A PROCEDURAL MODEL OF RECOGNITION FOR MACHINE PERCEPTION

by

William S. Havens

March 1978

Technical Report 78-3

Department of Computer Science
The University of British Columbia
Vancouver, British Columbia V6T 1W5

A PROCEDURAL MODEL OF RECOGNITION FOR MACHINE PERCEPTION

by

WILLIAM S. HAVENS

M.Sc., Virginia Polytechnic Institute, 1973
B.Sc., Virginia Polytechnic Institute, 1969

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

.....[illegible].....

.....[illegible].....

....

THE UNIVERSITY OF BRITISH COLUMBIA

March, 1978

Abstract

This thesis is concerned with aspects of a theory of machine perception. It is shown that a comprehensive theory is emerging from research in computer vision, natural language understanding, cognitive psychology, and Artificial Intelligence programming language technology. A number of aspects of machine perception are characterized. Perception is a recognition process which composes new descriptions of sensory experience in terms of stored stereotypical knowledge of the world. Perception requires both a schema-based formalism for the representation of knowledge and a model of the processes necessary for performing search and deduction on that representation. As an approach towards the development of a theory of machine perception, a computational model of recognition is presented. The similarity of the model to formal mechanisms in parsing theory is discussed. The recognition model integrates top-down, hypothesis-driven search with bottom-up, data-driven search in hierarchical schemata representations. Heuristic procedural methods are associated with particular schemata as models to guide their recognition. Multiple methods may be applied concurrently in both top-down and bottom-up search modes. The implementation of the recognition model as an Artificial Intelligence programming language called MAYA is described. MAYA is a multiprocessing dialect of LISP that provides data structures for representing schemata networks and control structures for integrating top-down and bottom-up processing. A characteristic example from scene analysis, written in MAYA, is presented to illustrate the operation of the model and the utility of the programming language. A programming reference manual for MAYA is included. Finally, applications for both the recognition model and MAYA are discussed and some promising directions for future research proposed.

TABLE OF CONTENTS

LIST OF FIGURES

Acknowledgements

"When a pickpocket meets a holy man,
All he sees is his pockets."
Anonymous

I would like to thank my doctoral committee for their invaluable guidance and perspective, Rachel Gelbart, Gordon McCalla, Michael Kuttner, Peter Rowat, Jan Mulder, and Roger Browse for many helpful discussions and criticisms, Marian Mackworth for proofreading this thesis, my housemates for their comraderie, tolerance, and understanding, especially Richard Rosenberg and Sheryl Adam for providing accomodation and transportation, my dear friend L. R. Floyd for many insights and his faithful companionship, and above all, Alan Mackworth for supervising my research, sharing my enthusiasm, and providing the constant support and faith of a friend.

This research was supported by a Postgraduate Scholarship from the University of British Columbia and a Research Assistantship from the Department of Computer Science.

CHAPTER 1: INTRODUCTION

The creation of intelligent automata has been a compelling dream of mankind for millennia. Each advancement in the sophistication of our technology has been seen as a new tool for the understanding of ourselves. Hydraulics, clockworks, the steam engine, and the telephone switchboard have each, in their time, been metaphors, taken as theories of the functioning of the mind (Rapoport, 1963). Only in the last few years, however, with the invention of the von Neumann digital computer has the realization of intelligent machines been a serious possibility. Such a possibility, encouraged by the early successes of Samuel (1963), Gelernter (1963), Newell and Simon (1963), and others, created high expectations. Unfortunately, these expectations have been maddeningly difficult to realize. In particular, we do not yet have an adequate theory of perception as part of an overall theory of machine intelligence. However, as Mackworth (1977c) points out, elements of such a theory are emerging.

This thesis is concerned with aspects of this developing theory of machine perception. This work is motivated by the belief that perception can be characterized as a recognition process guided by plans and expectations and driven by observation and experience. A theory of machine perception is seen as having two major parts - a formalism for the representation of knowledge and a model of the processes and control structures required to perform search and deduction on that representation. The distinction between representation and process is emphasized in order to point out an aspect of machine perception that has not been sufficiently developed. Presented in this thesis is a procedural model of recognition for perception. The model is intended as a computational paradigm for perception research and is based on the following characterization of perception.

Perception is a recognition process that composes new descriptions of observed experience in terms of stored stereotypical descriptions of the world. The new knowledge created in this process is abstract and relational, the formation of the description of a perceived concept. Perception is seen to exploit the sequential nature of everyday experience by assuming causal relationships among events and observations.

Perception is a non-deterministic process. Our sensory experience of the world can be ambiguous and often illusory. Likewise, the knowledge by which we interpret sensory experience is incomplete and often erroneous. Yet perception operates in this uncertain environment. The perceptual process must tolerate non-determinacy by exploiting context and allowing multiple partial interpretations to be hypothesized and their confirmation attempted concurrently.

Perception is both an active process guided by hypothesis and expectation and a passive process driven by events and sensory observation. Observations act as cues which stimulate both the formation of hypotheses and the activation of heuristic knowledge associated with specific hypotheses. Such hypothesis-specific knowledge is used to direct the recognition process by making observations, creating new expectations, and attempting to satisfy those expectations.

Perception is also a recursive process. Cues are not solely primitive observations but may be, in fact, the result of perception. The perceptual process uses the description of some successfully perceived concept as an abstract cue in the perception of higher concepts.

As an approach towards a theory of machine perception, a procedural model is presented based on these characterizations. The model provides an integration of top-down, hypothesis-driven search with bottom-up, data-driven search in hierarchical, schema-based knowledge representations. The model defines explicit mechanisms for employing recursive cue/model hierarchies in perception. Heuristic procedures, called <u>methods</u>, are used to guide the recognition process. Methods are associated with specific stereotypial schemata to drive the recognition of instances of those schemata. Methods may be applied in both top-down and bottom-up search modes and a number of methods may be active simultaneously. The model defines mechanisms of communication and coordination between concurrent methods and also defines a deductive method-scheduling technique based on the notion of computing a method's applicability to the perception process.

As an implementation of the perception model, a programming language called Maya has been developed. This programming language is designed as a multiprocessing dialect of LISP and provides data structures for constructing, manipulating, and accessing schemata-based knowledge representations. As well, Maya defines control primitives for integrating top-down and bottom-up processing. The language also provides mechanisms for creating and scheduling processes deductively and for coordinating the interaction of processes.

In presenting the model of recognition and its implementation, the thesis takes the following form: Chapter 2 reviews the contributions of recent research to the evolution of a computational theory of perception. Examined are specific representational theories, programming languages, and perception programmes. Chapter 3 presents the procedural recognition model in detail. Chapter 4 presents a small but characteristic example from computer vision to illustrate the benefits of the model and to demonstrate the utility of Maya as a programming language. Chapter 5 provides an overview of the design of Maya, a description of its features, and a small tutorial on Maya programming style. Chapter 6 re-examines other relevant artificial intelligence research from the perspective of the procedural model presented here. The relevance of this work to the study of machine perception is investigated and suggestions are given as to the possible directions of future research.

CHAPTER 2: MECHANISMS FOR MACHINE PERCEPTION

2.1 Introduction

If the dream of intelligent automata is to be realized, there must exist a body of underlying principles from which these machines will be built. The discovery of this body of knowledge will have a profound effect on mankind. Its principles will be manifest in mathematics, psychology, computer science, linguistics, philosophy, and all other branches of science concerned with human reasoning. Its ultimate implications for our society will be felt in now unimaginable ways.

An apparent convergence of ideas about the organization of memory, the understanding of language, the representation of knowledge, and the machine perception of visual images suggests that there must exist computational mechanisms governing perception. The fact that similar mechanisms are being investigated in the fields of cognitive psychology, artificial intelligence, and linguistics indicates that these underlying principles may reside not too far from the surface of our present knowledge. In this chapter, a review of some evidence from recent research supporting this view is presented.

2.2 Artificial Intelligence and Psychology

Both cognitive psychology and artificial intelligence are concerned with understanding the mechanisms of perception. The computer has given psychologists both an information processing metaphor for visualizing cognitive mechanisms and a laboratory in which to experiment with these mechanisms. In exchange, their experiments have given artificial intelligence a test of the validity of our computational mechanisms as a theory of human perception. The approach of many researchers (Rumelhart & Norman, 1973) (Collins & Quillian, 1972) (Newell & Simon, 1972) has been to propose an information-processing model of some particular aspect of perception, memory, or learning; then to compare the behavioural adequacy of the computer simulation to the behaviour of human subjects given the same task.

A significant early example of this approach is the GPS model of human problem solving (Newell,1963). The model uses a simple "back-chaining" scheme of breaking a problem down into smaller and smaller sub-problems until progress on some sub-problem can be made.

Another example is the EPAM model of verbal learning developed by Feigenbaum (1963). The model uses a discrimination tree as an associative memory for nonsense syllables. At each node in the tree, only sufficient information is retained to perform a binary discrimination test at the time the node is constructed. As more nonsense syllables are added to the network, the test becomes insufficient for proper discrimination. This leads to such retrieval errors as a failure to respond to a stimulus, confusion between similar stimuli, and oscillation between correct and incorrect responses.

An important contribution to the development of machine representations of knowledge is Quillian's (1968) proposal of semantic networks as a model of human memory. His work models memory as an arc-labelled directed graph structure in which nodes represent arbitrary concepts and arcs represent typed binary relationships between concepts. The meaning of a concept in the network is considered to be the entire network as viewed from the concept node.

As a representation scheme for machine perception, semantic nets have an appealing property. The meaning of a concept is not represented as a set of isolated facts, but as an encyclopedic network of relationships with other concepts. Although this representation is extremely rich in its structure, Woods (1975) has analyzed these relationships and points out a number of problems and misconceptions. Recently, Schubert (1975) has extended the representational power of semantic networks to incorporate logical quantifiers and connectives. Hendrix (1975) also has augmented the representation with a partitioning mechanism to incorporate quantification and hypothetical situations.

Quillian's original research and the more recent work of

Collins and Loftus (1975) have modelled human memory search as a parallel spreading activation process in a semantic network. From two concept nodes, the search proceeds in a breadth-first manner to each of their neighbours until a path intersection occurs. The types of arcs traversed during the search are supposed to represent the semantic relationship between the two concepts.

2.3 Programming Languages

A number of artificial intelligence programming languages suggest aspects of a computational theory. These languages include both a scheme for representing knowledge and a control structure scheme for operating on that representation. The most popular such language has been the partial implementation of Hewitt's (1972) Planner language, called Micro-Planner (Sussman, 1973). Hewitt's language provides a procedural realization of an incomplete higher-order logic system. In Planner, facts are represented declaratively as n-tuple patterns in a global associative database and as procedures, called theorems, associated with patterns. The language relies on three mechanisms: associative database retrieval, pattern-directed procedure invocation, and an automatic backtracking control structure. The best implementation of the original Planner proposal is the Popler language written in Pop-2 by Davies (1973) at Edinburgh.

The utility of the Planner paradigm was demonstrated by Winograd (1973a). However, as has been pointed out (Sussman & McDermott, 1972) (Hayes, 1973), there are serious problems, most notably the lack of a precise representational semantics and the dependence on automatic backtracking for generating alternative solution paths in a uniform and exhaustive depth-first manner. Backtracking reverses the side-effects of any rejected alternatives. The fact that each alternative at a decision point is treated independently is the source of the difficulty. No communication between competing alternative solutions is possible. Consequently, nothing is learned from failures. The problem is further aggravated by the intended modularity of the pattern-invoked Planner theorems. The language attempts to use all theorems matching a given pattern to achieve some goal or subgoal until one succeeds. However, each theorem is considered to be a modular method alone capable of achieving the goal. Each theorem is independent of all others and, as a result, each theorem is effectively ignorant of the efforts and methods of every other.

McDermott and Sussman (1973), in an attempt to solve these problems, designed and implemented a successor programming language, Conniver. The language supports multiprocessing by using the control structure model suggested by Bobrow and Wegbreit (1973). Conniver provides neither automatic backtracking nor automatic restoration of variables. Changes made to the database normally remain changed unless specifically restored by the programmer. This modification permits communication among "sister" processes exploring alternative solutions to a problem. Each process may consult the database to discover the results of her siblings. In order to permit processes to use hypothetical situations, Conniver provides a layered context mechanism. Any process may request a separate, experimental copy of the database. Any changes made to this new copy are not visible external to the context.

As previously mentioned, Conniver does not depend on automatic backtracking to generate alternatives. Instead, it defines a co-routine mechanism called generators which are procedures that can maintain an internal state between invocations. Generators may return multiple values in a communication port called a possibilities list. Instead of being embedded within an automatic backtracking control structure, alternatives are explicitly represented as data items in the possibilities list. Conniver also provides primitives for manipulating the possiblities list and recalling generators.

Conniver's authors intended to improve AI programming language technology by repairing some of the problems encountered in the use of Micro-Planner. Conniver has also provided some representational mechanisms decidedly more powerful and flexible than those realized in Micro-Planner. Conniver permits the representation of hypothetical worlds and allows arbitrary properties to be associated with patterns. In the next chapter, the utility of this last mechanism will be investigated.

Conniver does not propose a model of computation in its design. In fact, it defeats the primitive model of Micro-Planner. However, the language does support the creation and manipulation of multiple processes thus providing a capability, if not the facility, for using bottom-up search mechanisms. A second contribution of Conniver is its use of the possibilities list to represent processes as data structures to be manipulated by other processes.

A very recent programming language, KRL-0, has been proposed by Bobrow and Winograd (1977). They explicitly propose a model of recognition for machine perception based on a schematic representation and a notion of schema matching. KRL will be discussed in more detail in the next section.

2.4 Representation of Knowledge

Suitable mechanisms for the computer representation of knowledge are a major aspect of a theory of machine perception. The search for representations exhibiting desirable properties for perception has been an important research effort.

2.4.1 Logical Representations

First-order predicate calculus has been advocated by many as a computational paradigm for Artificial Intelligence

(McCarthy & Hayes, 1969) (Green, 1969). Predicate calculus offers the advantages of both a completely modular representation and a precise and formal semantics. All knowledge is represented factually and is specifically divorced from the proof procedures used to perform search on that representation. A number of proof procedures have been advocated, most notably the resolution principle of Robinson (1965). For the most part, these proof procedures are syntactic mechanisms utilizing a uniform interpreter. There is no general concept of process and control inherent in the logic system itself.

A number of strategies have been proposed for controlling the search process in predicate calculus systems, including dynamic pruning of the search space and attaching domain-specific heuristic procedures to axioms of the system. Reiter (1973) has advocated the use of a model to restrict the search space and to give advice to the proof procedure.

A number of researchers have advocated predicate calculus as a programming language (Kowalski, 1974) (VanEmden, 1977). In most implementations, a uniform proof procedure is used as the control structure model for the programming language. In an attempt to introduce logical semantics into the control mechanism, Hayes (1973) is defining a language of control structure operators deducible by the logic system during execution.

These efforts point to the need to have the deductive process guided by semantic knowledge instead of relying on a uniform syntactic procedure. What is required is an integration of the representation with a model of control and process.

2.4.2 The Procedural Reformation

The problems observed in the purely logical formalism have given impetus to the development of a procedural formalism for representing knowledge. As in any serious reformation, two competing schools, the proceduralists and the declarativists, quickly delineated their respective points of view thereby radicalizing those positions. A detailed discussion of these positions is outlined by Winograd (1975).

The proceduralists contend that knowledge is best represented in procedures. Their argument is that a large part of man's knowledge of the world is knowledge of process - knowing "how" instead of a factual knowing "what". The Actor formalism of Hewitt (1973) defines the extreme of this point of view. Hewitt states that his research is directed at putting semantics on a firm procedural basis. The knowledge of some entity is the behavior exhibited by the procedure representing that entity. Access to information in an actor is permitted only by sending the actor a message which it interprets by its own means. The formalism can alternatively be viewed as the decentralization of the system interpreter among the data objects of the system. Actors are a generalization of the formal notion of classes and objects introduced in the Simula programming language (Dahl, 1966).

It seems clear that method and process in general are best expressed procedurally because temporal relationships are handled automatically by the sequential nature of the representation. Procedures provide a natural way to specify interactions as operations and they are convenient for representing higher order knowledge. Winograd (1975) points out a duality between the modularity of declarative representations and the interaction inherent in procedural representations. From the declared goal of developing a computational theory of perception, that same duality can be seen as the distinction between a theory of representation - the declarative aspect, and a theory of recognition - the procedural aspect. What is needed is an integration of the modularity of a declarative representation with the interactions that are specifiable in a procedural representation. That integration cannot be a simple concatenation of techniques. Instead, there must be a synthesis that respects the inherent duality between representation and recognition, between form and process.

2.4.3 Schemata

Within the last few years, research into suitable representations of knowledge in such diverse fields as human memory research (Bobrow & Norman, 1975) (Pylyshyn, 1976), linguistics (Fillmore, 1968), and artificial intelligence (Minsky, 1975) (Bobrow & Winograd, 1977) has led towards the convergent notion of schemata. The term is attributable to the work of Bartlett (1932), although the concept has now been rediscovered under various names with many incarnations.

A general characterization of schemata includes the following aspects. Schemata are data structures for representing stereotypical concepts including objects, events, actions, situations, and sequences of events, actions, and situations. Schemata form network structures like the semantic networks of Quillian (1968) exhibiting the same rich encyclopedic organization. Each schema represents a generic concept. Concepts may be simple or complex, concrete or abstract. Complex concepts are represented as a composition of simpler schemata. Because knowledge is organized into conceptual modules, the interpretation of process can deal with large related amounts of information as single concepts, as units at a single level of detail, or, alternatively, the hierarchical data structure can be examined at a deeper level of detail when required.

Each schema is composed of a set of named relations with other schemata and primitive values. The representation also includes the notion of stereotype and instance. Stereotype schemata may be copied to yield multiple schema instances. Each schema stereotype initially may contain default values for some of its named relations. When the schema is copied to represent an instance of its stereotype, the default assignments serve two functions. First, they provide generic knowledge about the instance that must be generally true of most occurrences of the represented concept. Second, the default assignments are interpreted as expectations of what type of information may be used to replace the default values in the instance. The process of instantiating a schema instance becomes a search for particular data or embedded sub-schema instances satisfying the schema's expectations.

Schemata may contain both active and passive knowledge. In a stereotype, passive knowledge includes the expectations and default values. In a fully specified instance, passive knowledge consists of the values of the named relational variables comprising the description of the instance.

Minsky (1975) has proposed a schema-based representation which he calls <u>frame systems</u>. His work is primarily concerned with the development of schemata for computer vision knowledge representations, although he extends its applicability to other domains. Recently, Winograd (1975) has further specified the frames paradigm for use in natural language research. Schank and Abelson (1975) have developed a schema-based system for narrative story understanding called <u>scripts</u> which uses a small number of primitive actions to represent cause and effect relationships in simple narratives. Using a case parsing method (Fillmore,1968) to construct the schematic representation of a story, Schank's (1975) system can infer a paraphrase of the story including information not explicitly present in the original narrative.

Similarly, Charniak (1975) has proposed a schema-based story understanding system. In neither of these systems is the process of translation from the narrative to the schematic representation of primary concern. Their efforts are decidedly representational and assume the existence of suitable recognition mechanisms.

McCalla (1977) has recently modelled natural language dialogue using schemata. His system integrates both syntactic parsing using a case grammar and semantic analysis as message passing and interpretation among cooperating schemata.

Bobrow and Norman (1975) and Rumelhart and Ortony (1976) have presented a characterization of schemata for modelling human memory. As well, Norman, Rumelhart, et al. (1975) have proposed active structural networks as schemata for modelling memory processes in linguistic comprehension.

2.4.4 Search

A popular perspective in artificial intelligence has been to view machine intelligence as a complex search task guided by heuristic techniques (Slagle, 1971). From this perspective, recognition methods for machine perception can be characterized as having two major aspects - the development of powerful search mechanisms for particular representations and the discovery of powerful heuristics for particular knowledge domains to order and reduce the size of the search space. A number of search mechanisms have been advanced for schema-based representations.

Fahlman (1975) has advocated the use of parallel hardware. Rieger (1974) has proposed the use of unrestricted forward deduction. In the author's opinion, both of these proposals are attempts to solve the perception problem with a "bigger hammer". Although advances in the state of the hardware art may ease our programming plight, they should not be the basis of a theory of perception.

Kuipers (1975) has advocated a top-down, hypothesis driven recognition model for schema systems. In this model, schema stereotypes contain heuristic knowledge to guide the search process. As well, the stereotype's default expectations constitute hypotheses about what to look for to fill the slots of the instance. Schemata recognize instances by making external observations and by recursively calling on the efforts of other sub-schemata as subgoals. Unfortunately, this recognition scheme forces the use of purely goal driven search mechanisms thereby suffering from a number of serious drawbacks. Described below are three such deficiencies:

1. A schema must be explicitly hypothesized as a subgoal in order to recognize instances of its stereotype.

A schema may contain heuristic knowledge to guide the recognition process. In order for this knowledge to become available, the schema must be hypothesized explicitly as a subgoal by some higher schema. This forces a reliance on top-down, goal directed search strategies.

2. An ordering must be assigned to alternative hypotheses.

The top-down recognition model forces the choice of one subgoal at a time. Furthermore, the mechanism for activating each alternative subgoal is completely failure driven. Consider a schema containing a number of alternative subgoals. Which should be hypothesized first? One particular subgoal must be chosen as the most likely hypothesis and called. This choice must be made on "blind" expectation before the heuristic expertise of the subgoal schema is available to help make the decision. Each subgoal schema may contain heuristic knowledge to drive the recognition of its stereotype, yet that guidance is available only after a committment has been made to the schema as a subgoal.

3. Identical subgoals must be carried out independently.

A schema may be successful at achieving a number of its subgoals. If, however, another necessary subgoal should subsequently fail, the schema must itself return a failure to its caller. Later, the system may re-compute those identical subgoals. This behavior has been called <u>thrashing</u> (Bobrow & Raphael,1974).

Minsky (1975), anticipating this third objection, has proposed a mechanism, first used by Winston (1975), that attempts to avoid duplication of effort for identical subgoals. When a schema discovers from observation that it is not applicable to a given situation, it consults a <u>similarity network</u> which recommends a replacement candidate. The schema then attempts to map its correctly completed subgoals into the expectations of the new candidate schema and then passes control to it. This mechanism assumes both that a mapping exists between each failing schema and each next candidate and that the similarity network is sufficiently complete that relatively few inexplicable failures occur. Such surprises force the system to rely entirely on automatic backtracking to continue the search.

The above comments are applicable not only to schemata, but hierarchical hypothesis driven systems in general. Mackworth (1977b) has labelled the same phenomenon in vision research "the chicken and egg problem". Top-down, hypothesis driven search heuristically orders the search space by attempting more likely interpretations before trying less likely ones. However, heuristic ordering is not in itself sufficient to solve the recognition problem.

At the other extreme, bottom-up search is driven solely by evidence discovered from observation. Such evidence can be compared against domain specific knowledge to constrain the interpretation. Since no hypotheses need be formed, backtracking is not required. Unfortunately, bottom-up search mechanisms provide no guidance either since there is no expectation of future experience. For perception, techniques are needed which allow hypothesis directed search to give overall guidance to the recognition process, yet permit bottom-up, data-driven techniques to circumvent the inefficiencies of the purely top-down scheme.

One such approach is the use of multiprocessing to integrate top-down and bottom-up search. Kaplan (1973) has developed a natural language parsing system, GSP, based on a multiprocessing scheme. The system creates independent processes to look for each grammatical constituent in a sentence. GSP employs a priority queue scheduling mechanism and uses a <u>grammatical chart</u> as a communication mechanism between processes. The system is very flexible in that it can emulate, at one extreme, top-down recognition such as Woods' (1970) Augmented Transition Network parser and, at the other extreme, bottom-up recognition.

Recently, Bobrow and Winograd (1977), as mentioned earlier, have reported on the development of a schema-based programming language supporting multiprocessing called KRL. The language is designed as an integration of procedural and declarative forms of knowledge with a recognition model based on schema matching. In KRL, schemata are composed of modular entities called <u>descriptions</u> which may have associated procedures and attributes. A description is made up of multiple <u>descriptors</u>, each describing the schema's concept from a different viewpoint. As in Minsky's frames model, the concept of stereotype and instance are fundamental to the representation. Descriptions are basically intensional representations and may be composed into higher schemata called <u>units</u>. Units are intended as a mechanism for achieving <u>procedural attachment</u> by associating a set of descriptions with a set of procedures. As well, units possess a category type which indicates to the matcher how operations are to be performed on the descriptions contained in the unit. This semantic marker scheme provides a further level of specialization for the matching process.

Bobrow and Winograd propose a model of recognition based on an extended concept of description matching. The KRL matcher is designed to compare two forms syntactically, or at the other extreme, to drive the overall operation of the recognition process. The matcher uses both the syntax of the descriptors and domain specific knowledge encoded as semantic markers and procedural knowledge attached to units and descriptions. The model as described is essentially hypothesis driven. To avoid the problems noted with the top-down recognition model, the authors propose a multiprocessing scheme that provides a process priority queue with user-supplied strategies for scheduling and resource allocation.

2.5 Machine Vision

Recent research in machine vision has made a particularly significant contribution to the development of a theory of machine perception. The inherent complexity of vision has forced the confrontation of the problem from two major directions. First, machine vision research has expanded our understanding of the role of domain-specific knowledge in visual perception. The use of this heuristic knowledge is recognized as being essential to the perceptual process. Second, machine vision has, by necessity, been concerned with developing computational methods sufficiently powerful for this research domain. This second aspect of machine vision is particularly germane to this thesis and will be examined in this section. A more comprehensive review of machine vision can be found in Mackworth(1977b).

2.5.1 Roberts' Paradigm

The early research efforts of Roberts(1965) established a paradigm for machine vision which has provided a significant contribution towards a theory of machine perception. Roberts used a two-pass procedure to recognize scenes of simple polyhedral objects. The first pass reduced gray-level picture data to perfect line drawings from which the second procedure could perform object recognition. The ability of the first procedure to produce perfect line drawings from realistic data without performing higher-level interpretation has been doubted (Mackworth, 1977b).

From a perfect line drawing, Roberts' second procedure attempts to compute a scene interpretation using geometrical models of three prototypical polyhedra, specifically cubes, wedges, and prisms. Roberts used the predictive power of these geometric models to significantly constrain the search for a scene interpretation. He noted that the complexity of the search space could be reduced because the view of a particular prototypical object in some given picture is topologically invariant over a relatively wide range of viewpoints. Instead of searching in the picture domain for lines that belong to some polyhedron, a model can predict where in the picture to look for those particular lines. In other words, Roberts exploited the predictive ability of models to guide the recognition process.

A second contribution of his work is the use of picture cues to hypothesize particular models. The program's models are invoked by the discovery of specific cues that suggest the appropriateness of a particular model. Once a model is selected, that model directs the remainder of the recognition process by calculating, based on its partial instantiation, where in the picture to look for the remaining lines of the model prototype. If a model is found to be inappropriate, then the cue discovery process is continued to select another model. When a model is successful in recognizing a simple polyhedron, that object is "edited out" of the picture and the search for new cues resumed. This technique provides a crude mechanism for recognizing complex polyhedra as the composition of the three simpler polyhedra modelled by the program. The recognition model embodied in Roberts' program has been characterized as a cyclic process of discovering cues, activating a most likely hypothesis, attempting to verify that hypothesis, and following the consequences of a successful hypothesis (Mackworth, 1977b). On success, the recognized simple polyhedron is deleted from the picture and the process iterates.

A third contribution of Roberts' work is that the cue discovery process is realized as an ordered heuristic procedure. This procedure depends on the notion of an <u>approved polygon</u> which is defined as a view of a polygon face of any cube, wedge, or prism. The procedure first attempts to find a picture vertex surrounded by three approved polygons. If unsuccessful, it attempts to find a line joining two approved polygons. If this fails, the procedure attempts to find a polygon containing a three-line vertex. Otherwise, as a last resort, it looks for a three-line vertex as a cue.

The very early work of Roberts can now be seen to have made significant steps towards a theory of machine perception. First, he used geometric models of simple polyhedral objects to guide the recognition process. Second, he used picture cues as a mechanism for selecting a relevant model. This bottom-up search mechanism further constrained the search space by utilizing evidence discovered in the picture to select a viable hypothesis. And third, the process of cue discovery was itself a recognition task driven by a heuristic procedure.

In light of the present state of the art, Roberts' research can be criticized for a number of shortcomings. Most notably, the program uses only a single level of cue discovery. There is no notion of a hierarchy of cues and models. Cues are completely context-free discoveries that cannot themselves be the result of recognition. As well, the process of recognizing complex scenes is handled via a primitive composition mechanism which operates, not in the interpreted scene, but directly in the picture domain. And lastly, the recognition process is driven by a single global method, the iterative cycle of cue discovery, model invocation, and model satisfaction. There is no possibility of using specific heuristic methods for the recognition of particular models. A single global method must suffice for the recognition of all polygon types. Although Roberts' research can now be easily faulted, it still remains an amazing first step towards a theory for machine perception.

2.5.2 Guzman's SEE

Guzman's (1968) work diverged from the paradigm established by Roberts. Guzman's program, called SEE, attempts to partition regions of line drawings into polyhedral objects using only local corner junction information. SEE employs a two pass method. In the first pass, relational "connectedness" links are placed between adjacent regions as a function of the picture junction types that the two regions share. In order to cope with the inherent ambiguity of picture junctions, Guzman used a number of complicated inhibition rules to temper the link placing process.

The second pass attempts to compute the transitive closure of regions sharing two or more links while again using inhibition rules to moderate the process. The simple method of this second pass depends critically on the "tuned" performance of both its inhibition rules and the rules of the first pass. The first pass must create enough links so that a complete scene labelling can be obtained; the second pass method must then close enough regions so that a unique unambiguous interpretation results. Both methods, however, must be conservative enough to prevent the joining of separable objects to each other or the background. Guzman claimed that SEE performed recognition without the use of models, a divergence from the earlier paradigm of Roberts. Yet, as Mackworth (1977b) points out, the model-specific knowledge represented in Roberts' cue recognition procedure is hidden by Guzman implicitly in the complicated _ad hoc_ inhibition rules.

2.5.3 Huffman and Clowes

Huffman (1971) and Clowes (1971) later independently generalized the work of Guzman to use junction shapes appearing in the picture as cues for their interpretation as corners in the scene. By differentiating between the picture domain and the scene domain, both Huffman and Clowes reasoned that each picture junction can have only a few valid corner interpretations in scenes containing real three-dimensional polyhedral objects. Such physical constraints were seen to be unary predicates on the way a particular junction type can be labelled. As well, each such junction is further constrained by a binary relation along the picture edges it shares with other junctions. An edge must have the same scene labelling at both of the junctions defining its ends.

Clowes and Huffman significantly extended our knowledge of recognition mechanisms suitable for machine vision. Unlike Guzman, they refrained from trying to perform recognition in the picture domain using only local knowledge about junction type. Instead, they used picture junctions as cues to invoke parallel unary and binary constraints in the scene domain. They then satisfied the resulting system of simultaneous constraints by employing, in one case, a depth-first search and in the other, a breadth-first search.

Unfortunately, like their predecessor, both men neglected the virtues of using explicit object models to guide the recognition process. Instead, their models are effectively compiled into the sets of possible corner interpretations. As well, Huffman and Clowes used only primitive cues, the picture junction types given in the input data. Cue discovery is a trivial computation independent of the semantics of the particular scene being interpreted. In other words, cues cannot be recursively the result of the recognition process. Consequently, this recognition mechanism makes use neither of a hierarchy of object models, nor of a hierarchy of cues associated with those models.

