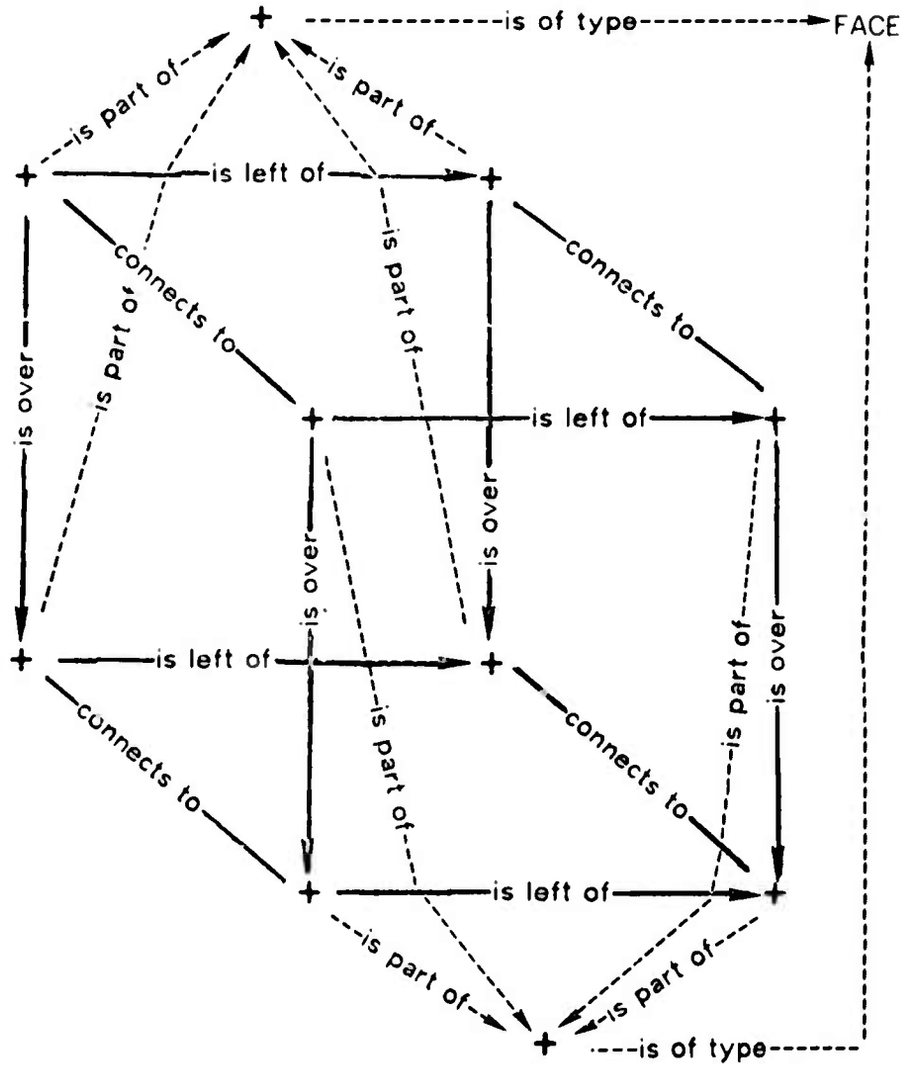


Figure 11.3 - Graph Representing Oriented Necker Cube

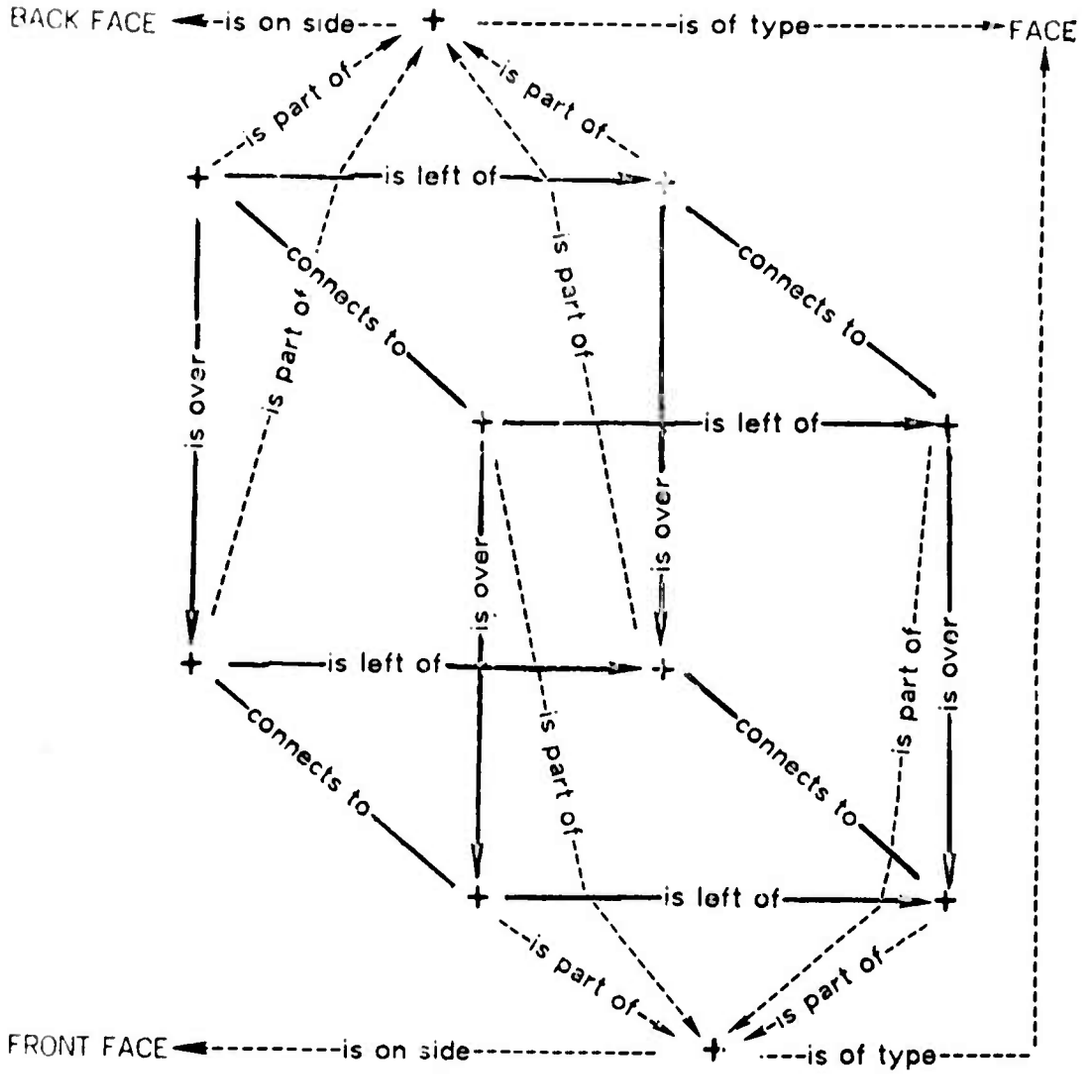


Figure 11.4 - Chunk of Oriented Cube

prompted for a corner to attend to.

The assimilation successfully adds depth information to the given information about the cube. Once the cube is assimilated, depth information is always present. Either of the two orientations may be indicated, but only in alternating sequence, never simultaneously. The inputted depth information is always consistent with Necker's hypothesis that the attended-to corner is on the front face.

The only three parts that are unique to the Necker task are the prompt for a corner mechanism, the mechanism that stores the resulting arc, the routine that merges the stored result with the chunk of the subsequent mapping. This latter (merge) mechanism is necessary in order to be able to remove the effects of a previous attention arc after a shift of attention has taken place. This differs from the string assembly tasks, where there was never any need to remove the arcs which represented previous input events.

There are two kinds of input graph ambiguity, one arising from alternate ways in which the next chunk is selected and on arising from the way the token assignments are selected. The cube task exemplifies the assignment type.

We see that the system can in a sense experience the successive perceptions of Necker cubes, but not express the ambiguity of the cube in the result graph.

The differences between the cube task and all of our others help in understanding the capabilities of our methods. The definitions of the geometric constraints on the directional relations are adequate for this task, but they do not constrain enough to be generally adequate. They permit constructs such as A above B to the left of C above D

to the left of A. The relation definitions could be extended to cover such cases in a way which caused the graph to always contain all the derivable "higher than" and "somewhere left of" relations; this would be unreasonably cumbersome for examples of interesting sizes. A different approach is called for.*

The system fails to take into account the adjacencies of vertices created by its own assignments. This lack leads to some errors in cases with symmetries. Thus for example in the graphs in Figures 11.1 and 11.2, the front face may map correctly and then the back face map incorrectly. The created adjacencies should affect the effective support for various proposed assignments. In the State system this would be a minor modification of the processing which occurs between assignments.

SUMMARY

We have learned several new things from the Necker Cube task. The task is accomplished using processes developed for parsing noisy symbol sequences, and yet it is exhibiting a particular effect from the phenomena of visual perception. This suggests that we are able to identify functional similarities between the two dissimilar tasks. Such similarities have both psychological and informational consequences.

---

* This is one of several cases which make us want to separate a sense of orderedness (vertical, horizontal, time and depth order are some of the important cases) from the rest of the representation and mechanisms.

The single-hypothesis property of the Slate memory is clea.ly exhibited. It is interesting that the cube topology did not introduce new difficulties for the match routine. The need for a dynamic notion of support in the match process has been demonstrated. This means that the match must be guided by its own local hypotheses as well as the initially-available information.

The weakness of Slate's relational formalism for representing geometric constraints has been exposed. A counterexample to the heuristic that any prior input information should always override the results of chunking has been exhibited.

REPRESENTATION OF INFORMATION.

CHAPTER 12-1

This chapter presents the directed graph representation schema of the Slate system, along with parts of the program which implement it.

DESIGN CONSIDERATIONS

The choice of a representation for a particular problem has pervasive consequences in the approach to solutions, since it determines what operations are easy to carry out. The sensitivity of problem solving processes to small changes in representation has been widely demonstrated on a diversity of tasks in the history of artificial intelligence. See [n71], chapter 2. Given a particular task to be performed, it is often possible to take great advantage of the structure of that task in choosing a representation, so that the most frequent operations required by the particular task are made easy. For example, the two kinds of organization expressed in the two halves of telephone books reflect such advantages.

We should therefore expect that an a-priori choice of a representation, on a basis which is not responsive to task content, would commonly carry a serious performance penalty relative to the performance of a task-specific problem solver. All of the evidence from the present project suggests that this is true for the Slate system and the tasks treated in this thesis.*

---

* It also appears now that much of the penalty for particular tasks is avoidable. Design of Slate-like systems which take speed and space advantages from the structure of particular tasks does not appear to be difficult.

There are a number of reasons for choosing to work on a general representation in spite of these penalties:

1. To investigate problems common to several tasks.
2. To develop approaches which can be applied to many tasks.
3. To develop a flexible tool for exploring new tasks.
4. To provide a performance reference for task-specific systems.
5. To explore the representation being used.

The chief reason for choosing a task-independent representation for this work is simply that we wanted task-independent results. We were seeking knowledge of problems and approaches which would span a number of tasks, and hopefully provide some insight into the entire class of problems.

RE-REPRESENTATION AND ITS CONSEQUENCES

A useful guideline in designing representations is that all of the regularities and structural features of the things represented should be simply identifiable in their representations. The justification for this is twofold: that any such information is potentially useful in reducing the difficulty of problem solving, and that the act of representing a structural feature tends to reveal the structure to the designer in a way which facilitates his designing. In a program, wherever a single item of knowledge is represented by more than one program item, then the representation is to that extent defective. It fails to represent the relationship between the multiple items, namely that they encode the same knowledge. To the extent that the programs must deal with comparisons of such items, the knowledge of their equivalence is obscured or lost. Depending on how the information is used, such re-representations can result in erroneous conclusions, failure to generalize, or other errors. For both the designer and

his program, rerepresentation also limits the possibility of constructing a meta-level of representation.

The principle that one item of knowledge should be represented in one place reduces the space of acceptable representations. The three-level program structure described below is in part a response to the desire to localize the program's representation of formal regularities which span many tasks.

GROSS PROGRAM ORGANIZATION

This section presents the organization of the system into functionally independent parts and the communication of the parts.*

The system is composed of a number of independent sections arranged into a linear sequence of levels, with each level communicating only with its immediate neighbors, as shown in Figure 12.1 .

---

* Programs and exercise data are available through the author.

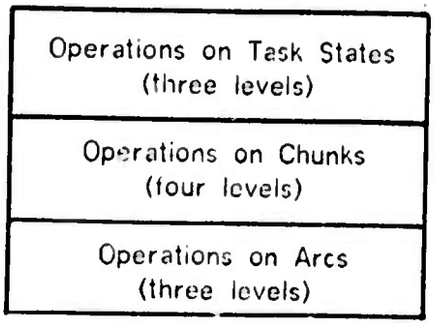


Figure 12.1- Strata of Groups of Operators.

Each level has one input port which it examines whenever it is called, and one output port where it deposits all of its results. Each level (except Level 1) has access to the immediately adjacent next lower level, in that it can deposit in its input port, examine its output port, and call it. All levels perform according to the same general discipline when they are called. It consists of three steps:

1. Examine the input port to identify the operator (command).
2. If it is an operator performed by this level, perform it, taking any operands from the rest of the input port. Otherwise submit the entire input port content to the next lower level and call that level.
3. Deposit result in output port and return control to caller.

The three levels which implement the operations on arcs are described in this chapter, and the remaining operators in the next. (In addition there are parts for communicating with the user and for debugging.)

DIRECTED GRAPHS

Let V and R be two disjoint sets whose members are called vertices and relations respectively. An arc consists of a relational-triple and a chunk-list. A relational-triple is an ordered set of three symbols, say

V1 r V2

where V1 and V2 are vertices and r is a relation. A chunk-list is an ordered set of zero or more vertices, called chunk names.*

> A chunk is the set of all arcs which have a particular chunk name on their chunk-lists.

Since an arc may have several chunk names on its chunk-list, the chunks of a graph do not partition the graph, but rather form overlapping sets.

There are several memories in the Slate system, each of which holds a graph. (The system never treats arcs from more than one memory as constituting one graph.) Each vertex known to the system is represented in two different ways, depending on context. There is an internal symbol (an odd integer) for each vertex, and a printable name. Each vertex is either a constant or a token. Each relation known to the system is represented in three different ways, depending on context. There is an internal symbol (an even integer) for each relation, and there are two printable names, called the direct name and the converse name.

---

* For reasons that are merely historical, chunks are identified by the names of vertices, and if a vertex Vi is in a particular chunk-list, then Vi is also part of a relational-triple in at least one arc which has Vi in its chunk-list.

If NV1 and NV2 are the printable names of V1 and V2, and Nr and NCr are the direct and converse names of r respectively, then the Slate system will take either of:

NV1 Nr NV2

NV2 NCr NV1

as a reference to the relational triple

V1 r V2.

LEVEL 1 - GRAPH STORAGE.

All of the graph storage of the system is located in Level 1. The three graph memories mentioned above reside in this level, and so any use of them requires that Level 1 be addressed.*

Each graph memory content is represented by a set of three-symbol groups in a table, with additional symbols in the table indicating in which chunks each arc is included.

---

* The operations on arcs presented below are derived in part from a system by Roger Elliott. [E65] He developed a "fact retrieval system" (by analogy to "information retrieval system") in which assertions were represented by arcs in a directed graph. The arcs were made readable to the system user, who could present queries answerable from the graph. Answers consisted of sets of arcs, counts or certain other simple responses. The examples developed included an airline reservation system, an organization chart and a qualitative map of city locations.

His system faced two problems the Slate system also faces: a need for consistency in the graph and a need to localize and suppress from user's attention the simple completion operations of transitive relations, symmetric relations and so forth. To meet these needs systematically, Elliott used a set of defining properties to be associated with relation names. Compatible sets of properties define relational classes. Compiled routines associated with particular classes kept the graph consistent and complete according to the properties. The user could also define interactions between arcs with different relation names by a "combine" statement, which was used interpretively.

Although the memories accomodate arcs which are not in any chunk, we have not used them. In practise, arcs in bulk memory are members of exactly one chunk, and arcs in the other memories are members of one to seven chunks, which has always been enough.

Level 1 performs explicit storage in its table and explicit retrieval.* The Level 1 operators are shown in Table 12.1 .

The operators which take partial specification (the two which set up generators) make it possible to search a graph. These operators are used heavily in searching the knowledge base and in mapping chunks into the graph.

---

* There is a design issue concerning which arcs should be stored and which should be found by computation from other information. (Arcs such as "30,000 is larger than 27,318," "A equals A," "New York City is east of Los Angeles," are candidates for computational recovery from other stored information.)

The design of the system allows for such redundant arcs to appear at the ports of Level 2. They can be produced by appropriate relation interpreters in Level 2 rather than being stored explicitly in Level 1. Thus Level 2 presents to the remainder of the system a set of virtual directed graphs for which the store/recompute issues have already been dealt with.

For the relations used in these experiments, the decision has always been to store all known redundant arcs explicitly rather than recompute any of them on retrieval. This leads to easy debugging and also to an empirical knowledge of the needs for recomputation.

Levels 1 and 2 have been separated from each other and from the rest of the system in order to make it possible to resolve store/recompute issues on a relation-by-relation basis.

Store	Enter 3 symbols representing relational-triple of arc into designated memory
Retrieve	Set up a generator which will retrieve arcs from memory on request according to a given partial or complete specification. (For example: JRST is the opcode of *, where * represents an unspecified vertex to be discovered.)
Getnext	Produce the next arc from a particular generator set up by a Retrieve operation.
Alter	Mark all arcs conforming to a given specification as members of a given chunk.
Mark retrieve	Set up a generator of arcs of a specified chunk.
Getmark	Produce the next arc from a specified generator set up by Mark retrieve.
Demarkit	Remove the most recently generated arc of a specified generator from a specified chunk; if this leaves the arc outside of all chunks, erase it.
Query	Test whether a particular arc is in a specified memory.
Storekill	Turn off a specified generator
Erase	Erase all arcs conforming to a given specification.
Clearmem	Remove all arcs from a specified memory.

Table 12.1- Level 1 operators.