2.5.4 Waltz's Algorithm

The scene analysis program of Waltz (1972) elaborated further the techniques developed by Guzman, Huffman, and Clowes. Waltz extended their approach in two important directions. First, he incorporated more knowledge specific to the visual world of toy blocks by expanding the set of junction labels used. The new set of labels included knowledge about crack edges between adjacent blocks and a crude representation of shadows. Such an expanded label set created a huge number of possible corner labellings for each junction type, thereby increasing considerably the complexity of the search space. Waltz, however, noticed that after applying to each junction type the unary constraint of what corner interpretations could appear in the "real world", the remaining set of valid labels was much reduced. Adding more dimensions to the labelling of polyhedron junctions increased the richness of the domains semantics without exponentially increasing the complexity of the search space.

Second, in order to cope with the expanded set of labels, Waltz developed a junction filtering algorithm to further constrain the search space before attempting a depth-first or breadth-first search for a global scene interpretation. This filtering algorithm is based on the notion of a consistency condition, "C", which holds true if, for every label assigned to a particular junction, there is either a matching label assignment at each labelled, neighbouring junction, or that junction has not yet been labelled.

The filtering algorithm operates by touring the set of picture junctions once in some arbitrary order. At each junction, the algorithm first attaches a list of all corner interpretations which satisfy the unary predicates for that junction type. Waltz noted that such lists were static and could be compiled once for each junction type. Next, the corner interpretations of each newly labelled junction are "pruned" against the label sets of each neighbouring junction sharing an edge with this junction such that condition "C" holds. That is, any corner interpretation of the new junction having an edge label that does not match an edge label of each already labelled neighbouring junction, is deleted. Then, in a spreading breadth-first search, each neighbouring junction prunes its label set against this junction and each of its neighbours do likewise until, once again, condition "C" holds throughout the network. The significance of this algorithm is that it requires only a single pass through the set of picture junctions. When it terminates, all inconsistent corner interpretations have been eliminated. Often, the algorithm yields a single labelling for each junction, thereby negating the need for a subsequent scene interpretation search.

Waltz both extended the use in machine vision of domain specific knowledge and introduced the use of constraint propagation techniques to the field. He demonstrated that by incorporating enough semantic information about a "blocks world" scene, an over-constrained network representation can be constructed which through the use of constraint propagation techniques can quickly yield a unique interpretation.

From the present perspective of developing a computational model of recognition for perception, Waltz can be criticized for the same deficiencies as his predecessors. His program makes no explicit use of models of the polyhedra it recognizes. Instead, it relies on the implicit knowledge of polyhedra embedded in the junction labels. Likewise, the cue discovery process is completely a context-free process. Cues are, in fact, primitive entities, the picture junctions given in the input data. Their discovery can be neither a function of the partial knowledge so far known about a particular scene, nor can they be complex abstract entities computed recursively as the result of recognition.

2.5.5 Mackworth's MAPSEE

The constraint satisfaction techniques developed by Waltz and others have been recently generalized by Mackworth (1975, 1977a) to a class of network consistency algorithms. These are shown to be more efficient search methods than automatic backtracking for search tasks which can be formulated as n-ary constraint satisfaction problems. Network consistency forms the basis of a recognition model for machine perception which applies general constraint satisfaction algorithms to networks of simultaneous constraints.

Mackworth (1977a) has recently used network consistency techniques for the interpretation of freehand sketch maps. The program, called Mapsee, interprets a hand-drawn map of an island according to the conventional semantics of cartography. The program begins by performing a very conservative partial region segmentation of the input sketch to yield a set of primary cues based on simple picture features. Cues are features derived from the sketch such as acute angles, point clusters, free-ends of lines, and junction types. These cues are then used to invoke primary models that provide partial interpretations of the map in the locale of the cue. Note that the interpretation provided by a model may be initially highly ambiguous. However, each model establishes constraining relationships with its geographic neighbours according to the cartographic semantics. The resulting network can be visualized as a hyper-graph whose nodes are pictorial objects (regions and chains of lines) and whose n-ary arcs are constraining relations derived from the models.

Mapsee then applies a network consistency algorithm to the network that progressively eliminates inconsistent interpretations for the various cartographic features represented by the models. If the conventional semantics of the models chosen is rich enough, and if a given sketch map is explicit in its representation, the resulting system is over constrained and the algorithm may converge to a single possible interpretation.

Mapsee demonstrates first that cue/model driven recognition can be combined with network consistency search techniques and that these methods may be applied to perception task domains outside the "blocks world". Second, Mapsee defines a cyclic recognition model for machine perception. Mackworth (1977b) has noted that picture segmentation requires scene interpretation and conversely that interpretation requires segmentation. He calls this phenomenon "the chicken and egg problem" for machine perception. Mapsee's initial conservative picture segmentation, although inadequate for a global interpretation, yields enough primary cues to invoke appropriate models. The subsequent constraint satisfaction among these models provides an initial interpretation which can then be used to guide a more context-sensitive re-segmentation. This process may be iterated until a complete interpretation is obtained. Third, network consistency algorithms provide a uniform syntactic control structure for searching declarative network representations. Consistency algorithms tend to converge towards a unique interpretation by focusing on those nodes in the network which remain the most ambiguous.

Each cycle in Mapsee's recognition process computes a new approximate scene interpretation that is used to drive a context-sensitive re-segmentation, thereby yielding semantically richer cues for the next cycle. This iterative mechanism is seen as a means of "bootstrapping" into an interpretation thereby avoiding the "chicken and egg problem". However, since Mapsee utilizes non-hierarchical descriptive models, cues must still be primitive features detected by the re-segmentation. They cannot be more complex entities recognized during the scene intepretation as part of a hierarchy of cues and models.

Network consistency techniques encourage the use of purely declarative knowledge representations and exhibit the familiar benefits and limitations of that representation (Winograd, 1975). Since models are realized as sets of constraining relationships among other models, network consistency is a very modular computational paradigm. New constraints and new models can easily be incrementally added and deleted from the network. As well, since all of the domain specific knowledge is embodied in the declarative models, the system is portable. It can be easily applied to other recognition tasks which exhibit a semantics expressable as a system of mutual simultaneous constraints. On the other hand, the divorce of the declarative models from the procedures used to search the network structure forces the use of a single global syntactic search method, the network consistency algorithm. No domain-specific knowledge such as heuristic search methods associated with particular models is possible. Mackworth (1977c) has noted the limitations of a uniform search method for non-hierarchical descriptive models and has advocated "exploring control strategies for schema-based theories of perception".

2.5.6 Freuder's SEER

Freuder (1976) has recently developed a recognition model for schema-based representations that is primarily concerned with the specification of control structures for machine perception. His program, SEER, recognizes a scene of a common machinist's hammer represented as gray-level video data. Freuder argues that most recognition schemes employ control algorithms which do not rely on computed partial results or the semantics of the scene being perceived. To the contrary, SEER employs the discovery of partial hammer components combined with general knowledge about hammers to guide the recognition process.

In SEER, knowledge is represented in two forms of hierarchical semantic network structures. General knowledge about hammers is represented in schema structures called GK networks, whereas knowledge specific to a particular hammer instance is represented in a schema instance called a PK network. The nodes of a GK network represent items of visual knowledge about hammers, such as handles and heads. The links between these nodes represent how these items may establish each other's recognition. On the other hand, a PK network represents a partially instantiated instance of a GK concept and inherits its structure and procedures.

Both the GK and PK networks form tree data structures. At the leaves of each PK tree are procedures which search for instances of the specific GK concept. The leaves of a particular PK tree structure represent the state of the procedural methods concerned with the recognition of that schema instance.

In Freuder's model, recognition proceeds using both top-down and bottom-up search within a PK structure. When a new feature is discovered, it is used as bottom-up evidence for the hypothesis of higher conjectures of which the feature may be part. A new PK structure is created to represent this new possible relationship. As well, the creation of new conjectures permits the top-down exploration of their subgoals thereby resulting in the creation of subordinate conjectures.

Since a number of conjectures can be active simultaneously, the control structure question centers about which conjecture to explore next. The mechanism used in SEER is the familiar priority-queue multiprocessing scheme. Conjectures are assigned a priority when placed on the queue and their priority may be changed during the recognition process. A global monitor then selects the highest priority conjecture and attempts to confirm it by activating one of its procedural methods.

The recognition model defined by SEER follows a cyclic process. A conjecture chosen by the scheduler is explored. If the conjecture is achieved, it may then be _exploited_ resulting in the hypothesis of suggested higher conjectures as new PK structures. These new schemata are added to the priority queue and the process is repeated.

Freuder's work has focused attention on an important aspect of machine perception, the control of the processor during the search process. He has combined the use of a schema-based representation with a hierarchy of cue invoked models. As well, he defines a priority queue multiprocessing scheme to integrate top-down and bottom-up search using multiple active hypotheses. SEER realizes top-down search by simulating the exploration of conjectures as subgoals and realizes bottom-up search by exploiting the consequences of successfully recognized conjectures.

As was pointed out for KRL, the use of multiprocessing to simulate parallel search suffers from a number of deficiencies. It is a syntactic, non-deterministic method of simulating parallel execution, and is inept at realizing intelligently guided parallel search. The requirement that some procedure compute a priority for a new process manifests the "chicken and egg problem" in two significant ways. First, it assumes that a procedure can assign a global priority to a process being placed on the priority queue based only on information local to that procedure. And, more importantly, this method requires that a priority be assigned to a process _before_ information is discovered in the scene to help decide which processes to run. The procedure that picks a priority for a process is, in effect, computing a non-deterministic scheduling of processes. This computation must be made before the information required to make this decision has been discovered. This mechanism operates essentially backwards. A mechanism is needed for simulating parallel search that schedules processes semantically by utilizing the discovery of particular cues during the recognition process to schedule those processes which can exploit the existence of those very cues.

CHAPTER 3: A PROCEDURAL MODEL

3.1 Introduction

This chapter presents the development of a procedural model of recognition for schema-based representations. The model is motivated by both the characterization of perception outlined in the first chapter and the current methodology of machine perception examined in the second chapter. First, an informal overview of the model will be given in order to highlight a number of its aspects. Then in the remainder of the chapter, techniques for realizing the model as a computational mechanism will be discussed in detail.

3.2 Model Overview

A theory of machine perception was characterized in the first chapter as having both a formalism for representing knowledge and a set of search mechanisms for performing recognition on that representation.

3.2.1 Schemata

In this model, knowledge is represented as schemata. A schema is a modular representation of everything known about some concept, object, event, or situation. That knowledge is manifest in three forms. First, each schema contains factual knowledge about the concept that the schema represents. Such facts form a description of the concept and may be represented declaratively, procedurally, or as some combination of data and attached procedures. Second, each schema may contain procedural heuristic knowledge to guide the search process for the schema's concept. And third, schemata form relations with other schemata thereby creating hierarchical network structures. This allows complex concepts to be represented by composition as networks of schemata and provides an encyclopedic retrieval mechanism analogous to that of semantic networks (Quillian,1968).

For example, Figure 3.1 illustrates a schema for a hypothetical vision system. The notation employed is similar to that used by Bobrow and Winograd (1977). This schema represents a stereotypical bicycle and consists of a set of named relations or slots (Minsky, 1975), each containing either a primitive value (often a name), a pointer to another schema, or an expectation indicating what type of information may be used to fill the slot. When the bicycle stereotype is used to represent an instance of a particular bicycle, the stereotype schema is copied to create a schema instance and its slots, initially

```
NAME:   BICYCLE

FRONT-WHEEL: (A WHEEL DIAMETER = (RANGE 19 27)
                      WITH (A TIRE WIDTH = NARROW)
                      TYPE = SPOKED
                      CONNECT (AND FRAME CRANKSET)
                      (TD-METHOD FIND-BIKE-WHEEL)
                      (BU-METHOD FOUND-BIKE-WHEEL))
REAR-WHEEL: (A WHEEL DIAMETER = (RANGE 19 27)

                     WITH (A TIRE WIDTH = NARROW)
                     TYPE = SPOKED
                     CONNECT (AND FRAME CRANKSET)
                      (TD-METHOD FIND-BIKE-WHEEL)
                     (BU-METHOD FOUND-BIKE-WHEEL))

FRAME:  (A FRAME TYPE = DOUBLE-DIAMOND
                (TD-METHOD FIND-BIKE-FRAME)
                (BU-METHOD FOUND-BIKE-FRAME))

CRANKSET: (A MECHANISM TYPE = CHAIN-DRIVE
                       WITH (A PEDAL-CRANK)
                       MAY-HAVE (A MECHANISM
                                    TYPE =DERAILLEUR)
                       (TD-METHOD FIND-CRANKSET))

STEERING-SET: (A MECHANISM TYPE = STEERING-FORK
                           WITH (A HANDLE-BAR)
                           (TD-METHOD FIND-STEERING-SET))

ISA:  VEHICLE

INSTANCES:  NIL
```

Figure 3.1: Bicycle Schema

containing expectations, are replaced systematically by information specific to the bicycle as it is discovered.

This stereotype bicycle schema illustrates a number of features of the recognition model. The first slot of the schema specifies that the name of the schema is BICYCLE. By naming each stereotype schema, it can be referred to either by a pointer or by simply using its name.

The next five slots in the schema represent composition knowledge about bicycles. A bicycle is composed of a front wheel, a rear wheel, a double-diamond frame, a power transmission mechanism called a crankset, and a steering fork mechanism called a steering-set. For perception, the composition relations in a stereotype schema define those structural and functional aspects of the concept that can be used to recognize instances of that concept. A bicycle is recognized by the discovery of its component parts composed in a way that represents the gestalt of a bicycle.

3.2.2 Schema Hierarchies

Schemata form hierarchical networks in two significant ways. Complex stereotypical concepts are represented by schemata which are a composition of other concepts represented by sub-schemata. The resulting hierarchical structure is called a <u>composition hierarchy</u>. This static hierarchy represents the composition of all possible instances of the class.

Figure 3.2 shows a composition hierarchy for the stereotypical bicycle schema. Bicycles are composed of wheels, a frame, and various mechanical mechanisms. Each of these generic components form a stereotypical class of objects represented by a stereotype schema. In turn, each of the these stereotypes is composed of its own generic components represented by stereotype sub-schemata. For instance, the WHEEL schema represents the class of all wheels. Each wheel instance will be composed of a tire, rim, and central hub assembly. A particular wheel will be represented by a specific tire instance of a particular type and by specific instances of the stereotype rim and hub schemas as well.

Figure 3.2 also illustrates the inclusion in the model of an <u>inverse composition relation</u> between schemata. For each stereotype schema having a composition relation with one or more sub-schemata, each of these sub-schemata have an inverse relation with that schema. This relation is usually called the "part-of" relation and is essential to performing bottom-up search within the schemata network.

Schemata form hierarchies in a second way. Each schema represents a stereotypical concept that may have many partially specified instances. These instances may themselves function as stereotype schemata each having a number of more fully specified instances. In this manner, schemata form <u>instance hierarchies</u>. At the top of an instance hierarchy is a schema representing an uninstantiated generic concept. Each of its sub-schemata

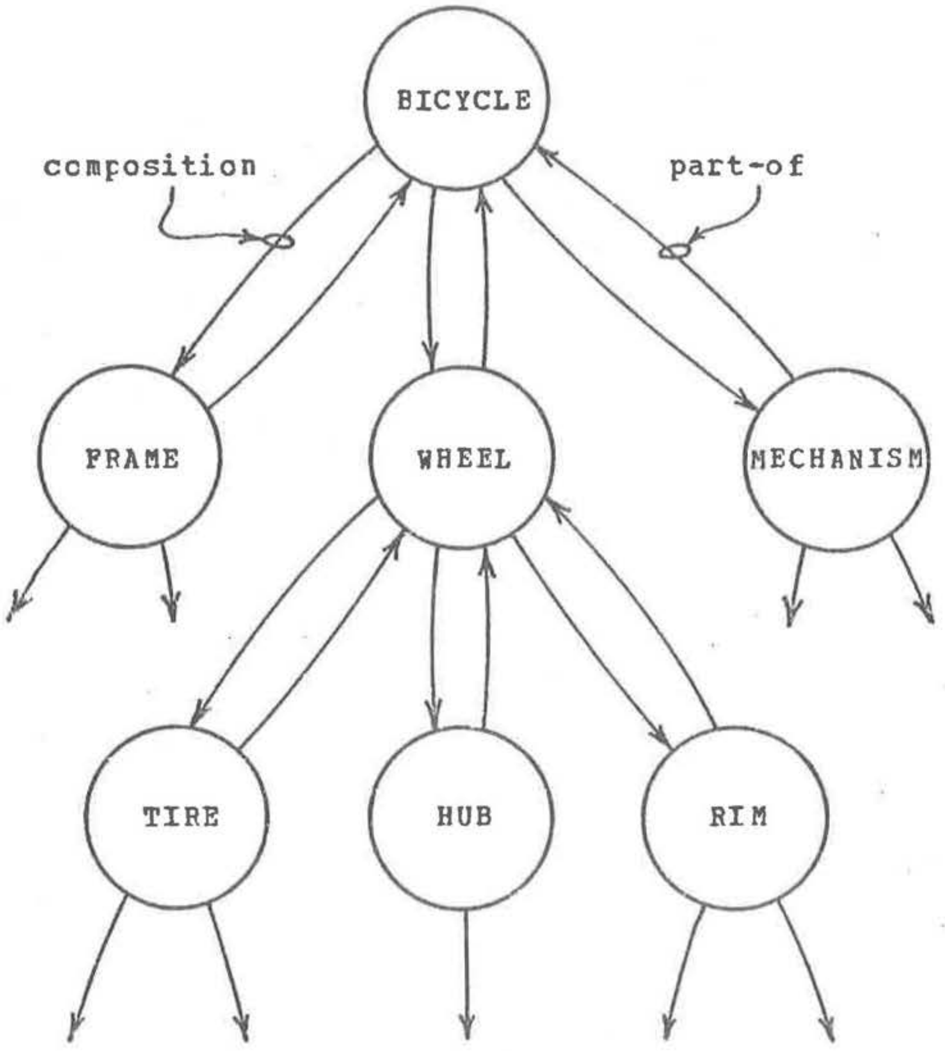


Figure 3.2: Bicycle Ccmposition Hierarchy

instances represent partially specified occurrences of that concept. Each of the descendants of these instances, in turn, represents more fully specified instances until, at the bottom of the resulting tree structure, completely specified instances become leaves of the tree. Instance hierarchies are also referred to as "ISA" hierarchies (Fahlman, 1975).

At each interior node in an instance hierarchy, the partially instantiated schema represents a non-deterministic description of a smaller class of concepts than its parent stereotype a level above. Schemata near the top of the instance hierarchy represent large classes of possible instances, whereas, schemata nearer the bottom represent smaller, more fully specified classes of concepts.

In the bicycle schema, the last two slots of the schema establish an instance hierarchy. Since bicycles are instances of the more general concept of vehicle, the ISA relation indicates that this schema is an instance of another stereotype, the VEHICLE schema. In this example, the bicycle schema has, as yet, no instances of its own, as indicated by the NIL value for the INSTANCES relation.

These two hierarchies serve different purposes. Composition hierarchies are static data structures that facilitate representing complex conceptual objects. The creation of a particular bicycle instance uses the bicycle composition hierarchy as a template from which to construct the schema instance. Each occurrence of an expectation for a frame, mechanism, or wheel in the bicycle stereotype will be replaced by an instance (perhaps only partially specified) of that stereotype.

In contrast, instance hierarchies are dynamic data structures that provide a primary associative retrieval mechanism upon which to base search over the schemata network. These hierarchies are viewed as taxonomies of concepts. Each node in an instance hierarchy is a stereotype schema that maintains an index of all its instances. For small data-bases, the schema's index can be a simple list of all its instances. For larger networks, each schema maintains an index of its sub-instances based on observable and recognizable cues. The instance hierarchy then becomes an inverted index structure for performing associative retrieval in the network. Analogous syntactic mechanisms include EPAM (Feigenbaum, 1963) and the associative retrieval of patterns in most artificial intelligence programming languages (Bobrow & Raphael, 1974). Patterns are indexed in a tree structured database by common pattern elements. A similar semantic mechanism is found in the Linnaean botanical taxonomy where, for instance, plant life is organized into a hierarchical database indexed by easily perceived cues. The cues used are physical observable properties of each class of plant life. The choice of cues is not made from _a priori_ considerations, but for convenience. Cues are recognizable features of each class that are easily observed and can function as reliable discriminators.

In this recognition model, instance hierarchies are indexed by cues that are easily recognized features of a stereotype class. For example, consider an instance hierarchy for the wheel schema. The stereotype wheel schema represents the class of all wheels including all partially and fully specified instances. For a small number of wheel instances, the database can be organized as a simple list of instances bound to a variable in the wheel stereotype. The advantage of this scheme's simplicity is balanced by the necessity of searching the list sequentially to find a particular wheel instance. Such a blind search makes no use of any observable features of the desired instance used as cues.

For larger databases, the inverted index structure is advantageous. Figure 3.3 illustrates an associative database for the wheel instance hierarchy using this scheme. Indices of the hierarchy are chosen to be readily observable features of wheels that can effectively discriminate among various classes of wheels. In this example, three different observable features of wheels are used. Neither the structure of the hierarchy nor the choice of indexed features is made from _a priori_ considerations. The choice of both structure and index is arbitrarily based on the ability to discriminate among various wheels using such available information as the type of the observed wheel's rim, the width of its tire, and its diameter. Each of these features, as in the Linnaean taxonomy, is a cue recognizable during the perception process.

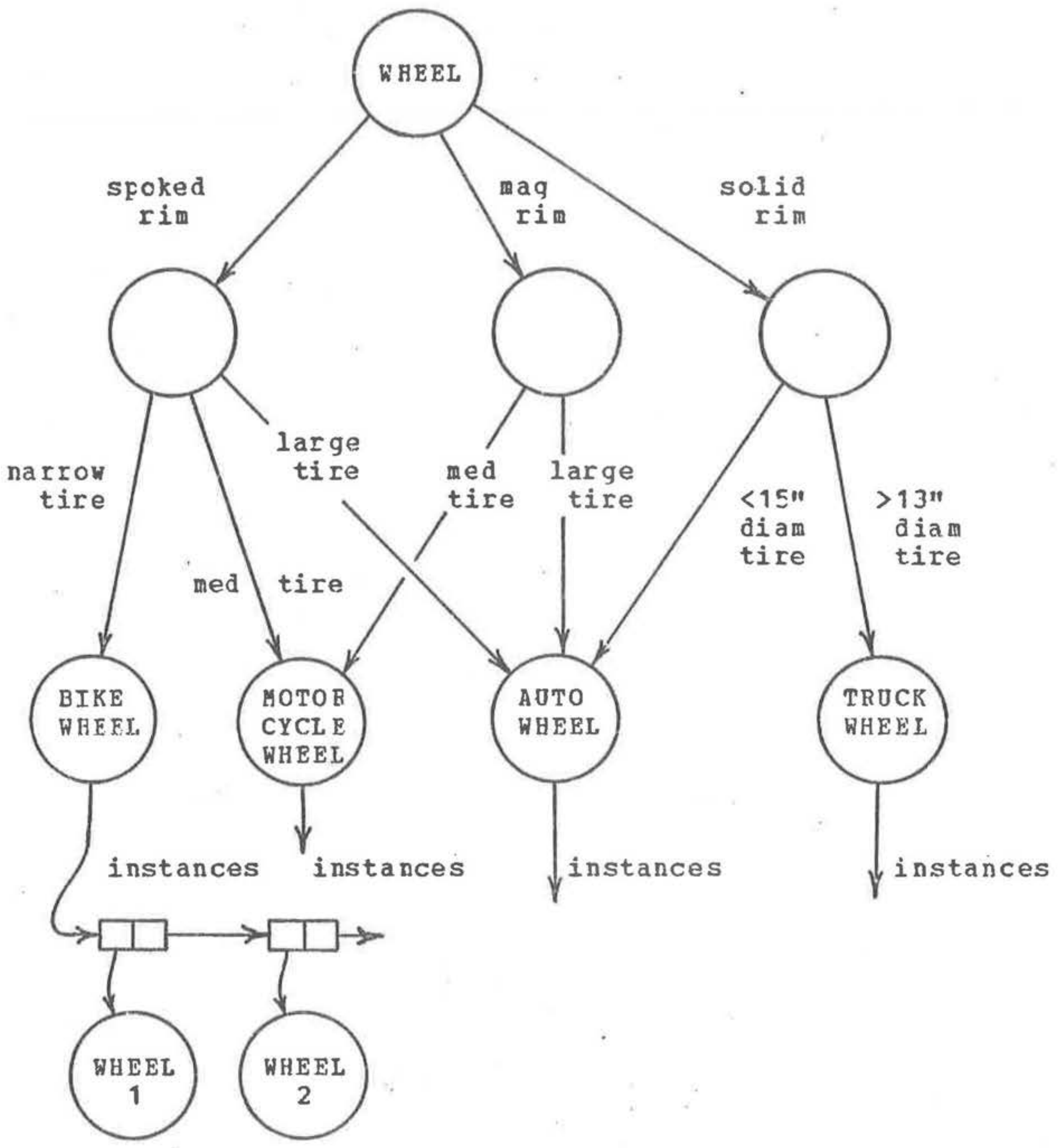


Figure 3.3: Indexed Instance Hierarchy

3.2.3 Recognizers

Perception is not a passive reflection of sensation but an active process motivated by plans, expectations, and desires (Bartlett, 1932). Perception is a recognition task that composes a description of a perceived concept from a sequence of external observations of the world. This concept is represented as a schema instance which is composed of relationships among other more primitive concepts.

In this model, each stereotype schema is considered to be an active _recognizer_ for its stereotypical concept, an individual recognizer in a system of such recognizers. Every schema contains the active knowledge necessary to guide the recognition process for its concept from sensory observations. Such active heuristic knowledge is called a _method_. Methods are procedures specifically tailored for the recognition of their associated schemata. Methods allow the exploitation of domain specific search techniques. Instead of relying on general search methods to conduct the search for every schema in the system, specific methods can be associated with particular schemata to exploit special techniques that are particularly effective for that schema.

The expectations associated with stereotype schemata play an important role in the recognition process. They are dynamic properties of each schema that change as the uninstantiated instance proceeds towards being fully specified. At each point in the process, the schema's expectations represent what additional information is required to complete its recognition. From a different perspective, they represent the schema's knowledge of the world: what it expects to occur next or be found next from observation. Expectations embody the notion of a plan or script (Schank & Abelson, 1975).

Expectations may be represented by simple default values to be replaced by observed values when they are discovered, or they may be represented by complex patterns with attached procedural methods. These attached methods use both top-down and bottom-up search mechanisms. Top-down methods are designed to search for information to satisfy the requirements of its expectation. Bottom-up methods are designed to continue the recognition of their schemata based on satisfaction of their associated expectations. The notions of these two different types of procedural methods are similar to the characterization of Servants and Demons proposed by Bobrow and Winograd (1977).

In the bicycle schema of Figure 3.1, expectations are represented as declarative specifications of the type of schema instances which may replace the expectation in the slot. Each of these expectations has either or both a top-down method and bottom-up method associated with it. For example, both the front and rear wheel slots of the bicycle schema contain expectations that specify a spoked wheel with a narrow tire having a diameter between 19 and 27 inches. Attached to these expectations are specific methods tailored for the recognition of bicycle wheels. Two different methods are specified. One method, FIND-BIKE-WHEEL, is a hypothesis-driven procedure which can be called upon to attempt top-down search to fill the wheel slot. The second method, called FOUND-BIKE-WHEEL, is a procedure for performing bottom-up search. It will be invoked when a wheel matching the specified expectations for a bicycle wheel has been recognized. Its function is to first replace the expectation in the wheel slot with the new recognized wheel instance, then to continue the recognition process for the bicycle utilizing the knowledge gained by the discovery of the new wheel.

3.2.4 Non-Determinism

Unfortunately, perception is not a deterministic process. Perception requires the search of a knowledge base to assign an interpretation to sensory input. For large knowledge bases, this search cannot be a simple top-down goal-directed process. Our knowledge of the world is far too complex to rely solely on goal-directed search mechanisms. Neither can the search be a completely bottom-up process driven by observation. Sensory data is too ambiguous to permit a reliance on purely data-driven mechanisms. Machine perception must utilize an integration of both goal-directed and data-driven search. Bobrow and Norman (1975) have called these two modes, conceptually-driven and data-driven. Goal-directed or conceptually-driven search provides active guidance based on domain-specific knowledge of the hypothesis being attempted, whereas data-driven search utilizes the observation of cues to intelligently select likely hypotheses.

In this recognition model, schemata may employ both hypothesis-driven and data-driven methods to perform the recognition process. Hypothesis-driven recognition involves a top-down search of a composition hierarchy. Schemata attempt to recognize instances of their stereotypes by making observations from sensory input and by recursively calling on the efforts of their sub-schemata as subgoals.

In order to recognize a bicycle using top-down search, the bicycle schema attempts to look for sub-schemata instances that satisfy its expectations. To do so it will invoke its own top-down methods asssociated with each of its expectations in some likely order. Each of these methods will then attempt to recognize suitable sub-schemata by calling on the methods of those schemata as subgoals.

To realize bottom-up, data-driven search in schemata networks requires that multiple hypotheses be allowed to exist simultaneously. Since the recognition of the schemata representing these hypotheses are conducted by procedural methods, these methods must be allowed to apply their heuristic techniques concurrently. In bottom-up search, therefore, methods are realized as concurrent processes. In this model, a multi-processing mechanism for simulating concurrent methods is based on the notion of deductive method scheduling. A particular method, realized as a process, applies its heuristic techniques to the recognition of its schema's stereotype until one or more of the schema's expectations prove difficult to achieve. The method may then suspend its execution by creating new expectations for the schema that describe its unrealized objectives. It remains attached to these new expectations until such time as more evidence is discovered matching those expectations and supporting the renewed probability of the schema's success.

When such matching evidence is discovered, the suspended method is resumed. Methods iterate through a cycle of being resumed by the discovery of matching evidence, then computing a new set of expectations about their evolving schema instances, and then suspending themselves and possibly other methods to those expectations. Since multiple methods may be attached to multiple expectations, these expectations represent diverse possible directions for a schema's script. The choice of search path is not made by blind hypothesis but is data-driven, chosen by the discovery of evidence matching a particular expectation. The method associated with that expectation is then activated to continue its schema's recognition. A branch in the schema's non-deterministic script has been taken.

In the following situation, for example, a wheel instance has been recognized in the input scene. Its features match the expectations of either the front or rear wheel slots of the bicycle schema. Both of the bottom-up methods associated with these expectations may be activated. We assume that the front wheel's expectation is matched first and its method invoked. This bottom-up method will exploit the fact that a front wheel has been discovered to guide the search for other parts of the bicycle. Since the front wheel has a CONNECT relation with the steering-set, the method looks in the neighborhood of its known wheel in the input scene for the remainder of the bicycle. Such an application of common-sense knowledge is realized as a bottom-up procedural method associated with a particular schema.

For this example, we assume that this bottom-up method does not easily discover significant information in the vicinity of the known wheel instance. Instead of retaining control wastefully, it computes a new set of expectations for the bicycle based on that information which it has discerned. Typical of the expectations which might be included is the discovery of a steering-set that must be connected to the specific wheel instance. Its orientation must be above the wheel in the scene, as is normally the case for bicycles. These expectations will act as constraining information on the search for new bicycle components.

The method suspends itself and possibly other methods, as well, to these new expectations. If evidence matching one of these expectations is discovered, its suspended method is resumed to continue its techniques using the new information.