Level 1 does not restrict the graphs that are stored in any way, except that only bona fide arcs are stored. The operations which maintain graph consistency are in level 2 and level 3.*

RELATION DEFINITIONS

This section presents a definition schema for relations, and its program implementation. These parts of the system serve two purposes: they exclude from memory some simple combinations of arcs which the definer regards as contradictory, and they cause inferential side effects to occur when certain combinations of arcs are stored in memory, the side effects always being the storage of additional arcs.

We first present relational properties, which have no independent identity in the system. The properties are grouped in various ways into classes, which do correspond directly to program parts. The classes are associated with relation names by a relation definition process, and are used by relation interpreters in Level 2 of the system.

---

* During system development, a complex method of generating the arcs of a chunk was developed and later discarded. Levels 4 and 5 contain these operators for this method: retrieve1, getnext1, retrieve2, getnext2, modelinit, modelget, killfgen. In addition, level 5 contains operators for manipulating binary marks (called paint) on vertices for bookkeeping purposes. The primary use for these marks is to mark vertices already accessed by processes in order to prevent reprocessing and loops. These operators are: paintit, unpaintit, strip, seeit.

RELATIONAL PROPERTIES

We express the relational properties by another directed graph representation, which is not the graph system of the Slate. A property consists of a test part and an action part. Let R = (A,P) be a set of relation names and V = (V1,V2,V3) be a set of vertex names.* Let G be the set of graphs which can be formed from the vertices of V and the relations of R. Then a property test part is a graph of G.

An action part is either the symbol "FAIL" or a graph of G which is restricted:

1. to contain vertices only from the corresponding test part,
2. to contain arcs on relation A but not relation P,
3. to contain no arcs contained in the corresponding test part.

Arc Va A Vb in a test part matches an arc V1 r V2, by substitution of V1 for Va, V2 for Vb. Similarly arc Va P Vb in a test part matches a directed path of arcs V1 r Vx, Vx r Vy, . . . Vz r V2 by substitution of V1 for Va, V2 for Vb. A successful match requires a consistent set of vertex substitutions such that all of the arcs of the test part are matched.

The definitions are used as follows: When a request to add to some target graph an arc, say X r Y, is presented to the Level 2 input port, the definition of r is used in an action which is equivalent to the following informally described sequence, using the properties for relation r and the set of arcs T which are the arcs on relation r contained

---

* A and P will match single Arcs and Paths respectively. Although the property definition graphs could be extended to more than three vertices, we have found no need to do so.

in the target graph.

First, for the properties which have a graph as the action part, match the test parts of the properties in all possible ways against the combination of T and the incoming arc. A successful match must include the incoming arc either by matching against an arc on relation A, or by the incoming arc completing a path in T, whose terminal vertices match an arc on relation P, such that there was not a corresponding path in T alone. For each completed match, the substitution values of the vertices of the test part allow us to define a set of arcs corresponding to the action part in the obvious way. All such arcs are stored with T to yield T'.

Second, all the properties which have FAIL as the action part are matched in T'. If any matches are successful, all the arcs in T' - T are removed from the target graph, leaving it in its previous condition, and a FAIL signal is returned. Otherwise, T' is retained in the target graph and a success signal is returned.

The Slate system relations are defined in terms of the properties shown in Figure 12.2 .*

---

* For completeness, all of Elliott's properties are shown, although not all are used. An A denotes a single arc, P a path.

PROPERTY NAME	TEST PART	ACTION PART
irreflexive	V1 A V1	FAIL
reflexive	V1 A V2	V1 A V1,V2 A V2
symmetric	V1 A V2	V2 A V1
asymmetric	V1 A V2,V2 A V1	FAIL
transitive	V1 A V2,V2 A V3	V1 A V3
one-follower	V1 A V2,V1 A V3	FAIL
one-leader	V1 A V2,V3 A V2	FAIL
no-regrowth	V1 A V2,V1 P V2	FAIL
unlooped	V1 P V1	FAIL
no-chain	V1 A V2,V2 A V3	FAIL

Figure 12.2- Relational Property Definitions

The classes are defined by sets of properties. Table 12.2 gives the class numbers used and the corresponding sets of properties.

We have used the closest analogous property names and class numbers from Elliott's definitions where applicable. Our classes are not quite equivalent to Elliott's because in his treatment of vertex variables, Elliott allowed one vertex name to match more than one variable, thus merging some topologically distinct cases. He used only 6 of his classes in his demonstrations. We use 9 of ours, including 6 analogous to his. As noted elsewhere, our addition of the no-chain property is a correction which makes possible correct definitions of relations which map between disjoint sets.

CLASS NUMBER	PROPERTY LIST
1	irreflexive, asymmetric, one-leader, one-follower, unlooped, no-regrowth.
3	irreflexive, asymmetric, one-leader, unlooped, no-regrowth.
4	irreflexive, asymmetric, unlooped, no-regrowth.
15	transitive, irreflexive, asymmetric, unlooped.
23	symmetric
26	(none)
33	no-chain, one-follower, one-leader.
35	no-chain, one-follower.
36	no-chain.

Table 12.2- Relational Classes and Their Properties

The use of the relational classes in the Slate system is shown in Table 17.1 .

CREATING RELATION DEFINITIONS

One of the operators at the teletype interface of the system is DEFINE RELATION. It prompts for the relation name, the converse name, the relation class number and a "mark word," which amounts to a required restatement of the one-leader and one-follower properties defined above. All of these are entered into a table of relation definitions which can be accessed by the internal symbol of the relation.

CREATING VERTEX DEFINITIONS

Similarly there is an operator DEFINE VERTEX which prompts for a vertex name and a property word which specifies whether the vertex being defined is a token. These two items are entered into a table which can be accessed by the vertex internal symbol. Vertices may also be entered into this table by an internal vertex creating routine, which creates a three-letter print name for the vertex.

LEVEL 2 - RELATION INTERPRETERS FOR ALL RELATIONS.

For each class there is a corresponding set of four routines in level 2 which constitute the interpreter for that class. For each of the following,

1. Store operator
2. Retrieve operator
3. Alter operator
4. Erase operator

there is a corresponding routine which is responsive to it for all requests which enter Level 2 and which specify a relation in that class. There is a similar set of four for pseudoclass 0; they are used whenever the relation name is not specified in the input port.

The store operator routine would be called, for example, if the Level 2 input port specified a store of the arc: "JRST is the opcode of BEW." It would call Level 1 enough times to determine that there was no conflicting arc in the memory, and then call Level 1 to store the arc.

In the event of conflict, the arc would not be stored, and this fact would be noted in the output port of Level 2. Since all storage requests must pass through Level 2, this

assures that the coocurrence and inference properties of each individual relation are maintained.

The store operator routine for class 15, a class of transitive relations, would be called if a store of "BEW appears before JOM" were requested. It would call Level 1 several times to determine all of the "appears before" arcs deducible from this one in conjunction with currently stored arcs, and would request storage of these as well.

For example, if the memory contained the arcs shown in Figure 12.3 then the memory would contain the arcs of Figure 12.4 after storing this arc. In this case a request to store a single arc results in the storage of six arcs.

Level 2 also has a Getnext operator, which is entirely analogous to the Getnext operator of Level 1, except that it operates on generators set up by the Level 2 Retrieve operator rather than the one in Level 1.

RELATION INTERACTIONS

There are relations which are jointly meaningful which must be used in a coordinated fashion to avoid incongruities. For example, in graphs denoting set membership, the relations for set membership and set inclusion must be coordinated so that the members of a set are also members of any set which includes that set. Another interacting pair of relations, used in the parsing of letter strings, is "is left of" and "starts with." See chapter 4. In building the parse tree from a letter string, we

Figure 12.3- State Before Storing Arc

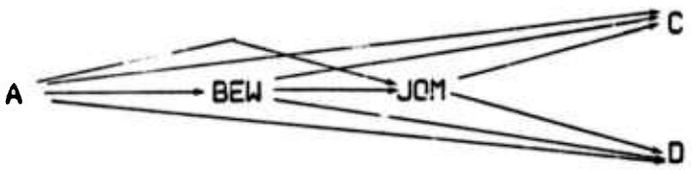


Figure 12.4- State After Storing Arc

want to keep track of the left-right relations between syntactic constructs as well as letters. In particular if we have, for some vertices AA, BB, CC, the arcs "AA is left of BB" and "CC starts with BB," then we want to infer arc "AA is left of CC."

Level 3 implements the kind of interactions exemplified above, using an operator called Fullstore.

We define an interaction specification, which consists of two arcs (called condition arcs) on a three-vertex graph, and a third arc (called an inference arc) on the same three-vertex graph. The vertices of this graph are called variable vertices, and the graph is called a variable graph.*

Whenever an arc with the same relation name as one of the condition arcs appears in a store request to Level 3, a search is made for an arc having the same relation name as the second condition arc, and for which there is a unique mapping CM of three distinct vertices onto the tree variable vertices such that CM maps the arc to be stored and the arc found by search onto the two condition arcs in the obvious way. If such a second arc is found, then Level 3 issues a request to store both the inference arc and the requested arc. This is repeated for all of the interaction specifications which match the request. Since the inference arc might itself lead to inferences, the store request for this arc is issued to Level 3 rather than Level 2. Level 3 is the only recursive level.

A second effect of the Level 3 operator, Fullstore, is to include both the Store operation and the Alter operation of Level 2 under a single operator, so that they need not be specified separately at higher levels. Fullstore accepts an arc bearing a set of chunk marks; it includes the arc in all of the chunks named in the set.

In addition to straightforward inference-making, the relation interaction mechanism is used to prevent certain combinations of arcs. For example, the relation "precedes" and the relation "begins with" interact to prevent an element from preceding the beginning

---

* This graph exists for exposition purposes only. The description does not represent the way it was implemented.

of a sequence. This is accomplished by making an inference from the undesired combination which is sure to be rejected according to the definition of its relation. The rejection leads to failure of the attempt to complete the combination. (A more aesthetically pleasing way would have been to implement class and fail notions like those for relations.)

The interactions used are shown in Figure 125 below.

ARC PROCESSING OVERVIEW

We can regard the Level 3 input port as a means of access to a set of graph memories which have their vertex names and relation names in common, for which consistency is guaranteed, and for which the simple (interaction) consequences of knowledge of an arc are automatically included in the graph.

This organization relieves all of the processes which use the graphs from the burden of keeping them complete and correct. Keeping the graph free of conflicts and discovering the inferences to be made from a given graph both involve a large number of contingencies, even in simple cases. The availability of a fairly sophisticated storage medium at the ports of Level 3 thus makes the specification of all of the higher levels much easier.

There is some overlap between the capabilities of the relation interaction mechanism and the inferential mechanism of the relation definition schema. Where possible, the

MATCH ARCS	INFERENCE ARCS
AA is a subset of BB CC is a member of AA	CC is a member of BB
AA begins with BB CC precedes BB	BB precedes CC
AA ends with BB BB precedes CC	CC precedes BB
AA precedes BB CC starts with BB	AA is left of CC
AA finishes with BB BB is left of CC	AA is left of CC
AA is left of BB CC starts with BB	AA is left of CC
AA is left of BB AA is part of BB	BB is left of BB
BB is left of AA AA is part of BB	BB is left of BB
AA is part of BB CC is the side of BB	CC is the side of AA
BB is part of AA CC is the side of BB	CC is the side of AA

Figure 12.5 - Relation Interaction Definitions

latter have been used, since we believe that they are faster, and the interaction mechanism is used only when two different relations are involved.

CHUNK PROCESSING OPERATORS, HEURISTICS AND EVALUATION

CHAPTER 13-1

OVERVIEW

This section describes the chunk processing parts of the Slate system. The purpose of the description is to provide a basis for evaluating its primary heuristics and modeling the effort expenditure of the program.

The system maintains three graph memories: the Slate, which holds the input graph in its current state of augmentation, bulk memory, which holds all of the chunks of the knowledge base, and query memory, which holds a query being addressed to the knowledge base.

A major portion of each task consists of attempts to augment the Slate using chunks from bulk memory. A single attempt to augment the current graph passes through the stages diagrammed in Figure 13.1.

The principal actions of these parts are as follows:

Query Formation - creates a graph which is a subgraph of the current Slate content. This graph represents the part of the Slate for which an augmentation is desired. If the query graph is smaller than the entire Slate content, then the subsequent processes take correspondingly less time.

Knowledge Search - identifies chunks in bulk memory which have possible overlap with the query graph. The possible overlap must exceed a threshold score computed on

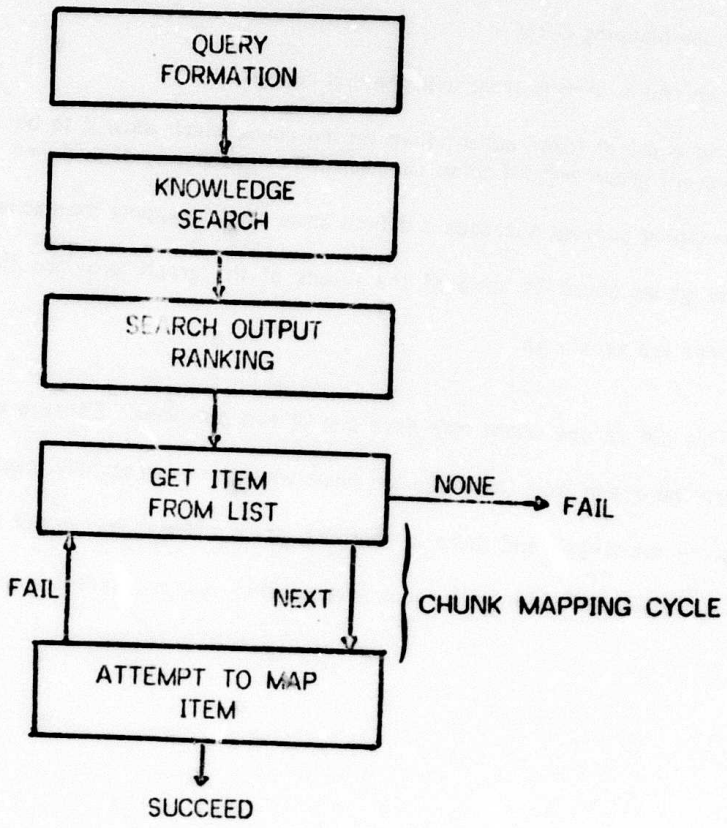


Figure 13.1 - Mapping Attempt Operations

an arc by arc basis.

Search Output Ranking - ranks the set of chunks delivered by the knowledge search on the basis of the number of different constants from the query which they include. Chunks not spanning at least two constants of the query are dropped.

Chunk Mapping Cycle -

For each chunk, in rank order until the first success, do:

Find a set of token substitutions on the chunk which allow it to be copied into the current graph without either contradiction or inadequacy of evidence.

The mapping process succeeds if it finds some partial mapping from some of the tokens of the given chunk to some of the tokens of the graph, provided that its rules of evidence are satisfied.*

This set of operations may have one of two outcomes. Either a new set of arcs enters the graph from the particular chunk which was successfully mapped, or else the graph is unchanged and there is no chunk which will map in. In our experiments the cycle is repeated whenever a chunk is successfully mapped, except in a few cases which are noted.

OPERATIONS ON CHUNKS

This section presents the operators which have effects which are of chunk-sized scope. They also operate on multiple memories rather than just one. Figure 13.2 shows these operators and their dependencies.

The functions of these operators are described below.

---

* The rules of evidence are summarized near the end of this chapter.

THE REGARD OPERATORS

The operator REGARD causes the system to attempt assimilation of the current Slate content. It is the highest task-independent operator. It accomplishes the actions shown in Figure 13.1 by sequential use of the following operators: Makequery, Searchsetup, Searchget (several times), and a cycle of Preparechunk and MAPCHUNK, ending with INSERTCHUNK if MAPCHUNK succeeds. The operators REGARD and its alternate REGARD2 should be understood as one, since REGARD2 is used only when its effects are identical to those of REGARD. It skips the query-formation step and uses the Slate content as the query.

QUERY FORMATION

The Makequery operator forms a query graph in query memory from the current graph.

We can regard the Slate content as consisting of an "uninteresting" already-explained part and an "interesting" unexplained part, corresponding to tokens which have or have not been mapped onto successfully. The query graph formed by Makequery consists of the union of all of the one-arc neighborhoods of tokens which have not been mapped onto more than a certain threshold number of times. This threshold is zero for all experiments except the machine code and Necker cube tasks, where indefinitely many mappings onto certain vertices are necessary.*

Notice that the selection of interesting vertices does not partition the Slate content along chunk boundaries. The old part of a newly introduced chunk is uninteresting, and the new part is interesting. Several earlier query formation methods tried to identify

the interesting chunks rather than vertices. These tended to produce excessively large queries, and therefore excessively large responses, leading to much irrelevant and repeated work. A method based on recency of acquisition and another based on chunk inclusion were both ineffective relative to the method described above.

SEARCHSETUP AND SEARCHGET

Two operators implement the search for chunks which plausibly could map into query memory (and therefore into the corresponding portion of the result graph.) One, called SEARCHSETUP, sets up a generator of chunks for a search of a specified memory. A second operator, SEARCHGET, discovers the next chunk which meets the threshold of acceptability in a search set up by SEARCHSETUP.

PREPARECHUNK

The first, PREPARECHUNK, creates a table of scores showing the degree of correspondence between the neighborhoods of tokens in the chunk and tokens in the target graph. This table, called the Support table, contains a score for every possible pairing of a token in the chunk and a token in the graph. For a particular chunk-token/graph-token pair, points are given for every arc in the chunk for which there is a corresponding arc in the graph, as shown in Table 13.1 below.

---

* For the latter, an unattainable threshold can be used, making all tokens "interesting." It is faster to skip selective query making by using the REGARD2 operator rather than REGARD for assimilation.

POINTS	CHUNK ARC	GRAPH ARC	
1 point	T R T	T R T	
20 points	C R T	C R T	T = any token R = Relation name C = Constant name

Table 13.1 - Support Table Scores

The effect of the higher score for arcs containing constants is to let arcs which correspond on constants outweigh any accumulation of arcs which correspond by substitution of tokens only. (Most vertices have many fewer than 20 incident arcs.)

The following is an example of the support computation. Figure 13.3 shows a graph of a section of machine code. It is part of the example shown in Figure 9.17 .

Figure 13.4 shows the chunk for which the support is to be computed.

Table 13.2 shows the support table resulting from the application of the PREPARECHUNK operator to this pair of graphs.