3.2.5 Recursion

Perception is also a recursive process. The recognition of some concept may be used recursively as an internal cue in the perception of more abstract concepts. In this model, cues can be primitive features of the external world or they can be abstract internal features represented as schema instances. When a method satisfies all its schema's expectations for a fully specified concept, that instance becomes an internal high level cue. By allowing cues to be arbitrarily complex concepts, a mechanism is realized for avoiding the "chicken and egg problem" for schemata (Havens, 1976). Starting at the sensory data level, primitive cues present in the input can be used to drive the hypothesis and recognition of low-level concepts. These features then behave as higher level cues stimulating the hypothesis of more abstract interpretations. This bottom-up recognition mechanism depends on the existence of the inverse composition relations (Part-of) in the schemata network. When a concept has been recognized, the completed instance uses its knowledge of what higher schemata in any composition hierarchies it might plausibly be part of. Attempts are then made to match the expectations of those schemata.

In this model, primitive cues are discovered from features extracted from the input image using context-free cue discovery methods. In the vision domain, such techniques include region segmentation and line-finding algorithms. The primitive cues are matched against the expectations of low-level schemata thereby activating their attached bottom-up methods. Each of these methods then attempts to confirm its own schema's hypothesis. When one or more of these methods succeed, the recognized schema instance becomes a higher-level cue using its inverse composition knowledge to attempt to match the expectations of higher schemata of which it may be part. This recursive process is seen to be a bottom-up recognition model for schemata driven by both primitive context-free cues and abstract context-sensitive cues recognized recursively.

To illustrate, when a wheel is found in the scene, its discovery becomes an abstract cue in the higher hypothesis of the bicycle schema. The fact that its recognized features, such as type and size, match the bicycle's expectations indicates that the bicycle is a likely hypothesis and that its methods should be invoked. This is characterized as a matching process between cue and expectation. Then, if the bicycle schema is eventually successful in recognizing a bicycle instance, that instance will become a higher cue in the recognition process, perhaps, in this example, hypothesizing a class of scenes containing bicycles.

3.3 Earley's Algorithm

In an attempt to develop computational mechanisms for realizing this model, the author investigated the formal recognition models used in parsing theory. The context-free parsing domain can be viewed as a highly restricted subset of the perceptual domain. Context-free parsing is a recognition task that assigns an interpretation to an input sentence based on a hierarchical knowledge base, that is, a context-free phrase-structure grammar. The interpretation is inherently a recursive process for which both top-down and bottom-up recognition algorithms have been developed exhibiting well understood properties. In particular, the bottom-up parsing algorithm of Earley (1972) has some interesting properties from the perspective of machine perception. Earley's algorithm is an efficient bottom-up recognizer that can operate directly from any non-left-recursive context-free grammar. The algorithm is quite elegant. It can operate in time order n^3 and space order n^2 where n is the length of the input sentence. Morever, it does not require backtracking to handle non-determinism. Appendix-A provides a formal presentation of the algorithm.

In studying his algorithm, not as a parser, but as a bookkeeping scheme for the simulation of multiple bottom-up processes simultaneously operating on the same input sentence, it was noticed that the algorithm dynamically inverts certain portions of a hierarchy (the grammar) based on a selection function (the input sentence). The algorithm operates from a context-free grammar and a set of valid parsers. Each parser attempts to recognize a complete sentential form derived from the right-hand-side of a particular rule in the grammar.

Initially, the set contains only a single parser which attempts to recognize a sentential form derived from the start symbol of the grammar. Such a derivation will be, of course, a complete sentence in the language. If a non-terminal symbol may appear next in the derivation of any parser, new parsers are created for every rule in the grammar having that non-terminal as its left-hand-side. This function is called _prediction_ in the algorithm.

The algorithm proceeds by scanning in a left-to-right order each symbol in the input sentence. If the observed symbol can be a valid next terminal symbol in the sentential forms of any of the active parsers in the current parse list, then those parsers are propagated into the next parse list. All other parsers are deleted. The algorithm scans the input once. It hypothesizes new parsers by prediction from the grammar when their rule can appear next in the derivation. Old parsers which cannot derive some portion of the input sentence to the current position are then destroyed.

Earley's algorithm exhibits some interesting properties from the standpoint of a model of recognition for computer perception. The algorithm provides a mechanism for implementing bottom-up search yet operates directly from a top-down hierarchy. Portions of the hierarchy are dynamically inverted, selectively, depending on what terminal symbols are observed in the input and what non-terminal symbols are recognized recursively from the input. These are appealing attributes since schema systems are organized as top-down hierarchies and it is desired to drive the bottom-up recognition process by the discovery of both primitive and recognized cues.

As a bookkeeping scheme, Earley's algorithm is applicable not only to parsing but also to multiprocessing in general. The algorithm systematically simulates multiple co-existent parsing processes operating from the same hierarchy and on the same input. As well, it provides a scheduling mechanism based, not on emulating parallel search, but on concurrent deduction. Parsers remain suspended until such time as the terminal or non-terminal symbol for which they are looking is discovered. Then, each such matched parser is resumed to continue its recognition.

The algorithm also handles non-determinism naturally and efficiently. When a terminal symbol is scanned or a non-terminal symbol is recognized, the algorithm propagates into the next parse list every suspended parser that was expecting that constituent. All others are deleted. No backtracking is required. Constituents need be found only once in the input and invalid interpretations are deleted as soon as possible.

Although the above properties are desirable, there are a number of problems with adapting Earley's algorithm as a search mechanism for the general and more complex domain of perception. Minker (1973) has considered and rejected its use as a problem space representation. Woods (1974) has favorably compared its abilities with those of his augmented transition network parsers.

There appear to be three major difficulties inherent in this approach. First, the process of predicting which terminal and non-terminal symbols may appear next in the derivation is too top-down, exhibiting aspects of "the chicken and egg problem". For a large knowledge hierarchy, the prediction process may have to tour very large branches of the hierarchy. In the parsing domain, this inefficiency can be tolerated by suitably restricting the grammars employed. For machine perception, no such restrictions of the knowledge base are feasible. In spite of this difficulty, Kaplan (1973) has proposed a similar prediction scheme for creating parallel interpretations in his parsing system, GSP.

Second, the algorithm depends upon the strict sequential nature of parsing to efficiently limit the proliferation of parsers. After the next input symbol has propagated all parsers still valid to the next parser set, all the remaining parsers can be deleted. They represent invalid interpretations of some portion of the input sentence. In perception, however, some aspects are encoded sequentially, such as the surface form of natural language. However, other aspects have very little sequential content, for example, image analysis. Where sequence is an explicit part of perception, it can be effectively used to constrain invalid interpretations, but an efficient perception mechanism must handle non-sequential aspects as well.

The last difficulty perceived with Earley's algorithm is that the mode of search employed is completely bottom-up. There is no top-down mode defined. As has been seen, for perception, both top-down and bottom-up mechanisms are essential.

3.4 Three Phases of Recognition

The context-free parsing algorithm of Earley was shown to exhibit a number of properties desirable for a recognition model for machine perception. A few difficulties were also noted. This section develops a new recognition model for perception using some of the techniques developed by Earley while, at the same time, avoiding the difficulties inherent in his algorithm.

This model supports both top-down and bottom-up search in schema-based representations. The bottom-up mechanisms will be explained first since they are derived, in part, from the previous discussion. The recognition model consists of three phases, called <u>expectation</u>, <u>matching</u>, and <u>completion</u>.

3.4.1 Expectation

The expectation phase of this model is analogous to the prediction function of Earley's algorithm. In the top-down recognition model for schemata given by Kuipers (1975), expectations are described as static properties of the stereotype which are systematically replaced by specific information as it is discovered. In this model, expectations are computed dynamically during the recognition process as a function of the current partial instantiation of the schema. At any point, a schema's expectations represent a non-deterministic description of all possible final instantiations of the schema. Bound to each expectation is a method designed to continue the recognition of the schema based on the satisfaction of that specific expectation.

An instance hierarchy is regarded, in this model, as a non-deterministic representation of a general class of objects. Recognition is characterized as the differentiation of the stereotype into a specific instance. This process of refining the expectations of a stereotype towards a fully specified instance has been called <u>specialization</u> (Bobrow & Winograd, 1977). In the top-down recognition model, however, a schema stereotype is portrayed as representing a particular class of objects such as tables or chairs. The recognition process results in the description of a particular table or chair. If, during this process, the schema is found to be inappropriate, a replacement schema must be selected by some substitution procedure. In this model, this substitution is not necessary. The schema instantiation is characterized as being a process of differentiation instead of selection.

3.4.2 Matching

The second phase of the model is the matching phase. The expectation and matching phases form an iterative recognition cycle. In bottom-up recognition, the expectations bound within schemata are satisfied by suitable observations from some input medium or by the recursive recognition of other schemata. After a schema has created a set of expectations and bound methods to those expectations, the expectations may be matched by observed or recognized evidence, thereby activating the associated methods. Each method first incorporates the new evidence into the evolving instance of its schema's stereotype. It may then calculate, based on this new evidence, a next set of expectations, suspending itself and perhaps other methods to these new expectations. This _expectation/matching cycle_ may co-exist over time with the recognition cycles of many other schemata. This cycle realizes a multiprocessing mechanism for simulating parallel search.

The matching phase depends on the availability of an associative retrieval mechanism in the schemata network. New evidence, discovered by observation and deduction, must be able to find those schemata containing expectations which it can match. Two such mechanisms are proposed, both of which involve pattern matching over schema systems and are similar to the extended concept of description matching advocated by Bobrow and Winograd (1977). In general however, the problem is quite complex. Indeed, a comprehensive theory of deductive associative retrieval over semantic networks is required. This problem can be seen to be, recursively, the recognition problem.

The first mechanism uses simple pattern matching and associative retrieval as is familiar in the newer Planner-like artificial intelligence programming languages (Bobrow & Raphael, 1974). In this system, the expectations of a schema are represented as n-tuple patterns in a pattern associative database contained within the schema. McCalla (1977) has advocated such an implementation for schemata in a natural language dialogue system.

The second mechanism involves using the instance hierarchies as associative databases. Such a taxonomic organization within the system of schemata provides the necessary retrieval mechanisms to support the matching phase of the recognition model. The matching process therefore involves a cue-driven search over the schemata network in conjunction with syntactic matching of expectations represented as patterns.

3.4.3 Completion

A schema completes the recognition of an instance when all its expectations are satisfied. It must return that success to higher schemata of which the instance can be a component part. This is the completion phase of the recognition model and is analogous to the completion function of Earley's algorithm

(Appendix-A). If the completing schema's method was called using top-down search as a subgoal, then the schema has an explicit caller and must eventually return a success or failure to that caller. On the other hand, a schema's method may be activated using bottom-up search by having an associated expectation matched by some other process. This method has no explicit caller and is referred to as a supergoal. The distinction between subgoals and supergoals is based on the way in which they are activated. Subgoals are activated as subroutines attached to the calling routine, whereas supergoals are activated as processes whose existence may continue after control has returned to the caller. Both types of methods are directed at completing the recognition of their schemata and both may use a combination of top-down and bottom-up techniques to do so.

When a supergoal has satisfied its schema's internal requirements for the recognition of an instance of the schema's stereotype, the recognized concept becomes an abstract cue. It enters the matching phase by attempting to match the expectations of those higher schemata of which the particular instance might be a plausible part. Such knowledge is available to the completed schema instance through the inverse composition relations of its stereotype.

Figure 3.4 illustrates both the cyclic and the recursive nature of this process. For example, the recognition of Schema-1 at the first level in the hierarchy may proceed over

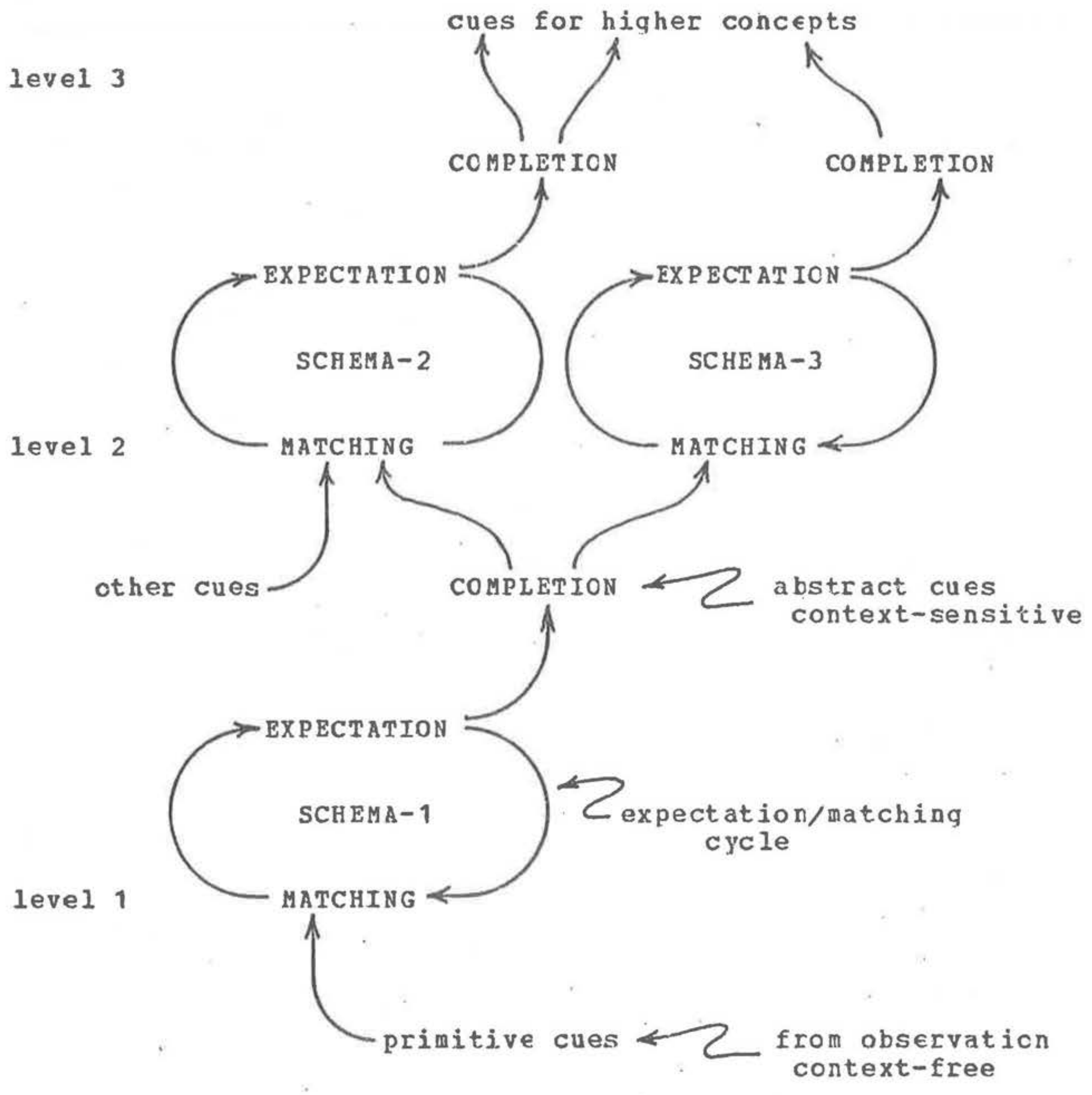


Figure 3.4: Recognition Cycles

time with the recognition of other schemata. Schema-1 may go through several cycles of creating expectations, suspending its methods to those expectations, and being resumed by matching primitive cues computed from the input data. When all the expectations of Schema-1 have been satisfied, it then enters its completion phase. Since it was not called as a subgoal of any higher schema, its description is, in effect, an abstract high-level cue. From the information contained in this description, it attempts to match the expectations of schemata at the second level in the hierarchy, in this case, Schema-2 and Schema-3. If the match is successful, the methods of one or both of these schemata are resumed as supergoals of Schema-1 to continue their recognition. Their recognition may also proceed through a number of expectation/matching cycles concurrent with other recognizers. But, unlike Schema-1, their expectations are matched by non-primitive cues recognized recursively as the result of perception.

Completion is seen as a "handle" in the recognition model for realizing a number of desirable features of a theory of machine perception. It provides a mechanism for simulating parallel search via a deductive method scheduling scheme, for realizing a recursive cue/model hierarchy, for realizing method hierarchies, and for coordinating the efforts of concurrent methods. The remaining sections of this chapter will explain how these features are achieved in the model.

In the analysis of Earley's algorithm, it was pointed out that the prediction phase may be computationally very expensive. In this recognition model, the problem is avoided by noting that the expectation phase for some particular schema need not be performed until such time as another completing supergoal or some input observation attempts to match that schema. When a supergoal attempts to match a schema of which it can likely be part, it must first search for a particular instance of that schema to match. If no such instance can be found, then the supergoal calls upon the schema stereotype to create a new instance. Thereby, the expectation phase is performed only when needed.

3.5 Scheduling

Proposals for simulating parallel search usually employ a multiprocessing scheme that relies on some global algorithm to allocate the processor. A popular technique is a weighted time-slicing mechanism based on a priority queue (Bobrow & Winograd, 1977). This type of mechanism is effective at simulating the concurrent evaluation of procedures but it is not effective at simulating the parallel application of non-deterministic search methods. The technique is directed at scheduling processes, not at the methods that are implemented as processes. The scheme is too low level. It typically forces the programmer to assign a global numeric priority to a process when it is placed on the priority queue.

A scheduling mechanism, operating at a higher level, is needed. Methods should become active when their applicability to the recognition process is discovered or deduced, not when a process spontaneously reaches the front of a priority queue.

To illustrate this further, consider an automatic deduction system that is implemented in a multiprocess environment. The system is attempting a complex proof. One particular deductive process has shown considerable promise but has been unable to achieve some result, say P(x). The process decides to suspend itself until such time as another process has succeeded in deducing P(x). In this priority driven system, which priority should be assigned to this suspended process? How does one assign a priority number to a process in order that it wait for a specific situation to occur? Obviously, a multiprocessing scheme based on deductive scheduling instead of simple parallelism is required. Processes should be scheduled when their applicability to the system's task has been computed. The bottom-up search problem is not to simulate parallellism, but to coordinate the simultaneous efforts of multiple methods. When a method achieves some intermediate result, the scheduling mechanism should ask, "What methods are suspended waiting for this result?".

The completion phase of the recognition model provides such a scheduling mechanism. In fact, the recognition model can be characterized as the computation of what methods should be scheduled next. Methods remain suspended to patterns representing expectations until resumed explicitly by some lower completing supergoal. A method is resumed when it is deduced that the method could be applicable to the recognition process. A completing supergoal may resume, in turn, the methods bound to expectations of all higher schemata that its schema instance can succeed in matching. Each such resumed method is activated also as a supergoal and proceeds to continue the recognition of its own schema. The recognition of a schema instance need be computed only once. By sequentially resuming more than one higher supergoal, a single schema instance can be part of the recognition of multiple higher schemata. No backtracking of subgoals or use of similarity network schemes is required.

Each completing supergoal behaves as a heuristic scheduling mechanism, resuming the methods of those higher schemata which it is successful at matching. That scheduling mechanism may be tailored specifically for each situation. A syntactic global scheduler is not required. For instance, a supergoal can resume higher methods in order of expected likelihood of applicability or it can resume first only the methods of those schemata which already have existing instances. If none of those succeed, then it can create new schema instances to match, thereby saving the expense of the expectation process until it is needed.

The completion mechanism can be characterized as a bottom-up generator in contrast to the top-down generators of Conniver. When a completing method matches a higher supergoal, it is generating a possible successor process. It tours a schema hierarchy in inverted, bottom-up order. The completing method's computations are suspended while the higher schema's supergoal attempts to complete its recognition. If control returns to the completing method, it will then generate the next possible higher supergoal. This mechanism is a generator for bottom-up search instead of top-down search.

3.6 Method Hierarchies

It is clear that machine perception must utilize heuristic domain-specific search knowledge in order to cope with the complexity of the perceptual process. By incorporating this knowledge as procedural methods associated with stereotypical schemata in a composition hierarchy, a _method hierarchy_ (Newell, 1972) isomorphic to the composition hierarchy is formed. Methods exhibit a tradeoff between applicability and power. Methods applicable to a large class of search problems are inherently inefficient for any specific problem. Conversely, methods heuristically engineered for the accomplishment of a narrow specific task can achieve power and efficiency. A comprehensive recognition scheme for perception must include provisions for such a method hierarchy to intelligently guide the search process.

In the purely top-down recognition model, a method hierarchy capability is straightforward. The methods associated with each schema may be created specifically to search for instances of that schema. The hierarchy is realized by the fact that, in top-down search, schemata call on the efforts of other schemata as subgoals. Each level of subgoaling applies a more powerful and specific method to the achievement of that particular subgoal.

In bottom-up search, however, achieving a method hierarchy is not so straightforward. Multiple methods may be active simultaneously. Which method should, at any given instant, direct the search process? Most bottom-up search schemes have used a single top-level method to drive the operation of the system (Waltz, 1972) (Mackworth, 1977a). Multiprocessing schemes typically alternate the application of methods as processes which are run and then suspended again on the priority queue. There is, however, poor coordination between alternating methods.

The following mechanism is utilized to realize a method hierarchy in bottom-up search. As has been described, a supergoal method can compute, based on the partial instantiation of its schema, a next set of expectations. It may then suspend other methods to patterns within the schema representing those expectations. After the method has performed the expectation phase, it can either terminate its execution, relinquishing control to some higher method, or it can remain active, retaining access to the processor. This mechanism provides effectively an extra "degree of freedom" in the search process. If the method relinquishes control, then it is relying on the efforts of other methods to discover evidence that will match its schema's expectations. On the other hand, if the method retains control of the processor, it may use its own specialized knowledge to direct the discovery and deduction of evidence to satisfy its own expectations.

This mechanism allows each schema the choice of applying its own specialized heuristic knowledge to direct the search process or yielding to the heuristics of higher, more general methods. In top-down search, each method, as a subgoal, is forced to direct the search process regardless of the schema's applicability or expertise. A major reason for the pathological behavior of pure top-down search (Sussman & McDermott, 1972) is that inapplicable methods do not know when to quit.

In bottom-up search, however, the choice of when to apply a method's techniques is not critical. The decision is based locally on a schema's anticipation of success. If a schema applies a bottom-up method, but the schema itself is inappropriate, control will soon propagate to the methods of more appropriate schemata. Evidence will be discovered which matches the expectations of those more appropriate schemata. Their methods will then assume control thereby correcting the mistake. The essential difference from top-down search is that this decision does not have to be made locally by each method. Instead, it is made globally by the discovery of evidence supporting the choice of a different method. In the worst case, all that is lost is some efficiency for a short while. There is no chronic pathological behavior. Mackworth (1977a) has noted a similar convergence effect in the use of his bottom-up constraint propagation algorithm, NC.

3.7 Coordination and Communication

Since multiple methods can be active concurrently, methods may simultaneously attempt to recognize different schemata or more than one method may attempt to recognize the same schema instance. It is necessary to provide coordination and communication among sister methods.

In this model, communication among sister methods is accomplished through their common schema instance. The instance is a data structure accessed by all methods associated with that schema. Each method contributes to the instantiation of the schema and is aware of the contributions of its sisters. When some particular method decides that the recognition of this schema instance has been completed, it must communicate that success to every active sister method. Such a mechanism is defined within the completion phase. When a method begins the completion phase, it is assumed that the efforts of all other methods associated with the recognition of this particular schema instance are no longer needed. All such sister methods are suspended within the completed schema instance.

As well, there must be coordination among the methods of a method hierarchy. A number of methods in the hierarchy may be simultaneously attempting the recognition of their associated schemata. Since the method hierarchy is isomorphic to the composition hierarchy, when a method at some level in the method hierarchy is successful at recognizing a schema instance at that level, the methods at lower levels are no longer needed. These lower methods were attempting to recognize components of the now completed schema instance and to discover cues to propagate its recognition. That recognition is now complete and these methods are also suspended.

Completion then is seen as a coordination mechanism among cooperating concurrent methods. When a schema is successfully recognized, all methods currently attempting the recognition of that schema or sub-schemata of that schema are suspended. Completion was characterized above as a bottom-up generator. This mechanism is realized by the conventions described here. The bottom-up generator is composed of all the methods suspended by the completion process. On failure, the generator is resumed to generate a new plausible higher supergoal, resuming all the suspended methods to continue the search for that next supergoal.

3.8 Integration

The presentation of this recognition model has so far concentrated on realizing bottom-up search mechanisms. One of the premises for the model's development was that it should provide an integration of top-down and bottom-up search. This section describes such an integration.

In this model, methods may execute either as subgoals or supergoals. A subgoal method may attempt to complete its task by using either top-down or bottom-up techniques. Top-down search is implemented in the familiar manner by methods recursively calling the methods of other schemata as subgoals. As well, a subgoal method may use bottom-up techniques by computing a set of expectations for its schema, then attempting to satisfy those expectations by making observations and deductions. This process may, itself, recursively use an integration of top-down and bottom-up techniques. The only restriction is that the subgoal method eventually return a success or failure to its explicit caller.

Likewise, supergoal methods may use either top-down or bottom-up search to achieve their tasks. Supergoals may call other methods as subgoals. Alternatively, a supergoal method may create a set of expectations for its schema, then either relinquish control to some other method, or using its heuristic knowledge, direct the discovery and deduction of information to satisfy its own expectations. In summary, both top-down and bottom-up methods can be intermixed freely.

When a completing schema is attempting to match the expectations of higher schemata, it must first find instances of those schemata to match. This device involves the semantic network matching described earlier. Part of the heuristic knowledge contained in a schema's methods consists of procedures for searching the schema network to look for higher schemata to match. In fact, there is no sharp distinction between when a schema ceases the recognition of an instance and when it begins the completion process. Overlapping may occur to the point that the completion process involves using both top-down and bottom-up search techniques and the expectation/matching cycle may involve aspects of completion. For instance, in order to efficiently calculate a next set of expectations, a method may need to match higher supergoals in order to constrain the number of expectations produced. This distinction is made more for conceptual than computational reasons. In a large schema system, it will be advantageous to blur the distinction in order to facilitate communication up and down the the schema hierarchy.

CHAPTER 4: AN EXAMPLE FROM MACHINE VISION

4.1 Perspective

This chapter presents a detailed example of the operation of the recognition model on a computer vision task. This example has been implemented as a running Maya program which is given in Appendix-C. In the next chapter, the implementation of the example will be covered as part of the description of Maya.

The example chosen is a small but characteristic machine vision problem from the "blocks world". The problem is to recognize from perfect line drawings a class of polyhedra including cubes, wedges, and pyramids. The recognizer operates from a schema representation of polyhedra and accepts input scenes consisting of vertices and lines connecting vertices.

The problem is not a vision task of current research interest. It is presented to illustrate the operation of the recognition model developed in this thesis. There are three major reasons for this choice. First, the problem is characteristic of machine perception tasks. The world of polyhedral objects is believed to be the simplest task domain that captures the essential aspects of scene analysis (Mackworth,1976). Second, the example is restricted enough in its scope that heuristic concerns specific to a particular vision task do not overshadow more general issues of representation and recognition. And third, this example is similar to an example given by Kuipers (1975) to illustrate the operation of the top-down model of recognition for schema systems. This choice of example, therefore, permits a comparison of the two recognition models.

The method of presentation will be to first describe the overall structure of the problem's solution, and then, by utilizing a protocol produced by the program, explain the operation of the program and the underlying recognition model. Although the program does not explore all the issues addressed by the model, it does provide a handle for their discussion.

4.2 The Problem

The knowledge of polyhedral objects is represented in this example as a schemata network. The program uses the simple composition hierarchy of Figure 4.1 to represent stereotypical scenes of polyhedral blocks. Each node in this hierarchy is a stereotype schema representing a named concept. Each downward-directed arc represents the composition relation between its schema and its sub-schemata. Scenes are composed of polyhedral objects. Polyhedra are composed of polygon faces which, in turn, are composed of primitive edges and vertices. Each upward-pointing arc represents the inverse composition relation, "part-of".

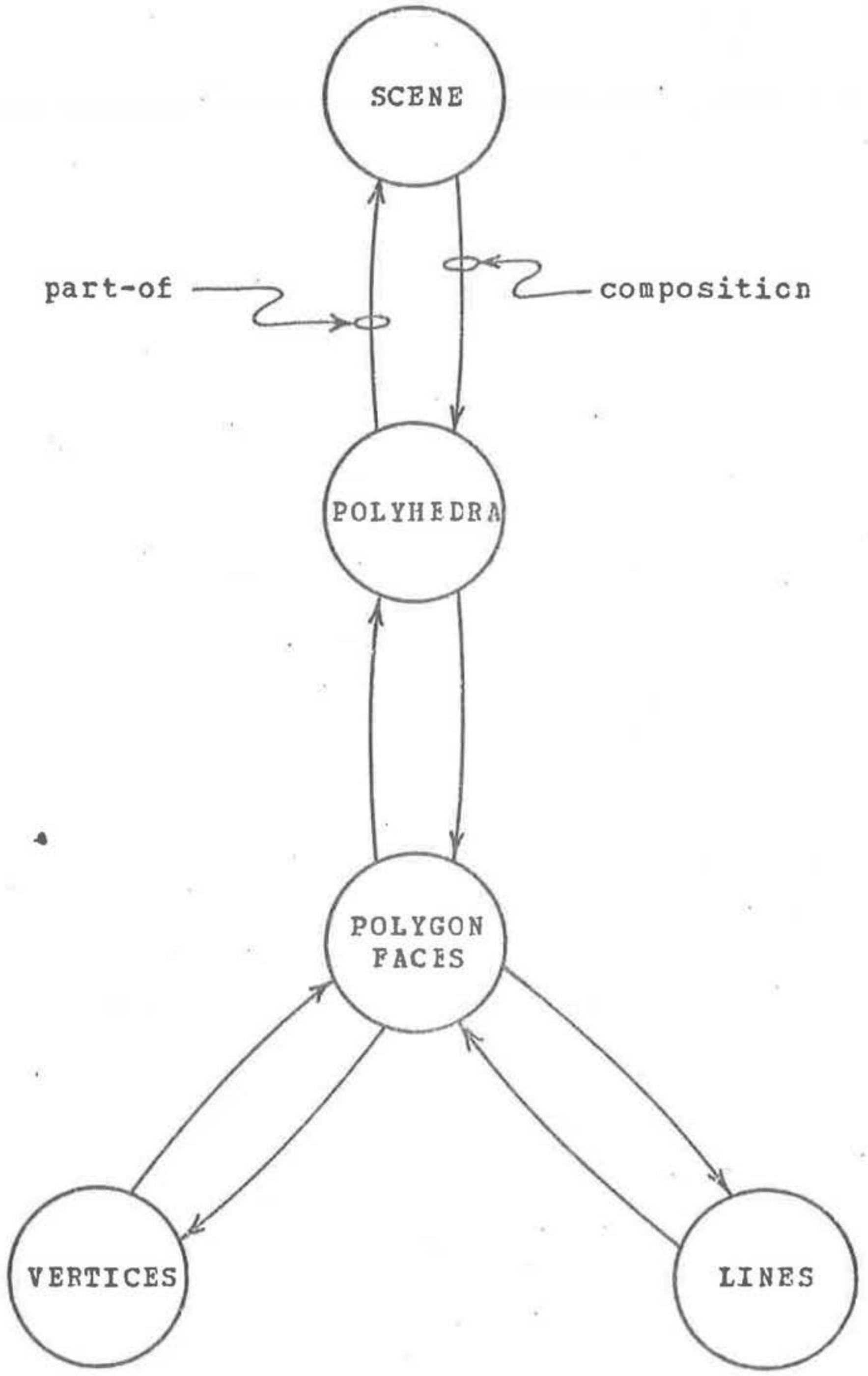


Figure 4.1: Scene Composition Hierarchy

In this example, polyhedral objects are differentiated into cuboids, wedges, and pyramids. Cuboids are polyhedra composed of parallelogram faces only. Wedges are objects composed of parallelogram faces and a single triangle face, and, pyramids are objects containing two or more triangle faces. Polygons are differentiated into triangles, parallelograms, general quadrilaterals, and arbitrary polygons of five or more sides called multilaterals. Notice that the differentiation of the generic polyhedron and polygon schemata into subclasses is not represented explicitly in the composition hierarchy. Polyhedra, for example, are not divided into cuboids, wedges and pyramids each of which, in turn, would be divided into triangles and parallelograms. To do so would expand considerably the size of the schema system and force an explanation of this larger structure in order to perform recognition. Instead, the knowledge of the division of stereotype classes into subclasses and finally into specific instances is represented procedurally in the methods attached to the stereotype schemata. The task of the polyhedron schema's methods is to recognize instances of cuboids, pyramids, and wedges. Likewise for the polygon's methods, their task is recognize from observation instances of triangles, parallelograms, quadrilaterals, and multilaterals. Initially, the taxonomic structure is represented within the procedural methods, but as specific instances of polygons and polyhedra are hypothesized and subsequently recognized, they are added to the instance hierarchy for those schemata. Thus the structures are created dynamically. The construction of the hierarchical representation of a scene is based on evidence discovered in the input, not on all possible compositions compiled before the recognition begins.