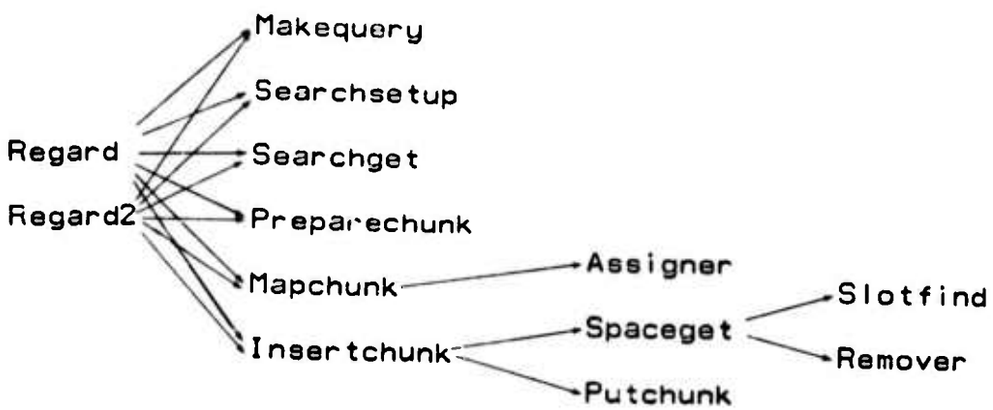


Figure 13.2 - Dependencies Among Operators on Chunks

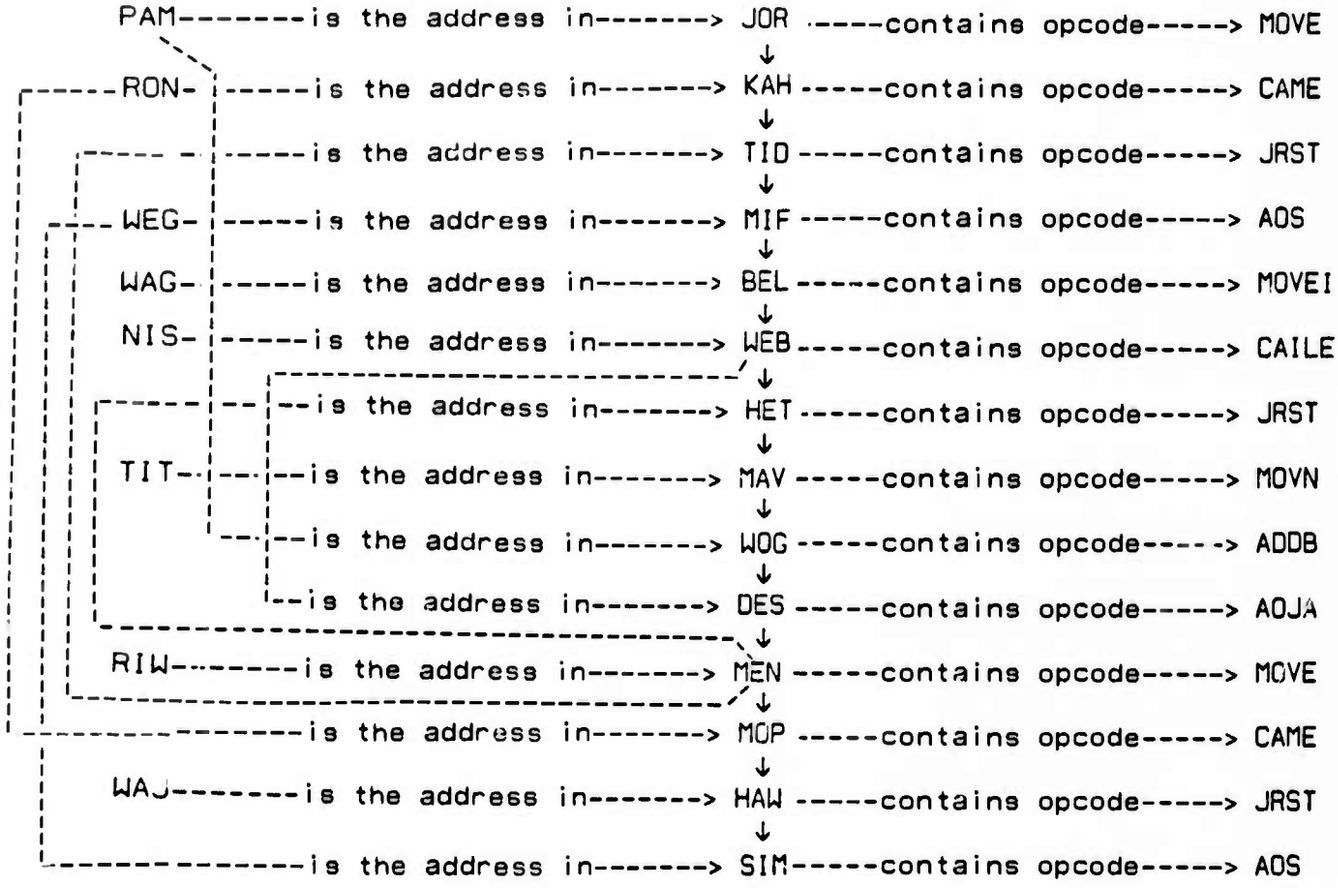


Figure 13.3 - Current Graph State For Support Computation

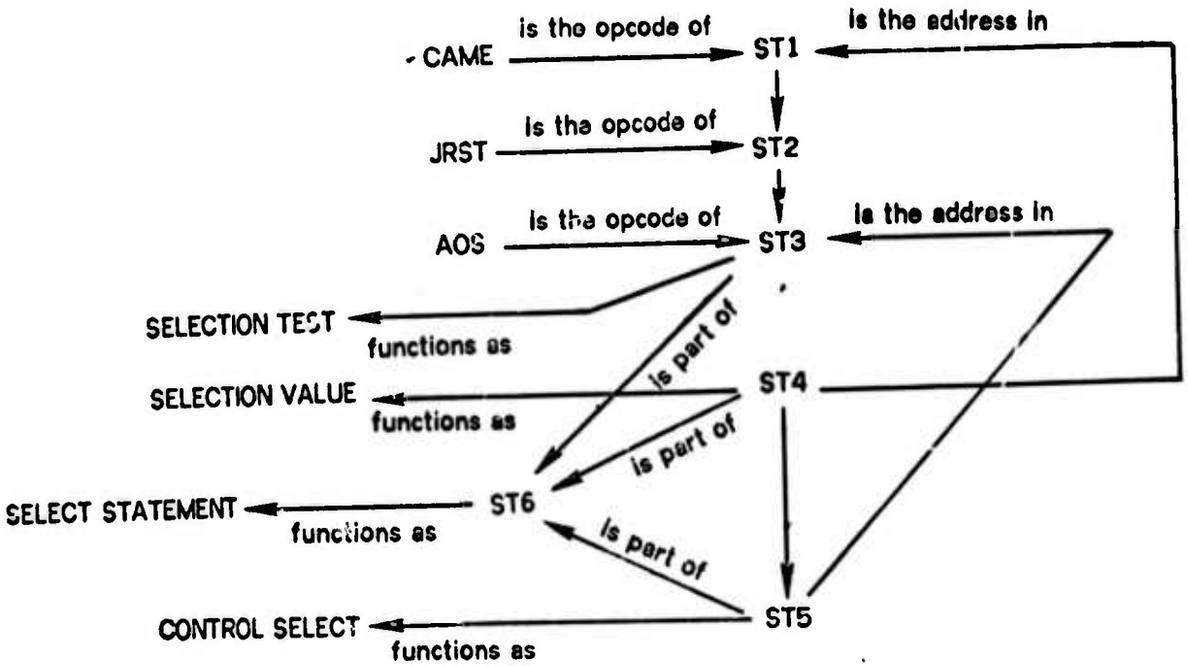


Figure 13.4 - Chunk for Support Computation

	ST2	ST3	ST1	ST4	ST5
JOR	1	1	2	1	0
KAH	2	2	22	1	1
TID	22	2	2	1	1
MIF	2	22	2	1	1
BEL	2	2	2	1	1
WEB	2	2	2	2	2
HET	22	3	3	1	1
MAV	2	2	2	1	1
WOG	2	2	2	1	1
DES	2	2	2	1	1
MEN	2	2	2	4	4
MOP	2	2	22	1	1
HAW	22	2	2	1	1
SIM	1	22	1	0	1
PAM	0	0	0	2	2
RON	0	0	0	2	2
WEG	0	0	0	2	2
WAG	0	0	0	1	1
NIS	0	0	0	1	1
TIT	0	0	0	1	1
RIW	0	0	0	1	1
WAJ	0	0	0	1	1

Table 13.2 - Computed Support Table.

The values in the support table are computed from the two graphs. For example, the value of 22 for ST3:MIF is computed as follows:

CHUNK ARC	POINTS	GRAPH ARC
AOS is the opcode of ST3	20	AOS is the opcode of MIF
ST2 precedes ST3	1	TID precedes MIF
ST5 is the address in ST3	1	WEG is the address in MIF
For the pair ST3:HET, 3 Points:		
ST2 precedes ST3	1*	WEB precedes HET
ST5 is the address in ST3	1	MEN is the address in HET
For pair ST4:MEN, 4 points:		
ST4 precedes ST5	1*	MEN precedes MOP
ST4 is the address in ST1	1	MEN is the address in TID
ST4 is the address in ST1	1	MEN is the address in HET

Table 13.3 - Sample Support Table Computations

The Preparechunk operator passes over the entire chunk once. simultaneously computing all of the values of the Support table.

MAPCHUNK

MAPCHUNK uses the support table to select the sequence of assignments of chunk token to graph token which it tries.

The flow of control of the MAPCHUNK operator is shown in Figure 13.5 .

---

* The method of accessing graph storage in Level 1 involves hash tables which allow arcs to be found more than once by a generator. These are always treated as separate occurrences, with the result that some support scores are higher than the correct value. In each of these two cases, one excess point of support is attributed to a pair. The mapping process is not particularly sensitive to this kind of variation.

MAPCHUNK iterates through the possible assignments offered by the Support table

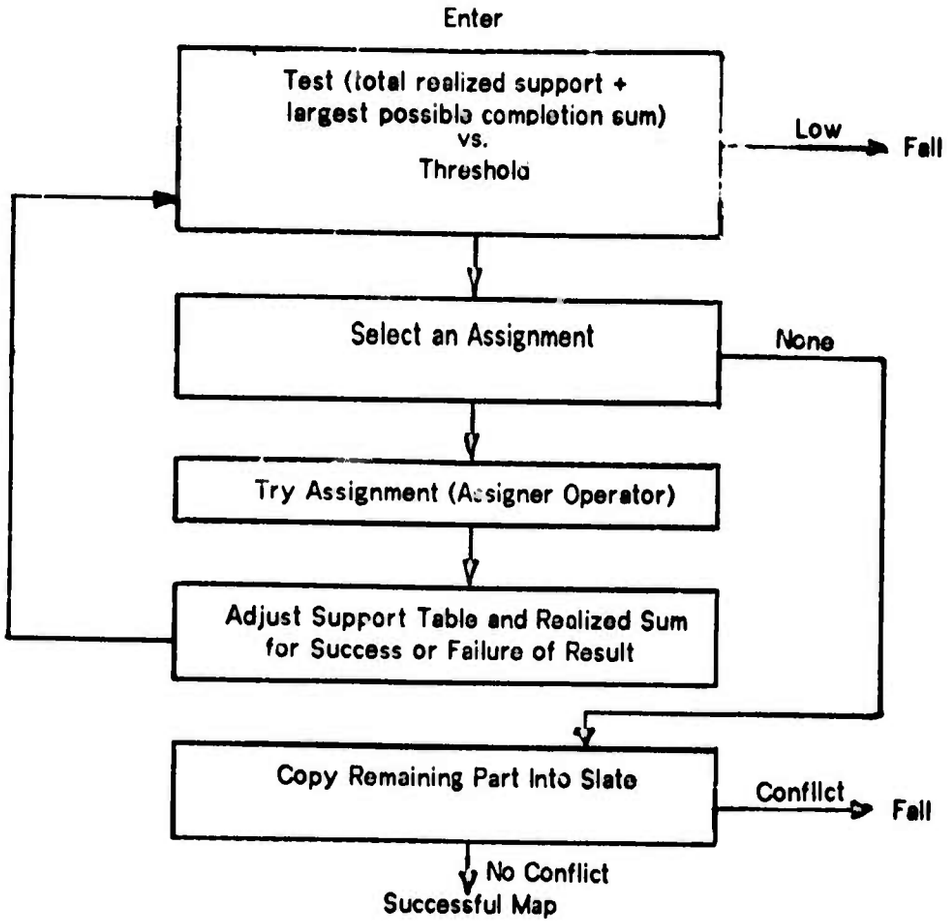


Figure 13.5 - Control flow under MAPCHUNK

until they are exhausted or hope is lost. If the possibilities are exhausted (indicating potential success,) a new token is created to correspond to each of the unassigned tokens of the chunk. The arcs on those tokens are mapped from the chunk, completing the copying of the chunk; if this final copying produces a conflict, the entire mapping of the chunk fails. Otherwise the mapping has succeeded, and all of its effects on Slate

content are retained. A detailed example appears below.

Assigner operator

The assigner operator processes a chunk-token/ Slate-token pair, P, specified by the MAPCHUNK operator. It can be thought of conveniently as a token-renaming copier of a local region of a chunk.

It uses a set M of token pairs, and maintains the correctness of the set. When ASSIGNER is called for the first time by MAPCHUNK, M is empty. Thereafter M contains all of the pairs for which assignment terminated successfully.*

ASSIGNER attempts to map the one-arc neighborhood of the chunk token into the one-arc neighborhood of the Slate token. For each arc A incident on the chunk token, ASSIGNER attempts to create a corresponding arc C(A) to add to the Slate token neighborhood. If Vc is a vertex of A, then its corresponding vertex c(Vc) is

Vc	- If Vc is a constant;
Vs	- if Vc is a token and Vc:Vs is in M(union)P;
undefined	- otherwise (i.e. if Vc is a token and no pair Vc:t is in M(union)P.)

If C(A) is V1 r V2 then C(A) is c(V1) r c(V2). If a vertex of C(A) is undefined then arc A is ignored.

---

* The operation of MAPCHUNK makes all of the pairs of M unique. If V1:Va and V2:Vb are pairs in M, then V1 neq V2 and Va neq Vb. Also, the sets of left and right members of pairs are disjoint, since the sets of chunk tokens and Slate tokens are disjoint.

M is maintained as a set of arcs on a bookkeeping relation, "maps onto," in query memory.

ASSIGNER attempts to store C(A) in the Slate. If no storage conflict results, ASSIGNER continues to process arcs from the neighborhood of the chunk token.

In the event of a conflict, ASSIGNER removes all of the arcs that it has stored in the Slate and returns a failure signal. If all of the arcs incident on the chunk token are successfully processed, then P is added to M and a success signal is returned.

One of the operands of ASSIGNER is the name of the chunk being built. All of the arcs added to the Slate are included in this chunk. Thus ASSIGNER operates to augment a particular chunk in the Slate.

There is a complication to the action of ASSIGNER which causes it in some circumstances to process more than one pair of tokens. For convenience it is discussed after the example.

INSERTCHUNK

INSERTCHUNK is the highest level Slate space management operator. It adds a designated chunk, whose arcs are already in the Slate and marked, to the list of chunks in the Slate. If necessary, it selects and deletes a chunk in order to keep the total below the fixed limit.

INSERTCHUNK uses the operator SPACEGET to provide an empty space in the Slate, and the operator PUTCHUNK to fill it.

SPACEGET

SPACEGET creates an empty space in the list of chunks in the Slate. It uses operator SLOTFIND to identify a slot to be filled, and if necessary it uses operator REMOVER to remove the chunk currently occupying the slot.

SLOTFIND

SLOTFIND finds a chunk slot in the Slate as follows:

1. If the Slate contains less than the limiting number of chunks, an empty chunk name slot is selected to be filled.

2. Otherwise, if there is a chunk which has been marked as included in some other chunk, ("covered"), then such a chunk is selected for removal.

3. Otherwise, a chunk is randomly selected for removal.

REMOVER

REMOVER removes a designated chunk from a designated memory. Chunk removal consists of removing the chunk name from all of the chunk-lists of all of the arcs in that memory, and removing any arcs which thereby have empty chunk-lists. Thus removing a chunk may or may not remove arcs.

PUTCHUNK

PUTCHUNK enters a designated chunk name into an empty slot in the list of chunks in the Slate.

EXAMPLE OF AUGMENTING THE SLATE GRAPH

Now we can follow the action of MAPCHUNK on the graphs of figure 13.3 and figure 13.4 . The sequence will result in the mapping of the chunk and addition of arcs to the Slate.

The first operation is to test the sum of the support values of past successful assignments (currently 0) and the most optimistic sum of future assignment scores. The latter is the sum of the support table column maxima (currently 74.) The reference for this test is a threshold which is preset externally. An acceptance threshold of 40 was used for the machine code experiments. This threshold in effect requires that two arcs containing constants be mapped or potentially mappable. If this test fails at any cycle the effort to map the chunk ceases, all of the arcs stored during this call on MAPCHUNK are removed and a failure signal is returned by MAPCHUNK. The test succeeds here, ($74 \leq 40$).

The next step is to select an assignment. The Support table is scanned and TID:ST2 is found to have a maximal support value. The ASSIGNER operator is called with MIF and ST2 as Slate and chunk tokens respectively. The assignment succeeds. No arcs have been added to the Slate, since any two-token arcs would have an undefined vertex, and the one arc having a constant in the chunk (AOS is the opcode of TID) was already in the graph.

The next cycle successfully assigns ST3 to MIF. A new arc, "SELECTION TEST is the function of MIF," has entered the Slate. The next cycle successfully assigns ST1 to KAH without adding new arcs.

The next cycle attempts to assign ST4 to MEN. This fails because of a conflict between the arc "RON is the address in KAH" in the Slate and the attempt to store arc "MEN is the address in KAH," which is a contradiction under the defined properties of class 3, of which "is the address in" is a member.