The program is run on the scene of Figure 4.2. The scene data is input to the program as a set of lines and vertices, each of which is represented as a primitive fully instantiated schema. As the data for each vertex is read, a schema instance is created and added to the vertex instance hierarchy. This function is illustrated in Figure 4.3. Vertices are divided into four classes in this hierarchy. They are ARROW, FORK, T, and L. Because there are only a few primitive vertex instances in the input data, the vertex hierarchy uses a simple organization. VERTEX maintains a simple list of its instances, FORK, L, T, and ARROW. Likewise, each of these stereotypes contains a list of its instances, the primitive input vertices. Besides organizing the database of vertices by vertex type, the hierarchy also provides an attribute inheritance mechanism. For instance, Vertex-1 inherits the schema attributes of its stereotype, ARROW, which, in turn, inherits the more general properties of VERTEX. Inherited attributes can include procedures, variable bindings, pattern databases, and methods, thereby implementing method hierarchies.

Figure 4.4a illustrates a typical primitive vertex schema, Vertex-i. Each such vertex contains a number of named slots. The "ISA" slot (Fahlman, 1975) represents the inverse instance

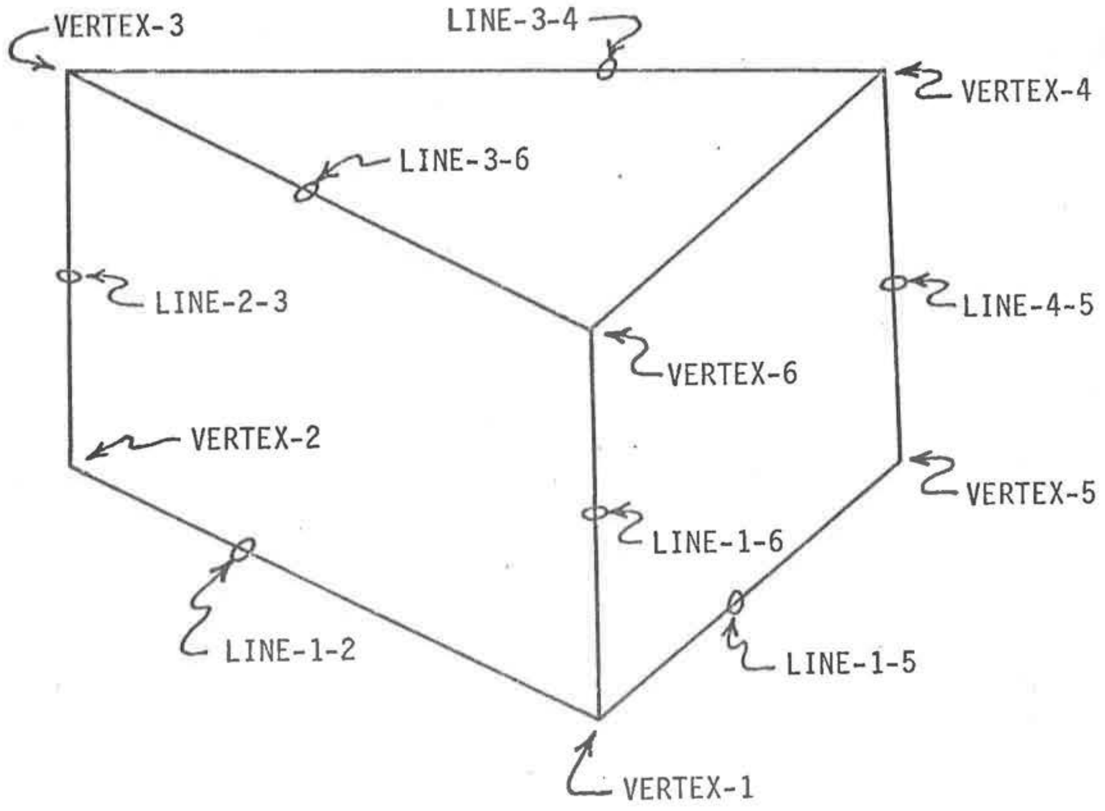


Figure 4.2: Input Scene

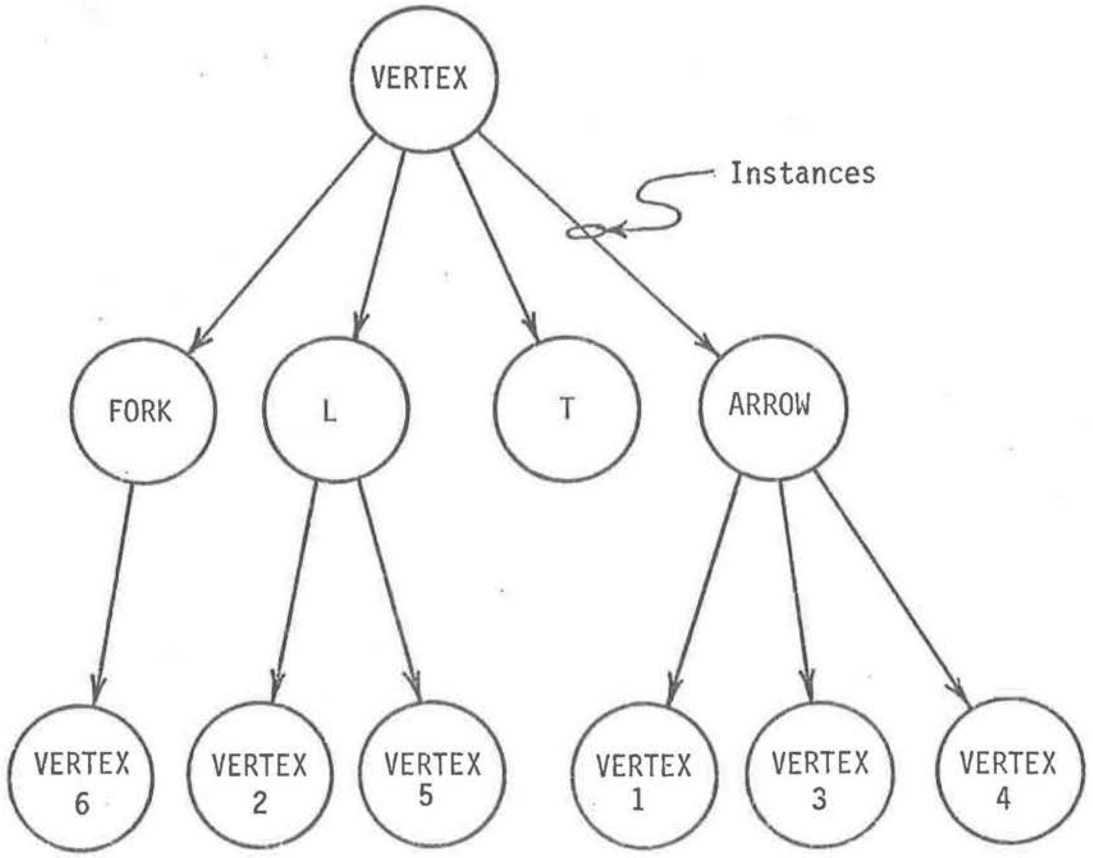


Figure 4.3: Vertex Instance Hierarchy

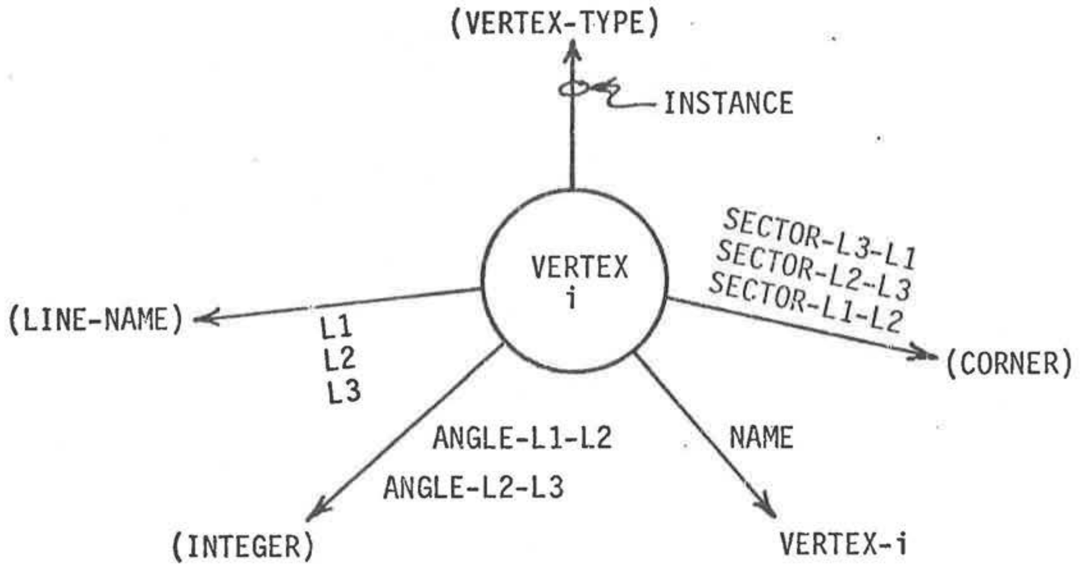


Figure 4.4a: Vertex Schemata

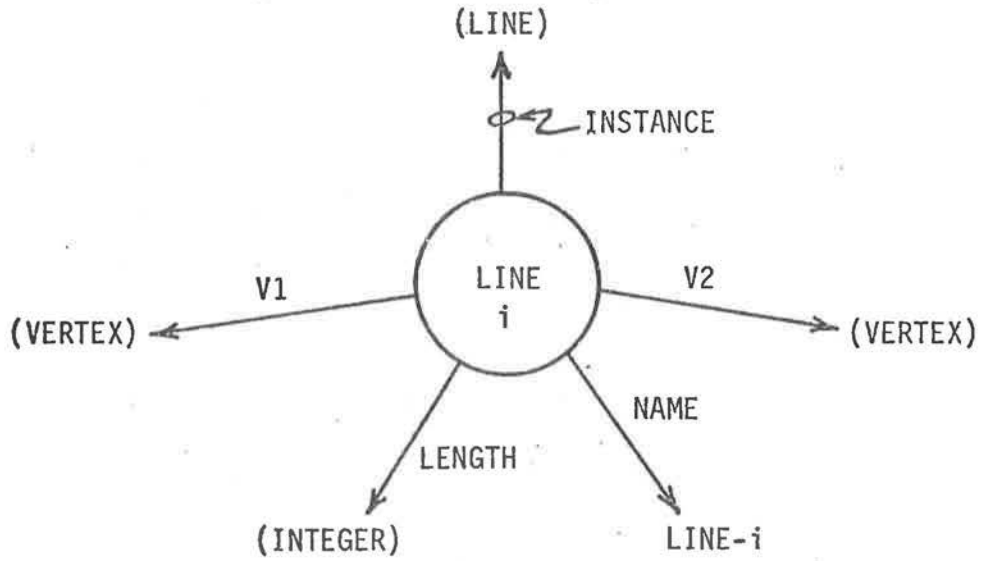


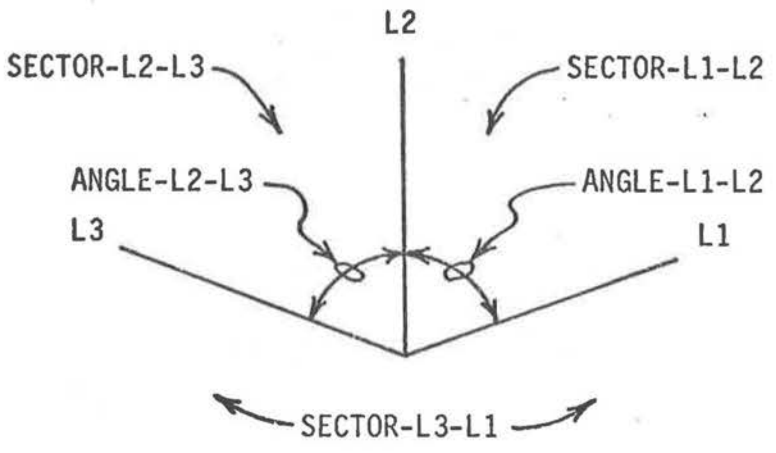
Figure 4.4b: Line Schemata

relation between every instance and its stereotype. This relation is automatically created by Maya when the instance is defined. It will point either to the FORK, L, T, or ARROW schema.

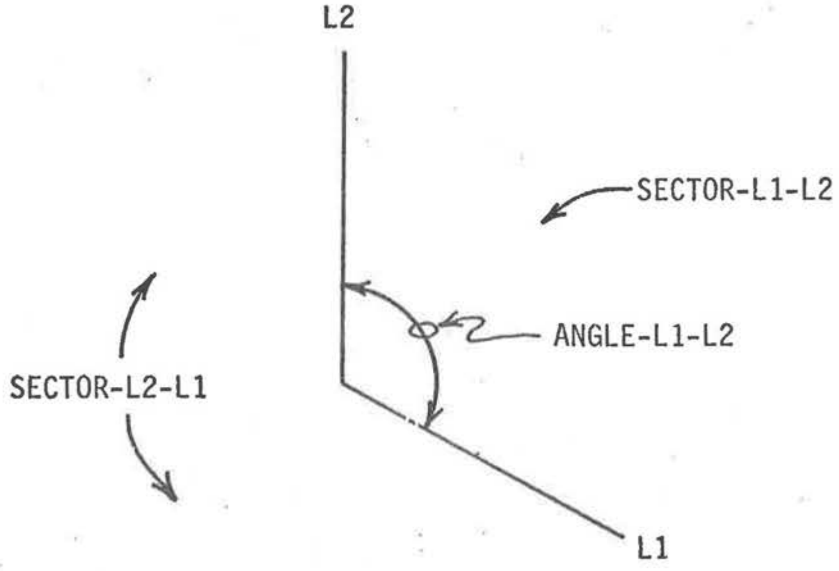
A second slot in the vertex instance, called NAME, points to an atom representing the schema's name, Vertex-i. Three additional slots in the instance (two for L-vertices) are used to represent sectors of the picture plane that each vertex imposes on its immediate locale, as shown in Figure 4.5. When the vertex has been recognized as part of one or more polygon faces, these sectors will be corners of those faces.

The remaining slots of the vertex instance represent data for the particular vertex. The slots labelled L1, L2, and L3 point to the corresponding line instances given in the input picture. Likewise, the slots labelled ANGLE-L1-L2 and ANGLE-L2-L3 are used to indicate approximate angles of each of the sectors, as is illustrated in Figure 4.5.

In a similar manner, a schema instance is created for each line in the input data. Figure 4.4b depicts the structure of primitive line schemata. Each line schema is an instance of the generic LINE schema and has slots for its name, its length in the picture, and the names of the two vertices, labelled V1 and V2, connected by it in the picture. As a matter of convenience, all of the input data is read and represented as schema instances before the recognition process begins. The output of the program consists of a hierarchical description of the



For ARROW, FORK, and T-Vertices



For L-Vertex

Figure 4.5: Vertex Labelling Conventions

recognized scene composed of fully specified schema instances, or conversely, a report of failure.

4.3 Annotated Protocol

In the following pages, a protocol is used to facilitate the explanation of the model. The sentences which are preceded by an asterisk and printed in upper case are the statements of the protocol produced by the program. All others are the author's commentary on the behavior of the program and the model. It may be useful to use Figure 4.6 to follow the recognition process. The face schema instances created during the recognition are shown superimposed on the lines and vertices of the picture.

The recognition process begins by sending the top-level schema a message to interpret the input data as a scene. In order to recognize a scene, the SCENE schema must find a polyhedral object in the data. The schema has the choice of applying top-down or bottom-up techniques. If it chooses to conduct a top-down search, the difficulties mentioned with the top-down recognition model will appear. Specifically, the SCENE schema will be forced to hypothesize alternatively cube, wedge, and pyramid schemata as subgoals. If a hypothesis is incorrect, the schema will have to choose a nex subgoal based only on that failure. Alternatively, the SCENE schema may begin its recognition using bottom-up search. Later, when sufficient

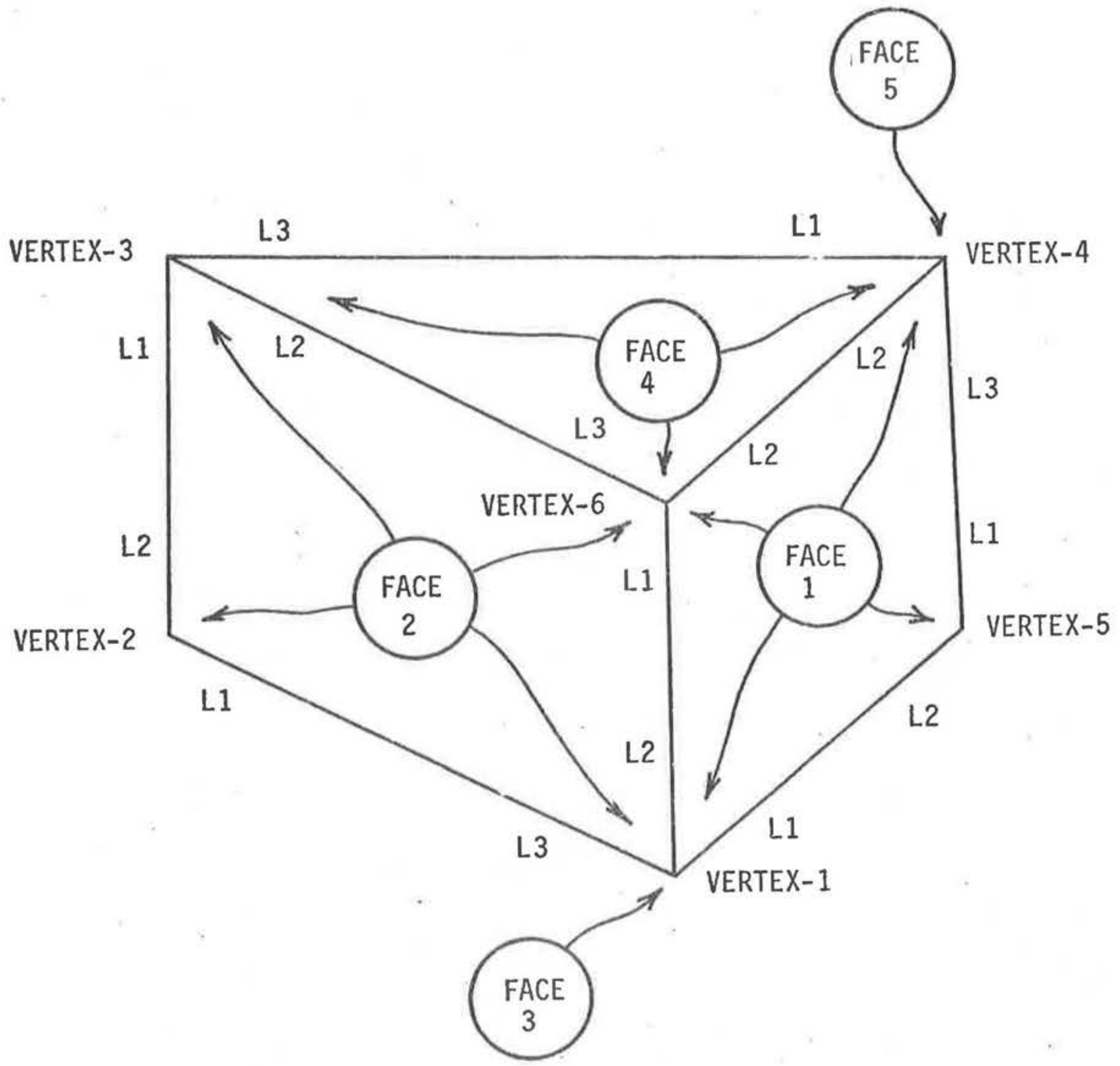


Figure 4.6: Recognition of a Wedge

supporting evidence for a single hypothesis has been found, the hypothesized schema can confirm its recognition using top-down techniques.

* METHOD: TOP-LEVEL SCHEMA EXHAUSTIVELY OBSERVES EACH VERTEX

Since no evidence has yet been discovered supporting any particular hypothesis, the top-level Scene schema chooses to use bottom-up search techniques by making observations in the input data. The heuristic method used by this schema is the same as that given by Huffman (1971). The schema exhaustively activates each primitive vertex schema in the scene, beginning with the peripheral vertices as they are less ambiguous interpretation cues than interior vertices.

The observation process consists of activating each vertex schema to perform its completion phase. Since the recognition of each vertex is given as the input data, the vertex need only compute of which higher schemata in the network it may be plausibly part. In this example, vertices may be component parts of both triangle and parallelogram faces.

* SCENE SCHEMA OBSERVES VERTEX-1

Vertex-1 is an ARROW vertex which, in turn, is an instance of the generic Vertex schema. ARROW, FORK, and T-vertices contain three sectors since they divide the picture plane locally into three regions. Vertices containing three sectors are taken to be default and the generic vertex schema provides a procedure for performing completion for these vertices. Each such vertex inherits this method from the Vertex schema by the inverse instance relation maintained automatically by Maya. The only vertex having two sectors is the L-vertex which has a completion procedure defined locally and need not inherit it from the vertex hierarchy.

```
* FOR SECTOR-L1-L2 OF VERTEX-1,
* CREATE A NEW SCHEMA: FACE-1 CONTAINING A CORNER FOR VERTEX-1
```

A vertex can be part of more than one polygon face because each sector of a vertex is possibly a corner of a different face. The completion process for vertices therefore consists of finding and attempting to match polygon face schemata for each of these sectors. For this first vertex, there are, as yet, no instances of faces to match. Thus, the vertex matches the generic polygon schema, FACE, which, in turn, creates a new instance, Face-1. The observation of Vertex-1 is seen by FACE as a cue to hypothesize the existence of another face instance. The matching of Face-1 by Vertex-1 activates a method associated with the new schema instance as a supergoal which promptly incorporates this vertex as a corner of its newly evolving description of a polygon face.

At this point, a face has been hypothesized containing

Vertex-1. Face-1 can now continue the search via one of two mechanisms. The method could use familar top-down search by calling the methods of other sub-schemata as subgoals. For this polygon, that would involve hypothesizing a particular polygon face type (either parallelogram, triangle, quadrilateral, or multilateral), then predicting what types of vertices each of the possible polygons could be composed of, and finally looking for those vertices. This mode is rejected by Face-1.

The second mode of search uses a bottom-up mechanism. The active method of Face-1 could compute dynamically a new set of expectations for the schema as a function of the information provided by Vertex-1. That is, Vertex-1 constrains the possible final interpretation of Face-1. Later, if evidence is discovered matching these expectations, the method will be resumed as a supergoal to continue Face-1's recognition. The schema can repeat this cycle of computing a new set of expectations and waiting for some expectation to be satisfied. Alternatively, at some point in the cycle, it can apply top-down techniques to the verification of its hypothesis.

* CREATE TWO PROCESSES BOUND TO EXPECTATIONS FOR LINE-1-6 AND LINE-1-5

The method of Face-1 creates two expectations and binds a new method to each. Expectations are indices of the schema associative retrieval mechanism. These indices must be knowable by both the schema creating the expectations and every other schema that can match them. Expectations are constraints on the final interpretation. For this problem, the lines connecting vertices are such a convention. Since both Face-1 and any vertices that can be part of Face-1 have access to these lines, they can be used as constraining information on the interpretation of vertices as polygon faces (Mackworth, 1975, 1977b).

After creating a set of expectations for Face-1, the schema's method is still active. Since the method is designed for the recognition of polygons, it can apply its techniques to the further recognition of its own schema, or it can relinquish control to the method of Vertex-1 which activated it. Should the method apply its techniques or defer to the more general knowledge of the scene schema at the top of the method hierarchy? The choice is completely heuristic, and is based on the methods appraisal of its probability of success. The decision is made from local information, such as how much evidence has been collected supporting the schema's hypothesis. Since very little evidence has been discovered supporting the recognition of Face-1, its method suspends itself.

```
* FOR SECTOR-L2-L3 OF VERTEX-1,
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-1
```

Control returns to Vertex-1 which is still actively pursuing its completion phase. This time, however, it finds a polygon face instance to match since the polygon instance hierarchy contains Face-1. The match fails. Sector-L2-L3 cannot be part of Face-1 because Sector-L1-L2 of the same vertex is already part of this face.

* CREATE A NEW SCHEMA: FACE-2 CONTAINING A CORNER FOR VERTEX-1

A new face recognizer, Face-2, is now hypothesized and its method activated as a supergoal of Vertex-1. As in the case of Face-1, the method creates the first corner of Face-2 by incorporating Sector-L2-L3 of Vertex-1 into its description.

* CREATE TWO PROCESSES BOUND TO EXPECTATIONS FOR LINE-1-2 AND
LINE-1-6

Again the face schema creates two expectations, binds methods to them, and returns control to Vertex-1 which activated it.

* FOR SECTOR-L3-L1 OF VERTEX-1,
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-2
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-1
* CREATE A NEW SCHEMA: FACE-3 CONTAINING A CORNER FOR VERTEX-1
* CREATE TWO PROCESSES BOUND TO EXPECTATIONS FOR LINE-1-5 AND
LINE-1-2

Vertex-1 continues looking for higher schemata to match, this time for its third sector, Sector-L3-L1. Attempts at matching the two existing face recognizers fail, and a new schema, Face-3, is created as before. Vertex-1 has now finished its completion phase by searching the polygon instance hierarchy for all schema instances of which its sectors might plausibly be part. In this example, this search involves only attempting to match the expectations of each face instance. In general, the search for higher matching schemata within the schema hierarchy can be arbitrarily complex, perhaps recursively involving recognition.

```
* SCENE SCHEMA OBSERVES VERTEX-2
```

Control has returned to the scene schema which continues to observe vertices in the heuristic ordering mentioned earlier. Vertex-2 is activated next to perform its completion phase. The vertex will attempt to find face recognizers to match, that is, face hypotheses of which it can be part.

```
* FOR SECTOR-L1-L2 OF VERTEX-2,
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-3
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-2
* FACE-2 HAS BEEN MATCHED BY VERTEX-2
```

Vertex-2 has found Face-2 and matched one of its expectations, namely, that it shares Line-1-2 with the matching vertex. The method associated with that expectation is activated; it uses the following scheme for recognizing faces. Two methods are defined to follow the periphery of the face being recognized. One method follows the periphery of the face in a clockwise direction, the other in a counter-clockwise direction. The search proceeds from some vertex via a connecting line to the next vertex and so on until the region comprising the face has been closed.

* VERIFY FACE-2 USING TOP-DOWN SEARCH

When enough evidence has been discovered using bottom-up search to conclude that the recognition of a polygon face is likely, the schema applies a top-down method to the verification of the hypothesis. The discovery of two neighbouring vertices is considered by the face recognizer to be enough evidence to switch from bottom-up to top-down search mode. In general, the choice of search modes is a heuristic decision made by a schema's methods.

* GET NEXT CLOCKWISE NEIGHBOUR VERTEX: VERTEX-3 FROM LINE-2-3
* INCORPORATE THIS VERTEX INTO A NEW CORNER OF FACE-2

The top-down method uses a clockwise tour of neighbouring vertices until a closed figure is formed. As each vertex is discovered, it is composed into a corner in the evolving description of the polygon face. If another face recognizer is discovered during this tour attempting the recognition of the same face, then its partially completed description is merged into this face recognizer's description and the tour is continued. Note that this transfer of information from one schema instance to another is straightforward and does not violate the criterion of schema modularity for the recognition model. Because both schemata are instances of the same stereotype, they already have access to information about the internal structure of each other.

```
* GET NEXT CLOCKWISE NEIGHBOUR VERTEX: VERTEX-6 FROM LINE-3-6
* INCORPORATE THIS VERTEX INTO A NEW CORNER OF FACE-2
* GET NEXT CLOCKWISE NEIGHBOUR VERTEX: VERTEX-1 FROM LINE-1-6
* THIS VERTEX IS ALREADY CONTAINED IN FACE-2
```

The top-down method continues to incorporate vertices into the description of Face-2 until it attempts to add Vertex-1. Since this vertex is already part of this face, the top-down method has found a closed region, the polygon face.

```
* COMPARE FACE-2 TO POLYGON MODEL
* COMPLETED FACE-2 IS A PARALLELOGRAM
```

It is the task of Face-2 to decide the type of its polygon from information obtained during its recognition. In this example, part of that information is the size of the angle of each vertex sector in the face's description. Face-2 compares this information to an internal model of triangles, parallelograms, quadrilaterals, and multilaterals, and decides that it is a parallelogram. In general, the models that a schema has of its stereotype concept guide the search process and are manifest as the methods associated with the schema.

* MATCH THIS FACE TO THE EXPECTATIONS OF POLYHEDRON RECOGNIZERS

Face-2 now begins its completion phase as a high-level internal cue in the recognition process. Part of the knowledge contained in the face schema is of what higher concepts in the schema composition hierarchy polygons can be a part. In this example, polygons can be part of only polyhedral objects. Face-2 will attempt to stimulate the recognition of particular polyhedron schemata by sending messages to each of the recognizers.

The completed face schema is characterized as a bottom-up generator of possible higher schemata in the composition hierarchy. The algorithm used by this generator is to attempt to match a single polyhedron schema instance of which it _must_ be a part. Failing to find such a schema, it will attempt to match every polyhedron schema that it _may_ be part of. In this latter case, a new polyhedron recognizer must also be created with this polygon as its first component face because it could, conceivably, be part of no existing polyhedron instance. Since there are no existing polyhedra instances to match, Face-2 matches the generic polyhedra schema.

```
* IS FACE-2 COMPATIBLE WITH THIS CLASS OF POLYHEDRA?
* YES
```

The polyhedron instance analyzes Face-2 to decide whether it will accept the completed face as part of its description. As shown by this example, the polygon schema can recognize more types of polygons than the polyhedron schema can accept as valid faces of polyhedra. This illustrates a modularity of the recognition model. A schema need only know about recognizing instances of its own stereotype. It does not need to know the requirements of other schemata. If a schema can be part of some higher schema, then it will be able to match the expectations of that schema during its completion phase. The creation of expectations within a schema and the matching of those expectations by another schema is characterized as a procedural constraint satisfaction process between two schemata.

```
* CREATE A NEW SCHEMA: POLYHEDRON-1 CONTAINING FACE-2
```

Parallelograms are valid faces of polyhedra in this example. A new schema instance is created and parallelogram Face-2 is incorporated into its new description. This recognizer could use either top-down or bottom-up techniques to further the recognition of a polyhedron. To use top-down search would involve hypothesizing the existence of particular polygon faces and then activating those sub-schemata as subgoals. Bottom-up search, on the other hand, would not require a commitment to a particular hypothesis. Instead, the polyhedron, based on its partial instantiation from Face-2, can create new expectations of what polygon faces would have to be discovered to propagate its recognition. Unlike top-down search where a committment must be made to a single hypothesis at a time, using bottom-up mechanisms, the polyhedron schema can create expectations for multiple possible polygon faces, and wait for the discovery of such a face or faces. In this example, since there is little evidence supporting a particular hypothesis, Polyhedron-1 will use the bottom-up mechanism.