The next 7 attempts are:

assigning:	STORING THIS	conflicts with THIS
ST5:MEN	MEN is the address in MIF	WEG is the address in MIF
ST4:WEB	WEB is the address in KAH	RON is the address in KAH
ST4:PAM	PAM is the address in KAH	RON is the address in KAH
ST4:RON	(succeeds)	
ST5:WEB	RON precedes WEB	BEL precedes WEB
ST5:PAM	PAM is the address in MIF	WEG is the address in MIF
ST5:WEG	(succeeds)	

This exhausts the support table. Of the 132 possible assignments, 89 had non-zero support. Of these, 11 have been tried, of which 5 were successful. (Selection is done in a space containing 190,050 possible mappings for this case.) There is a remaining unassigned vertex ST6 in the chunk.

The mapping is completed by copying the entire chunk in, performing all of the substitutions of the existing assignments, creating a new token to be the assignment of any yet unassigned token. The set of arcs added by the mapping process is shown in the graph in Figure 13.6 .

At this point there is an arc in the graph which corresponds directly to each arc of the chunk. The chunk mapping process is an all-or-none process in this sense. Any single failure to map one arc will cause all of the accessions for a particular chunk to be

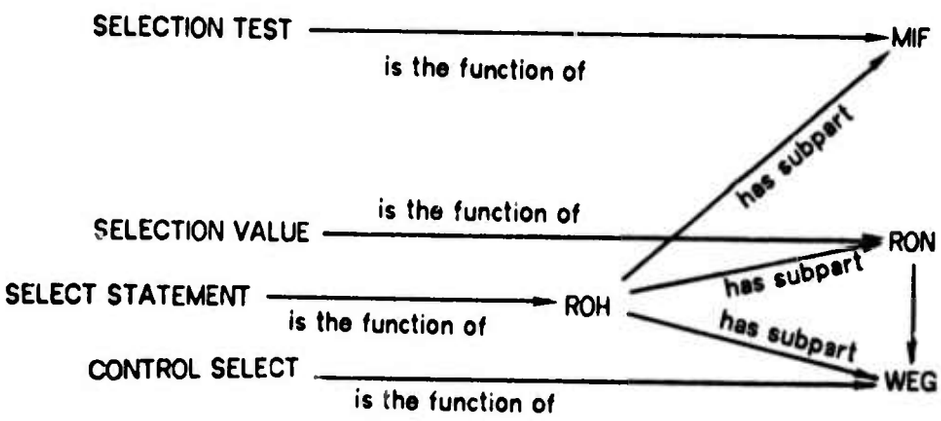


Figure 13.6 - Arcs Added to Graph of Figure 13.3
by Mapping One Chunk

removed.

FORCED ASSIGNMENTS

The mapping process exemplified above is simpler than the one used in the experiments in one important respect. When the success of a particular assignment makes the assignment of certain other pairs inevitable, then the ASSIGNER operator carries out the assignment of these other pairs as well.

Consider the state of the above example just before the first failure of assignment. The State contains arc "RON is the address in KAH", the chunk contains arc "ST4 is the address in ST1", and ST1 has been assigned to KAH. To map this chunk, the arc on ST4

and ST1 must be mapped in somehow. Any assignment of ST4 other than ST4:RON will conflict with the former arc, since by the properties of relation "is the address in", (Figure 12.2 - the one-leader property) the subgraph -------->,<--------- on this relation is contradictory. So it is necessary to assign ST4:RON. The first assignment failure is caused by this particular contradiction.

The key to this necessity is that relation "is the address in" is single-valued for vertices on which its arcs are positively incident.

The method described below forces all of these "necessary" assignments to be made as soon as their necessity can be determined.

When each relation is defined, there is associated with it a "mark word" which contains a pair of symbols which specify whether it is positively and/or negatively single-valued. When ASSIGNER finds an unspecified arc (a two-token arc with the second token unassigned) it uses these symbols to decide whether the assignment of the other token is forced by this arc. If so, ASSIGNER calls upon itself to perform the other assignment immediately. There is a switch in the system to determine whether this forced-assignment test is performed.

What beneficial and harmful effects does the introduction of the forced assignment heuristic have on the system? The evaluation of the forced-assignment heuristic has an efficiency aspect and an accuracy aspect. The use of forced assignments avoids the effort which would be expended otherwise in pursuing assignments that could not possibly succeed. It also causes some attempts at mapping chunks to succeed where

they would otherwise fail. This occurs because a set of individually successful assignments may be incompatible. If, in the example above, we tried assigning ST1:KAH and ST3:SIM, both of these assignments would succeed. Both also have maximal support, so that this sequence is feasible. However, there remains no feasible assignment for ST2. The two assignments are consistent with their immediate neighborhoods, but not with the requirements of the chunk. There are no means for removal of successful assignments in the system.

Tests on letter-sequence mapping indicate that when forced assignment is turned off, these failures to perform compatible assignments are the principal error source of the system.

For the example, with forced assignment turned on, all of the assignments after the first are forced.

The effort taken by the two cases can be compared as follows:

	ASSIGNER CALLS	LEVEL 1 ACCESSES
Forced assigns	5	280
No forced assigns	11	681

Table 13.4 - Forced Assignment Effort Effect

This saving is representative of a general increase in speed which occured when forced assignments were introduced into the system.

From the effect on accuracy, the significant saving of effort of assignment, and the fact that the test for the necessity of a forced move requires trivial effort, we can

conclude that the forced-assignment heuristic is a beneficial addition to the basic mapping method.

THE USE AND EVALUATION OF THE SUPPORT HEURISTIC

Finding a good partial match of a chunk to the Slate content potentially involves combinatorially large numbers of possibilities. For the small example above, there are 6 tokens in the chunk and 22 in the Slate, so there are 74613 ways to assign all tokens, 158004 ways to assign all but one, and so forth. One way to select a good partial match would be to evaluate the goodness of fit of each of these and select the best, a prohibitively extensive solution.

The Slate system constructs only a single partial match in this situation, and engages in no overall comparisons of match fits at all, and so avoids the combinatorial difficulty. The heuristic which we used is the assignment selection method based on the Support table.

Why would we expect the method to work? The hypothesis is that a satisfying global result involves some combination of good local fits, and that these are reflected in the Support table well enough to guide the sequence of assignments adequately. Whether the hypothesis is correct in a particular context depends on a great many contingencies involving both the content of the Slate and the content of the chunk. The chunk must somehow "depend on the right things," and the Slate content must be developed enough so that the right information is present and suitably represented. It seems reasonable

to expect that we can assimilate effectively this way if the sequence of chunk mappings can be made to represent small steps of induction on given information.

In order to understand the use of the Support table, we examine the rules for assignment selection in detail to see how altering them affects the results. Assignments are then chosen according to the following 5 rules:

1. Assignments are tried one at a time in descending order of support.
2. An assignment with zero support is never tried.
3. No token is assigned more than once.
4. If the sum of the support values of the successful assignments does not exceed a preset threshold, then the attempt to map the chunk fails.
5. Any successful assignment is retained.*

SUPPORT AND ACCEPTABILITY

The primary influence on the acceptability of the result is the choice of the local measure used in the computation of the support matrix. The Support computations can be regarded as perfect for the machine code tasks, the oriented version of the Necker Cube task, the parsing task and Bower's task, since the first encounters of chunk and

---

* These rules are complicated somewhat for chunks which have been previously mapped into the Slate, as described in the section on remapping. However, the uses and effects of the Support heuristic are the same in both cases.

graph produce mappings in which all of the intended correspondences of graph and chunk tokens appear in the result graph, and no spurious correspondences appear.

We have varied the support table content in a number of situations in order to judge the sensitivity of the result to the rules mentioned above. In particular we would like to know the qualitative importance of the emphasis on constants, the avoidance of zero-support assignments, the threshold requirement on successful mappings and the differences from task to task in the chunks and input graphs used.

Three conditions were tried. In the first condition, the support table was replaced by a random table which had a non-zero support value for every possible pair. This was tried on various Necker graph states and on several of the machine code assimilation examples already presented. The results were always qualitatively the same:

1. Any chunk could be mapped into any graph somehow.
2. Several of the correct token correspondences were missed.
3. Tokens which should have mapped onto new tokens in the Slate were mapped onto existing tokens.
4. Several tokens were added to the Slate to represent entities for which tokens were already present.

This was in a sense a maximal perturbation of support since it simultaneously disturbed the occurrences of zero, the differential response to constants, and the order relations among pairs. The wholly unacceptable result shows that something about support is vital, but fails to show what the vital part is.

A second condition varied the degree of the emphasis on constants, comparing the normal value of 20 with values of 1 and 0. For the latter two conditions an acceptance threshold of 2 rather than 40 was used. It turned out that the first two examples tried nearly span the spectrum of possible cases. For the first mapping of the Necker Cube task, the results were entirely independent of the value of the score of arcs on constants. We can see from the graph why this is so. (See Figure 11.4). In terms of their use of constants, all of the corners of the cube are alike. It is the different ways in which the relations "is over" and "is left of" relate the corners which distinguish them. There are enough arcs on the face vertices to ensure that they are mapped before the corners. The entire result than is guaranteed, provided that the front face token of the chunk does not get mapped onto the back face token in the Slate. This depends on the support computation in the ordinary case and on a fortunate order of discovery under this modification.

A contrasting case concerns (the first Bliss example first chunk.) If the emphasis on constants is zero, then all of the support values are zero, since no two-token arcs of the chunk correspond to any in the Slate. All of the support information is in arcs on constants. Any non-zero constant emphasis, with a proportional threshold, will yield the same correct result.

A third condition involves the state of the same example in which the first mapping has succeeded normally and a second is to be attempted. Using a constant emphasis of 1 was a sufficient disturbance to allow a chunk for loop control (a construct which was not present) to map in.

The way which it mapped in suggests some simple ways in which the use of support could be improved. Examining the graph, it turned out that the Slate content before mapping and the part of the graph representing the new chunk had no arcs in common. The two parts had two tokens in common which gave enough accumulated support to satisfy rule 4 above. This could not have happened if constant emphasis were 20, since that would in effect require that there be arcs on constants in common. Since rule 4 is intended to eliminate cases of inadequate overlap of the Slate graph and the chunk as mapped, a substitute rule which accumulated realized overlap rather than potential (support) overlap would be preferable.

Aside from this bug, can we find examples of chunks which would map improperly because of the demotion of constants? Consider any three-letter word chunk being mapped into the input graph produced by three successive "noise" events in the noisy sequence retention task. The input graph carries the form of the sequence but does not bear any evidence favoring any particular word. The two arcs on "precedes" would satisfy rule 4 and thus allow the three-letter word to map in. It is reasonable to reject this mapping simply because the population of words having the same form (sequences of letters) is large, whereas the population having the same constants in order is much smaller. The same sort of mismapping would occur on sequence of instructions having only mutual references and order specified, and not the opcodes. It seems quite plausible that a wide variety of different program control structures would yield the same form, whereas the use of particular opcodes is much more specialized. So we would reject mapping of the control structure chunks into such a graph as being inadequately evidenced on the same grounds.

We thus see that emphasis of constants is vital to both the efficiency of finding mappings and correctness of the result.

EFFORT EFFECTS OF THE SUPPORT HEURISTIC

We can evaluate the support heuristic in terms of the amount of effort which it requires to do its job compared to various references. The job is to select a set of assignments which yields a high proportion of judgments of reasonableness by the user. The space in which it selects is the set of all sets of assignment pairs which obey the rule that a token is assigned at most once.

The first thing we notice is that this space is combinatorially large, even for small problems. For n tokens in one graph and m in the other, the number of possible assignment sets is

$$\sum_{j=0}^{min(n,m)} \binom{m}{j} \binom{n}{j} j!$$

where the number of assignments to produce one member of the set is j. Counting assignment operations is a convenient measure of the overall effort. The corresponding formula for the number of assignments Q necessary to serially examine the set of assignment sets is

$$Q = \sum_{j=0}^{min(n,m)} j \binom{m}{j} \binom{n}{j} j!$$

For the small example used to illustrate support table computation, Q > (7 * 1018). Other examples in this thesis have much larger values of Q.

Q is a kind of upper bound reference for evaluating effort, since it expresses the number of assignments of a simple program which is sufficient in principle to do the generation part of the effort (but which does not evaluate).

A second reference is the number of assignments which achieve the correct answer directly. This is a lower bound on effort. The problem of selecting an acceptable set of assignments and the problem of deciding that no set of assignments is acceptable require different amounts of effort. It turns out that our efforts to minimize one have also tended to minimize the other.

We would like to evaluate the Support heuristic in isolation to understand how it affects the effort requirement of the system. However, the selection process is organized so that the support table, the assignment operator and the assignment selector operate cooperatively. The support table acts as a generator of assignments and the ASSIGNER as a filter, and we agree to accept the set of assignments based on a threshold criterion. These three interact in the total effort, and so we will evaluate them together.

The effort requirements of a representative case, the integer sum program previously introduced, is shown in tables 13.5 and 13.6 below.

	Count	Time
Chunks Accepted	10	39

Chunks Rejected due to Remapping	23	155
Chunks Rejected for other Reasons	4	16
Total Count	37	

Table 13.5 - Mapping Efforts by Chunk Fates

	Count	Time
Successful Assignments*	20	6.3
Assignments Failing due to Remapping*	126	2.2
Assignments Failing for other reasons*	227	81.6
Total Count	373	

Table 13.6 - Mapping Effort by Assignment Outcomes

We see that there are few successful assignments, only 20 to successfully map 10 chunks and reject the others. Since there are many more than two vertices per chunk, we see that forced assignments are very effective as a means for avoiding search for a suitable permutation of tokens. The effort required to reject chunks from remapping in places already mapped is substantial. Even so, the basic purpose of the support heuristic, to limit the search for a permutation of tokens, is fulfilled, since only about 10 assignments per chunk are tried.

---

* These do not include "forced assignments."

We thus conclude that the support heuristic and the forced assignment heuristic are effective means for limiting the effort of graph matching.

When serial input information is being assimilated, it is possible to include some vertices in the Slate which never are part of any successful support computation. This "anticipation" effect can drastically reduce the effort of assimilating redundant information, making it more nearly proportional to the input information rate rather than the input event rate.

REMAPPING OF CHUNKS

After a chunk has been mapped into the Slate it is necessary to prevent mapping the chunk into the same place again. At the same time it must be allowed to map into other places which may or may not overlap the first site. For example, consider the chunk for case statements in the machine code task. There are disjoint sets of vertices onto which the case statement chunk must map, one for each case statement in the source program. The chunk maps several times onto each set. In each set, the chunk maps once for each case section in the statement, with the pairing of one vertex

There is a bookkeeping relation, "maps onto", which is used to keep track of past and current successful assignments, in query memory and the Slate respectively. When a chunk mapping is started, each attempted assignment is checked to determine whether it has been made before. If a previous assignment on the same pair exists, the

assignment fails in a special way called refailing. The mapping algorithm is altered so that the target threshold of accumulated support is reduced to the score of one arc containing a constant. Further refallures are treated as failures. However, if a successful (necessarily new) assignment occurs, the support table is reinitialized, the successful assignment is retained, and no further testing for previous mappings occurs during the processing of the chunk. The previous threshold of acumulated support is reapplied.

The effect is that the system is forced directly to find one new way to map an arc containing a constant. If none is found, the effort to map the chunk fails quickly for lack of support.

Once a chunk has been mapped into the Slate, only the query formation process will keep it from being presented repeatedly for remapping. The filtering effect of the query formation process is inadequate for these purposes. This leads to a large amount of processing devoted to rediscovery and rejudgement of previously discovered chunks. This processing is avoidable. Chapter 15 discusses the details of this processing and suggests some more efficient methods.

SUMMARY OF RULES OF EVIDENCE GOVERNING CHUNK ACCEPTANCE

This section describes the basis on which chunks are rejected or accepted for mapping into the Slate. These rules determine what is regarded by the system as an acceptable change of state of the Slate. Since the effect of choosing not to consider a

chunk is equivalent to the effect of rejecting the chunk after consideration, all of the means by which selectivity is introduced are discussed together. The selective acts occur in four activities of the system which have already been introduced: query making, bulk memory search, chunk ranking and mapping.

1. Selectivity in constructing the query graph -

The query graph is constructed by copying the one-arc neighborhoods of particular vertices into query memory. The vertices are tokens which are selected as follows: Knowledge of the number of successful mappings onto each vertex is maintained by the system. If this number is below a threshold value for some token in the Slate, then that token neighborhood is copied. These vertices are called query-class vertices. The value of the threshold is 2 for all of the experiments except the machine code and Necker cube tasks. For the latter tasks, since the number of correct mappings onto a vertex has no intrinsic limit, either the threshold is set very high or (equivalently) a queryless form of the Interpretation operator is used, letting the entire Slate content act as the query graph.

The values used for each task are high enough to access all of the chunks of interest. Thus the effect is to keep already-interpreted parts of the Slate graph out of the query for efficiency. Processing is focused on the unexplained parts of the graph.

The mechanism is not used in a selective way.

2. Selectivity in Searching Bulk Memory -

The search of bulk memory uses the constants in the query graph as starting points. All chunks having at least one constant in common with the query graph are tested. Acceptance is based on having two arcs containing constants and one other arc in common (up to substitution of tokens.)

3. Selectivity in Chunk Ranking -

Chunks are ranked by the number of different constants which they have in common with the query. Chunks having only one constant in common with the query are rejected, rather than being ranked. This element of selectivity appears very similar to the requirement during search that two arcs containing constants be matched. However it differs in not dealing with the relation names, and also in that the two arcs may involve the same constant. Despite the similarity, the second filter rejects a significant number of chunks not rejected by the first.

4. Selectivity in Mapping -

Two related methods are used for rejecting chunks during mapping. Particular token-to-token correspondences are rejected because they have zero support or because they lead to inconsistencies. A mapping, consisting of a set of correspondences, is rejected if it does not have enough accumulated support or if it leads to inconsistencies after free vertices are assigned to unmapped chunk tokens. Details of these processes are found in the Support heuristic section.

CHAPTER 14-/

TASK PROCESSING

There are two groups of operators in this set of three levels. One group is involved in all of the sequential input symbol processing tasks, including digit encoding, the interference task, and the parsing of clean and noisy letter sequences.

These operators were present in some form from the point at which the first of this group, the octal encoding task, was taken up.

The second group implements the Necker Cube reversal task discussed in chapter 11. These were added after the system development was essentially complete. The table below identifies which controlling operator was used for each of our six tasks.