* COMPUTE EXPECTATIONS ABOUT OTHER FACES OF POLYHEDRON-1

Polyhedra are composed of polygonal faces connected by common edges and vertices. To recognize instances of polyhedra, the polyhedron schema compares the instances of polygonal faces that it has matched with its own internal model of polyhedra. For this example, the model is based on the notion of edge connectedness as used by Guzman (1968). Any two polygonal faces sharing a common edge which is the shank of an ARROW vertex are part of the same object. These are called "connect" edges. Other edges shared by two faces are called "maybe-connect" edges. The use of this scheme is intended to demonstrate the concept of model guided recognition. A whole variety of information about the interpretation of three-dimensional scenes (Mackworth, 1977b) has been ignored for the sake of simplicity.

```
* FOR LINE-1-6 OF FACE-2,
* CREATE A PROCESS BOUND TO THE CONNECT EXPECTATION: LINE-1-6
* FOR LINE-3-6 OF FACE-2,
* CREATE A PROCESS BOUND TO THE CONNECT EXPECTATION: LINE-3-6
```

Polyhedron-1 creates expectations for each "connect" edge of Face-2, binding a method to each expectation. The "connect" edges of Face-2 are Line-1-6 and Line-3-6 which are the shanks of ARROW vertices 1 and 3 respectively.

```
* FOR LINE-2-3 OF FACE-2,
* CREATE A PROCESS BOUND TO THE MAYBE-CONNECT EXPECTATION:
          LINE-2-3
* FOR LINE-1-2 OF FACE-2,
* CREATE A PROCESS BOUND TO THE MAYBE-CONNECT EXPECTATION:
          LINE-1-2
```

Different methods are bound to "maybe-connect" expectations for each other edge of Face-2.

After the active method of Polyhedron-1 has completed the expectation phase, it may choose to terminate its execution or it may attempt to stimulate the recognition of polygon faces as cues to drive its own recognition and subsequently the recognition of a scene. Again, the choice is a heuristic decision. Polyhedron-1 must estimate its likelihood of success. However, the choice is not critical. If Polyhedron-1 decides to direct the observation of vertices from the input scene but it is an invalid hypothesis for this scene, control will soon migrate away from this schema. The vertices observed by Polyhedron-1 will stimulate the recognition of faces which will attempt to match the expectations of polyhedron schemata. If Polyhedron-1 is the wrong hypothesis, these faces will instead match other, possibly new, polyhedron recognizers thereby activating their methods instead.

On the other hand, if Polyhedron-1 is indeed a valid hypothesis but the method yields control to the weaker method of the Scene schema above it in the method hierarchy, this method will discover a polygon face in the scene that will succeed in matching an expectation of Polyhedron-1 thereby reactivating one of its methods. Thus search is seen to converge towards a valid interpretation. To the contrary, in the top-down model, a bad hypothesis can dominate the search process for a catastrophically long time.

For these reasons, Polyhedron-1 decides to apply a bottom-up technique for stimulating its own recognition.

* METHOD: SEARCH FOR OTHER FACES OF POLYHEDRON-1 BY OBSERVING VERTICES OF FACE-2 THAT MAY BE PART OF MORE THAN ONE FACE

This heuristic observes that each sector of the vertices of this polyhedron can be part of more than one face. This method is both more powerful than the simple enumeration of vertices used by the scene schema and more specialized because such a second face is likely to be part of this same polyhedron. However, its expertise is good only for recognizing polyhedra, not scenes. This method hierarchy is seen to incorporate a trade-off between power and applicability.

* POLYHEDRON-1 OBSERVES VERTEX-6

The next vertex observed is a FORK vertex, Vertex-6. Sector-L1-L3 of this vertex is already part of Face-2, but the remaining two sectors may be parts of two other, but, as yet, unknown faces.

* FOR SECTOR-L1-L2 OF VERTEX-6,
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-3
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-1
* FACE-1 HAS BEEN MATCHED BY VERTEX-6

Vertex-6 conducts its completion phase by attempting to match the expectations previously created by Faces-3 and 1. Vertex-6 cannot be part of any interpretation of Face-3 and that match fails. It is, however, successful at matching an expectation of Face-1, specifically, that it shares Line-1-6 with some vertex. The bottom-up method that had been bound to this expectation is activated as a supergoal of Vertex-6. This is the second vertex discovered for Face-1 and, as before, the Polygon schema attempts to verify the existence of this face instance using top-down search. In this example, the top-down search at the vertex level is achieved by a simple touring of the input schema instances because vertices were given as data. In general, however, goal directed search can be of arbitrary complexity, perhaps involving recursively the use of bottom-up mechanisms. A subgoal at any level can create expectations, bind methods to those expectations, and then attempt to make observations matching those expectations. The only restriction on the search mechanisms used by a subgoal is that it eventually must return a success or failure to its caller.

```
* VERIFY FACE-1 USING TOP-DOWN SEARCH
* GET NEXT CLOCKWISE NEIGHBOUR VERTEX: VERTEX-4 FROM LINE-4-6
* INCORPORATE THIS VERTEX INTO A NEW CORNER OF FACE-1
* GET NEXT CLOCKWISE NEIGHBOUR VERTEX: VERTEX-5 FROM LINE-4-5
* INCORPORATE THIS VERTEX INTO A NEW CORNER OF FACE-1
* GET NEXT CLOCKWISE NEIGHBOUR VERTEX: VERTEX-1 FROM LINE-1-5
* THIS VERTEX IS ALREADY CONTAINED IN FACE-1
```

* COMPARE FACE-1 TO POLYGON MODEL

* COMPLETED FACE-1 IS A PARALLELOGRAM

Face-1 has found a closed figure composed of vertices-1, 6, 4, and 5. A description of this polygon face is created with a corner for each vertex. The face is compared against the polygon schema's internal model of polygons. The figure is a quadrolateral having equal opposite angles, and is labelled as a valid parallelogram.

* MATCH THIS FACE TO THE EXPECTATIONS OF POLYHEDRON RECOGNIZERS

Face-1 begins its completion phase. It must find instances of polyhedron recognizers having expectations of being matched by a parallelogram. Face-1 will resume each successfully matched schema as a supergoal process. This supergoal activation is seen as an intelligent process scheduling mechanism. Instead of activating processes through some global scheduling algorithm, each completing schema can use its own domain specific scheduling algorithm. Processes are scheduled by the discovery of evidence suggesting their applicability as supergoals.

Face-1 will first attempt to match the expectations of an existing polyhedron recognizer that it shares a "connect" edge with. If it is successful, the method bound to the polyhedron's expectation will be resumed as a supergoal process. If Face-1 is able to find such a "connect" expectation and match it successfully, its task is done. Otherwise, Face-1 must attempt in sequence to match the expectations of all polyhedron recognizers with which it shares a "maybe-connect" edge, and it must also create a new polyhedron schema instance in case it is not part of any existing interpretation. For each polyhedron recognizer matched by the face schema, the method bound to the matched expectation is resumed as a supergoal process.

```
* ATTEMPT TO MATCH THE CONNECT EXPECTATIONS OF POLYHEDRON-1
* TRY LINE-4-6
* TRY LINE-1-6
*  A  CONNECT  EXPECTATION  OF  POLYHEDRON-1 HAS BEEN MATCHED BY
        FACE-1
```

Face-1 finds and attempts to match Polyhedron-1. This face schema shares a "connect" edge with Face-2 of the polyhedron, so the match is successful and the polyhedron's method is resumed.

```
* IS FACE-1 COMPATIBLE WITH THIS CLASS OF POLYHEDRA?
* YES
```

The method of Polyhedron-1 compares the polygon instance to its internal model of polyhedra. In this system, polyhedra are composed of both triangles and parallelograms. Face-1 is accepted and its interpretation results in the propagation of

Polyhedron-1's recognition.

```
* COMPUTE TRANSITIVE EDGE CLOSURE FOR THIS FACE
* FOR LINE-1-5 OF FACE-1,
*  CREATE  A  PROCESS  BOUND  TO  THE MAYBE-CONNECT EXPECTATION:
          LINE-1-5
* FOR LINE-4-5 OF FACE-1,
* CREATE A  PROCESS  BOUND  TO  THE  MAYBE-CONNECT  EXPECTATION:
          LINE-4-5
* FOR LINE-4-6 OF FACE-1,
* CREATE A PROCESS BOUND TO THE CONNECT EXPECTATION: LINE-4-6
```

Polyhedron-1 incorporates the new face instance into its description by computing which edges of Face-1 are edges already contained in Polyhedron-1. This is called its transitive edge closure. At the same time, a new set of expectations are created for each remaining edge of Face-1 not closed with edges of the polyhedron. These edges represent the cues by which other completing face schemata will be able to match this schema's expectations.

```
* DOES DESCRIPTION OF POLYHEDRON-1 SATISFY THE  CRITERIA  FOR  A
          COMPLETE POLYHEDRAL OBJECT?
* NO
```

At this point, the polyhedron checks to see if its instance is fully instantiated. It is not. There are still "connect" expectations remaining unsatisfied.

* METHOD: OBSERVE VERTICES THAT WILL DRIVE THE RECOGNITION OF NEIGHBOURING FACES

Having finished its expectation phase, Polyhedron-1 applies a bottom-up method by observing vertices which should stimulate the recognition of neighbouring faces. The recognition of such faces will hopefully satisfy its own expectations. It does this by observing three-line vertices of its component faces that may be part of some yet unrecognized face. Notice that there are now two methods of Polyhedron-1 simultaneously active. Both methods are observing vertices to stimulate the recognition of neighbouring faces. They may communicate with each other through their common data structure, the schema instance of Polyhedron-1. Both can contribute to its recognition, but their efforts must be coordinated. If one of the methods should decide that either a fully instantiated instance has been found, or that the polyhedron is a bad hypothesis, the other method must be suspended. It is no longer applicable to the discovery of a scene.

* POLYHEDRON-1 OBSERVES VERTEX-4
* FOR SECTOR-L1-L2 OF VERTEX-4,
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-3

* CREATE A NEW SCHEMA: FACE-4 CONTAINING A CORNER FOR VERTEX-4
* CREATE TWO PROCESSES BOUND TO EXPECTATIONS FOR LINE-4-6 AND LINE-3-4

The polyhedron method first observes Vertex-4 because it is a vertex of one of its component faces that could be part of another undiscovered face. Vertex-4 begins its completion phase by trying to find existing uncompleted face recognizers to match. Only Face-3 remains uncompleted, but the match fails. Sector-L1-L2 of Vertex-4 cannot be part of the background region represented by Face-3. As a result, Vertex-4 requests the polygon schema to create a new polygon face instance. Face-4 is created, added to the polygon instance hierarchy, and one of its methods is activated. The new recognizer incorporates Vertex-4 into its description and uses the vertex to create a new set of expectations for the schema. As before, these expectations consist of methods bound to patterns containing line cues, in this case, Line-4-6 and Line-3-4. Face-4 has finished its expectation phase and returns control to Vertex-4.

* FOR SECTOR-L3-L1 OF VERTEX-4,
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-4
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-3
* CREATE A NEW SCHEMA: FACE-5 CONTAINING A CORNER FOR VERTEX-4
* CREATE TWO PROCESSES BOUND TO EXPECTATIONS FOR LINE-3-4 AND LINE-4-5

Vertex-4 still has one remaining uncommitted sector, Sector-L3-L1. Neither of the lines of this sector can match Face-4. Nor can they match the expectations of Face-3, though they are part of the same region. There is not yet constraining information (possibly provided by Vertex-5) to link them to same interpretation. Thus, a new face, Face-5, is created in the manner described for Face-4.

The behaviour of Vertex-4 is characterized as a bottom-up generator of possible higher supergoals. Each time control is returned to Vertex-4, it generates another plausible polygon face of which to be a part. As well, it generates these faces in heuristic order, first attempting to activate those already existing polygon recognizers which it can match successfully. Only if that fails, will it generate a new polygon face instance.

```
* POLYHEDRON-1 OBSERVES VERTEX-6
```

Vertex-4 has finished generating supergoals and returns control to Polyhedron-1 which proceeds with its method of observing likely vertices, this time activating Vertex-6.

```
* FOR SECTOR-L2-L3 OF VERTEX-6,
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-5
* ATTEMPT TO MATCH THE EXPECTATIONS OF FACE-4
* FACE-4 HAS BEEN MATCHED BY VERTEX-6
```

Sector-L2-L3 is the only sector of Vertex-6 not already part of some polygon face. Vertex-6 first attempts to match Face-5 as a possible higher supergoal, but the match fails. An attempt at matching Face-4 succeeds, thereby activating a method of this face.

```
* VERIFY FACE-4 USING TOP-DOWN SEARCH
* GET NEXT CLOCKWISE NEIGHBOUR VERTEX: VERTEX-3 FROM LINE-3-6
* INCORPORATE THIS VERTEX INTO A NEW CORNER OF FACE-4
* GET NEXT CLOCKWISE NEIGHBOUR VERTEX: VERTEX-4 FROM LINE-3-4
* THIS VERTEX IS ALREADY CONTAINED IN FACE-4
* COMPARE FACE-4 TO POLYGON MODEL
* COMPLETED FACE-4 IS A TRIANGLE
```

Face-4 searches for its remaining vertices by hypothesizing another vertex, finding that vertex by following Line-3-6. It then confirms that its component lines and vertices form a closed figure and that its type is a triangle.

```
* MATCH THIS FACE TO THE EXPECTATIONS OF POLYHEDRON RECOGNIZERS
* MATCH THE CONNECT EXPECTATIONS OF POLYHEDRON-1
* TRY LINE-3-6
*  A  CONNECT  EXPECTATION  OF  POLYHEDRON-1 HAS BEEN MATCHED BY
          FACE-4
* IS FACE-4 COMPATIBLE WITH THIS CLASS OF POLYHEDRA?
* YES
```

Face-4 finds Polyhedron-1, the single member of the polyhedron instance hierarchy, and matches a "connect" expectation of this recognizer. Both Face-4 and Face-2 share the common "connect" edge, Line-3-6.

Expectations are equated with constraints. Line-3-6 constrains the interpretation of Face-4 and Face-2 to be part of the same object. The recognition process can be viewed as a procedural constraint propagation, where the flow of control through the schemata network is directed by the procedural methods attached to the nodes in the network.

```
* COMPUTE TRANSITIVE EDGE CLOSURE FOR THIS FACE
* FOR LINE-3-4 OF FACE-4,
* CREATE A PROCESS BOUND TO THE MAYBE-CONNECT EXPECTATION:
          LINE-3-4
* FOR LINE-4-6 OF FACE-4,
* LINE-4-6 MATCHES A CONNECT EXPECTATION OF POLYHEDRON-1
```

Polyhedron-1 incorporates the new face into its description by computing the transitive edge closure of Face-4 with its other faces.

```
*  DOES DESCRIPTION OF POLYHEDRON-1 SATISFY THE CRITERIA FOR A
          COMPLETE POLYHEDRAL OBJECT?
* YES: POLYHEDRON-1 IS A WEDGE
```

The addition of Face-4 has been enough new information to constrain the interpretation of Polyhedron-1 to a fully differentiated instance. There are no "connect" expectations remaining in the schema and each face in the schema's description is labelled "connect" with every other face with which it shares an edge. The completed instance is compared against the schema's internal model of polyhedra and is labelled a wedge since it is composed of two parallelograms and a single triangle.

* MATCH COMPLETED POLYHEDRON-1 TO THE EXPECTATIONS OF THE SCENE SCHEMA

The method of Polyhedron-1 now begins its schema's completion phase. However, it is not the only method of this schema currently active. There are two other concurrent methods observing vertices. In fact, these other methods were instrumental in observing the vertices that stimulated Polyhedron-1's successful recognition. These methods have now performed their task and they must be suspended.

The mechanism for achieving this coordination among concurrent methods is realized within completion. When a completing schema matches another schema as a supergoal, the method performing the completion and _all_ other active methods associated with the same completed schema instance are suspended. The implementation of this control structure mechanism as a Maya language primitive is discussed in the next chapter.

In this example, Polyhedron-1 performs its completion phase rather deterministically. It knows that polyhedra are only part of scenes. The scene schema has an expectation waiting to be matched by a completing polyhedron. The match is successful, and the method of the scene schema is activated.

* SCENE RECOGNIZER HAS FOUND A SCENE COMPOSED OF POLYHEDRON-1

For brevity, it is assumed that a scene is composed of a single polyhedral object. The scene schema has completed its recognition, finding a scene composed of Polyhedron-1. The program terminates sucessfully, returning to top-level the hierarchical description of the scene.

4.4 Conclusion

This example provides a high level description of the operation of this recognition model in a scene analysis program. Many of the ideas embodied in the model could only be partially illustrated by this single example. The precise specification of these ideas is made manifest in this thesis as an artificial intelligence programming language called Maya. The next chapter describes the design of this programming language.

CHAPTER 5: MAYA

5.1 Introduction

Maya is a multiprocessing LISP dialect that defines a number of extensions to the data types and control primitives of the LISP language. Maya generalizes the OBLIST and property lists of LISP to a new primitive data type called the object which can be used to represent schemata and to construct frame systems and semantic networks. Maya defines specific control structures for integrating top-down and bottom-up search in schema-based representations. The language provides, as well, primitives for pattern matching and for creating and scheduling multiple processes. An extensive interactive debugging system modelled after that of INTERLISP (Teitelman, 1974) is also provided.

Although Maya, as a programming language, is concerned with developing programming technology, the motivation behind the language has been the experimental implementation of the recognition model presented in this thesis. The natural realization of such a model is a programming language because it provides a general experimental vehicle with which to evaluate the ideas of the model. Maya focuses on general questions of representation and process involved in machine perception.

5.2 Language Overview

This section provides a general overview of Maya by describing the data types defined in Maya and their evaluation. A number of other general features of the language will also be mentioned. Issues of schema representation and recognition will be covered in subsequent sections of this chapter. A familiarity with LISP must be assumed in this discussion. The reader is referred to Appendix-B, the Maya Language Reference Manual, for details of the various Maya primitives involved.

5.2.1 Data Types

Maya extends the primitive data types of LISP to facilitate programming in schema-based systems. Maya is embedded in LISP and its data primitives are realized using LISP forms. Below is a BNF grammatical representation of the LISP implementation of Maya's data types. Angle brackets are used to delimit non-terminal symbols and the asterisk is used as the Kleene star, indicating zero or more repetitions.

```
   <FORM> --> <ATOM>
          --> <VAR>
          --> <LIST>
          --> <TUPLE>
          --> <OBJECT>
    <VAR> --> ?<ATOM>
   <LIST> --> (<FORM>*)
  <TUPLE> --> (@ <FORM>*)
 <OBJECT> --> (@OBJECT@ <TYPE><PAIR>*)
   <ITEM> --> (@ITEM@ <RES><PAIR>*)
   <PAIR> --> <ATOM><FORM>
```

A form in Maya may be an atom, variable, list, n-tuple, item, or object. In LISP, an atom is used both as the name of a set of properties and as a variable. In Maya, an atom, <ATOM>, is differentiated from a variable, <VAR>, because it is desirable to distinguish between the name of an object and the value of a variable. Names are represented by atoms, whereas variables are represented by atoms prefixed by a question mark.

Likewise, Maya differentiates between a list, <LIST>, and an n-tuple, <TUPLE>. Lists are used, as in LISP, to encode both actual lists and function calls. The value of a list is the result of applying the function indicated by the CAR of the list to the CDR of the list. The value of a tuple, however, is a tuple of the values of its elements. Tuples are used extensively by the pattern matching functions.

The most significant extensions to the data types defined of LISP are the inclusion of objects and items. The object, <OBJECT>, subsumes the property lists, usually called plists, of LISP atoms. It can be used to form schemata, frame systems, and semantic networks. A schema or frame may be thought of as a collection of named slots or relations. A node in a semantic network may be considered to be a set of named attribute/value pairs. <OBJECT>s can conveniently represent both of these structures.

An object is composed of a LISP list having the prefix,

@OBJECT@, a user-supplied type indicator, <TYPE>, and a set of named attributes, <PAIR>s. It should be noted that the different prefixes associated with objects, tuples, and variables permit Maya to type check the use of each data type. Within the definition of objects, an extra type indicator is provided to allow further user supplied type checking. Each <PAIR> in an object associates an atomic name with a <FORM>. A name is said to be defined by its binding in some object. Objects are created by the Maya primitive, OBJECT, which takes as arguments a type and a list of names and their new definitions. The function returns an object as value.

Whereas objects associate atomic names with their definitions, items associate variable names with their local values. In Maya, generators, such as the pattern matcher, always return items, i.e., they return a set of local variable bindings computed within the generator. An item, <ITEM>, is composed of a LISP list having the prefix, @ITEM@, followed by a processor reserved field, <RES>. Each <PAIR> of the item represents an association between a variable name and its value.

Maya maintains a stack composed of objects, items, and a number of internal structures used by the processor for recording procedure invocations and the states of generators and the pattern matcher. The object nearest the top of the stack is called the enclosing object. It represents the current schema, semantic network node, or immediate context in which the user is "operating".

A number of Maya primitives operate on the enclosing object. For instance, PUT* adds definitions to the enclosing object. This function takes two arguments, an atomic name and a form to be bound to that name in the enclosing object. If that name already exists in the object, its definition is replaced. The function, GET*, returns the definition of a name from the enclosing object, whereas the primitive, REM*, removes the definition of a specified name from within the object. A fourth primitive, called SELF, returns as value the entire enclosing object. This function provides a mechanism for obtaining a pointer to the current schema.

In order to make Maya and LISP as compatible as possible, a few of LISP's SUBRs have been altered to accept objects instead of PLISTs. LISP property list functions now recognize the header at the front of each object. This overhead is justified because it allows Maya to consider all of LISP's database, i.e., the OBLIST, as the <u>global object</u> of its database. Each LISP atom which has properties associated with it will have a PLIST of the following form bound to its CDR:

(@OBJECT@ @PLIST@ <PAIR>*).

For example, DEFUN and DEFINE always add their definitions to the enclosing object. This permits objects to contain function definitions local to the object. This mechanism will be used as one way to incorporate procedural knowledge into schemata.

Maya considers LISP's variables to be the set of global Maya variables. Maya uses a deep-binding scheme to access variables on the processor stack. These variable bindings may appear in LAMBDA expressions, PROG expressions, or as variables defined in items contained on the stack. The assignment functions, SET and SETQ, which are analogous to their LISP counterparts, will alter the value of the first variable of the specified name that they find on the stack. Otherwise, they will assign a new value to the LISP atom of the same name.

5.2.2 Evaluation

The evaluation algorithm for Maya data types is presented below in pseudo-LISP code. The behavior of the evaluator is then elaborated in some detail. Finally, the function types recognized by Maya are described.

When Maya is asked to evaluate a form, the following algorithm is used (but not the following implementation of that algorithm):

```
(DEFUN EVALUATE (FORM)
  (COND ((ATOM FORM)(FETCH-DEFINITION-FROM-STACK FORM))
        ((VARP FORM)(FETCH-VALUE-FROM-STACK FORM))
        ((TUPLEP FORM)(APPLY '@ (CDR FORM))
        ((OBJECTP FORM)(ERROR))
        ((ITEMP FORM)(ERROR))
        ((ATOM (CAR FORM))
            (APPLY (FETCH-FUNCTION (CAR FORM)) (CDR FORM)))
        (T(APPLY-LAMBDA (CAR FORM)(CDR FORM)))))
```

If FORM is an atom, then the definition of that name is fetched from its first occurrence on the stack. If it is not found on the stack, it is fetched from the global object, the PLIST of the LISP atom. When presented an atom, Maya attempts to find its definition within the first enclosing object. If that fails, it attempts to locate the definition within the next enclosing object, and so on. If it is not to be found in any enclosing object, then the definition of the atom present in the global object is returned. Note that for every atom there will always be a definition for it in the global object, although that definition may be the null object, NIL.

If FORM is a variable, its value is fetched from the first occurrence of the variable's name on the stack. The variable may appear on the stack in two different ways. It may be a local variable of some LAMBDA or PROG expression. Or it may appear as a variable bound in an item returned from a generator. In either case, the first local binding found is returned as the value of the variable. If, however, the processor cannot find the variable's name on the stack, Maya fetches the global value of the variable, that is, the LISP value.

If FORM is a tuple, it is evaluated by the tuple evaluator using inverse quote mode, that is, atoms and pattern variables are treated as constants. The value of a tuple is a new tuple of its values.

If FORM is an object or an item, an ERROR occurs.

If the CAR of FORM is an atom, the atom is assumed to be the name of a Maya or LISP function. The function is fetched from the first <u>definition</u> of the atom on the stack, else from the LISP PLIST of the atom. If the function fetched is of the types recognized by Maya, it is applied to the CDR of the form. If more than one function by the same name exists, the first one found is used. If no recognizable function can be found, an error occurs. A discussion of the function types recognized by Maya is presented below.

Otherwise, the CAR of FORM is not atomic and is assumed to be a Maya LAMBDA or QLAMBDA expression. The expression is then applied to the CDR of FORM. If the CAR is not a LAMBDA or QLAMBDA expression, an error occurs.

5.3 Representation

This section discusses the use of Maya language primitives for implementing schemata networks and for realizing procedural message passing and interpretation.

5.3.1 Schemata

Schemata and schemata networks are realized in Maya as objects. Maya defines a mechanism for considering an object as a schema stereotype and then creating multiple instances from that stereotype. The function, NEW, when applied to an object, creates an incremental copy of the object and assigns it type,

@INSTANCE@. Any changes made to the instance are reflected in the incremental copy and not the stereotype object. For example, a stereotype schema to represent the concept of "dog" could be constructed as follows:

(PUT* 'DOG (OBJECT 'GENERIC NAME 'DOG CLASS 'MAMMALIA)).

This expression creates within the enclosing object, a definition of the name, DOG, as an object having type, GENERIC, a reflexive pointer to its own name, and the indicator that dogs are of class, MAMMALIA. To create a specific instance of dog, the following expression could be then evaluated:

(NEW DOG NAME 'FLOYD OWNER 'BILL).

The name and ownership attributes of the dog instance are defined within the instance and not the stereotype. Fetching the name, CLASS, from the instance will return MAMMALIA but fetching NAME will yield FLOYD, the definition local to the instance. Advantages of this scheme are that it makes instances computationally inexpensive and changes made to a stereotype are immediately reflected in each descendent instance, unless specifically redefined by the instance.

The Maya primitive, SEND, is the basic mechanism for accessing schemata networks, for traversing arcs in semantic networks, and for procedural message passing and interpretation. Its form is:

(SEND <A1><A2> . . . <An>).

The arguments, <A1> through <An>, are called a message sequence. SEND evaluates each of its arguments in sequence returning the evaluation of its last argument, <An>, as its value. If the value of some <Ai> yields an object or an item, then it is pushed onto the processor stack thereby augmenting the environment for either atom definitions or variable bindings respectively.

The sequence, <A1> through <An>, is an encoding of a search procedure through the network. By pushing some object, <Ai>, onto the stack, the processor in effect "goes to" that object. Now the evaluation of <Ai+1> is computed from within the new enclosing object, perhaps itself yielding another object or item.

Knowledge may be represented in schemata in three different ways. First, it can be represented declaratively as either atoms defined within the schema object or as patterns in a tuplebase contained within the object. We saw above how atoms could be associated with definitions by creating a new object using either OBJECT or NEW. Atom bindings can also be added to existing objects by using SEND and PUT*. For example, to add a new schema slot to the generic schema, DOG:

```
          (SEND DOG (PUT* 'VIRTUE 'MANS-BEST-FRIEND)).
```

SEND first evaluates DOG which yields an object, the dog schema. This object is then pushed onto the processor stack, thus becoming the enclosing object. The evaluation of PUT* then adds the new definition of the atom, VIRTUE, to the enclosing object. That is, a new slot is added to the schema, DOG.

To access slots in a schema, it is again necessary to "goto" that schema's object. For example, the form:

```
(SEND DOG (PRINT CLASS) VIRTUE),
```

will first print MAMMALIA and then return MANS-BEST-FRIEND. Declarative knowledge may also be realized as patterns in associative databases defined within schemata. Mechanisms for pattern matching in schemata are discussed in the next section.

The second way knowledge can be represented in schemata is procedurally using local function definitions. In Maya, the property list of an atom is represented as an object bound to that atom. Since the EXPR function property is no different than any other property, function definitions can be local properties of objects. For example,

```
(DEFUN FOO (X) X)
```

creates, as expected, a LAMBDA expression bound to the name, EXPR, on the property list of the atom, FOO. However, evaluating the following form:

```
(SEND FOO EXPR)
```

returns the binding of the atom, EXPR, within the object, FOO, which is:

```
(LAMBDA (X) X).
```

As well, a new definition of FOO can be locally associated with a particular object, for instance:

```
(SEND DOG (DEFUN FOO (X) 'WOOF)).
```

Evaluating (FOO 'A) yields A but evaluating:

```
(SEND DOG (FOO 'A))
```

yields instead WOOF. Furthermore, the data structure created by this example is itself a network of objects which can be accessed as follows:

```
                    (SEND DOG FOO EXPR)
```

which yields

```
                    (LAMBDA (X) 'WOOF).
```

The third mechanism for representing knowledge in schemata is procedural attachment (Winograd, 1975). In Maya, both generators for performing top-down search and processes for realizing bottom-up search can be associated with tuple patterns in tuplebases local to schemata. Such procedures attached to patterns are Maya's mechanism for associating top-down and bottom-up methods with a schema's expectations, represented as patterns. Top-down and bottom-up methods will be discussed in sections 5.5 and 5.6 respectively.

5.3.2 Messages

The evaluation of SEND can also be defined recursively in terms of sending messages. The value of SEND applied to the sequence, [<A1><A2> . . . <An>], is recursively the result of sending <A1> the message, [<A2> . . . <An>], and so on, returning finally the value of <An>. Thus, evaluating a message sequence is sending the <u>value</u> of the CAR of the sequence a message, the CDR of the sequence.

An example follows which illustrates in Maya the construction of instance hierarchies, and the use of some of the Maya primitives to interpret simple procedural messages. This example is extracted from a Maya program to play "Twenty Questions". In this program, an instance hierarchy is used as a discrimination tree to interpret a series of responses from the user. Each node in the net contains a question plus the possible branches of the tree to take depending on the user's response. Although the example is simplistic, it demonstrates the use of Maya network structures and simple message passing and interpretation.

```
(DEFUN TWENTY-QUESTIONS @EXPR NIL
  (PRINT '"PLEASE THINK OF SOME OBJECT")
  (SEND TOP-NODE
        (ASK 20)))
```

The top-level function sends the Top-Node of the discrimination net a message to ASK 20 questions.

```
(DEFUN ASK @EXPR (N)
  (AND (ZEROP ?N) (RETURN '"YOU WIN" 'TWENTY-QUESTIONS))
  (PRINT QUESTION)
  (SEND (EVAL (READ))
        (ASK (SUB1 ?N))))
```

In this example, the message sent to each node is procedural. It says: "Check to see if we have asked more than 20 questions; if so, then lose. Otherwise, print your question and recursively send this same message (minus 1 from N) to _your_ choice of next node." A few of the semantic net nodes of this program follow:

```
(PUT* 'WIN-NODE (OBJECT 'NODE))
(SEND WIN-NODE
      (DEFUN ASK @EXPR (N)
        (PRINT '"I WIN !!")
        ?N))

(PUT* 'NODE4
      (OBJECT 'NODE
         QUESTION '"IS IT A SNAKE?"
         YES WIN-NODE
         NO  NODE7))

(PUT* 'NODE1
      (OBJECT 'NODE
         QUESTION '"HOW MANY LEGS DOES IT HAVE?"
         ZERO NODE4
         TWO  NODE5
         FOUR NODE6))

(PUT* 'TOP-NODE
      (OBJECT 'NODE
         QUESTION '"IS IT ANIMAL, VEGETABLE, OR MINERAL?"
         ANIMAL    NODE1
         VEGETABLE NODE2
         MINERAL   NODE3))
```

In the recursive calls to ASK, how does the recursion terminate? On failure, that is, after the twentieth question, the first line in the body of ASK will terminate the recursion. On success however, the process is quite different. At NODE4, if the user answers "yes" to the question, "Is it a snake?", a node called WIN-NODE is sent the message,

(ASK (SUB1 ?N)).

Within WIN-NODE is a local definition of the function, ASK. WIN-NODE has its own interpreter for this particular message. It always interprets the message as a successful end of the game.

This example serves to illustrate a few basic features of

Maya. Through the use of PUT* and OBJECT, semantic networks can be constructed. As well, SEND can be used to send simple procedural messages between objects. In this example, an answer to the current question is elicited from the user. If his answers are restricted to atoms, then the expression,

(EVAL (READ))

will fetch an object, the next node of the discrimination tree. SEND pushes this object onto the stack, and subsequently evaluates in this new context the rest of the message, which is the recursive call to ASK.

SEND fetches each function from its name's first definition on the stack. This provides the mechanism to allow the object receiving a message to perform its own local interpretation of that message. In this example, when WIN-NODE is sent the message, ASK, the definition of ASK defined locally is used because SEND had previously pushed WIN-NODE, including its local definition of ASK, onto the stack.

Suppose, however, that the user types an atomic answer that the program is not expecting. The result would be unpredictable because Maya will fetch the first definition of the atom it can find. For a random answer, the definition is likely to be NIL. If we replace the form, (EVAL (READ)) in ASK with:

(GET* (READ) '(WHAT?))

and define a new function, we now have a solution:

```
   (DEFUN WHAT? @EXPR NIL
     (PRINT '"WHAT ?")
     (RETURN (ASK ?N) 'ASK))
```

GET* always fetches definitions from within the current enclosing object. If GET* fails to find the name defined in the enclosing object, the second argument is evaluated. In this case, (WHAT?) which just asks the same question again.

5.4 Pattern Matching

Syntactic pattern matching provides a primitive level of comparison based only on syntactic form. Maya defines a number of pattern matching primitives similar to those provided in most recent artificial intelligence programming languages, (Hewitt, 1972) (McDermott, 1973) (Davies, 1973). Maya uses its pattern matcher to compare tuples, to perform associative retrieval of tuple patterns from databases, to implement generators and processes, and as a syntactic base upon which to build semantic associative retrieval over schemata systems. This higher network matching may involve active search and deduction (Bobrow & Winograd, 1977). Semantic matching over schemata structures is, as has already been pointed out, another characterization of the recognition problem.

The pattern matcher in Maya is called MATCH. The form of a call to MATCH is as follows:

```
(MATCH <pattern><db>[<else>]).
```

Its first argument, <pattern>, is matched against a datum, <db>, which is either another tuple or an associative database of tuples called a tuplebase. The matcher, on a successful match, returns an item composed of the bindings of any pattern matching variables assigned during the match. On failure, an optional third argument, <else>, is evaluated. This optional argument is called a failure exit and is used to control failure driven search.

An example of pattern matching in Maya is given by the following:

```
(MATCH '<ON !:X !:Y> '<ON B1 TABLE> '(FOO)).
```

The evaluation of this expression results in a successful match returning as value an item containing the variables X bound to B1 and Y bound to TABLE. If the match had failed, the form, (FOO), would have been evaluated and returned as the value of MATCH.

The item returned from the pattern matcher may also contain a reactivation tag. This tag is a pointer to the current invocation of the matcher in order that it may be recalled for another match. Because the pattern matcher is implemented as a generator, MATCH provides a non-deterministic search mechanism for tuplebases.

Tuplebases are accessed by the pattern matcher, various database maintenance functions, and a number of control structure functions for scheduling both top-down and bottom-up methods. Any tuple pattern in a tuplebase may have an associated value which is included in the item return from the matcher as the value of the distinguished variable, "?*". Tuplebases are composed of an inverted index structure which uses Maya objects to represent each level of the index. Since they are formed from a primitive data type, tuplebases may as well be operated on with ordinary object accessing functions. For example, to create a new tuplebase of assertions within the enclosing object:

```
(PUT* 'ASSERTIONS (OBJECT 0))
```

Notice that as many tuplebases as desired may be created in this manner.

In order to add a pattern assertion to this tuplebase, the following form could be evaluated:

```
(PADD '<WOOF BE DOG> ASSERTIONS).
```

Finally, to delete assertions from this tuplebase:

```
(PREMOVE '<PUSHKIN HAS FLEAS> ASSERTIONS).
```

5.5 Top-down Methods

Top-down search in Maya is based on the notion of generators which are functions that may be recalled a number of times for a single invocation. Alternatively, generators may be thought of as functions that retain an internal state between calls. There are four types of generators in Maya:

1. MATCH
2. QLAMBDA expressions and QEXPR functions.
3. B1ST and D1ST.
4. NEXT and FAIL.

The pattern matching function, MATCH, has been already described. The second type of generator is realized by combining the concept of QLAMBDA expressions from QLISP (Davies, 1973) and Maya items. The mechanism is a generalization of the generators defined in CONNIVER (McDermott & Sussman, 1973). Each QLAMBDA expression has a tuple pattern as its argument list. QLAMBDAs are invoked by matching a pattern in the form of a tuple against either the pattern argument of a specific QLAMBDA expression or against a tuplebase of QLAMBDA expressions. QLAMBDAs always return items as values. As in MATCH, the item is composed of the bindings of the pattern variables assigned during the pattern match plus a possible reactivation tag for the generator. QLAMBDA expressions may be given names by creating QEXPR type functions with DEFUN.

QLAMBDAs and QEXPRs take optionally one or two arguments:

```
((QLAMBDA <arg-pattern><body>) <pattern>[<else>]) or
(<qexpr><arg-pattern>[<else>]).
```

The first argument, <pattern>, must be a tuple and is the pattern to be matched against the argument list, <arg-pattern>. The second argument, <else>, is evaluated if the pattern match fails or if the QLAMBDA expression evaluates a call to FAIL. <Else> defaults to NIL. Since QLAMBDAs and QEXPRs are treated by the interpreter as any other function, they may be applied, mapped, and evaluated.

QLAMBDA expressions may be stored in tuplebases by binding them to patterns identical to their pattern arguments. A mechanism is thus realized for associating top-down methods with patterns. Methods are added to a method tuplebase as follows:

```
        (PADD <arg-pattern><method-base><qxl>),
```

where <arg-pattern> is the pattern argument of the method, <qxl> is either a QEXPR name or a QLAMBDA expression, and <method-base> is a tuplebase of methods. Methods are deleted from tuplebases in a similar fashion:

```
                        (PREMOVE <arg-pattern><method-base>).
```

The third type of generator is used to access and then invoke QLAMBDA expressions in tuplebases by matching their associated patterns. Two different mechanisms are provided. D1ST searches a tuplebase of QLAMBDA expressions in depth-first order. The form of calls to this function are:

```
                        (D1ST <pattern> <db> [<else>]).
```

D1ST will fetch from the tuplebase, <db>, a QLAMBDA expression matching <pattern> and then invoke that function. A successful value returned from the QLAMBDA will also be returned from D1ST with the addition of a reactivation tag. If the tag is later used to restart the generator, D1ST will recall the QLAMBDA expression to generate another item. D1ST will continue to recall that same QLAMBDA until it fails to generate any new items. Then D1ST will return to the tuplebase to search for another QLAMBDA expression matching <pattern>.

Similarly, B1ST searches a tuplebase of QLAMBDA expressions in breadth-first order. The form of this primitive is:

(B1ST <pattern> <db> [<else>]).

B1ST will invoke the first QLAMBDA expression it finds that matches <pattern>, and return the item yielded by the QLAMBDA with the addition of a reactivation tag as its own value. However, when this generator is resumed again, B1ST will attempt to find a different matching QLAMBDA expression in the tuplebase. Only after it has invoked once all matching QLAMBDA methods will it recall each suspended QLAMBDA for a second time, and then each for a third time, and so forth.

The last type of generator defined in Maya includes the control primitives for restarting generators from the reactivation tags returned in their items. The functions, NEXT and FAIL, are used for loop-driven and failure-driven search respectively. For a comparison of the two approaches, see Sussman and McDermott (1972).

NEXT, which is analogous to Conniver's TRY-NEXT, is a function of two arguments:

(NEXT <item> [<else>]).

<Item> must be an item created by another generator. NEXT looks for a reactivation tag in the item. If it finds such a tag, the attached generator is resumed. Otherwise, if no tag is found or the resumed generator fails, NEXT evaluates <else>, its optional failure form. For example, to find in a tuplebase, TOYS, the names of all boxes, the following expression could be used:

```
(PROG (X)
  (SETQ X (MATCH '<BOX !:B> TOYS '(RETURN NIL)))
```

```
   LOOP
    (SEND ?X (PRINT ?B))
    (SETQ X (NEXT ?X '(RETURN NIL)))
    (GO LOOP))
```

Failure-driven search is realized in Maya by using the FAIL function in conjunction with another primitive, ELSE. FAIL is a function of no arguments that causes Maya to begin discarding control frames from its processor stack until an item is found on the stack containing a reactivation tag. The generator attached to this tag is then restarted from that failpoint.

The function ELSE provides the necessary mechanism for controlling failure-driven search. It is a very simple function of one argument which creates an explicit failpoint with that unevaluated argument. ELSE returns an item containing only a tag bound to this failpoint. When the tag is reinvoked by FAIL, the argument is finally evaluated, thereby providing a mechanism for capturing failure. The mechanism is called a failure block and is illustrated in the following example:

```
  (SEND (ELSE '(FAILURE-EXIT))
        (MATCH '<BLOCK !:X> TOYS '(FAIL))
        (MATCH '<COLOUR ?X BLUE> TOYS '(FAIL))
         .
         .
        (SUCCESS-EXIT))
```

This program segment attempts to find a blue block from the database of toys. If it is successful, control passes through the block. Otherwise, a failure exit is taken. Failure blocks define a local backtrack search mechanism that follows separate control paths depending upon whether the local search succeeds or fails.

5.6 Bottom-up Methods

Bottom-up search is realized in Maya via multiprocessing. Processes consist of semi-autonomous procedures. A number of processes may co-exist simultaneously and may or may not be associated with particular schemata. Processes may be created, invoked, destroyed, and resumed by other processes. A process may terminate or suspend itself or it may via the COMPLETE function suspend itself and all other processes associated with a particular schema.

The procedure body of each process consists of a QLAMBDA expression. When a process is created, a specified pattern is matched against the pattern of the QLAMBDA expression. Processes may be suspended to patterns in a specified tuplebase of processes. Suspended processes may be resumed by matching those patterns.

Processes are created in Maya via the function, PROCESS, which has the following form:

(PROCESS <schema> <ql> <pattern> [<else>]).

<Pattern> is matched against the pattern argument of the QLAMBDA expression. Should the match fail, <else> is evaluated if present or defaults to NIL. Otherwise, a new process is created with the QLAMBDA expression as the body of the process. The process is associated with an object, <schema>, or if NIL is

specified, with no object. The association of processes with schemata is utilized by the system function, COMPLETE, which is described later.

Once a process is created, its procedure body is executed until it terminates or is suspended. A process may be terminated by executing the last form in its procedure body or by executing the EXIT or FAIL functions. Processes may be suspended via the SUSPEND or COMPLETE functions. SUSPEND is a primitive of two arguments, a tuple, <pattern>, and a tuplebase, <db>, as indicated in:

(SUSPEND <pattern> <db>).

The current running process is suspended to the pattern and is stored in the specified tuplebase. Control returns to the process which invoked the now suspended process.

Maya recognizes two types of processes in tuplebases; <u>suspended processes</u> and <u>named processes</u>. A suspended process consists of the segment of the stack representing the current state of evaluation of the process plus an associated schema. A named process consists of either a QLAMBDA expression or a QEXPR name plus an associated schema. Suspended processes are added to a tuplebase via the SUSPEND function. Named processes are added to a tuplebase as follows

(PADD <pattern><db>(NAMED-PROCESS <ql><schema>)).

This form adds to the tuplebase, <db>, under the pattern, <pattern>, a named process whose procedure body, <ql>, is a QLAMBDA expression or a QEXPR name. <Pattern> and the pattern argument of <ql> must be identical. If <schema> is an object, the process is associated with the schema represented by the object. If <schema> is NIL, the process is not associated with any schema.

The major differences between named processes and suspended processes are the following. A suspended process may be resumed only once for each appearance in a tuplebase. When it is invoked, the pattern to which it was bound is deleted from the tuplebase. On the other hand, a named process may be invoked multiple times from the same pattern in a tuplebase, each invocation resulting in the implicit creation of a new process, and the pattern is not deleted from the tuplebase.

Suspended and named processes are resumed by the RESUME function which takes three arguments: a pattern, <pattern>, a tuplebase, <db>, and an optional failure form, <else>:

(RESUME <pattern> <db> [<else>]).

<Pattern> is matched against the tuplebase, <db>. If the match fails, <else> is evaluated. Otherwise, the process bound to the matched pattern in the tuplebase is resumed.

It will be noticed that all the top-down and bottom-up control primitives return items as values. Therefore, functions such as RESUME are also generators. In the case of process primitives, they are <u>bottom-up generators</u>. The item returned as value from RESUME can be operated on by NEXT and FAIL to sequentially generate and then to resume every process matching pattern.

A schema may have more than one of its associated processes active at any one time. The stack thus may contain a number of processes associated with the same schema. All of these processes have a common purpose, to whit, the recognition of an instance of the schema's stereotype. When a schema is successful in its recognition, the efforts of all its processes need to be suspended. This is called completion in the recognition model. The following function, COMPLETE, provides a mechanism for suspending all the processes of a schema and resuming the process of another specified schema or schemata:

(COMPLETE <pattern> <db> [<else>]).

<Pattern> is matched against the tuplebase of processes, <db>. On success, the current process, P1, is suspended as described below to a reactivation tag. COMPLETE then resumes the process matching <pattern>, P2.

This complex function is the main control structure mechanism in Maya for realizing the control aspects of the recognition model. COMPLETE supports supergoals, heuristic method scheduling, and method hierarchies. The mechanism is based on the assumption that for machine perception tasks all top-down and bottom-up methods associated with a particular schema are concerned with the ultimate recognition of that schema instance. Therefore, when a method associated with a particular schema has concluded that the recognition of its instance is complete, all concurrently active methods associated with that same schema are no longer needed. They must be suspended. In order to achieve this result, COMPLETE searches the processor stack to find every occurrence of processes associated with the same schema as the current process (labelled P1 above). Every such process and its sub-processes are then suspended. Finally, P2 is resumed.

5.7 Conclusion

This chapter has presented a brief overview of a new artificial intelligence programming language. The language was discussed from the perspective of realizing the procedural mechanisms defined in the recognition model. Although it is impossible to completely characterize a high-level programming language that introduces new complex control and data structures in the space available here, a number of its salient features have been discussed. A tutorial on Maya, designed to augment the descriptions presented here and to supplement the language reference appearing in Appendix-B, is now in preparation.

CHAPTER 6: RECOGNITION REVISITED

6.1 Perspective

In this final chapter, the author takes the opportunity to step back and re-examine the work of this thesis. A recapitulation of the recognition model is made from the perspective of its contribution to a theory of machine perception. Finally, applications for the model are discussed indicating some promising directions for future research.

6.2 Recognition Revisited

Perception has been characterized as an active recognition task. Such a view of perception is held by Bobrow and Winograd (1977):

> Reasoning is dominated by a process of recognition in which new objects and events are compared to stored sets of expected prototypes, and in which specialized reasoning strategies are keyed to these prototypes (p.4).

This process is more than a simple passive retrieval from memory of a stored description of the thing perceived. Experience is much too varied and complex to depend on such a mechanism. We are constantly experiencing new situations, seeing new objects, understanding new sentences. We probably never perceive exactly the same experience in the same situation more than once. Thus, perception is a generative process, composing new descriptions of experience in terms of a stored finite knowledge of the world (Chomsky, 1957). Jaynes (1976) argues that perceiving an experience is a process of arriving at a metaphor to describe that experience.

> Generations ago we would understand thunderstorms perhaps as the roaring and rumbling about in battle of superhuman gods. We would have reduced the racket that follows the streak of lightning to familiar battle sounds, for example. Similarly today, we reduce the storm to various supposed experiences with friction, sparks, vacuums, and the imagination of bulgeous banks of burly air smashing together to make the noise. None of these really exist as we picture them. Our images of these events of physics are as far from the actuality as fighting gods. Yet they act as metaphor and they feel familiar and so we say we understand the thunderstorm (p.52).

Thus, descriptions of experience are metaphorical. We perceive the sensory world in terms of our stored descriptive knowledge of that world. In order to develop a theory of machine perception, we must characterize mechanisms for emulating this generative process. The specification of a theory faces two major issues: representation and recognition. What is the form and organization of memory and what types of procedural mechanisms can perform recognition on that representation?

The psychological question of memory organization is far from resolved. There is experimental evidence supporting both imagery and structural representations (Yuille, 1976). However, Chase and Simon (1973) have shown that chess masters are much better at remembering chess board positions from actual chess games than from random board arrangements. The expert's ability at remembering random boards approaches that of non-chess players, thus indicating a definite dependence on structural descriptions. It seems clear that high-level reasoning depends predominantly on schematic mechanisms (Pylyshyn, 1976), although perception most certainly makes use of both imagery and schemata.

A major aspect of this thesis is the characterization of perception as an active process that exploits heuristic knowledge of the world. The value of active heuristic knowledge has been demonstrated by Winograd's (1973a) natural language system, SHRDLU. His system was a significant advance in the state of the art, incorporating procedural semantics in a language understanding system. The apparent lesson of his research was that procedural semantics is more powerful than declarative semantics for performance-based systems. A more insightful lesson is that procedural semantics coupled with hypothesis-driven search provides a mechanism for introducing heuristic knowledge to guide the perception process. Procedures are only the vehicle for the improvement in performance, not the reason. The same capability can be obtained in purely logical systems, as Hayes (1973) has argued, by defining deductive control operators within the formal logic system. Again, such mechanisms provide the vehicle for introducing active knowledge into the perceptual process.

Unfortunately, the incorporation of heuristic guidance to hypothesis-driven recognition has not been a sufficiently powerful mechanism to solve the machine perception problem. Winograd (1973b) has noted the appearance of a "complexity barrier" to the advancement of the art. The barrier arises from the dependence on top-down search mechanisms. Such methods require the system to hypothesize the correct interpretation for some sensory input before it can be found. The deficiency is not hypothesis-driven search, but the fact that a particular schema must be chosen as a plausible interpretation and attempted before any of its heuristic recognition knowledge becomes available to direct the search process. Its expertise comes too late!

Hypothesis-driven recognition has also been proposed for schemata representations (Minsky, 1975). The recognition model described by Kuipers (1975) is probably the best known. As was pointed out, his model attempts to avoid the inadequacies of top-down search by appealing to a failure-driven similarity network to recommend likely alternative hypotheses. This mechanism does not confront the real problem. Sensory data is a highly ambiguous encoding of experience. The interpretation of sensory data requires a methodology that exploits contexts and can tolerate large degrees of non-determinism. The top-down model appeals implicitly to the "little man in the head" or homunculus theory of perception (Pylyshyn, 1973). Consider the following verbal example given by Kuipers (1975):

> A frame [schema] represents a certain limited domain, and hence a range of variation for objects which belong to that domain. As we saw in the room scenario, the features of a frame may be frames in their own right, embodying ranges of variation. On entering a room, you are prepared for certain typical pieces of furniture. A park bench or diamond-encrusted throne would be outside the permissible range of variation in this frame. Such an anomaly may indicate to the correction mechanism that another frame is called for (p.159).

In this description, the perception problem has been finessed by assuming that it is easier to recognize a park bench than a room. They are, in fact, problems of the same order of complexity. Postponing the problem will not solve it. Relying on the expectations of the room schema to handle all or even most possible contents of a room abdicates the responsibility of the search process to failure mechanisms. Upon entering a room containing an unanticipated object, the search process will blindly select one bad hypothesis after another until an "appropriate" schema is found. Only then will that schema's domain-specific knowledge be available to guide the recognition. That knowledge was needed much earlier. One must conclude that hypothesis-driven recognition is not the mechanism upon which to build machine perception regardless of the type of auxiliary attachments added to improve its performance.

In order to surmount this complexity barrier, machine perception requires both hypothesis and data-driven recognition. Hypothesis-driven search applies heuristic methods associated with the recognition of a particular schema once a commitment has been made to that schema. Data-driven search provides the means to select likely hypotheses based on the discovery of supporting evidence and cues. Top-down and bottom-up search methods can be integrated in a synergistic manner. Bottom-up search drives the activation of plausible higher schemata as supergoals. After being established as likely hypotheses, these schemata attempt to confirm their recognition both by using top-down search via subgoaling to sub-schemata and by observing cues in the input data to drive the bottom-up search for other schemata.

Recently, a number of other models have been proposed for perception. Mackworth (1977c) has offered the cyclic model depicted in Figure 6.1a which he attributes to the original work of Roberts (1965). In this model, perception is seen as an iterative process. The discovery of cues invoke appropriate models. Models attempt to verify their hypotheses by observation. Successfully recognized hypotheses cause the elaboration of the consequences of their models, resulting in the discovery of new cues.

Figure 6.1b illustrates a similar cyclic model of perception given by Neisser (1976). Schemata represent expectations which direct the exploration of the sensory world. Exploration results in observations which match these

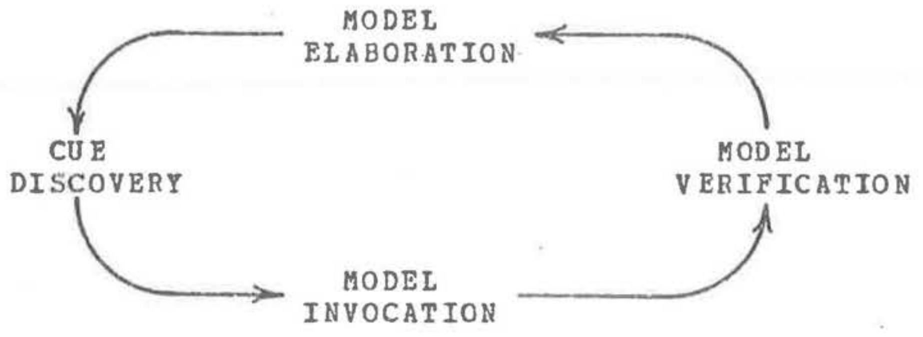


Figure 6.1a

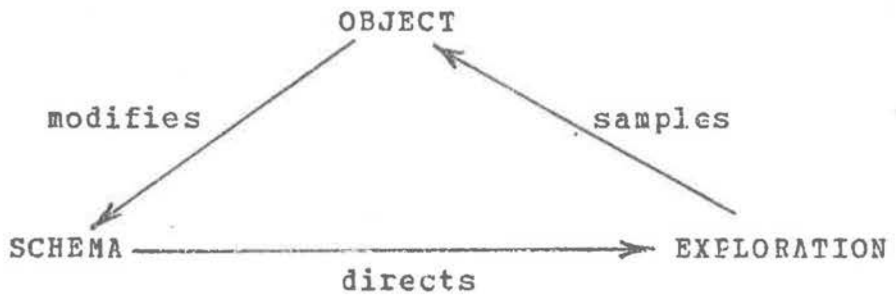


Figure 6.1b

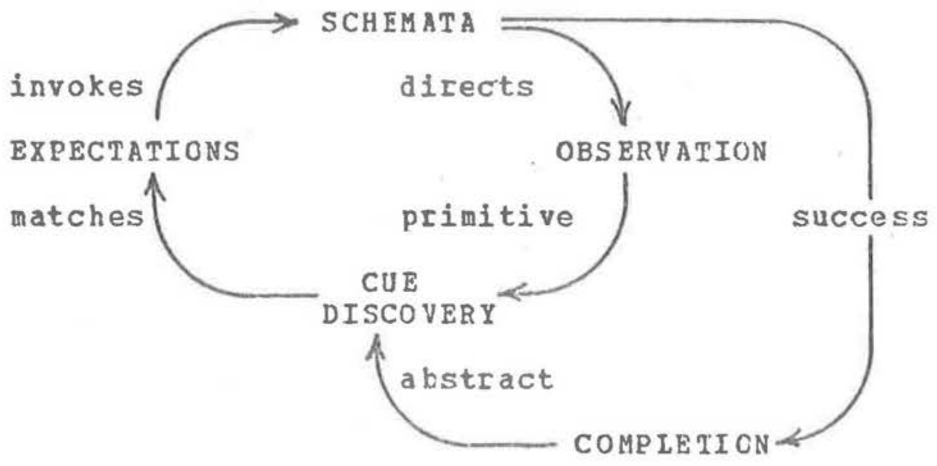


FIGURE 6.1C

expectations thereby modifying the schemata and propagating the process. Neisser uses this model to stress the inherently sequential nature of perception involving the modification of schemata over time.

Although each author allows for the existence of a cue/model hierarchy, in neither model is a mechanism for the perception of cues as abstract entities sufficiently elaborated. Figure 6.1c illustrates the recursive model of perception presented in this thesis. The model can be seen to generalize the purely cyclic models of Figures 6.1a and 6.1b. As in these models, recognition follows a cyclic path of cue discovery and schemata invocation. A particular schema may pass through this cycle a number of times. However, when a schema's recognition is completed, the recognition cycle ascends one level in a hierarchy of cues and models. Schemata recognized at one level becomes cues in the recognition at the next higher level in the hierarchy.

Despite the fact that the necessity of cue/model hierarchies in the cycle of perception has been clear for some time (Mackworth, 1976, 1977c), a mechanism for achieving this goal has not been specified. The major contribution of this thesis is the precise characterization of such a mechanism.

The notion of completion provides an explicit mechanism for capturing the recursive nature of perception. When a schema has recognized a fully specified instance of its stereotypical concept, it must return that success to one or more higher schemata. If the schema was activated as a subgoal by a higher schema, it must return its completed description to that schema. Otherwise, the completed schema has been recognized using bottom-up methods and has no explicit caller. It then exists as an abstract cue which attempts to match the expectations of higher schemata activating them as supergoals. Rumelhart and Ortony (1976) have addressed similar control structure issues for schemata.

> It may be helpful to think of these processing issues in terms of a computer programming metaphor, for one can think of a schema as being a kind of procedure. Procedures have subroutines and one can think of the activation of a schema as being like the invocation of a procedure. The variables of a schema are thus analogous to the variables of a procedure while the sub-schemata are analogous to the subroutines which may be invoked from within it. The activation of subschemata within a schema is like the calling-up or invocation of the subroutines within a procedure. This is the paradigm case of conceptually-driven processing. However, unlike ordinary procedure calls, in which the flow of control is only from procedure to subroutine, the flow of control in a schema system operates both ways. It is as though a given procedure not only could invoke those procedures in which it was itself a subroutine (data-driven processing). Finally, one must imagine these procedures as all operating simultaneously (p.46).

The realization of this programming metaphor as an operating programming language is the second major contribution of this thesis. Maya defines explicit language primitives for creating schemata and schemata networks, for associating procedural methods with schemata, and for invoking those methods both as subgoals using conceptually-driven search and as supergoals using data-driven techniques. As well, Maya utilizes the completion aspect of the recognition model as a multiprocess scheduling mechanism for simulating the concurrent application of methods.

6.3 Applications and Future Research

The issues addressed in this thesis are currently of interest in a number of research areas. For this reason, the recognition model and its realization as Maya should have general applications in such perception research as machine vision, natural language understanding, and episode understanding. Moreover, problems of control in automatic deduction systems are similar to the control structure issues in machine perception. Issues of integrated hypothesis-driven and data-driven recognition are analogous to similar issues of backward and forward deduction. The ideas developed in this model concerning active heuristic guidance and concurrent methods should also have application there.

Future research will focus on a specific task domain that exhibits the following four criteria. First, the task must have a well-defined semantics, preferably an explicit conventional semantic representation such as exhibited by sketch maps (Mackworth, 1977a). Second, in order to exploit fully the advantages of integrated top-down and bottom-up recognition, the problem should have a highly ambiguous input data representation. Third, the problem should inherently have a hierarchical knowledge representation in order to demonstrate the advantages of recursive cue/model hierarchies in machine perception. And finally, the domain must be generally accepted as a a perceptual task for which previous recognition mechanisms have been shown to be inadequate. Possible research tasks exhibiting these criteria include the interpretation of LANDSAT video images, the analysis of electronic circuit schematic diagrams (Stallman & Sussman, 1977), and the understanding of handwriting.

BIBLIOGRAPHY

AHO A. & ULLMAN, J. (1972) The Theory of Parsing, Translation, and Compiling, vol 1, Prentice Hall, Englewood Cliffs, N.J., p.320.

BARTLETT, F.C. (1932) Remembering, Cambridge Univ. Press, Cambridge, England.

BOBROW, D.G. & NORMAN, D.A. (1975) Some Principles of Memory Schemata, in D.G.Bobrow & A.Collins (eds.), Representation and Understanding, Academic Press, New York.

BOBROW, D.G. & RAPHAEL, B. (1974) New Programming Languages for A. I. Research, Comp. Surveys, vol 6, pp.153-174.

BOBROW, D.G. & WEGBREIT, B. (1973) A Model and Stack Implementation of Multiple Environments, CACM, Oct. 1973, vol 16, #10, p.591.

BOBROW, D.G. & WINOGRAD, T. (1977) An Overview of KRL: A Knowledge Representation Language, Cognitive Science, vol 1, #1, Jan 1977.

CHARNIAK, E. (1975) Organization and Inference in a Frame-Like System of Common Knowledge, Proc. Theoretical Issues in Natural Language Processing, Cambridge, Mass., June 1975, p.46.

CHASE, W. G. & SIMON, H. (1973) Perception in Chess, Cognitive Psychology, #4, pp.55-81.

CHOMSKY, N. (1957) Syntactic Structures, The Hague: Mouton and co.

CLOWES, M.B. (1971) On Seeing Things, Artificial Intelligence, vol 2, #1, pp.79-112.

COLLINS, A. & LOFTUS, E. (1975) A Spreading Activation Theory of Semantic Processing, Psychological Review, vol 82, #5.

COLLINS, A. & QUILLIAN, M.R. (1972) How to Make a Language User, in Organization of Memory, E.Tulving & W.Donaldson (eds.), Academic Press, New York.

DAHL, O. & NYGAARD, K. (1976) SIMULA-An Algol-based Simulation Language, CACM, vol 9, Sept. 1976.

DAVIES, D. J. (1973) Popler-1.5 Reference Manual, TPU Report#1, School of Artificial Intelligence, Univ. Of Edinburgh, Edinburgh, Scotland.