Interference Task	RECALL WORDS
Digit Encoding	REPEAT SEQUENCE
Noisy Sequence	REPEAT SEQUENCE
Control Structure Discovery	REGARD2
Control Structure Completion	REGARD2
Necker Cube	NECKR

Table 14.1- Highest Operator For Each Task

OPERATORS FOR SEQUENCE TASKS

There are six sequence task operators. Their names, levels and dependency relations are shown in Figure 14.1 .

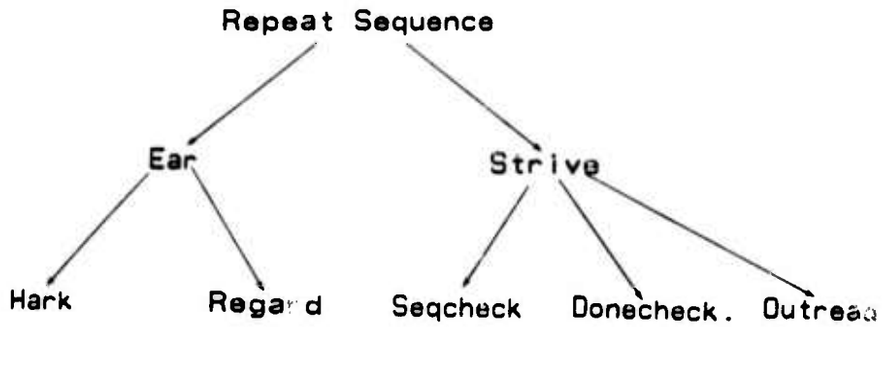


Figure 14.1 - Operators For Sequence Tasks

These operators work as follows:

REPEAT SEQUENCE

The top operator, REPEAT SEQUENCE, is invoked for the two sequence repetition tasks. It relies entirely on two operators, EAR and STRIVE, which it calls in alternating sequence, starting with EAR. REPEAT SEQUENCE has a goal of receiving and repeating back a sequence of symbols. It receives one symbol from the external source each time EAR is invoked. The symbol is given its own chunk in STM, and, as a side effect of receiving the symbol, an attempt is made to chunk the resulting STM content.

STRIVE is called upon each completion of EAR. It tests whether the current state of STM represents a complete sequence. If so, it prints the portion which it can recover from STM and sends a completion signal to REPEAT SEQUENCE.

RECALL WORDS

The operator RECALL WORDS is a minor variant of REPEAT SEQUENCE which acts similarly but gives an ordered report after termination of the sequence. Since the sequence may be incomplete, it reports by first finding the beginning of the sequence (if it is present) in STM and reporting successive symbols. If this succession does not terminate at the end of the sequence, it then locates the end and reports (in forward order) the successive symbols of the fragment of the sequence found at the end. Any other remaining symbols in STM must be in isolated middle fragments of the sequence. They are not reported. It inserts "-UH-" into the sequence presented to the user under 4 conditions:

1. It cannot find the sequence beginning.
2. It cannot find the next symbol at a non-final point in the sequence.
3. It cannot find the symbol for a token which is part of the sequence of tokens having symbols.
4. It cannot find the sequence end.

EAR

EAR calls on operator HARK to get the external symbol and put it in place as a chunk. If the symbol was anticipated, no chunking is attempted. Otherwise, REGARD is called to interpret the current State content.

Under external control, EAR may be directed to call REGARD only if the number of available chunks is below a preset threshold. If one or more chunks is mapped into STM by REGARD, the incoming chunks will usually absorb more than one chunk in the Slate, increasing the number of available chunks. For example, several one-letter chunks may be absorbed into a word chunk.

STRIVE

STRIVE relies entirely on 3 operators to do its work. First, it calls on operator SEQCHECK. SEQCHECK is a dummy operator which does nothing. It was expected that some verification and correction of Slate content would be performed after the accession of new chunks based on the special non-chunk task-dependent knowledge of what a sequence should be. However, the need did not develop on these tasks. Hence SEQCHECK serves as a place-holder indicating where we think such operations belong.

The second operator called is DONECHECK. DONECHECK tests for the presence of a completed sequence in the Slate. If the arcs representing sequence termination are present, STRIVE calls on operator OUTREAD to deliver as much of the sequence as it can to output, and to mark those symbols in the Slate as already spoken.

HARK

HARK interacts with the external interface of the system to receive a symbol of the sequence, which must be a vertex name or a special symbol "QUIT." In the latter case, arcs representing the completion of the sequence are stored in the token representing the previous symbol received. HARK works directly with the low level operators to modify the graph. The effect of "QUIT" is roughly analogous to representing detection

of downward inflection of a previously heard symbol.

SEQCHECK

SEQCHECK has no effects, as noted above.

DONECHECK

DONECHECK tests for a particular completion arc which is stored when the quit symbol is received. It uses only low level operators.

OUTREAD

OUTREAD is an operator which takes the name of a sequence to be presented to the user, and prints as much of it as it can. It inserts "-UH-" into the sequence presented to the user under 4 conditions:

1. It cannot find the sequence beginning.
2. It cannot find the next symbol at a non-final point in the sequence.
3. It cannot find the symbol for a token which is part of the sequence of tokens having symbols.
4. It cannot find the sequence end.

DSETUP

There is also an operator DSETUP which prompts the user for the particular vocabulary of the sequence tasks.

OPERATORS FOR NECKER CUBE TASK

The Necker cube task is the only one for which special single- task operators were developed. The basic reasons for doing so were:

1. the need to represent attention input,

2. the need to allow information acquired from the input interface (rather than from chunking) to be deleted as a result of later processing (in this case, a shift of attention.)

The features which distinguish the Necker cube task operators are not specific to this task. We expect them to reappear on a variety of other tasks involving attention or visual motion, as described in chapter 17.

The dependency and levels of the three operators used by this task are shown in figure 14.2 .

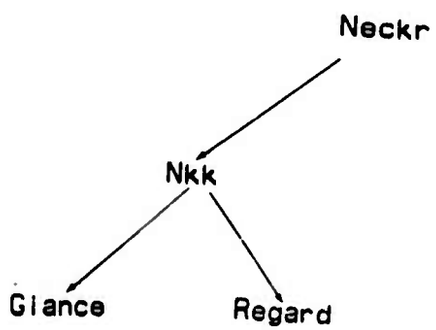


Figure 14.2 - Dependencies Among Necker Cube Task Operators

Operators NECKR, NKK and GLANCE are directly analogous to Repeat Sequence, EAR and HARK of the sequence group respectively. NECKR processes the entire task, NKK

alternates input and interpretive phases, and GLANCE performs a single input operation. The group is simpler because the task is non-terminating, so that NECKR never has to worry about quitting, and because there is no output part, the graph states which the system passes through being the objects of interest. Details of differences between the treatment of this task and the treatment of other tasks are in the description in the chapter on the task.

OTHER OPERATORS

All of the operators of the Slate system have been described above except those used in initialization and debugging. Every level has an operator called "Clear," which clears its storage and the levels below.

TASKS WITHOUT TASK OPERATORS

There are no operators for the machine code tasks. These tasks were performed by simply putting the correct graphs into the Slate and the bulk memory and calling on the Regard2 operator. The result graphs for these tasks are the final states of the Slate.

SYSTEM USE OF TIME

CHAPTER 15 - I

This chapter presents a snapshot of the current state of the Slate system and its use of time. This is a view of the system which is relatively independent of the other kinds of descriptions in the thesis, and thus is useful for understanding the system. It also turns out to be useful as a basis for considering system performance limits, changes, and understanding the nature of assimilation tasks.

Time usage is sensitive to many details of system implementation. We will see below that the current system is not representative of what can be accomplished in the time actually used. Since almost all of the system time use on some problems is avoidable, the use of time provides a better insight into potential improvements than into ultimate limits.

A detailed model of time use for one task is first developed and projected to asymptotic large-task behavior. Methods for avoiding particular components are described and their effects modeled and used to develop some expectations for other assimilators.

Time information was gathered by special programs available for timing Bliss programs, using the 10-microsecond resolution clock which is part of the CMU PDP-10 computer. The basic unit timed by these programs is a Bliss routine. Information is available on the number of calls, the amount of time spent in the routine and the cumulative amount of time spent in the routine and those which it calls.

A MODEL OF TIME USE

In order to develop a more detailed understanding of time use we have selected the control structure discovery task for analysis, using the first major program example from chapter 9 as a representative case.

Equation (1) below represents the basis categories of time use.

Total time used = Search time + Ranking time +

Mapping time + Unaccounted time (1)

OR

$$T = S + R + M + U \qquad (1)*$$

For that example we have the values in in Table 15.1 .

S =	= 14%
R =	= 6%
M =	= 78%
U =	= 2%
	SEC.= 100%

Table 15.1 - Time Use in Discovering Control Structure for the Program in Figure 9.1 .

---

* Query preparation time is zero because query preparation is omitted by the REGARD2 operator.

We can break down the largest element, M, as follows:

P	Preparation of Support tables	12%
	Mapping chunks, by outcomes:	
F	First-time failure	5%
FF	Remapping failure	49%
G	Successful mappings	12%
		78%

$$M = P + F + FF + G \qquad (2)$$

$$T = S + R + P + F + FF + G + U \qquad (3)$$

All of these terms are sums representing many events. We can break them down to event counts and average times. To do so it is helpful to define several event count terms:

Count Term Symbol	Interpretation	Example Value
c	(chunkings) is the count of successful mappings into the Slate.	10
s	(searches) = c + 1 since one final search fails to lead to a successful chunking.	11
d	(discoveries) average count of discovered chunks from a single search.	8.2
v	(variety) number of different chunks which map successfully in the course of the problem.	6

Table 15.2 - Count Variable Definitions

So we can reexpress the terms of (3) as follows:

Symbol	Interpretation
t1	average bulk search time
t2	average time to compute a rank for one chunk
t3	average time to prepare a Support table
f	average count of first-failing chunks per search
t4	average time to perform a first-time failure
r	average number of chunks which refail per search
t5	average time to refail a chunk
t6	average time to succeed in mapping a chunk

This yields, for the terms of (3):

S = s t1
P = s d t3
R = s d t2
F = s f t4
FF = s r t5
G = c t6

Equation (3) under this representation becomes:

$$T = s\ (t1 + (d\ (t2+t3)) + f\ t4 + r\ t5) + c\ t6 \qquad (4)$$

There is a simple view of the system's operation which explains some of the values above. Recall that when a chunk is mapped into the Slate, a kind of copy of it is made in the Slate. This copy is part of the content used to decide what chunks might map well into the Slate on all further searches. Since the chunk corresponds to its copy so well, it gets a very high rank. As a first approximation, a mapped chunk outranks all chunks which have not yet been mapped. Then the mapper must consider all of the previously mapped chunks before mapping any new one. For a particular chunk about to be mapped successfully, there are on the average half of the other chunks, (v-1)/2, which must be attempted first. Under this model we expect

$$r \approx (v-1)/2$$

In the example, r = 2.1 and (v-1)/2 = 2.5, confirming the estimator. Observation of the chunk trial sequences also confirms the appropriateness of this approximation.

PROJECTED TIME USE ON LARGE TASKS

Projection of the system time use may be useful for a variety of practical and theoretical purposes. It provides an independent means for understanding both systems and tasks. The estimates below turn out to reveal much more about the potential successors of the Slate system than they do about performance limits.

In order to project to large tasks, we use some of the characteristics of the control structure discovery task that distinguish it from others. The 17 chunks which we use represent a significant portion of the knowledge of Bliss control structures. Variations

in the use of opcodes and inclusion of decrementing loops, simple conditionals and coroutine control might double or triple the number. Still it is small relative to realistic language tasks which might be projected from the Miller-Isard experiments, for example, where the vocabulary contains thousands of words. For control structure identification, it is plausible that all of the chunks would be used in a single large program.*

We denote this total number of chunks by a (all), and assume that all chunks are mappable on the large task. We assume further that all chunks are able to pass the search tests at the beginning of the task, so that d = a and v = a. We characterize the size of the task by c, the number of successful chunk mappings, and assume for convenience s = c and r = v/2. Under these assumptions, f = 0 since all chunks map somehow. We also assume that the number of tokens in the Slate is proportional to c. Then equation (4) can be written as:

$$T = s\,(t1 + a(:2 + t3 + t5/2) + t6) \qquad (5)$$

In (5), the part of the work retained in the result is represented by the successful mappings:

$$Tu = s\ t6$$

and the waste by:

---

* We estimate that there would be a chunk for each 5 to 10 instructions in the projected condition. The examples above are deliberately richer in control structure than typical programs.

$$Tw = s\ (t1 + a(t2 + t3 + t5/2))$$

All of the search is superfluous since the outcome is constant. The term for which a is a factor represents three ways in which the entire knowledge base gets in the way of new mapping operations.*

How do the time terms vary as the size of the task, c, increases? What is the asymptotic behavior of the system?

In this asymptotic case (but not in the normal case) the search term t1 depends on the number of available chunks, which we have assumed fixed. The ranking time t2 depends on retrieval from the Slate of arcs containing the chunk's constants. Since the retrievals need access only one such arc each, the time required to perform them does not increase with Slate content size. So t2 does not vary with increasing c. The support table size is proportional to the number of tokens in the Slate, and so varies linearly with c, causing t3 to vary linearly with c. The refailure time t5 includes rejection of all of the potential mapping sites for each remapping chunk, so it varies linearly with c. Finally s = c varies linearly with c.

The asymptotic behavior is to use time proportional to the square of c. Other terms are increasing in the approach to the asymptotic region, so that over shorter spans we would expect time use to increase faster than the square of c.

---

* Of course, the selective parts that are superfluous in the limit are not superfluous on smaller tasks.

FASTER METHODS

This section presents a series of small changes to the Slate system which, in combination, would allow non-sequential tasks such as control structure identification and the cube task, to be treated as sequential tasks, with an effective query making process and chunking confined to small regions at a time.

The combination of changes can be regarded as an attention mechanism.

First we must unravel the functional circumstances which led to the use of the whole Slate content as the query for the non-sequential tasks. Consider the chunk for case statements, Figure 9.15. Since a case statement may have indefinitely many sections, it must be possible for this chunk to map into the Slate indefinitely many times, with all of the tokens except CS5 mapping repeatedly to the same places, and CS5 ranging over the sections. In order to map in repeatedly, it must be discovered repeatedly, and so the tokens on which the repeated mapping takes place must be in the query repeatedly. The method previously described of counting to an adequate number of mappings on a token, cannot therefore be used to eliminate tokens from a query. Every token enters the query, and the query becomes the Slate content. Note that this result does not depend on large-problem assumptions.

The system does not understand the iterative way that chunks like the case statement chunk are used. If we were to permanently mark the iterating vertices of the chunks, then the Postmap operator could perform all of the iterated mappings at once, simply by restoring the support for all iterated vertices after each successful mapping. Notice that in doing so we also provide a partial representation for the knowledge of

indefinite plurality. We also have provided a basis for recognizing the situation which led to infinite looping on iterative chunks in the explication tasks.

Given that all iterated mappings can be performed together, the method of counting the number of mappings onto a selected vertex can be used to locate incompletely mapped regions in the Slate. Thus the principal wastage described above, arising from reprocessing of chunks, is eliminated.* This is not a full solution for large problems, since such problems would start processing with the entire Slate content incompletely mapped and therefore part of the query. If we provide another means of limiting the size of query graphs and confining queries to connected local regions, then the bulk memory searches will be correspondingly small, independent of problem size.

The system nas no methods which respond to relationships between chunks (except for the overlap which occurs after mapping.) Yet the chunks come in semantically related groups, such as the three for case statements which always map jointly. If there were a way of using such knowledge, the corresponding search effort would be significantly lower.

None of these proposed methods reduce the generality of the system.

The combination of the above methods would result in chunking of many tokens which never pass through the query process at all. For example, suggestion of the presence of a case statement structure at a certain place would lead to mappings which

---

* We might also reduce such processing by recognizing the previous mappings during the search and discounting the overlaps appropriately.

extended beyond the query to include all of the machine code which implemented that structure. For efficiency, the processing should properly proceed from existing chunkings rather than broad memory searches, reserving the query making process for beginnings and impasses.

There remain in the projected system no operations within the mapping cycle which require time propc tional to problem size. We therefore expect time use to grow proportional to problem size rather than problem size squared.

In the example, the time spent on the 20 unforced token assignments which succeeded was 2.0% of the processing time. In a sense this 2.0% includes all of the useful work. The need for many of these assignments would be eliminated by iterated chunking. A speed improvement by a factor of 100 seems to be a feasible goal.

For tasks in which exhaustion of the chunk allottment of the Slate is possible, there is another kind of effort management which appears attractive. Chunking effort can be confined to conditions under which either:

1. There is a real threat of loss of unchunked information, or
2. The assimilated result is needed.

We have evaluated this idea only informally. Under the STRIVE operator, there is an internal switch (named Lazy) which controls chunking effort relative to a threshold. If, after an input has entered the Slate, there are more than 2 available chunk slots, chunking is inhibited. This often avoids the large effort required to chunk when there is marginally sufficient evidence in the Slate. In a world in which, if we are patient,

redundant information for most recognitions appears, this heuristic would lead to large savings of effort.

On some tasks, the system seemed to spend nearly all of its effort on full Slates containing marginal evidence. One new input chunk would be processed, leading to mapping of one chunk from Bulk Memory, with consequent loss of one other chunk, often a large expensive one. This seems to be a detectable overload condition, in which there is a diminished return for chunking effort, in which chunking should be inhibited. It would be better under such conditions to have some slack, perhaps using 12 chunk slots to manage the assimilation of about 7 items. Each new item could be chunked with low effort on an abundance of evidence.

TIME USE AND KNOWLEDGE

Several times during the development of the system a method was found which made the system both more accurate and significantly faster. The forced assignment heuristic was one of these. The introduction of the optimistic support sum was another. Similarly, the methods proposed above improve the representation and reduce time requirements. The dynamic support suggestion has the same character.

These suggest a pattern. Use of some kind of previously ignored information about the structure of the assimilation task makes the rapid construction of correct results possible. Slight increases in complexity yield large gains in system competence.

What are the conditions for this continual openness to significant improvement? Why does it occur for some system problems and not others? Certainly there are varieties of task domains, with corresponding varieties of structuredness. The general assimilation task turns out to be a suitably richly structured task.

The fact that the task is constructive rather than generate-and-test or enumerate-and-select means that added knowledge can be used to eliminate classes of false steps of construction rather than simply changing the nature of final stages. Keeping the task a constructive one seems vital to exploiting its structure effectively.

The inherent redundancies in the input information of assimilation tasks is helpful. Somehow, given enough clues to the correct synthesis, some useable clues are included.

There are numerous structural features of the assimilation domain which we have not taken advantage of at all. The number of such features increases as time, repeated experience, automatic chunk derivation and the like are introduced into the task. Experience in constructing the present system would indicate that increasing task complexity will be met by increasingly complex, but much faster, methods.

REINTERPRETATION OF THE SLATE SYSTEM

AS A SET OF PRODUCTION SYSTEMS

CHAPTER 16-/

For better understanding of the Slate system, this section presents an alternate representation of the work of the Regard operators. The system is represented as a set of interacting production systems. The general production system model fits well, with some striking variations from the familiar tradition of Markov algorithms, the Snobol family of languages and other systems.

A production system (ps) accomplishes all of its work by application of rules, called productions, having the form:

CONDITION ==> ACTION

where the condition specifies a matching operation and the action specifies a change to be made if the match succeeds.

We can describe a production system by answering five questions:

1. What rules can be stated?
2. Where does matching take place?
3. How is a match of a condition part achieved?
4. How does an action part act?
5. How are productions selected for match attempts?

We describe the system in slightly idealized form below, in particular omitting details which we have not sought to control in the Slate system, such as the order of retrieval of arcs.*

ORGANIZATION

The entire Slate system can be regarded as a cascade of 5 subsystems as sketched in Figure 16.1 .

The lower 4 parts correspond to the Regard operator.

The finding of an acceptable set of assignments corresponds to the matching of condition parts of rules in the chunk ps, and storing the remainder of the chunk corresponds to the action. In the arc pair ps the defined interaction pairs correspond to condition parts of its rules, the storage of inference arcs to the action parts of its rules. In the relation ps, the productions correspond to the relational property definitions presented in Chapter 12, Figure 12.2 and Table 12.2 .

The chunk ps delivers sets of arcs to the arc pair ps, the arc pair ps delivers augmented sets of arcs to the relation ps, and the relation ps delivers further augmented sets for storage. As an alternative to delivering a set of arcs down, either the arc pair ps or the relation ps may deliver a FAIL signal up.

---

* A system that worked exactly as described below might be hard to distinguish from the one we have.

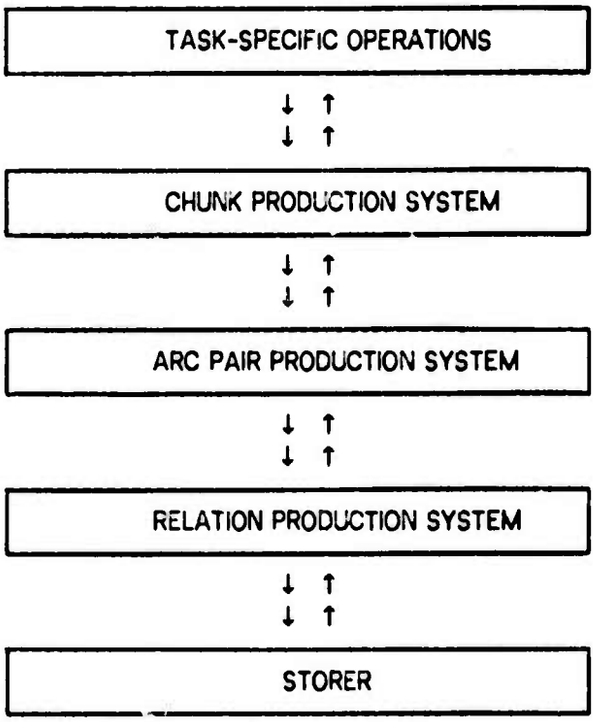


Figure 16.1 - Production System Interfaces

CHUNK PRODUCTION SYSTEM

RULES: The rules are all defined by interpreting our chunks in a particular way. Let Lc be the set of graphs obtainable by deleting some nonempty subset of arcs of a chunk C. For each member of Lc we will have a rule, and the set of rules derived from one chunk will be called a family, Fc. Each member of Lc is the condition part of one rule of the family Fc, and all of the right sides of Fc consist of chunk C itself. The rules are partially ordered: If A ϵ Lc can be obtained from B ϵ Lc by deletion of arcs, then B is higher than A.

MATCH SITE: Matching occurs in the content of the Slate.

MATCHING: The tokens of the condition parts are the match variables. Matching is successful iff tokens of the Slate can be substituted consistently one-for-one for tokens of the condition part to transform the condition part into a subset of the arcs of the Slate. Within this requirement for complete matching, the selection of Slate tokens for the match is non-deterministic.*

---

* Non-deterministic selection is selection of one item from a set of alternatives in an unspecified way. (The alternatives for this case are sets of chunk-token to Slate-token correspondences.) If a non-deterministic selection leads to a failure, the failing alternative is removed from the set and another is tried. The cycle continues until the set is exhausted or a non-failure occurs. There are three non-deterministic selections in this description, all located in the chunk production system.

ACTION: The action is to deliver to the next lower ps the set of arcs of the chunk, after substitution of tokens as follows:

For each token substituted for in the matching process, its substitute.
For each other token, an unused token.

PRODUCTION SELECTION: Selection of families is non-deterministic. Selection of a production within a family is nondeterministic among the highest rules in the partial order which in fact will match. (This reflects the fact that the system uses any consistent support.)

ARC PAIR PRODUCTION SYSTEM

RULES: There is a rule which corresponds to each defined interaction of the Slate system. (See the description of Level 3 in chapter 12.) The condition part is a connected graph containing one or two arcs. The action part is a single arc or the FAIL symbol.

MATCH SITE: Matching occurs in the union of the set of arcs delivered to the ps, the Slate content and the set of arcs produced by action rules in this ps.

MATCHING: All rules are applied in all possible ways to produce additional arcs, and to the augmented set, recursively to completion. (Completion necessarily occurs since no vertices are added, and the set of relation names is finite.) The variables of the match are all of the vertices found in the condition parts of rules.

ACTION: The arc which is obtained by substituting the values of the vertex variables of the condition part match into the action part arc is added to the set. If any rule having an action part of FAIL matches, then the action of the ps is to signal FAIL to the next higher ps. Otherwise a set of arcs is delivered to the next lower ps, consisting of the arcs input to this ps and the arcs produced by the action parts of rules.

PRODUCTION SELECTION: All productions are selected. Each is matched with its vertex variables assigned in all possible ways.

RELATION PRODUCTION SYSTEM

The relation ps has the same description as the arc pair ps, with one difference. One of the arcs in the condition part may be marked as denoting a path. If V1 R V2 denotes a path, then it matches a set of two or more arcs V1 R a, a R b, b R c, ... y R z, z R V2.

The principal differences between these two systems in the Slate system is that each rule of the relation ps always involves only one relation name, and that the relation ps is compiled while the arc pair ps is interpretive.

If the relation ps delivers a set of arcs they are stored in the Slate.

COMPARISON TO CONVENTIONAL PRODUCTION SYSTEMS

The principal differences between the pss described above and the kind represented by Markov algorithms and Snobol are:

1. The domain is graphs rather than strings.
2. There are several ps interacting, rather than just one.
3. Operations on sets predominate.
4. The chunk productions occur in families rather than singly.
5. All the chunk productions are completion productions.
6. The chunk ps is responsive to degrees of matching.
7. Production selection is non-deterministic.

How do these differences exhibit the nature of the system?*

Several of the differences express ways in which the order constraints of conventional systems are treated as inessential. The order of selection of chunks, the order of processing of arcs, selection of vertices where mapping will take place, the order of application of rules in the lower systems, all are uncontrolled. The graph proceeds to an equilibrium which is acceptable, if not unique.

Since the operands of the pss are all sets of arcs, and since the rules are all either unordered or partially ordered, the system has a highly parallel appearance. Independent rules operate on independent members of sets of arcs to yield the result.

---

*The description of the system above is entirely post hoc. The Slate system as implemented has a strong serial one-arc-at-a-time flavor, with the actions of the three parts described as production systems above freely mingled in time.

Why does the description have such a strong parallel appearance when the design and implementation were forced to be serial by the single instruction stream of the available computer?

The character of the operations is such that order of events is of little importance; what happens at one vertex of the graph and what happens short distances away are nearly always independent. When they interact, it is not in an order-dependent way. The system accumulates evidence for its assertions, and what matters is how much evidence supports a set of assertions rather than how the accumulation was compiled. Because the order of accession of arcs into a graph is not represented in the form of the graph, it is permissable to vary that order freely.

We conclude that the character of the results of chunking, sets of assertions about the data being chunked, permits many parallel operations. It is a feature which rests ultimately on the character of the task rather than on constraints of the available methods of performing the task.

We have been led to this conclusion from an analysis of the processor rather than from evidence for seriality or parallelism in people's acceptance and rejection of chunks. Whether the order invariances of the task are represented by multiple simultaneous operations when people perform such tasks is an entirely different question. However, it would be reasonable, in a search for psychological parallelism, to seek analogues of parallel operations known to be sufficient to perform the task, such as those indicated in the production system representation of the Slate system.

Because all of the chunk productions are completion productions, this set of production systems is strictly weaker than the conventional types, which have the full power of Turing machines. Arcs are never erased in this model, since they are delivered in sets which are either stored or abandoned on a FAIL signal.

NEWELL'S PRODUCTION SYSTEM MEMORY MODELS

Since we have reconstrued the Slate system as a set of production systems, we can compare it to the family of human performance models constructed by Allen Newell for cryptarithmetic, stimulus encoding and other tasks. [NS72], [N73]. In these systems, STM is a small set of linear expressions, and the productions which match and manipulate these expressions constitute the LTM. As in the Slate system, each STM element has a significant amount of internal symbolic structure. The principal contrast which causes many of the details to diverge is that Newell includes the task control information in STM, whereas in the Slate system it is in programmed task-dependent operators. His LTM content thus resembles a collection of Snobol programs much more than do the bulk memory chunks of the Slate. In the terms used in other comparative sections, Newell's match methods (like many others and unlike the Slate system) perform complete matches rather than partial matches. The fact that one uses expressions while the other uses graphs makes the representations appear to be quite different. Newell's productions must address a different mix of problems in order to perform task control.

His representations include an "absence of" operator which has no correspondent in the Slate system. The actions also include erasure of parts of STM content, which the Slate system never needs to do since all of its families of productions perform completion. These differences make some problems particularly easy for the Slate system because of its graphs, and others easy for Newell's systems because of their more general actions. On the other hand, mixing the knowledge base for assimilation with the knowledge of assimilation methods (task control) introduces complexities.

The two differ in their approach to the problem of re-matching: Newell's productions alter the match site so that a production cannot re-match, while in the Slate system the match site is marked with information about matches at that site.

The approaches to retention of active STM content also differ. The Slate system uses inclusion of chunks in larger chunks, where Newell uses rearrargement of the sequence of STM elements. The most recently matched expression is moved to the front of the STM sequence, and all deletions for overflow are from the back. These mechanisms surely have different interference properties and different ways of representing rehearsal.

Although both are interprable as production systems, the two systems represent very different approaches to memory modeling, each with considerable potential for further development.

SUMMARY

The Slate system can be represented as a set of production system without serious misrepresentation of its processes or effectiveness. It is seen as a cascade of dependent production systems which differ primarily in the localness of their fields of action.

Some of the differences between this system and conventional ones are important to its effectiveness.

The highest-level production system is responsive to degrees of matching.

The use of partial match techniques results in productions being grouped into indivisible families rather than occuring singly.

The use of graphs rather than strings strongly simplifies nearly all of our assimilation tasks.

Operations on sets predominate.

Only completion productions occur at the highest level.

This representation brings out the strongly parallel nature of the chunking processes and task. Parallel processing appears suitable for assimilation tasks.

GENERALITY

CHAPTER 17-/

This chapter examines the prospective capacity of the methods used in the system to perform effectively on other problems. The class of problems of interest is the broad class of assimilation problems. We would like to be able to describe beforehand the results of applying these methods to other assimilation problems.

The general pattern of applying a problem solver to an assimilation problem is indicated in Figure 17.1 .

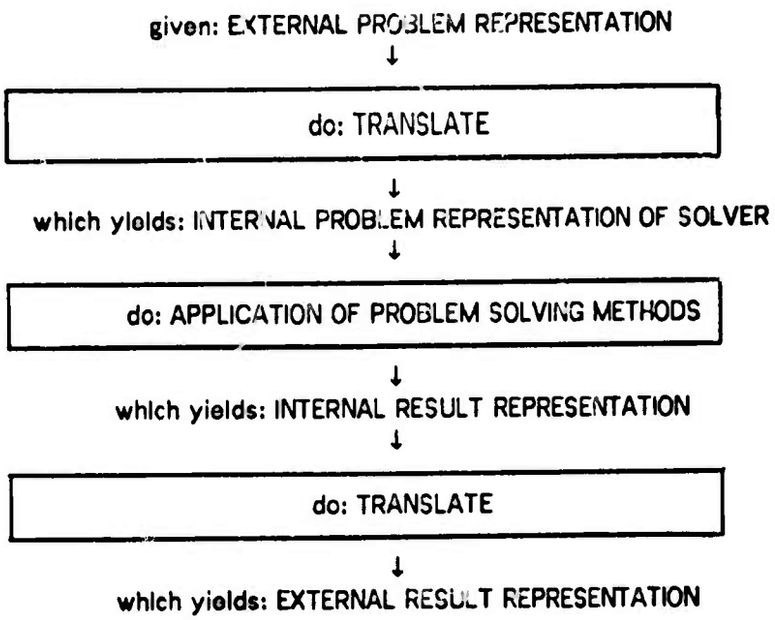


Figure 17.1 - Steps in Applying a Problem Solver

The generality of particular assimilation methods can be considered in terms of the limitations imposed on the three kinds of action indicated in Figure 17.1: translation in, method application and translation out.

These actions are instantiated in our tasks as follows:

a. Translation of the external problem representation into Slate system internal representation:
There are three parts here:
1. Invention of the relational code to be used internally. This includes selection of names for relations and constants, selection of relational properties.
2. Encoding of the knowledge into chunks.
3. Encoding of the given assertions into input graphs.

All three of these are currently manual processes.

b. Application of the problem solving methods:
All of the tasks which we have approached with the Slate system depend primarily on the chunk mapping operator for success. Thus there is essentially one method being applied. There are several freedoms in its application, the principal ones being determination of query content and the setting of acceptance thresholds.

c. Translation to external representation:
This is a manual process which has been made very simple and straightforward by the design of the internal representation.

Five questions are of interest in judging the generality of any assimilator system:

1. How easy is it to add new tasks?
2. How much of the system is used in common across several tasks?
3. What task-dependent information is built into the system?
4. To what extent does the system's generality really rest on the process of representing problems to the system rather than the system operations?
5. What are the known limitations of the system?

EFFORT OF ADDING TASKS

The Slate system was developed on a variety of tasks, and there were changes in the principal operators as each new task was added. We would like to anticipate the magnitude and character of changes that might be needed for further increases in scope.

We have informal evidence that the Slate system is capable of accepting some new tasks with modest effort:

1. The changes necessary to add the Necker Cube task to the system's repertiore were small (a few days of programming and testing) mainly confined to the user's input area. The operators were minor variants of those which existed for assimilating noisy letter strings.

2. Each task added to the system took much less effort to achieve than the previous one.

TASK-DEPENDENT INFORMATION IN THE SYSTEM

We are concerned here with two kinds of evidence for generality:

1. A system (or parts of a system) is likely to be useful for new tasks if it is already useable on a diversity of tasks.
2. A system is likely to be useful for new tasks if it does not rely on specific features of previous tasks in an essential way, that is, if there is relatively little task-dependent information built into its methods.

We deal with both of these related kinds of evidence in this section.

Many of the other existing programs for assimilation tasks are one-task programs. For example, the SEE program by A. Guzman assimilates only line drawings of groups of blocks, given in a coordinate representation. [G68] Some or all of the knowledge of the task is built into the processes which operate on the given information. In order to have a general program for the domain, any task-specificity must be included in such a way that it does not interfere with the processing or effective organization of tasks for which it is inapplicable. One way to accomplish this is to organize the program into independent sections, one for each task, as indicated in Figure 17.2 .

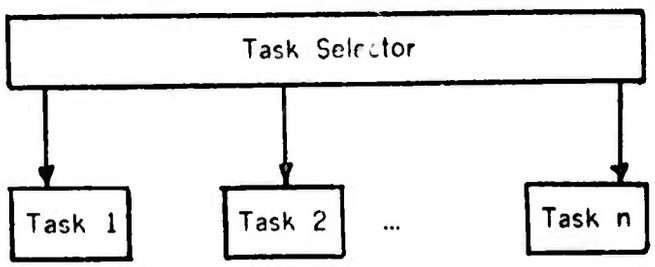


Figure 17.2 - Big Switch Generality

This approach is sometimes called the "big switch" approach because it relies on a single task selector to achieve independence of methods of task performance. It does not provide much insight into means for approaching new tasks, since it offers almost no resources for any new task to be added to the repertoire.

Organization of a system so that it has big-switch generality indicates only that several specialized methods are coexisting, not that significant spanning of a range of

tasks has been achieved. We will show below that the Slate system generality is of a different kind.

TASK DEPENDENCY OF DEFINITIONS

The defined relations used in the tasks we have described, and their use across tasks, are shown in Table 17.1 and Table 17.2 .

RELATION NAME	CLASS	TASKS				
		DIGIT CODING	LETTER-PARSE	NOISY PARSE	MACHINE CODE	NECKER CUBE
appears before	15				x	
begins with	1					
ends with	1					
executes after	15					
finishes with	1		x		x	
fulfills	33			x		
expects	33			x		
is linked by	33			x		
has subpart	3	x	x	x	x	x
identifies	26	x	x	x	x	x
is a member of	4	x				
is left of(syntax)	4		x	x		
is left of(cube)	3					x
is over	1					x
is the address in	3				x	
is the content of	3				x	
is the function of	35				x	
is the opcode of	35				x	
is the source of	35		x	x		
is the symbol of	35	x	x	x		
is the side of	35				x	
is the type of	35		x	x		
is the value of	3				x	
maps onto	36	x	x	x	x	x
precedes	1	x	x	x	x	
spoken after	1		x	x		
starts with	1		x	x	x	
has first part	1			x		
has second part	1			x		
has third part	1			x		

Table 17.1 - Relation Usage by Tasks

CLASS	CODING	PARSE	NOISY	MACHINE	CUBE	SUM
1	x	x	x	x	x	5
3	x	x	x	x	x	5
4		x	x			2
15				x		1
23					x	1
26	x	x	x	x	x	5
33			x			1
35	x	x	x	x	x	5
36	x	x	x	x	x	5
INTERACTIONS		x	x	x	x	4

Table 17.2 - Relation Class Usage by Tasks

Only 4 of the 9 classes are not used on all of the tasks. Of the 30 relations, only 3 are used in all tasks. Thus the relational classes tend to be used over and over, while the relation names tend to be task-specific. Since it is the classes rather than the names which correspond to the fixed parts of the system, we can conclude that the relation processors are being used in a general rather than a task-specific way.