EARLEY, J. (1970) An Efficient Context-Free Parsing Algorithm, CACM, vol 13, #2, Feb 1970, pp.94-102.

FAHLMAN, S.E. (1975) Thesis Progress Report: A System for Representing and Using Real-World Knowledge, AIM-331, A.I. Lab, MIT, Cambridge, Mass.

FEIGENBAUM, E.A. (1963) The Simulation of Verbal Learning Behavior, in Computers and Thought, Feigenbaum, E.A. & Feldman, J.(eds.), McGraw-Hill, New York, p.297.

FILLMORE, C. (1968) The Case for Case, in E.Bach & R.I.Harris (eds.), Universals in Linguistic Theory, Holt, Rhinehart & Winston, New York.

FREUDER, E.C. (1976) A Computer System for Visual Recognition using Active Knowledge, Ph.D. Thesis, AI-TR-345, MIT AI Laboratory, Cambridge, Mass.

GELERNTER, H. (1963) Realization of a Geometry Theorem-Proving Machine, in E.A.Feigenbaum & J,Feldman (eds.), Computers and Thought, McGraw-Hill, New York.

GREEN, C. (1969) Application of Theorem-Proving to Problem Solving, Proc. IJCAI1, Washington, D.C., May 1969.

GUZMAN, A. (1968) Computer Recognition of Three-Dimensional Objects in a Visual Scene, MAC-TR-59, Project MAC, MIT, Cambridge, Mass.

HAVENS, W. S. (1976) Can Frames Solve the Chicken and Egg Problem?, Proc. First CSCSI/SCEIO Nat. Conf., UBC, Vancouver, Canada, August 1976.

HAYES, P. J. (1973) Computation and Deduction, Proc. 1973 MFCS Conf., Czechoslovakian Academy of Sciences.

HENDRIX, G. (1975) Expanding the Utility of Semantic Networks through Partitioning, Proc. IJCAI4, Tbilisi, Georgia, USSR, Sept. 1975, pp.115-121.

HEWITT, C. (1972) Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, Ph.D. Thesis, A.I. Lab, MIT, Cambridge, Mass.

HEWITT, C., BISHOP, P., & STEIGER, R. (1973) A Universal Modular ACTOR Formalism for Artificial Intelligence, IJCAI3, Stanford Univ., Stanford, Calif., August 1973.

HUFFMAN, D.A. (1971) Impossible Objects as Nonsense Sentences, in _Machine Intelligence 6_, B.Meltzer & D.Michie (eds.), Edinburgh Univ. Press, Edinburgh, Scotland.

JAYNES, J. (1976) _The Origin of Consciousness in the Breakdown of the Bicameral Mind_, Houghton-Mifflin, Boston.

KAPLAN, R. (1973) A General Syntactic Processor, in R.Rustin (ed.), _Natural Language Processing_, Algorithmic Press, New York.

KOWALSKI, R. A. (1974) Predicate Logic as a Programming Language, Proc. IFIP74, North-Holland, pp.569-574.

KUIPERS, B.J. (1975) A Frame for Frames: Representing Knowledge for Recognition, in _Representation and Understanding_, D.G.Bobrow & A.Collins (eds.), Academic Press, New York.

MACKWORTH, A.K. (1975) Consistency in Networks of Relations, TR-75-3, Comp. Science Dept., Univ.of British Columbia, Vancouver, Canada, also _Artificial Intelligence_, vol 8, #1, pp.99-118.

MACKWORTH, A.K. (1976) Model Driven Interpretation in Intelligent Vision Systems, _Perception_, vol 5, pp.349-370.

MACKWORTH, A.K. (1977a) On Reading Sketch Maps, TR-77-2, Dept.of Computer Science, Univ.of British Columbia, Vancouver, Canada, also Proc. IJCAI-77, MIT, Cambridge, Mass., August 1977, p.598.

MACKWORTH, A.K. (1977b) How to See a Simple World, in _Machine Intelligence 8_, E.W.Elcock & D.Michie (eds.), Halsted Press, New York.

MACKWORTH, A.K. (1977c) Vision Research Strategy: Black Magic, Metaphors, Mechanisms, Miniworlds, and Maps, Proc. Workshop on Comp. Vision Systems, June 1977, U. Mass, Amherst, Mass.

McCALLA, G. (1977) An Approach to the Organization of Knowledge for the Modelling of Conversation, Ph.D. Thesis, Comp. Science Dept., Univ.of British Columbia, Vancouver, Canada.

McCARTHY, J. & HAYES, P. (1969) Some Philosophical Problems from the Standpoint of Artificial Intelligence, in Machine Intelligence 4, B.Meltzer & D.Michie (eds.), Edinburgh University Press, Edinburgh, Scotland.

McDERMOTT, D.V. & SUSSMAN, G. (1973) Son of CONNIVER: The CONNIVER Reference Manual, MIT AI Lab, Cambridge, Mass.

McDERMOTT, D. V. (1975) Very Large Planner-Like Databases, MIT AI Lab, Memo 339, Cambridge, Mass., Sept. 1975.

MINKER, J. & VANDENBRUG, G.J. (1973) The Earley Algorithm as a Problem Representation, Tech. Report TR-247, Comp. Science Center, Univ.of Maryland, College Park, Maryland.

MINSKY, M. (1975) A Framework for Representing Knowledge, in The Psychology of Computer Vision, P.Winston (ed.), McGraw-hill, New York.

NEISSER, U. (1976) Cognition and Reality, W. H. Freeman & co., San Francisco.

NEWELL, A. & SIMON, H. (1963) GPS: A Program that Simulates Human Thought, in Computers and Thought, E.A.Feigenbaum & J.Feldman (eds.), McGraw-Hill, New York, p.279.

NEWELL, A. & SIMON, H. (1972) Human Problem Solving, Prentice-Hall, Englewood-Cliffs, New Jersey.

NILSSON, N. (1971) Problem Solving Methods in Artificial Intelligence, McGraw-Hill, New York.

NORMAN, D.A., RUMELHART, D.E., et.al. (1975) Explorations in Cognition, W.H.Freeman & Co., San Francisco.

PYLYSHYN, Z. W. (1973) What the Mind's Eye Tells the Mind's Brain: a Critique of Mental Imagery, Psycological Bulletin, 1973, #80, pp.1-24.

PYLYSHYN, Z. W. (1976) Imagery and Artificial Intelligence, in W.Savage (ed.), Minnesota Studies in the Philosophy of Science, vol IX, University of Minnesota Press, Minneapolis.

QUILLIAN, M.R. (1968) Semantic Memory, in Semantic Information Processing, M.Minsky (ed.), MIT Press, Cambridge, Mass., p.227.

RAPOPORT, A. (1963) Technological Models of the Nervous System, in K.M.Sayre & F.J.Crosson (eds.), The Modelling of Mind, Simon & Schuster, New York, p.25.

REBOH, R. & SACERDOTI, E. (1973) A Preliminary QLISP Manual, Stanford Research Institute AI Lab, Tech Note #81, August 1973.

REITER, R. (1973) Semantically Guided Deductive System for Automatic Theorem Proving, Proc. IJCAI3, Stanford Univ., Stanford, Calif., Aug. 1973, p.41.

RIEGER, C. (1974) Conceptual Memory: A Theory and Computer Program for Processing the Meaning Content of Natural Language Utterances, Ph.D. Thesis, AIM-233, Stanford Univ., Stanford, Calf.

ROBERTS, L.G. (1965) Machine Perception of Three-Dimensional Objects, in _Optical and Electro-Optical Information Processing_, J.T.Tippet et.al. (eds.), MIT Press, Cambridge, Mass., pp.159-197.

ROBINSON, J.A. (1965) A Machine-Oriented Logic Based on the Resolution Principle, _JACM_, vol 12, #1, pp.23-41.

RUMELHART, D. & NORMAN, D. (1973) Active Semantic Networks as a Model of Human Memory, Proc. IJCAI3, Stanford Univ., Stanford, Calif., Aug. 1973, p.450.

RUMELHART, D.E. & ORTONY, A. (1976) The Representation of Knowledge in Memory, Tech. Report #55, Center for Human Info. Processing, Dept.of Psychology, Univ.of Calif. At San Diego, La Jolla, Calif.

SAMUEL, A. L. (1963) Some Studies in Machine Learning Using the Game of Checkers, in E.A.Feigenbaum & J,Feldman (eds.), _Computers and Thought_, McGraw-Hill, New York.

SCHANK, R. (1975) Using Knowledge to Understand, Proc. Theoretical Issues in Natural Language Processing, MIT, Cambridge, Mass., June 1975, p.131.

SCHANK, R. & ABELSON, R. (1975) Scripts, Plans and Knowledge, Proc. IJCAI4, Tbilisi, Georgia, USSR, Sept. 1975, pp.151-157.

SCHUBERT, L. (1975) Extending the Expressive Power of Semantic Networks, Proc. IJCAI4, Tbilisi, Georgia, USSR, Sept. 1975, p.158.

SLAGLE, J. R. (1971) _Artificial Intelligence: The Heuristic Programming Approach_, Mc-Graw-Hill, New York.

STALLMAN, R. M. & SUSSMAN, G. J. (1977) Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, Artificial Intelligence, vol 9, #2, Oct. 1977, p.135.

SUSSMAN, G. & McDERMOTT, D. (1972) Why Conniving is Better than Planning, AIM-255A, A.I. Lab, MIT, Cambridge, Mass.

SUSSMAN, G.J., WINOGRAD, T., & CHARNIAK, E. (1973) MICRO-PLANNER Reference Manual, AI Lab memo #203A, MIT, Cambridge, Mass.

TEITELMAN, W. (1974) INTERLISP Reference Manual, Xerox Palo Alto Research Center, Palo Alto, Calf.

Van EMDEN, M. H. (1977) Programming with Resolution Logic, in Machine Intelligence 8, E.W.Elcock & D.Michie (eds.), Halsted Press, New York, pp.266-299.

WALTZ, D. L. (1972) Generating Semantic Descriptions from Drawings of Scenes with Shadows, Ph.D. Thesis, AI-TR-271, MIT, Cambridge, Mass.

WINOGRAD, T. (1973a) Understanding Natural Language, Academic Press, New York.

WINOGRAD, T. (1973b) Breaking the Complexity Barrier (Again), Proc. ACM SIGIR-SIGPLAN Interface Meeting, Nov. 1973.

WINOGRAD, T. (1975) Frame Representations and the Procedural-Declarative Contraversy, in Representation and Understanding, D.G.Bobrow & A.Collins (eds.), Academic Press, New York, pp.185-210.

WINSTON, P. H. (1975) Learning Structural Descriptions from Examples, in The Psychology of Computer Vision, P.H.Winston (ed.), McGraw-Hill, New York.

WINSTON, P. H. (1977) Artificial Intelligence, Addison-Wesley, Reading, Mass.

WOODS, W.A. (1970) Transition Network Grammars for Natural Language Analysis, CACM, vol 13, #10, pp.591-606.

WOODS, W.A. (1974) Recursive Transition Networks and the Earley Recognition Algorithm, unpublished working paper, Bolt, Beranek, & Newman, Cambridge, Mass.

WOODS, W.A. (1975) Whats in a Link, in Representation and Understanding, D.G.Bobrow & A.Collins (eds.), Academic Press, New York, pp.35-82.

YUILLE, J. C. (1977) The Role of Imagery in Models of Cognition, _Journal of Mental Imagery_, 1977, #1.

APPENDIX A

EARLEY'S PARSING ALGORITHM

The following presentation of Earley's algorithm is intended to supplement the informal discussion of Chapter 3. A still more formal treatment is provided by Aho and Ullman (1972).

We are given a context-free grammar, G=(P,N,K,S), where P is a set of production rules, N is the set of non-terminal symbols, K is the set of terminal symbols, and S is the start symbol which is a distinguished symbol in N. The algorithm operates on an input sentence, w=a(1)a(2) . . . a(n) and determines whether w is contained in L(G), the language generated by G. Upper case letters are used to represent non-terminal symbols, the lower case letters, "i", "j", "k", and "n" represent indices, and a(i) is used to represent the i'th symbol in the input sentence. Other lower case letters represent sentential forms composed of both terminal and non-terminal symbols.

For $0 \leq j \leq n$, the algorithm constructs <u>parse lists</u> of items. An item, [A->m.q,i] $0 \leq i \leq j$, is an element of the parse list I(j) if and only if a sentential form, rAu, with r=a(1)a(2) . . . a(i) can be derived from S and a(i+1) . . . a(j) can be derived from m. That is, i through j bracket the portion of w derivable from m, and the production rule, A->mq, can be used in the generation of w up to position j. All items in a parse list, I(j), represent derivations which agree with w up to position j. The items in a given parse list, I(j), can be viewed as independent parsers, each attempting to recognize an instance of its own production rule from the input sentence.

The algorithm is initialized by forming a parse list I(0) containing the single seed item [S->.w,0]. As each new input symbol a(j+1) of w is read, the algorithm generates a new parse list I(j+1) from I(0) . . I(j). This process continues until the last symbol a(n) in w is read and I(n) generated, or until no new parse list can be generated, indicating w is not contained in the language of G. If, at the end of this process, an item, [S->m.,0], is contained in I(n), then w is in L(G).

The algorithm proceeds by the cyclic application of three functions, called the <u>predictor</u>, the <u>scanner</u>, and the <u>completer</u>, until the last symbol is read from w or until some I(j) is found to be empty. First, the predictor computes from the rules of G and the derivation found to date what derivations may possibly follow. It spawns new parsers to look for these derivations. If [A->m.Bq,i] is an element of parse list I(j), and B->r is a rule in P, then [B->.r,j] is added to I(j). The index, j, in the item indicates at what position, j, in the parse a new parser was created to look for the right-hand-side of the rule, B->r. See Figure A.1.

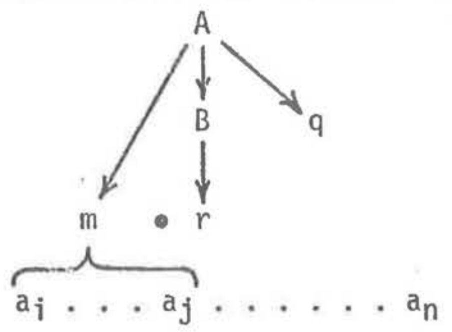


Figure A.1: The Predictor Function

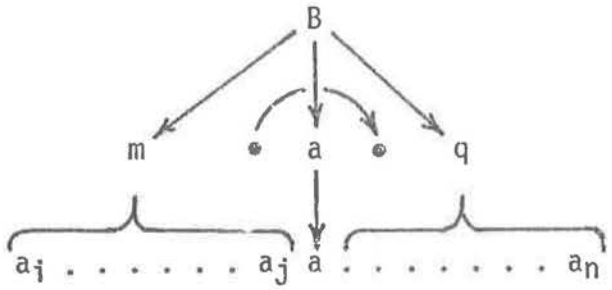


Figure A.2: The Scanner Function

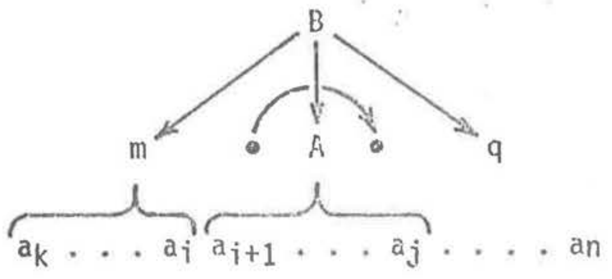


Figure A.3: The Completer Function

Next, the scanner function, by reading the next symbol, a(j+1), from w generates a seed item for the next parse list, I(j+1). For each [B->m.aq,i] that is contained in I(j) and w=a(1) . . a(j)a . . a(n) then [B->ma.q,i] is added to I(j+1). The scanner propagates all parsers in I(j) to the next parse list that were expecting a(j+1) to appear next in the input sentence. As is illustrated in Figure A.2, the scanner increments the internal state of an item by moving the parsing dot one terminal symbol to the right.

The completer function performs bottom-up reductions of sentential forms that appear as right-hand-sides of production rules in P to their non-terminal left-hand-sides. If [A->r.,i] is an element of I(j), then the non-terminal symbol, A, has been recognized in w. From I(i), the generating item, [B->m.Aq,k] is found and [B->mA.q,k] is added to I(j+1). The completer acts as a scanner for non-terminal symbols, as is shown in Figure A.3.

APPENDIX B

MAYA LANGUAGE REFERENCE MANUAL

This manual is a description of an Artificial Intelligence programming language called MAYA. Included in this language are facilities for performing pattern matching, primitives for constructing semantic networks and schemata, primitives for creating and manipulating processes, and control structures for integrating top-down and bottom-up search techniques. The language is designed as a dialect of LISP having a number of extensions and a few restrictions. The extensions will be described in detail in the following pages and the restrictions will be noted as well.

B.1 Relating to MAYA

This section explains the operation of the interpreter. Since this language is experimental, it is subject to occasional changes in behaviour. Most changes will be upwardly compatible and this document will be promptly edited to reflect those changes.

To run MAYA under MTS:

```
$RUN CS:LISP SCARDS=CS:LOADER+*SOURCE*
(RESTORE MAYA:SYSGEN)
```

MAYA will be started in a top-level EAR and the creation date for the current version of the interpreter will be printed.

The following list of functions are the basic mechanisms for controlling the interpreter. The form of each function call is given followed by the type of the function. EXPR, NEXPR, and FEXPR type functions can be used from both MAYA and LISP. Square brackets indicate optional arguments, and the asterisk is used as the Kleene star indicating zero or more repetitions.

1. (MAYA) {EXPR}

   Executed from LISP. Initializes the processor and enters a a top-level EAR.

2. (HALT [<form>]) {@FSUBR}

   The value of <form> is returned to LISP. If no argument is supplied, NIL is returned. HALT leaves the current invocation of the processor intact.

3. (RESTART <form>) {NEXPR}

   Executed from LISP. Restarts the interpreter from the previous HALT. Computation proceeds with <form> substituted for the call to HALT. RESTART and HALT can be used as a co-routine mechanism between MAYA and LISP. HALT returns a value to LISP at the point the interpreter was called, leaving the process intact. RESTART returns a value to MAYA at the point that the process was previously halted.

4. (@INIT) {EXPR}

   Reinitializes the processor. Meaningful only from LISP. If evaluated from MAYA, a warning is given and NIL is returned.

5. (EAR) {@SUBR}

   Enters a READ-EVAL-PRINT loop. Since MAYA permits multiple co-existant processes, this function permits the user to create multiple READ-EVAL-PRINT loops. Reads forms from each <file>. Each form is MAYA EVALed.

6. (INFILE <file>*) {@SUBR}

   This function is analogous to DISKIN in LISP. There is however no DISKOUT analogue because MAYA function definitions may be local to objects in the data base.

Since MAYA makes use of both read and print macro characters, it is unwise for the user to prefix his own atoms with these characters. The characters reserved by MAYA are:

"?", "!", "¢", "$", and "@".

The "@" is not actually a macro character but all MAYA internal names begin with this character. If it is inconvenient to abide by these conventions, then the following two functions may be used.

7. (@OFF-STATUS) {EXPR}

    Turns off read macro processing.

8. (@ON-STATUS) {EXPR}

    Turns on read macro processing.

To facilitate communication between LISP and MAYA, two prefixes are provided. From LISP, ¢<form> will return the MAYA value of <form>. From MAYA, $<form> will return the LISP value of <form>. For example, from LISP, ¢$¢$¢$¢$¢$(FOO) will return the LISP value of (FOO). Note that (MAYA) is (@INIT) followed by ¢(EAR).

B.2 The Database

Many of the primitive functions defined in MAYA are concerned with creating objects, forming semantic networks from objects, searching arc paths through these networks, and interpreting objects as schemata. These functions create and manipulate objects and the schemata and nodes that can be created from objects.

9. (OBJECT <type><pair>*) {@FSUBR & FEXPR}

    Each <pair> is of the form, <name><form>. OBJECT creates a new object of the user type, <type>, having for each <pair> a binding defined between the name, <name>, and its definition, <form>. OBJECT evaluates its odd arguments, and its even arguments must be <ATOM>s. Note that both a MAYA and LISP version of this function are included in the system.
    Example:

```
_ (OBJECT 'NODE CLASS 'TABLE HAS-AS-PARTS (LIST 'LEGS 'TOP))
!   (@OBJECT@ NODE CLASS TABLE HAS-AS-PARTS (LEGS TOP))
```

10. (NEW <object> <pair>*) {@FSUBR & FEXPR}

    If <object> is not an object, an error occurs. NEW creates a new instance of <object> of type, @INSTANCE@.

The new object consists of the bindings of <pair>s concatenated with <object>. The new instance is of the form:

```
        (@OBJECT@ @INSTANCE@ <pair>*) || <object>,
```

where || indicates list concatenation. Note that both a MAYA and LISP version of this function are included.
Example:

```
_  (PUT* 'DOG (OBJECT 'GENERIC NAME 'DOG CLASS 'MAMMALIA))
!    (@OBJECT@ GENERIC NAME DOG CLASS MAMMALIA)
_  (NEW DOG NAME 'FLOYD OWNER 'BILL)
!    (@OBJECT@ @INSTANCE@ NAME FLOYD OWNER BILL @OBJECT@
!         GENERIC NAME DOG CLASS MAMMALIA)
```

11. (OBJECTP <thing>) {EXPR}

If <thing> is an object, its type is returned. Else NIL is returned.

12. (STEREO <object>) {@SUBR}

If <object> is an instance, its stereotype object is returned. If <object> is object but not an instance of some other object, <object> itself is returned. Else NIL is returned.

13. (GET* <name> [<else>]) {@SUBR & EXPR}

Fetches the definition of <name> from within the enclosing object. Both a MAYA and a LISP version are provided. If there is no <name> defined in the enclosing object, <else> is evaluated. If <else> is not given, NIL is returned. If there is no enclosing object on the stack, the global object is taken to be the enclosing object.
Examples:

```
_  (GET* 'CONS)
!    (@OBJECT@ @PLIST@ SUBR *)
_  (GET* 'DFDFDGGGG ''ELSE)
!    NIL
_  ;NOTE: GLOBAL OBJECT HAS EVERY NAME DEFINED.
_  (SEND (OBJECT 'TYPE A 'ADEF)
_         (GET* 'A))
!    ADEF
_  (SEND (OBJECT 'TYPE A 'ADEF)
_         (GET* 'B ''ELSE))
!    ELSE
```

14. (PUT* <name><form>)   {EXPR}

Puts within the enclosing object the binding of <name> to its new definition, <form>. <Name> must be atomic. If a previous definition existed, it is replaced. If no enclosing object exists on the stack, the binding is added to the global object, i.e., the CDR of <name> is set to <form>. If the enclosing object is an instance of some parent sterotype object, the stereotype object is guaranteed to remain unaltered. PUT* returns <form>.
Examples:

```
_  (PUT* 'A 'ADEF)
!    ADEF
_  A
!    ADEF
_  (CDR 'A)
!    ADEF
_  (: (PUT* 'B (OBJECT 'NODE))
_      (PUT* 'C 'CDEF)
_      (SELF))
!    (@OBJECT@ NODE C 'CDEF)
_  B
!    (@OBJECT@ NODE C 'CDEF)
```

15. (REM* <name>)   {@SUBR}

Removes the definition of <name> from the enclosing object. If there is no enclosing object on the stack, the CDR of the atom, <name>, is set to NIL. If the enclosing object is an instance, its stereotype object remains unaltered. REM returns the binding of <name>. If the binding of <name> is not defined in this object or within this instance, NIL is returned.
Examples:

```
_  B
!    (@OBJECT@ NODE C 'CDEF)
_  (SEND B (REM* 'C))
!    NIL
_  B
!    (@OBJECT@ NODE)
_  (REM* 'B)
!    NIL
_  B
!    NIL
```

16. (SELF) {@SUBR}

Returns the current enclosing object from the stack. If there is no enclosing object, an error occurs.
Examples:

```
_ (: (OBJECT 'TYPE A 'ADEF)
_     (PUT* 'A 'NEW-ADEF)
_     (SELF))
!   (@OBJECT@ TYPE A NEW-ADEF)
_ (SELF) = ERROR: NO ENCLOSING OBJECT ON THE STACK
```

17. (ITEM <pair>*) {@SUBR}

Each pair is of the form, <name><value>. ITEM creates a new item containing a new instances of each variable, <name>, having value, <value>. ITEM evaluates its even arguments, and its even arguments must be atoms. The new item is returned as value.
Example:

```
_ (ITEM X 'XVAL Y (LIST 'YYY))
!   (@ITEM@ NIL X XVAL Y (YYY))
_ (SETQ X 'XVAL)
!   XVAL
_ (: (ITEM X 'ZZZ Y ?X)
_     (PRINT ?Y)
_     ?X)
!   XVAL
!   ZZZ
```

18. (SET <pair>*) {@SUBR}

Each <pair> is of the form, <name><value>. SET binds each variable name, <name>, to its new value, <value>. SET evaluates all its arguments and its odd arguments must evaluate to atoms. SET returns the value of the last <pair>.

19. (SETQ <pair>*) {@FSUBR}

Each <pair> is of the form, <name><value>. SETQ behaves as SET except it evaluates only its even arguments. Both SET and SETQ search the stack looking for the first occurrence of the variable, <name>. If no variable exists on the stack having name, <name>, the LISP value of the atom, <name>, is changed to <value>. Note that only the variable, <name>, is changed and not the definition of the atom, <name>. SET and SETQ bind values whereas PUT* binds definitions.

20. (DEFUN <defn>) {FEXPR}

DEFUN has been extended to permit the definition of @EXPR, @NEXPR, @FEXPR, and QEXPR type functions. DEFUN may be used from either MAYA or LISP to define any of the function types from both languages. DEFUN always adds its definition, <defn>, to the enclosing object and may therefore be used to provide function definitions local to specfic objects. DEFUN honors instance boundaries.
Examples:

```
_ (DEFUN TEST NIL (PRINT 'OUTSIDE) T)
!   TEST
_ (SEND (OBJECT 'CONTEXT)
_       (DEFUN TEST NIL (PRINT 'INSIDE) NIL)
_       (TEST))
!   INSIDE
!   NIL
_ (TEST)
!   OUTSIDE
!   T
```

21. (@ P1 P2 . . . Pn) {@FSUBR & FEXPR}

The tuple evaluator. The value of a tuple is a new tuple of the values of its elements, P1, P2, . . . Pn. The tuple evaluator uses <u>inverse quote mode</u> during evaluation. Atoms and matcher variables are treated as constants, i.e., they evaluate to themselves. All other forms are EVALed, note that (@ P1 P2 . . . Pn) can be abbreviated using angle brackets as <P1 P2 . . . Pn>. Both a MAYA and a LISP version are included. Note also that angle brackets cannot be used as LISP super-parentheses.
Examples:

```
_ <A B C>
!   <A B C>
_ <'A (LIST 'B 'C)>
!   <A (B C)>
_ (SETQ Y 'YVAL)
!   YVAL
_ <A <B !:X> ?Y>
```

```
    !   <A <B !:X> YVAL>
```

22. (TUPLEP <thing>) {EXPR}

Returns T if <thing> is a tuple, else returns NIL.

23. (VARP <thing>) {EXPR}

Returns the name of <thing> if it is a MAYA variable, else NIL. Pattern variables return NIL.

24. (ITEMP <thing>) {EXPR}

Returns T if <thing> is an item, else NIL.

B.3 Evaluation

The following paragraphs describe the function types included in MAYA. Valid LISP functions are acceptable and are executed directly by LISP for MAYA. In addition to LISP's function types, the interpreter also recognizes the following types:

@SUBR, @FSUBR, @EXPR, @NEXPR, @FEXPR, and QEXPR.

If desired, the interpreter may be extended to include other function types as well. Please see me for details.

@SUBR's and @FSUBR's are the system supplied functions of MAYA. @SUBRs evaluate their arguments but @FSUBRs bind the list of unevaluated arguments to their single parameter, as expected. @EXPR, @NEXPR, and @FEXPR are MAYA's user defined function types that are analogous to their LISP counterparts. QEXPR type functions are somewhat analogous to the Q-type functions of QLISP (Reboh, 1973). QEXPRs are implemented via QLAMBDA expressions and take a single tuple as argument. They return as value an item representing the result of a pattern match between the tuple argument and the tuple pattern of the QLAMBDA expression. Please see the sections on pattern matching, Generators, and Recognizers.

The interpreter uses the following algorithm when applying a function to its arguments:

- If the function is an @SUBR or an @EXPR, its arguments are evaluated and the function is applied to their values.

• If the function is an @FSUBR or an @FEXPR, then the function is applied to the list of unevaluated arguments.

• If the function is an @NEXPR, it is applied to the unevaluated arguments.

• If the function is a QEXPR, the tuple argument is evaluated and the pattern matcher called on the value. If the match succeeds, the function is applied to the result.

• Else the function is a LISP form. If it is a SUBR or EXPR, the arguments are evaluated and the function is LISP APPLYed to the result. If the function is an NEXPR, FEXPR, NSUBR, or FSUBR, then the form is LISP EVALed.

Functions which make no use of the parallelism, control structures, or data structures of MAYA can be written in LISP and executed from MAYA thereby gaining a considerable increase in efficiency. While in LISP, any forms defined within MAYA can be evaluated via the "¢" prefix and references to MAYA variables can be fetched via the "?" prefix.

25. (AVAL <form>) {@SUBR}

MAYA evaluates its argument, <form>. If <form> is a list whose CAR is an atom, AVAL fetches the MAYA function from the first definition of the atom on the stack. If there is no function definition within the first occurrence of the atom on the stack, an error occurs.

26. (EVAL <form>) {@SUBR}

MAYA evaluates its argument, <form>. If <form> is a list whose CAR is an atom, EVAL fetches the function definition from within the global object. That is, EVAL fetches function definition from the PLIST of the atom. EVAL is therefore faster than AVAL, but does not recognize function definitions local to enclosing objects. MAYA functions use EVAL for evaluation unless otherwise noted.

27. (SEND <A1><A2> . . . <An>) {@FSUBR}

SEND AVALuates each element in the sequence, <A1><A2> . . <An>, in a left-to-right order. If the value returned from <Ai> is an object or an item, then it is

pushed onto the stack. Next, <Ai+1> is evaluated in this new environment. The final value returned from SEND is the value of <An>. SEND may be abbreviated by a single colon, ":".
Examples:

```
_ (SEND PRINT)
!   (@OBJECT@ @PLIST@ SUBR *)
_ (: PRINT SUBR)
!   *
_ (SETQ NET (OBJECT 'NODE))
_ (: ?NET
_     (PUT* 'A (OBJECT 'NODE))
_     (PUT* 'B (OBJECT 'NODE))
_     (PUT* 'C 'CDEF))
!   CDEF
_ (SEND ?NET (PRINT A) (PRINT B) (PRINT C))
!   (@OBJECT@ NODE B (@OBJECT@ NODE C CDEF))
!   (@OBJECT@ NODE C CDEF)
!   CDEF
!   CDEF
```

B.4 Error Conditions and the DEBUG System

There are two classes of errors that can occur during the execution of MAYA. They are errors trapped by MAYA and errors trapped by LISP. Errors trapped by MAYA cause an "appropriate" error message to be printed and MAYA's DEBUG System to be called on the form causing the error. This form is called the Breakform. Note that there is one exception. If the error detected by MAYA is an undefined name or an undefined variable occuring at the top-level of an EAR, then the error message is printed but an immediate return is made to the EAR. In this case, there is no form to BREAK on.

The second class of errors cause a LISP BREAK to be entered for one of the following reasons:

- A LISP error occurred in a user's LISP function.
- An unexpected condition occurred within the interpreter causing it to abnormally terminate.
- An error occurred within a MAYA EXPR, NEXPR, or FEXPR type function.