The relation and vertex definitions are not part of the Slate system. They are created in a preliminary dialogue with the user before the problem is presented to the program. The relation definition schema was not developed to meet task requirements, having been implemented before any of these tasks were taken up. Except for the classes for relations which map between disjoint sets, the classes used were those developed for a very different set of tasks by Elliott. His tasks were fact-retrieval tasks representing airline reservation information, organization forms, personnel information, city locations, and others.

The new classes added to Slate really represent a correction to Elliott's set, since he used some of his classes for mapping between disjoint sets: "is the head of" mapping personal names to organization names, for example.

Elliott also found that the scheme provided more than enough classes for his needs:

> "Sufficient experience has not been obtained to know how useful each of the 32 permitted classes of relations are. Preliminary observations, however, would indicate that less than 10 classes categorize 90 per cent of the useful relations, several more are used rarely, and some of the 32 will probably never be used. At least there are some classifications for which the author has never been able to find examples."*

We can look for task-dependency using the commonalities of the set of tasks accomplished as a guide. The machine-code interpretation and explication tasks were chosen partly to exhibit some of the potential diversity of the system. The other tasks all impose various kinds of organization on strings of symbols. The machine code inputs are graphs with many arcs between vertices that are not adjacent by the "precedes" relation. Thus their input graphs are topologically unlike those of the string tasks. The explication task is unlike parsing in that it develops the details to correspond to the abstract constructs. The subject matter is also from a technical domain which is not subject to the constraints on human speech.

On the other hand, all of the problems deal with sets of qualitative (rather than quantitative) assertions which are interpretable without recourse to assertions which are not given.

---

* Elliott, page 79.

The Necker Cube task is a special case in that it has its own set of driving operators. However, the task-dependency incorporated in these operators is only superficial. Since they all have strong analogous operators in other tasks, we need only discuss their specific differences, which all involve the relationship between prior input information and newly elicited input information.

The central novelty in these operations is that new input information (about a corner being on the front face) can completely contradict older input information. For correct assimilation, the older input information and the chunk which it caused to enter the Slate must be removed. This is accomplished by causing each arc of input to be merged into the next assimilated chunk. then when another part of that chunk contradicts some input, the chunk is removed, and with it the old input.

Is this kind of action specific to the Necker Cube task? No. We would use the same strategies for a variety of other assimilation tasks in which the validity of input information was subject to decay or to refinement by allocation for additional processing. Considering other visual assimilation problems, this approach seems appropriate for:

1. Assimilation through the fovea what has previously been assimilated through the periphery.

2. Assimilating scenes which include motion or change.*

Both of these are recurrent conditions. It is therefore reasonable to regard the special features of the Necker Cube operators as representative of general features for which we happened to have only one task.

In summary, we have found serious task dependencies only in the section of the system which holds the task-dependent operators. The system is flexible with respect to topology of input graphs and the specific content of the knowledge represented. The definition methods used span a variety of kinds of subject matter and logical form. The relational classes are broadly useful, with some classes being usable for several different relations in each task. Some tasks require that there be parts of the system which reflect their particular structure, and these, of course, will be task dependent in this system, since it does not provide any general representation for the goals, methods or other information structures which might be used to define tasks. There seems to be no reason to think that this system would be incompatible with a general task representation structure such as GPS. [EN69] We expect that the methods used in the Slate system would serve effectively as tools in such a system.

BULK OF SPECIFICATION

It is well established that any computable function can, in principle, be computed by a Turing machine with suitable input, given adequate time. So we can regard the Turing machine as being ideally general over our domain of interest. However, this is

---

* this leads to an interesting hypothesis, that multistability effects may be artifacts of visual methods for dealing with motion.

generality in principle, and it does not necessarily give us workable approaches to assimilation problems. One way which the usual representation of a Turing machine departs from feasible methods is in requiring unbounded amounts of input specification. The knowledge of the problem solving method must be encoded in rather circuitous ways in the input. If we are unable to specify the problem solving method to any machine, then we are also unable to specify the proper problem input for the Turing machine.

There is a continuum of degrees of specification required by various problem solving methods. Of course, we prefer short and simple specification of a problem if it works; the classical Turing machines tend to operate at the other end of the spectrum. One of the important dimensions of evaluation of any assimilator concerns the degree of specification required in order to accomplish a task. This dimension is discussed for the Slate system in the next section.

This section assesses the amount of specification of problems which is required by the Slate system. We would like to know whether the Slate system has the kind of excess specification problem to which the classical Turing machines are subject. We would also like to know within broad limits how graph specification compares with other representations.

Problem specification for the Slate system consists of three parts: specification of chunks of knowledge, provision of input graphs, and provision of a task-specific operator or operators which uses the other operators of the system as major resources. Without specifying the particular task which we want to project, we have no way to

estimate the latter part of the specification. However, we have a basis for considering the other parts, since we can compare knowledge and input graph specifications to specifications in other media.

In particular, we can compare graphs to natural language assertions which they encode. For example, we may use various combinations of assertions in English for representing the word "SUCCESS".*

Consider how we might represent the word "SUCCESS," giving enough information so that the word could be recognized in text and included in a syntactic parsing according to a simple grammar. What do we know about "SUCCESS" which must be represented? These assertions, in some form, seem essential:

1. It is a word.
2. It has seven parts which are letters.
3. The first letter is "S."
4. The following letter is "U."
5. The following letter is "C."
6. The following letter is "C."
7. The following letter is "E."
8. The following letter is "S."
9. The following and last letter is "S."
10. It is a noun.

This English-language representation of the word has 10 assertions containing a total of 52 symbols.

---

* We use English simply as our available representative natural language. We expect the same properties of Russian or Hopi.

"SUCCESS" is a seven letter noun spelled "S", "U", "C", "C", "E", "S", "S". This description contains 14 symbols.

Assertions such as 8. above really carry 4 different items of information:

It is a letter,
It follows a certain entity,
Its value is "S",
It is part of the word.

The set of assertions refers to eight different entities, the word and the seven letters which are referred to by the pronoun "it" or by implicit reference above. We can give these arbitrary names, X1, X2, ... through X8.

Separating assertions and using explicit reference to the entity names, we have:

X1 has symbol value "SUCCESS"
X1 is a "WORD"
X1 is of type "NOUN"

X2 is a "LETTER"
X2 has symbol value "S"
X1 begins with X2

X3 is part of X1
X3 is a "LETTER"

. . .

The explicated version, still in English, contains 129 symbols, and thus is an order of magnitude larger than the compact statement above. It is rather close to an arc-by-arc reading of the corresponding word chunk.

The corresponding graph has 20 arcs, but only 34 symbols, since vertices occur only once in graphs. Here each occurrence of a relation name on an arc is counted as a symbol.

In terms of this gross symbol counting we can see that the symbol count of 34 of the graph falls in the span of some comparable representations of the same information in English, (14 to 129 symbols), and that the more efficient English representation contains about half as many symbols as the graph. Since we are not using a particularly rich segment of English here, we should expect that some specifications will be much more concise in natural language than in graph counterparts.

We can use the machine code example above to consider natural language, program source language, machine language and graphs.

> "The program computes the sum of the integers in the inclusive interval from NUM to 0. It has a routine which computes the correct result for positive parameter values, using the formula n*(n+1)/2. The program calls the routine with a positive parameter and then negates the result if NUM was negative."

Let us first compare this description, which has 59 symbols, with the program which it represents. The overlap of symbolization is quite small. The program, which contains 56 symbols, is mainly a particularization of aspects which were left unspecified in the description above, which did not even name the source language. The program does not mention the sum-of-integers notion or the fact that the routine always receives a positive parameter. It does express the existence of the routine and the two-case character of the computation expressed by "if NUM was negative."

The machine language program produced by the Bliss compiler contains 26 instructions. The instructions vary in their opcode, address and register fields, so we may consider each instruction to be 3 symbols. Thus the compiled program can be regarded as an arrangement of 78 symbols. The comparable result graph, represented

by the combination of Figures 9.12 and 9.16 has 131 arcs and a total of 202 symbols. The input graph for this example had 58 distinct symbols. (Figure 9.5). It represents part of what was expressed in the Bliss program and part of what was expressed in the compiled instructions.

Counting symbols yields only a crude measure of the bulk of specification, and our examples are further complicated by the differences between the contents of various objects above. However, it is clear that the expression of input information in graph form does not require amounts of specification which greatly exceed the amounts required in natural language or algebraic programming language.

We could argue that it is unfair to compare a natural language statement to a graph which "expresses the same thing" because the content of the natural language expression is necessarily judged after interpretation by a person. In order to make proper comparison we should consider the effects of interpretation by the graph processor.

To make this comparison we must use the graph processor as an explicator. We find a result graph that "expresses the same thing" as a particular natural language statement; then we may compare the statement with the input graph which led to that result graph. Consider the first explication task. "This is a Bliss select statement which has three sections." That statement has 10 symbols. The graph given for explication, Figure 10.1, also has 10 symbols. It is arguable whether the statement carries all of the information which is made explicit in the result graph shown in Figure 10.2, which contains 35 symbols in 47 arcs. For most Bliss programmers, it does not, since they

need not be familiar with the machine code produced.

The notion of information in two different representations "expressing the same thing" thus involves both interpreters, and some kind of parity of the knowledge presumed by each. It is interesting that in this example the symbol counts are of the same order of magnitude.

As the number of chunks in a system increases, it may be that the size of the input graph required to produce particular result graphs tends to decrease, the remainder being supplied by further explication. We would then say that the system was requiring decreasing amounts of input specification, or alternatively that its behavior corresponded to that of a more knowledgeable human.

KNOWN LIMITATIONS OF METHODS

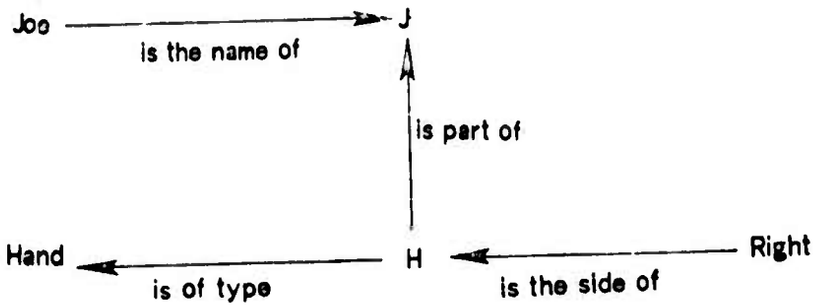
LIMITATIONS OF METHODS IN INPUT TRANSLATION

This section examines the two input translation processes which produce the graph representations of knowledge and input assertions from the external problem. Currently these are manual processes.

What constraints are placed on the representation of problems by the necessity to encode them in graphs? There are a number of underdeveloped areas of the graph notation which are made visible by attempts to represent various kinds of input. We will review a number of these, together with approaches to improving the graph

notation in order to accommodate them. For most of these, the problem can appear in translating either the knowledge base or the input assertions, depending on the particular task at hand. We will therefore discuss the two input representation processes together.

The rules for acceptance of arcs allow any arc to be inserted which is not inconsistent with arcs which are present. It is therefore easy, for example, to assert part-whole relationships of various kinds, but difficult to assert a lack of a part-whole relationship. If we want to describe a person who has no right hand, the sort of relations which we have been using will not suffice. Simply failing to assert:



will not work, since the system fills in expected parts freely (as in the explication tasks.)

A related problem involves the completeness of parts of graphs. Suppose we wish to represent that X, Y and Z, and only these, are owners of W. The joint ownership can be posited easily in simple relations, but we have no simple way of excluding other owners.

Two approaches to these problems appear attractive. The first involves structure within chunks. The chunks used in all of the examples were unstructured in the sense that any arc of the chunk had only one role, that of member. We can define a new role for arcs, that of denied arc, to prevent improper assertions and to make some desired ones. In this example, the input graph might appear as shown in Figure 17.3 .

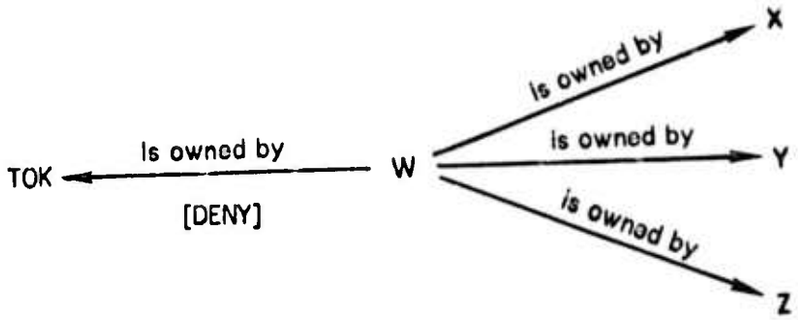


Figure 17.3 - Denial of Relation Arc

The processing which manages storage would have to prevent storage of any arc which matched a denial arc up to substitution of tokens.

A second approach involves the use of denial chunks in input or knowledge. The rule is that if the chunk can be mapped onto a set of present assertions (i.e. without adding new assertions to the graph), then the combination of assertions onto which it maps is denied. The effect would be to reject the last accession of the set.

Another minor representation problem concern the present methods of using tokens. Suppose that we want to have a chunk which will map onto any occurrence of a double letter, in tasks done in the style of the letter and digit string tasks already presented.

We might consider the chunk of Figure 17.4 to be appropriate.

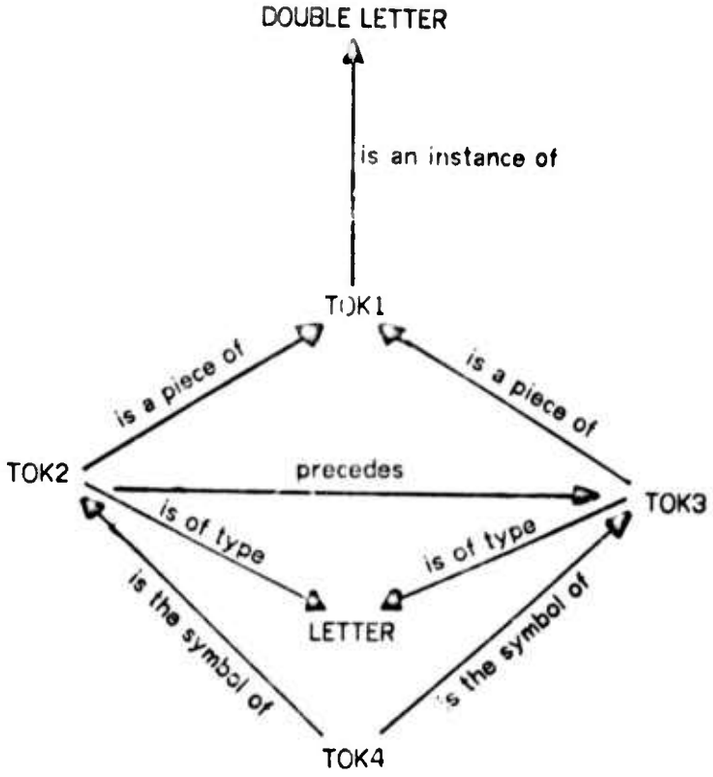


Figure 17.4 - Chunk for Recognizing Double Letters

This chunk will not work. The difficulty is that the intended range of TOK4 is certain constants, the names of the letters, whereas our mapping discipline maps tokens only onto other tokens, and constants only onto themselves. A class of vertices whose range was constants would probably remedy this problem, at a small cost of additional processing.

Many situations require description of sets having a plurality of members. "Our galaxy contains about ten thousand visible stars." The present descriptive scheme would have to be expanded to provide for such assertions. Description style for typical members of sets needs to be developed. The special feature of such descriptions is that single arcs represent multiple assertions which must be separated under some circumstances and (for efficiency) not under others.

Similarly, the notation does not yet express interval notions well. We would like to be able to say, in the machine code task, for example, that all of the code from the start of X to the finish of Y implements one contiguous source code expression. This can be done, but only very awkwardly.

These deficiencies, involving denial, token mapping range, plurality and intervals, are relatively minor and would seem to be correctable without wholesale redesign of the system. In contrast, the system has no representation for time, and no immediate prospect for dealing with time dependency notions. The present system develops consistent sets of jointly held assertions which never change their truth values. A system representing time phenomena must handle assertions whose truth values change. The present system offers a base for developing those portions of a time-dependent representation which are "instantaneous," but leaves the instant-to-instant motions to other processes. It remains to be seen whether this is a viable factorization.

This is not to say that no time phenomena can be handled by the system; the Miller-Isard examples certainly dealt with time-sequenced phenomena for which temporal order was important. We finessed the time-representation problem there by making

assertions about the compositions of sequences, using only the sorts of assertions which have single truth values.

Another interesting kind of representation problem involves exceptions to parts of entities which would normally be represented by chunks. If we have a chunk describing a "book" which asserts that it has a plurality of "pages," we should be able to use the chunk to recognize a "book without pages," a kind of exceptional book. The present notation does not provide for such exceptions.

An approach to expressing the exceptions is to provide a chunk structure role which marks arcs as being exceptions to expectations set up by chunks. Methods for dealing with such arcs would be needed for both the condition in which they appear as part of the input and the condition in which they appear as part of the knowledge.

AMBIGUITY AND REPRESENTATION OF ALTERNATIVES.

A different kind of expressive limitation involves ambiguity or alternatives of chunking the input. In the tasks which we have implemented, we see the Slate acting as a one-hypothesis memory in which a particular view of the input graph is worked out by the mapping operator under what ever task processor is in use. The fact that only single views of the input appear in the slate is not inherent in the mapping scheme. Rather it arises from the constraints used and the forms of the chunks. There was no explicit effort in the design of the system to exclude alternate valid chunkings of an input graph. It is the measures to disallow improper chunkings, sets of arcs which are

not faithful representations of the content of the input, which have turned out to disallow valid alternate chunkings as well.

For the word assembly parts of our STM tasks, only one word is allowed to contain a particular letter symbol. This constraint is enforced by the properties of the "is the symbol of" relation. In the syntactic parts, only one higher construct is allowed to contain a lower construct, as enforced by the properties of the "has subpart" relation. By excluding errors which would violate these constraints, we exclude overlapping proper analyses of the letter strings as well.

How is the particular analysis which appears in the slate chosen? Given an input graph and a knowledge base, there are two principal modes of variation of the result graph:

1. Different sequences of presentation of chunks for mapping yield different sets of accepted chunks.

2. Particular chunks may map into a graph in more than one way.

The slate system presents chunks for mapping in a rank order which is built up during the search for chunks to map. Chunks are ranked by the number of constants which they have in common with the query by which they were found. Chunks which span larger numbers of constants tend to be those which coordinate separate parts of a graph. These tend to be chosen over those which contribute only to small regions. For all of our tasks so far, this has been an effective way to choose. A previous version of the system which presented chunks in order of discovery also worked reasonably well, and saved some search effort.

Differences in chunk acquisition sequence need not result in differences in final result. Often several permutations of chunk acquisition order will yield the same result graph. Certainly for the machine code problems in which several kinds of control structure were represented in one graph, any of the control structure groups could have been recognized first without affecting the outcome.

For other tasks, the earlier selections determine later ones. If we are performing word assembly on the sequence:

THEREPEAL

we might consider the following 9 words as candidates to map in:

THE	(3)
THERE	(4)
HE	(2)
HER	(3)
HERE	(3)
ERE	(2)
REPEAL	(5)
PEA	(3)
PEAL	(4)

Many others would fail on boundary or order constraints, such as OTHER and PEAR.

Given the whole string, the system would chunk THE REPEAL since REPEAL has a span of 5 different letters. Inputting serially from the left, we might get THERE PEAL.

Others, like HERE PEA, could not occur because THERE would always look better than HERE. There is no way that two of these stable (unextendable) chunkings could coexist in the result graph memory, because the "has subpart" relation used to relate words to letters excludes the possibility.

The other kind of result variation involves differences in the set of accepted assignments. A particular chunk may be mapped into a graph in more than one way. This has been shown in detail in the chapter on the Necker Cube.

SUMMARY

This chapter evaluates evidence that the information processing methods used in the Slate system can be applied to assimilation problems in general. The methods are found to be task-independent and effective on a broad diversity of tasks. Almost every part of the system contributes to its performance on several tasks. The amount of specification required as input to the system is roughly comparable to the amount required by a programming language or natural language. The primary limits on its generality are limitations on the range of concepts which are easily expressible in the notation.

COMPARISON WITH OTHER SYSTEMS

CHAPTER 18 - I

There are several complex systems which include major parts for match specification and control. They include the SNOBOL family of languages, QA4, CONNIVER and the PLANNER and MICROPLANNER systems. It is of interest to understand their similarities and differences relative to the Slate system in order to identify its methodological contributions. We choose SNOBOL4 as representative of its family, and QA4 as representative of the latter group, which are all LISP-based. [GPP71], [R72] The control structure discovery task is a convenient assimilation problem for these comparisons.

The Slate system is similar to these other systems in that it has a fixed match process and is uncommitted on how the control of the use of the match process shall be done for particular tasks. They differ in that the match proceses of SNOBOL4 and QA4 are embedded in general programming languages, whereas the matcher of the Slate system is not. To compare a them, we consider these questions:

1. How could the match process of the language be used to find the acceptable correspondences of input data with stored chunks (corresponding to the set-of-assignments selected by the Slate system?)

2. What are the differences between the existing Slate system and the projected implementation?

3. Which differences represent inherent features of assimilation for which no appropriate control structure is provided? (These are therefore not merely limitations induced by correspondence with the Slate system.)

STRING MATCHING ASSIMILATION

In order to be able to consider input information in a definite string form, assume that a possibly incomplete assembly listing of a compiled program is available.*

The representation for an abstract object to be matched in SNOBOL4 is called a pattern. We can consider having a different pattern for each chunk to be matched against given information. Patterns may have alternatives as subparts and a subpart may itself be an arbitrarily complex pattern. The SNOBOL4 interpreter includes a pattern matcher which is used to control the search for a matching instance of a pattern in a given string. Failure of a subpattern to match leads the pattern matcher to backtrack over previous subpattern matches in order to find new subpattern matches which will cause the entire pattern to match. The overall match operation may fail or succeed. If it succeeds, there are ways to make specified complex changes in the region of the string which constitutes the matching part.

If several parts of a string are required to be equal for a pattern to match, the conventional way of expressing this is to use several subparts in the pattern. The first is given a side effect of assigning the value of its matching substring to a variable as soon as it matches. The other subparts are matches for the presence of the value of the variable. In order to discover that a particular assignment in the first subpattern is incorrect because there are no acceptable equal substrings to match the other subpatterns, the entire string must be searched, and the matching of the intervening

---

* We may assume that the form of this listing is constrained in various ways which make the task easier. The difficulties which arise are not at the level of finding the meaning of the content of the listing.

parts of the pattern must take place in order to reach the proper context for matching the latter subpatterns. This leads to gross inefficiencies in instances in which there are subparts of the given which are required to be equal but are separated by other subpatterns.

Since the information we are assimilating may be incomplete, the parts needed to match the first subpattern may be missing, leading to an inability to use the subsequent subpatterns even though the parts of the string which they should match are present. This case is detectable, but it leads to a necessity to anticipate the possibilities, and in general to enumerate the acceptable incomplete inputs as pattern alternatives. (Processing such alternatives one at a time would lead to further multiplication of effort.) The enumerated incomplete input pattern alternatives would also have to contain provisions for side effects which completed the input with information known within the pattern at success time.

The difficulties above, and several more, arise from the fact that SNOBOL4 is designed to perform complete pattern matches, whereas assimilation must occur over incomplete given information. Even the arbitrary-string pattern element of SNOBOL4, called ARB, must be located in places which are anticipated at pattern formation time rather than at match time.

There are further control complexities which we need not spell out in detail:

1. The need to make the result tend toward a best-match, not just a sufficient one. Since order of occurrence of possible matches is fixed by the SNOBOL4 interpreter, it would have to use a comparative method if the pattern matcher were in control of the match. The Slate system uses a constructive method.

2. The need to reject matches that violate constraints, such as the restriction that a particular instruction can be a part of only one kind of control structure.

These become complicated by other problems if tasks such as Necker cube chunking are considered. The fact that the statement of the given information may be found in any of several equivalent permutations must be anticipated if the given information is nominally string structured.

The Slate system gains real advantages from the differences between graphs and strings. Chunks are small connected graphs which can be explored in their entirety by short local searches. They may span what would be long intervals (of machine code, for example) in corresponding string representation. Also, they have immediate means for determining equality, since joint use of a token is the representation for equality in two assertions. All other occurrences of a token are directly accessible from the first occurrence, so that tests need not be postponed until a later rediscovery. That this is important is indicated by the very beneficial effect of the forced assignment heuristic, which makes use of this immediate access. Groups of related parts form small neighborhoods in graphs, where they may necessarily be dispersed in the corresponding strings.

On the other hand, maintaining the graph representation for strings in the current Slate system is relatively expensive.

The principal deficiencies of the SNOBOL4 match facilities for the assimilation problem arise because it performs only matches which are complete with respect to the

given pattern, that matches occur in a fixed rather than a controllable order, and that the methods for testing for equality of parts lead to serious inefficiencies.

QA4 MATCHING

There are a number of kinds of QA4 objects that can be used somewhat as SNOBOL4 patterns are used. A representative one for our purposes is the SETQ expression, which binds variables found in one arbitrarily complex expression to values derived from another arbitrarily complex expression. QA4 has unordered sets as one of its object types. The SETQ selects correspondences of variables with values by a matching process which recognizes the fact that there may be multiple ways to select the correspondence, as in the following example:

> EXAMPLE: (SETQ (SET ←X ←Y) (SET 1 2) BACKTRACK).
> Evaluation of this expression will establish a backtrack point and bind X to 1 and Y to 2. Failing back to the SETQ establishes the binding X to 2 and Y to 1. If a failure occurs once more in the subsequent program, no new binding is possible and the statement fails.

They all share two of the chief drawbacks of SNOBOL4, that complete matches (assignments of all variables) are being performed, and that the interpreter tries alternatives in a fixed order not subject to program control. The handling of incompleteness is thus forced outside of the matcher, as it was in the SNOBOL4 case. There is no opportunity to seek the best match first while letting the match controller do the selections of assignment. An assimilation program which used the match controller for its basic correspondence testing would necessarily have to use comparative rather than constructive means for finding good ways to map chunks, with

disastrous enumerative consequences. While the QA4 match processes might be useful in other ways in a reimplementation of the Slate system, they are inappropriate for the top level matching for assimilation.

We have seen in the examples above that the differences between complete match methods and partial match methods which seek "good" matches are pervasive, and that there seems to be no immediate advantage in trying to use the complete match control structures for selective construction of partial matches. Minsky and Papert have noted related consequences in studying the speed of various match methods: [MP69]

> GLOOMY PROSPECTS FOR BEST MATCHING ALGORITHMS
> The results in ... showed that relatively small factors of redundancy in memory size yield very large increases in speed, for serial computations requiring discovery of exact matches. Thus, there is no great advantage in using parallel computational mechanisms. ...
> But when we turn to the best match problem, all this seems to evaporate. In fact, We conjecture that even for the best possible (comparand) ... pairs, the speed-up value of large memory redundancies is very small, and for large data sets with long word lengths there are no practical alternatives to large searches that inspect large parts of the memory.
> We apologize for not having a more precise statement of the conjecture, or good suggestions for how to prove it, for we feel that this is a fundamentally important point in the theory of computation ...

It is clear that the differences between complete and partial match methods are pervasive, and that both have significant places in a theory of general intelligence.

CONCLUSIONS

CHAPTER 19 - 1

The principal contributions of this thesis toward a general theory of intelligence are summarized below. We have defined the class of "assimilation problems," which are problems of making available information useful by application of prior knowledge, and have developed a representation and a process (the Slate system) for solving some of them. The resulting theory contributes to psychology as a model of human short term memory, and to information science as an effective collection of new general methods. The vehicle for study is a computer program, the Slate system, which manipulates knowledge and experience represented as labeled directed graphs. As a model of short term memory, it explains how people group information into meaningful units, (chunking,) as they do when hearing familiar phrases or grouping digits according to a code. It also explains why their capacity for remembering items over short intervals varies with the task content, according to the "shared workspace effect." The presence of items belonging to unassembled groups reduces people's capacity to remember assembled groups. The larger the group size, the more of their capacity must be used for unassembled partial groups. The variation in capacity is explained as the use of a small memory having a fixed capacity of items, shared between chunked and unchunked items.

The model also identifies serial behavior in assimilation as a restriction against holding two mutually- and locally- contradictory hypotheses in memory at once. In the Slate system this is part of the memory function itself, and the processes of knowledge search and application are potentially highly parallel.

The system performs assimilation tasks with features that other models are not able to deal with. It operates on given information which represents several different objects presented together, with each being incompletely specified. This contrasts with process models of human perception which must receive their stimulus information one whole stimulus at a time. The system is responsive to the syntactic organization of incompletely specified (noisy) information representing word sequences. It is able to analyze, hold and repeat grammatical noisy sequences which are much longer than the longest ungrammatical sequence which it can repeat.

The model embodies a useful new representation for concepts and experience. The representation is concise and applicable to a wide variety of tasks. It has been used to express knowledge which relates computer instruction sequences to control structure constructs in a programming language, to represent the Necker cube and model its "reversal," to represent codes for grouping binary digits, the letter structure and syntactic categories of English words, and simple grammatical constructs. All of these have been applied to assimilate suitable given information.

The process for assimilation relies on a match process which differs from conventional ones in that it seeks a good partial match of a pair of graphs, allowing both to have unmatched parts. There is a copying process which transfers and particularizes the knowledge in the concept chunk into the Slate with each successful match. To combat the combinatorial difficulty of selecting a good partial match on two graphs, several directive heuristics have been developed. The match is a constructive process with its backtracking confined to a very local context involving a single assignment.

The present implementation has been pushed to rather soft limits on speed and ability to represent phenomena. It is clear that a much more capable system could be developed on the basis of present knowledge, and that the methods developed can be used as a basis for dealing with a wide variety of other phenomena and tasks.

BIBLIOGRAPHY

[A68] Anderson, J. A., FRAN: A Simulation Model of Free Recall, in **Psychology of Learning and Motivation 5**, G. Bower, ed., Academic Press, New York (1972).

[A71] Attneave, F., Multistability in Perceptic , **Scientific American 225**, 6 (December 1971), 62-71.

[BGST73] Bigelow, R. H., Greenfeld, N. R., Szolovits, P., Thompson, F. B., Specialized Languages: An Applications Methodology, California Institute of Technology REL Report Number 7, (February 1973).

[B69] Bower, G. H., Chunks as Interference Units in Free Recall, **JVLVB 8**, (1969), 610-613.

[BW69] Bower, G. H. and Winzenz, D., Group Structure, Coding, and Memory for Digit Series, **JEP Monograph Vol. 80, No. 2**, Part 2. (May 1969).

[BF71] Bransford, J. D. and Franks, J. J., The Abstraction of Linguistic Ideas., **Cognitive Psychology 2**, (1971), 331-350.

[C65] Chomsky, N., **Aspects of the Theory of Syntax**, MIT Press, Cambridge (1965), 9.

[E61] Eastman, P. D., **Go, Dog, Go!**, Beginner Books Division of Random House, (1961).

BIBLIOGRAPHY

[E65] Elliott, R. W., A Model for a Fact Retrieval System, Ph.D. Thesis, University of Texas, (May 1965).

[EN69] Ernst, G. and Newell, A., **GPS: A Case Study in Generality and Problem Solving**, Academic Press, New York, (1969).

[F59] Feigenbaum, E. A., An Information-Processing Theory of Verbal Learning, Ph.D. Thesis, Carnegie Institute of Technology, (1959).

[FS62] Feigenbaum, E. A. and Simon, H. A., Generalization of an Elementary Perceiving and Memorizing Machine, **Information Processing 1962: Proceedings of IFIP Congress 62**, Munich, (1962), 401-406.

[GS67] Gregg, L. W. and Simon, H. A., An Information-Processing Explanation of One-Trial and Incremental Learning, **JVLVB 6**, (October 1967), 780-787.

[GPP71] Griswold, R. E., Poage, J. F. and Polonsky, I. P., **The Snobol4 Programming Language**, (2nd Ed)., Prentice-Hall, Englewood Cliffs (1971).

[G68] Guzman, A., Computer Recognition of Three-Dimensional Objects in a Visual Scene. Ph.D. Thesis, MIT, (1968).

[M68] Martin, J. G., Temporal Word Spacing and the Perception of Ordinary, Anomalous, and Scrambled Strings, **JVLVB 7**, (1968), 154-157.

[M68B] Martin, J. G., A Comparison of Ordinary, Anomalous and Scrambled Word Strings, **JVLVB 7**, (1968), 390-395.

[M56] Miller, G. A., The Magical Number Seven Plus or Minus Two: Some limits on Our Capacity to Process Information, **Psychological Reports 63**, (1956), 81-97.

[MI63] Miller, G. A. and Isard, S., Some Perceptual Consequences of Linguistic Rules, JVLVB 2, (1963), 217-227.

[MP69] Minsky, M. L. and Papert, S., **Perceptrons: An Introduction to Computational Geometry**, MIT Press, Cambridge (1969).

[M73] Moran, Thomas P. The Symbolic Imagery Hypothesis: A Production System Model. Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University (December 1973).

[M69] Morton, J., A Functional Model for Memory, in **Advances in Psychology 7**, (1969).

[M70] Morganstein, S., Effect of Encoding on Short-Term Memory, **JEP 86**, 3, (December 1970), 387-392.

[N66] Neisser, U., **Cognitive Psychology**, Appleton-Century-Crofts, New York, (1966),221.

[N73] Newell, A. et al, Speech Understanding Systems: Final Report of a Study Group, published for Artificial Intellegence by North-Holland/American Elsevier, Amsterdam/New York, (1973). Chapter 7.

[NS72] Newell, A. and Simon, H. A., **Human Problem Solving**, Prentice-Hall, Englewood Cliffs (1972).

[N71] Nilsson, N. J., **Problem-Solving Methods in Artificial Intelligence**, McGraw-Hill, (1971).

[PJ65] Pollack, I. and Johnson, L. B., Memory-Span with Efficient Coding Procedures, **American Journal of Psychology 78**, (1965), 609-614.

[Q64] Quillian, M. R., Semantic Memory, Ph.D. Thesis, Carnegie-Mellon University Computer Science Department, (1966).

[Q69] Quillian, M. R., The Teachable Language Comprehender: A Simulation Program and Theory of Language, CACM 12, No.8, (August 1969), 459-476.

[R72] Rulifson, J. F., QA4: A Procedural Calculus for Intuitive Reasoning, Ph.D. Thesis, Computer Science Department of Stanford University. (November 1972)

[SF64] Simon, H. A. and Feigenbaum, E. A., An Information-Processing Theory of Some Effects of Similarity, Familiarization, and Meaningfulness in Verbal Learning, JVLVB 3, (October 1964), 385-396.

[SG73] Simon, H. A. and Gilmartin, K., A Simulation of Memory for Chess Positions, Cognitive Psychology 5, No. 1, (July 1973), 29-46.

[S70] Shulman, H. G., Encoding and Retention of Semantic and Phonemic Information in Short-Term Memory, JVLVB 9, No. 5, (October 1970), 499-505.

[SF61] Simon, H. A. and Feigenbaum, E. A., A Theory of the Serial Position Effect, The RAND Corporation report P-2375, (July 18, 1961).

[T64] Treisman, A.M., Monitoring and Storage of Irrelevant Messages in Selective Attention. JVLVB 3, (1964), 449-459.

[T68] Tulving, E., In: Dixon, T. and Horton, D. (eds.), Verbal Behavior and General Behavior Theory, Prentice-Hall, Englewood Cliffs, N. J., (1968), page 10.