If a LISP BREAK is entered for one of the above reasons, and the user desires to be instead in a MAYA BREAK, a transfer may be made using the function, OOPS, described below.

The DEBUG System provided in MAYA is modelled after the LISP/MTS DEBUG Package. Most of the facilities included in the LISP system are provided in MAYA and a familiarity with the LISP DEBUG Package is assumed here. MAYA's DEBUG System can be called in a number of ways. It can be called explicitly on a form via the functions, DEBUG and BREAK. The system may be called implicitly on the body of a function via setting a Breakpoint in that function. And lastly, the DEBUG System is called by the interpreter whenever a MAYA error occurs.

The LISP BREAK functions, BREAKF, UNBREAKF, and UNSET, have been modified to work with @EXPR, and QEXPR type functions as well as EXPRs. As in LISP/MTS, Breakpoints can set on an entire function, or a form within a function. Both types may have an optional predicate which determines whether the break is acknowledged when encountered. However, care must be taken with the LISP global atoms, "?BREAKSW", "?DEBUG", and "?BROKEN". If they are to be altered or examined, they must be enclosed in double quotes to prevent them from being treated as variables by MAYA or alternatively, read macro processing can be disabled via @OFF-STATUS.

Care must also be taken with local function definitions. To set a Breakpoint on FOO within some object, FR1:

```
                (: FR1 (BREAKF FOO)).
```

To negate the Breakpoint, you must again "GOTO" that object:

```
                (: FR1 (UNBREAKF FOO)) or

                (: FR1 (UNSET)).
```

28. (DEBUG <form>) {@FSUBR}

Calls the DEBUG System explicitly on <form>.

29. (BREAK <message>*) {@SUBR}

BREAK is provided as a user errror handler. It prints each <message> on @ERROUT then calls the DEBUG System on the current form being evaluated, i.e., in most cases BREAK itself.

30. (OOPS) {EXPR}

A very useful function! OOPS allows the user to recover from a catastrophic error that has caused a LISP break to be entered. OOPS will return control to MAYA's Break package with the form that was being evaluated in MAYA at the time of the error as the Breakform.

OOPS may also be used to Break on MAYA @SUBRs and @FSUBRs by setting a LISP Breakpoint on the internal MAYA routine. When the LISP Break is acknowledged, typing (OOPS) will transfer control back to MAYA's Debug System with the MAYA @SUBR or @FSUBR as the Breakform.

31. (@BK) {EXPR}

Prints a possibly long backtrace of the stack.

32. (@TRACE) {EXPR}

Begins the printing of a trace of the evaluation of each non-atomic form. The form is printed before its body is entered.

33. (@UNTRACE) {EXPR}

Turns off tracing.

A summary of the commands recognized by the DEBUG System follows:

BK [n] Abbreviation: none
Prints a backtrace of the stack starting at the stack-pointer for length "n" which defaults to 10.

BKO [n] Abbreviation: none
Prints a backtrace of all objects and items on the stack starting at the stackpointer for a length of "n" which defaults to 10.

BKE [n] Abbreviation: none
Prints a backtrace of all MAYA forms on the stack starting from stackpointer. "n" defaults to 10.

* Abbreviation: none
Prints the Breakform. Note that in MAYA, the Breakform is never on top of the stack at the time the BREAK is entered.

PP Abbreviation: none
Pretty-prints the form where the stack-pointer points.

PP* Abbreviation: none
Pretty-prints the Breakform.

PRINT Abbreviation: P

Prints the form where the stackpointer points.

TOP Abbreviation: none

Resets the stackpointer to the top of the stack. Note that since Breakform has not been pushed onto the stack at the time the BREAK occurs, the top of the stack and Breakform do not coincide. The top of the stack is always the form that called Breakform.

FIND <loc>* Abbreviation: F

Finds a form on the stack and sets the stack-pointer to that point. Each <loc> is either a number of forms to move the stackpointer down the stack or the name of a function on the stack.

GO [<loc>*] Abbreviation: G

Finds a form on the stack as in the FIND command, then restarts computation BREAKing on that form. <Loc> may be ommitted.

RETURN <form> Abbreviation: RET

Evaluates <form> and returns it as the value of the Breakform.

RESTART [<form>] Abbreviation: RES

Restarts computation from where the stackpointer points using <form>. If <form> is not coded, computation is restarted using the previous form on the stack.

CONTINUE Abbreviation: C CO

Continues with Breakform. If Breakform has been previously evaluated via the EVAL command, it will not be re-evaluated.

STEP [n] Abbreviation: S

Steps through the evaluation of the next "n" forms and generates a new BREAK. "n" defaults to 1.

NEXT Abbreviation: N NX

Evaluates Breakform and breaks on the next form. If Breakform has been previously evaluated via the EVAL command, it will not be re-evaluated.

EVAL Abbreviation: E

Evaluates Breakform and prints its value. Note that Breakform is AVALUATEd.

UP Abbreviation: |

Causes the DEBUG System to ascend one level. If there is no higher BREAK level, control is returned to top-level.

STOP   Abbreviation: NIL ||
        Causes a return to top-level.

EDIT [<loc>*]   Abbreviation: ED
        Calls EDITE on the form where the stackpointer points. If <loc>* is coded, an implicit FIND command is executed.

Any form typed at DEBUG other than the above commands or there abbreviations will be AVALed and its value printed. Note that the DEBUG System always uses AVAL for all evaluations, i.e., all function definitions are fetched from the first definition found on the stack.

B.5 Input/Output

When MAYA is running, all I/O is performed through buffers separate from the ones used by MTS/LISP. Reading is performed through @LISPIN with the file prefix character "_". Writing is performed through @LISPOUT with the file prefix character "!". All I/O functions behave, except for the choice of I/O buffers, as they do in LISP.

34. (TPRIN1 <form> [<IO-atom>])   {EXPR}

PRIN1's <form> in a terse mode. Printing descends only a set number of levels into the given structure. Any non-atomic structures greater than this level are represented by an "&". If the second argument, <IO-atom>, is given, <form> is written in its buffer. Else <form> is written in @LISPOUT. TPRIN1 returns T.

35. (TPRINT <form> [<IO-atom>])   {EXPR}

Same as TPRIN1 except that the buffer is emptied both before and after <form> is written.

36. (TPRINT-LEVEL <n>)   {NEXPR}

Sets the level to which TPRIN1 and TPRINT will descend. Default is (TPRINT-LEVEL 2).

B.6 Pattern Matching

This section describes the pattern matching functions and pattern element definitions provided in MAYA.

37. (MATCH <pattern><db>[<else>]) {@SUBR}

<Pattern> must be a tuple. <Db> must evaluate to either a tuple or an object representing a tuple database. See below. The <pattern> is matched against the <db>. If a match can be found, MATCH returns an item containing the bindings of any pattern variables contained in the <pattern> plus a reactivation TAG. This TAG is an stack segment which permits the matcher to be recalled again from the point of its last successful match. If <db> is a tuplebase, the value associated with the matching datum is bound to the variable, "*", in the returned item. If <db> is a tuple, "?*" defaults to T. If MATCH fails to find a match between <pattern> and <db>, <else> is evaluated. <Else> defaults to NIL.
Examples:

```
_  (MATCH '<A B C> '<A B C>)
!    (@ITEM@ NIL * T @TAG@ . . . )
_  (MATCH '<A B C> '<A B>)
!    NIL
_  (MATCH '<A B C> '<A B> '(LIST 'A))
!    (A)
```

A tuple data base, called a <u>tuplebase</u>, is the associative data base mechanism in MAYA. Each tuplebase is composed of an object having a type indicator of either "@INDEX@" or some positive integer number. Tuplebases are operated on by the system functions: MATCH, NEXT, PADD, PREMOVE, PDELETE, B1ST, D1ST, SUSPEND, RESUME, COMPLETE, etc. The user may as well operate on tuplebases by treating them as ordinary objects.

When MAYA is loaded three empty tuplebases are created within the global object. By convention, their names and purposes are:

@TUPLE@	for declarative patterns,
@METHOD@	for generator methods, and
@PROC@	for suspended processes.

To create a new, empty tuplebase within some enclosing object under the name, <name>, one may write:

(PUT* <name> (OBJECT 0)).

38. (PADD <datum><db>[<form>]) {@SUBR}

The <datum>, which must be a tuple, is added to the tuple database, <db>. Associated with <datum> in the database is the value of <form>. If <form> is not coded, <datum> is given the value, T. If <datum> is already present in the <db> its value is replaced with <form>. PADD returns the value of <form>. If <db> is not a tuplebase, an error occurs.
Examples:

```
_ (PADD '<A B C> @TUPLE@)
!   T
_ (MATCH '<A B C> @TUPLE@)
!   (@ITEM@ NIL * T @TAG@ . . . )
_ (PADD '<A B C> @TUPLE@ 'FFF)
!   FFF
_ (MATCH '<A B C> @TUPLE@)
!   (@ITEM@ NIL * FFF @TAG@ . . . )
```

39. (PREMOVE <datum><db>) {@SUBR}

Removes <datum> from the tuple database, <db>. <Datum> must be a tuple and <db> must be a tuplebase. PREMOVE returns <datum> if it was present in the tuple database. If it was not present NIL is returned.
Examples:

```
_ (PADD '<S D> @TUPLE@)
!   <S D>
_ (PREMOVE '<S D> @TUPLE@)
!   <S D>
_ (MATCH '<S D> @TUPLE@)
!   NIL
_ (PREMOVE '<S D> @TUPLE@)
!   NIL
```

40. (PDELETE <pattern><db>[<else>]) {@SUBR}

Matches <pattern> against <db>. If the match succeeds, PDELETE deletes the matching datum from the tuplebase, <db>, and returns an item containing the bindings of <pattern>, the variable, "", bound to the value associated with the matched datum, and a reactivation tag for PDELETE. If the match fails, <else> is evaluated which defaults to NIL.

Note that PREMOVE does not call the pattern matcher. The datum in the tuplebase that exactly matches <pattern> syntactically, is removed. PDELETE on the other hand deletes the first datum which matches <pattern> and it returns a tag for subsequent deletions.

41. (PDUMP <db>) {@SUBR}

Prints a dump of the patterns and their associated values from the tuplebase, <db>.
Example:

```
_ (PADD '<A B C> @TUPLE@)
!   T
_ (PADD '<A B> @TUPLE@)
!   T
_ (PADD '<WOOF> @TUPLE@ 'FLOYD)
!   FLOYD
_ (PDUMP @TUPLE@)
!   <A B> = T
!   <A B C> = T
!   <WOOF> = FLOYD
!   NIL
```

Patterns in MAYA are represented as tuples. Each tuple is composed of pattern elements called Patels. There are three types of Patels:

1. Patels that match values,
2. Patels that yield values to be matched,
3. Patels that are recursively sub-patterns, i.e., tuples.

The following paragraphs will explain in some detail the different pattern elements that are defined in MAYA. Because the type-2 Patels are somewhat more straightforward than the others, they will be explained first.

Type-2 Patels are pattern elements that yield values to be matched. Type-2 Patels can match Type-1 Patels. They can also match type-3 Patels and other type-2 Patels on EQUAL. The following forms are included as type-2 Patels:

```
<ATOM>
<LIST>
?<var>
!?<var>
!¬<var>
```

Note that some of the Patels have prefixes. These prefixes determine the manner in which the Patels yield their values. The semantics of Type-2 Patels are presented below:

<ATOM> or <LIST>

Atoms and lists are considered constants in MAYA patterns. Their values are themselves. This format is sometimes called "inverse quote mode". They yield their values immediately.
Examples:

```
_ (MATCH '<A (B C)> '<A (B C)>)
!   (@ITEM@ NIL * T @TAG@ . . . )
_ (MATCH '<A (B C)> '<A <B C>>)
!   NIL
```

?<var>

Yields the current MAYA value of <var>. Matches whatever its value will match as a constant. If <var> is not bound, an error occurs.
Examples:

```
_ (SETQ Y 'B)
!   B
_ (MATCH '<A B> '<A ?Y>)
!   (@ITEM@ NIL * T @TAG@ . . . )
_ (MATCH '<A ?Y> '<A C>)
!   NIL
_ (MATCH '<B ?Y> '<?Y ?Y>)
!   (@ITEM@ NIL * T @TAG@ . . . )
```

!?<var>

Matches the current matcher value of <var> from the same side of the match. If <var> has not been bound on the same side of the match, the current match fails.
Examples:

```
_ (MATCH '<!:X !?X> '<A A>)
!   (@ITEM@ NIL * T X B @TAG@ . . . )
_ (MATCH '<A !?X> '<A A>)
!   NIL
_ (MATCH '<!?X !:X> '<A A>)
```

```
    !   NIL
```

!¬<var>

This patel is valid only on the <db> side of the match or as a Patel in a QLAMBDA expression. It does not yield a value immediately but only at the time the match succeeds. It therefore can be used as a mechanism for returning a result from a method or the datum side of a match. This patel can only match type-1 Patels.
Examples:

```
_  (MATCH '<!:X B> '<!¬Z !:Z>)
!    (@ITEM@ NIL * T X B @TAG@ . . . )
_  (MATCH '<!:X B> '<!¬Z B>)
!    NIL
_  (MATCH '<A B> '<!¬Z !:Z>)
!    NIL
```

Type 1 patels include all of the pattern variables defined in MAYA. They match values yielded by the other two patel types and bind their variable names to those values. If the matcher succeeds, the bindings of all Type 1 patels on the pattern side of the match are composed into the item returned from the match.

MAYA permits <u>pattern actors</u> (Hewitt, 1972) to act as predicates on Type 1 patels. Each pattern variable is of the form:

<var-prefix><atom>   or
<var-prefix>(<atom><pred>*)

where <atom> is the name of the pattern variable, <var-prefix> restricts the types of values that the pattern variable can match, and <pred>* is zero or more forms whose evaluation must be non-NIL for the match to succeed. The predicates are evaluated in a left-to-right order. If one of the predicates fails, the remainder are not evaluated and the matcher attempts a different match. <Atom> is bound to its matching value <u>before</u> the predicates are evaluated and each <pred> may reference <atom> as a free variable, e.g.,

!:(X (PRINT ?X)(FOO ?X ?Y)).

The following pattern variables are currently included in MAYA:

!:<var>   or !:(<var><pred>*)

```
        !*<var>    or !*(<var><pred>*)
```

!:<var>

Matches a single pattern element which must be either a Type-2 patel or a Type-3 patel containing no pattern variables.
Examples:

```
_ (MATCH '<!:X !:Y> '<A <B C>>)
!   (@ITEM@ NIL * T X A Y <B C> @TAG@ . . . )
_ (SETQ Y 'YVAL)
!   YVAL
_ (MATCH '<!:X B> '<<A ?Y> !:Z>)
!   (@ITEM@ NIL * T X '<A YVAL> @TAG@ . . . )
_ (MATCH '<!:X B> '<<A !:Z> B>)
!   NIL
```

!*<var>

Matches a segment of patels of length zero or longer. The segment must contain no Type-1 patels. <Var> is bound to a copy of the segment.
Examples:

```
_ (MATCH '<A !*X> '<A B C>)
!   (@ITEM@ NIL * T X <B C> @TAG@ . . . )
_ (MATCH '<A !*X B> '<A B>)
!   (@ITEM@ NIL * T X <> @TAG@ . . . )
_ (MATCH '<A !*X> '<A <B ?Y> C>)
!   (@ITEM@ NIL * T X <<B YVAL> C> @TAG@ . . . )
_ (MATCH '<A !*X> '<A !:Y B>)
!   NIL
_ (MATCH '<A !*X> '<A !*Y B>)
!   (@ITEM@ NIL * T X <B> @TAG@ . . . )
```

Type-3 patels are tuples, that is, recursive sub-patterns. Each patel in a sub-pattern may be either a Type-1, Type-2, or Type-3 patel.

B.7 Generators

42. (D1ST <pattern><db>[<else>]) {@SUBR}

Generates items in depth-first order, i.e., in a tuplebase of QLAMBDA methods, D1ST will recall the same QLAMBDA generator repeatedly until it fails to return a next item.

D1ST will then attempt to match another QLAMBDA expression in the tuplebase. <Pattern> is matched against <db>. <Pattern> must be a tuple and <db> must be a tuplebase of QLAMBDA expressions or QEXPR names. If the match fails, <else> is evaluated which defaults to NIL. If the match succeeds, the matched QLAMBDA expression is evaluated using the item from the datum side of the pattern match as the actual arguments to the function. If the QLAMBDA successfully terminates, D1ST returns an item from the match plus a reactivation tag for the generator. If however, the QLAMBDA terminates unsuccessfully by executing the FAIL function, D1ST will attempt to find another match in <db>.
Example:

```
_  (DEFUN GENA QEXPR <A !¬X>
_    (SETQ X 'A1)
_    (POST)
_    (SETQ X 'A2))
!    GENA
_  (PADD '<A !¬X> @METHOD@ 'GENA)
!    GENA
_  (PADD '<B !¬Y>
_         @METHOD@
_         '(QLAMBDA <B !¬Y>
_             (PROG (N)
_               (SETQ N 0)
_              LOOP
_               (SETQ N (ADD1 ?N) Y (MKATOM 'B ?N))
_               (AND (EQ ?N 3)(EXIT))
_               (POST)
_               (GO LOOP))))
!    (QLAMBDA '<B !¬Y> . . . )
_  (D1ST '<!:AB !:V> @METHOD@)
!    (@ITEM@ NIL AB A V A1 @TAG@ . . .)
_  (: (ELSE '(RETURN 'DONE ':))
_     (D1ST '<!:AB !:V> @METHOD@ '(FAIL))
_     (PRINT ?V)
_     (FAIL))
!    A1
!    A2
!    B1
!    B2
!    B3
!    DONE
```

43. (B1ST <pattern><db>[<else>]) {@SUBR}

Generates items in breadth-first order. The arguments and behavior of B1ST are identical to D1ST except that B1ST calls only once each generator in the specified tuplebase, <db>, that matches the pattern, <pattern>. Only after it has called all possible matching QLAMBDAs will it recall each suspended generator for the second time, then each for the third time, etc.
Example:

```
_ ;FROM THE TUPLEBASE OF METHODS DEFINED ABOVE
_ (B1ST '<!:AB !:V> @METHOD@)
!   (@ITEM@ NIL AB A V A1 @TAG@ . . .)
_ (: (ELSE '(RETURN 'DONE ':))
_    (B1ST '<!:AB !:V> @METHOD@ '(FAIL))
_    (PRINT ?V)
_    (FAIL))
!   A1
!   B1
!   A2
!   B2
!   B3
!   DONE
```

44. (EXIT) {@SUBR}

Terminates immediately the current enclosing generator or process. The exited generator or process returns an item from the previous invoking pattern match. If it is a process associated with a schema, then the highest process on the stack associated with that schema is exited.

45. (POST) {@SUBR}

Returns an item from a generator. POST is analogous to the AU-REVOIR function of CONNIVER. POST causes the first enclosing generator to return an item representing the match that invoked the generator plus a reactivation tag. When the generator is resumed, POST returns NIL.
Example: See D1ST.

The last generator type is used for re-invoking a generator from a reactivation tag and for implementing a restricted automatic backtracking capability.

46. (NEXT <item>[<else>]) {@SUBR}

<Item> must be an item containing a reactivation tag. NEXT uses the tag in the item to resume the generator in order to seek a new match. If the generator succeeds again, NEXT returns the new item containing the new bindings of the pattern variables from the pattern side plus a new reactivation TAG. If the generator fails, NEXT returns the evaluation of <else> which defaults to NIL.

47. (FAIL) {@SUBR}

Provides a simple mechanism for realizing backtrack search. FAIL locates the first item on the stack containing a reactivation tag. The stack is truncated just past the item and the function which created the item is resumed. If the resumed function returns a new item, computation proceeds forward again. If however, the function returns failure, FAIL evaluates itself recursively, i.e., it continues failing up the stack. If FAIL encounters a generator or process marker on the stack, the generator or process immediately returns NIL.

48. (ELSE <form>) {@SUBR}

Creates an explicit failpoint. ELSE returns an item containing a single binding, a reactivation tag bound to <form>. If NEXT is applied to the item or if the item is encountered on the stack by FAIL, <form> is evaluated.

B.8 Processes and Recognizers

49. (PROCESS <schema><ql><pattern>[<else>]) {@SUBR}

Creates a new process and begins its execution. <Ql> must be either a QLAMBDA expression or the name of QEXPR function. The pattern of QLAMBDA expression is matched against <pattern> which must be a tuple. If the match succeeds, a new process is created and associated with <schema> which must be either an object or NIL. If NIL is specified, the process is not associated with any schema. Control is passed to the body of the QLAMBDA expression which is used as the body of the process. If the match fails, <else> is evaluated. <Else> defaults to NIL. When the process terminates or is suspended, PROCESS returns an item from the match of <pattern>.

50. (SUSPEND <pattern><db>) {@SUBR}

The current process, P1, is suspended to <pattern> in the tuplebase, <db>. <Pattern> must be a tuple and <db> a tuplebase of processes. Control returns to P2, the process which invoked P1. When resumed, SUSPEND returns an item representing the match to <pattern>.

51. (RESUME <pattern><db>[<else>]) {@SUBR}

Resumes suspended processes and creates new named processes. <Pattern>, which must be a tuple, is matched against <db> which must be a tuplebase of processes. If the match fails, <else> is evaluated which defaults to NIL. If the match succeeds, the matched process is resumed if it is a suspended process or a new process is created and begun if it is a named process. When the process returns, RESUME returns an item representing the match to <pattern> plus a reactivation tag for RESUME.

52. (COMPLETE <pattern><db>[<else>]) {@SUBR}

Suspends all active processes associated with the same schema as the current process, P1, to a reactivation tag, then resumes a specified process, P2. If P1 is not associated with any schema, only P1 is suspended. <Pattern>, which must be a tuple, is matched against <db>, which must be a tuplebase of processes. If the match fails, <else> is evaluated which defaults to NIL. If the match succeeds, P1 is suspended to a reactivation tag. P2 is resumed if it was a suspended process or created if it was a named process. Included in the item returned to the invocation of P2 is the reactivation for P1. If P2 terminates via any function except EXIT, P1 will be resumed from the COMPLETE function. COMPLETE returns an item representing the match of <pattern> plus a reactivation tag for subsequent invocations of the function.

APPENDIX C

EXAMPLE PROGRAM LISTING

```
;*** A POLYHEDRAL BLOCKS RECOGNIZER ***
; W. S. HAVENS, UBC, VANCOUVER, CANADA, MAY 1977.
;
;TOP-LEVEL CALL.
(DEFUN RECOGNIZE @EXPR NIL
  (: SCENE
     ; CREATE A TOP-LEVEL PROCESS.
     (PROCESS (SELF)
       '(QLAMBDA <!¬DESCRIPTION>
          (COMMENTARY '"METHOD: TOP-LEVEL RECR EXHAUSTIVELY
                          OBSERVES EACH VERTEX")
          ;EXHAUSTIVELY OBSERVE EACH VERTEX.
          (MAPC '(LAMBDA (V) (: ;SEND THIS VERTEX.
                                (AVAL ?V)
                                (COMMENTARY '"SCENE RECR OBSERVES"
                                   ?V)
                                ;A MESSAGE TO COMPLETE.
                                (RESUME '<OBSERVE-VERTEX> PROC)))
                ?VERTEX-LIST)
          ;A SCENE HAS NOT BEEN FOUND.
          (PRINT 'FAILURE)
          (FAIL))
       '<!:DESCRIPTION>
       '(RETURN NIL 'RECOGNIZE))
     ;SUCCESS!
     (BREAK '"** SUCCESS! **" ?DESCRIPTION)))

; CREATE GENERIC SCHEMATA
(: (PUT* 'SCENE (OBJECT 'SCHEMA NAME 'SCENE
                                COMPOSITION 'POLYHEDRON
                                PROC (OBJECT 0)))

  ;SCENE COMPLETION PROCESS.
  (PADD '<SCENE-RECR !:POLY> PROC (NAMED-PROCESS 'SCENE-COMPLETE
          (SELF)))

  ;SCENE COMPLETION PROCESS.
  (DEFUN SCENE-COMPLETE QEXPR <SCENE-RECR !:POLY>
    (COMMENTARY '"SCENE RECOGNIZER HAS FOUND A SCENE COMPOSED OF"
                 (: ?POLY NAME))
    (SETQ DESCRIPTION ?POLY)
    (EXIT))

 ) ;END OF SCENE SCHEMA.
```

```
(PUT* 'LINE (OBJECT 'SCHEMA NAME 'LINE))

(: (PUT* 'VERTEX (OBJECT 'SCHEMA PROC (OBJECT 0)))

  ;VERTEX COMPLETION PROCESS.
  (PADD '<OBSERVE-VERTEX> PROC (NAMED-PROCESS 'OBSERVE-VERTEX))

  ;OBSERVATION OF A VERTEX
  ;DEFAULT IS 3-LINE VERTICES
  (DEFUN OBSERVE-VERTEX QEXPR <OBSERVE-VERTEX>
    (: ;SAVE POINTER TO THIS VERTEX.
       (ITEM SELF (SELF))
       (ELSE '(RETURN NIL ':))
       ;FOR EACH SECTOR OF THIS VERTEX...
       (GEN-SECTORS '<?SELF !:L-LINE !:R-LINE !:ANGLE !:SECTOR>
                    (FAIL))
       ;IF ALREADY OBSERVED THEN FAIL
       (AND (GET* ?SECTOR)(FAIL))
       (COMMENTARY '"FOR" ?SECTOR '"OF" (: ?SELF NAME))
       ;GOTO FACE
       FACE
       ;EITHER MATCH THIS SECTOR TO AN EXISTING FACE RECR...
       (OR
         ;FOR EACH FACE RECR...
         (MAPC '(LAMBDA (RECR)
                   (: ?RECR
                      ;MATCH IT CWISE OR CCWISE
                      (COMMENTARY '"ATTEMPT TO MATCH THE
                                    EXPECTATIONS OF"
                                    (: ?RECR NAME))
                      (OR  (COMPLETE '<FACE-RECR
                                       CWISE
                                       ?R-LINE
                                       !:(FV (NEQ ?FV ?SELF))
                                       ?SELF
                                       ?L-LINE
                                       ?ANGLE
                                       ?SECTOR>
                                  CONSTRAINTS)
                           (COMPLETE '<FACE-RECR
                                       CCWISE
                                       ?L-LINE
                                       !:(FV (NEQ ?FV ?SELF))
                                       ?SELF
                                       ?R-LINE
                                       ?ANGLE
                                       ?SECTOR>
                                  CONSTRAINTS)
                           (RETURN NIL 'LAMBDA))
                      ;MATCH SUCCEEDED.
                    (RETURN 'T 'MAPC)))
              DNET)
         ;OR A NEW ONE.
         (: (NEW-FACE-RECR '<?SELF ?L-LINE ?R-LINE ?ANGLE ?SECTOR>)))
       ;RETURN FOR NEXT SECTOR.
       (FAIL)))

 ) ;END OF VERTEX.


; DEFINE VERTEX TYPES.
(PUT* 'ARROW-VERTEX (NEW VERTEX TYPE 'ARROW))
(PUT* 'FORK-VERTEX (NEW VERTEX TYPE 'FORK))
(PUT* 'T-VERTEX (NEW VERTEX TYPE 'T))
(PUT* 'L-VERTEX (NEW VERTEX TYPE 'L))
(: VERTEX (PUT* 'INSTANCES (LIST ARROW-VERTEX
                                 FORK-VERTEX
                                 T-VERTEX
                                 L-VERTEX)))


; DEFINE FACE SCHEMA
(: (PUT* 'FACE (OBJECT 'SCHEMA NAME 'FACE
                               COMPOSITION '(VERTEX LINE)
                               DNET NIL
                               NRECRS 0))

  ;NEW FACE RECOGNIZER
   (DEFUN NEW-FACE-RECR QEXPR <!:FIRST-VERTEX
                                 !:L-EDGE
                                 !:R-EDGE
                                 !:ANGLE
                                 !:SECTOR>
     (: ;CREATE AN INITIAL CORNER FOR THIS FACE
        (ITEM CORNER (OBJECT 'SNET
                       VERTEX (: ?FIRST-VERTEX NAME)
                       L-EDGE ?L-EDGE
                       R-EDGE ?R-EDGE
                       ANGLE ?ANGLE))
        ;VERTEX SECTOR IS THIS NEW CORNER.
        (: ?FIRST-VERTEX (PUT* ?SECTOR ?CORNER) T)
        ;CREATE A NEW FACE RECOGNIZER
        (NEW FACE
          NAME (MKATOM '"FACE-" (ADD1 (: FACE NRECRS)))
          CONSTRAINTS (OBJECT 0)
          CORNERS (LIST ?CORNER)
          EDGES NIL
          CCW-CORNER ?CORNER
          CW-CORNER ?CORNER)
        (ITEM SELF (SELF))
        (COMMENTARY '"CREATE A NEW RECR:" NAME '"CONTAINING
            A CORNER FOR" (: ?FIRST-VERTEX NAME))
        (: ?CORNER (PUT* 'PART-OF ?SELF) T)
        ;CREATE 2 PROCESSES TO EXPECT VERTICES CWISE AND CCWISE
        ; AROUND THIS FACE.
        (COMMENTARY '"CREATE TWO PROCESSES BOUND TO EXPECTATIONS
            FOR" ?L-EDGE '"AND" ?R-EDGE)
        (PROCESS ?SELF 'CW-FACE-RECR '<?L-EDGE ?FIRST-VERTEX>)
        (PROCESS ?SELF 'CCW-FACE-RECR '<?R-EDGE ?FIRST-VERTEX>)
        ;RECORD THIS RECR INSTANCE IN THE GENERIC FACE SCHEMA
        FACE
        (PUT* 'NRECRS (ADD1 NRECRS))
        (PUT* 'DNET (CONS ?SELF DNET))))

(DEFUN CW-FACE-RECR QEXPR <!:EXPECT !:FIRST-VERTEX>
  (: ;GET THIS SCHEMA.
        (ITEM SELF (SELF))
        ;SUSPEND THIS PROCESS TO CWISE EXPECTATIONS
        (SUSPEND <FACE-RECR
                    CWISE
                    ?EXPECT
                    ?FIRST-VERTEX
                    !:VERTEX
                    !:NEXT-EXPECT
                    !:ANGLE
                    !:SECTOR>
                 CONSTRAINTS)
        (COMMENTARY (: ?SELF NAME) '"HAS BEEN MATCHED BY"
          (: ?VERTEX NAME))
        ;CREATE A NEW CORNER FOR THIS VERTEX.
        (ITEM CORNER (OBJECT 'SNET
                       PART-OF ?SELF
                       VERTEX (: ?VERTEX NAME)
                       R-EDGE ?EXPECT
                       L-EDGE ?NEXT-EXPECT
                       ANGLE ?ANGLE
                       NEXT-CCW CW-CORNER))
        ;VERTEX SECTOR COMPRISES THIS NEW CORNER.
        (: ?VERTEX (PUT* ?SECTOR ?CORNER) T)
        ;ADD TO LIST OF CORNERS FOR THIS FACE
        ?SELF
        (PUT* 'CORNERS (CONS ?CORNER CORNERS))
        ;UPDATE LIST OF EDGES FOR THIS FACE.
        (PUT* 'EDGES (CONS ?EXPECT EDGES))
        ;EXCHANGE CWISE AND CCWISE POINTERS.
        (: CW-CORNER (PUT* 'NEXT-CW ?CORNER) T)
        (: ?CORNER (PUT* 'NEXT-CCW CW-CORNER) T)
        ;UPDATE CW-CORNER
        (: (PUT* 'CW-CORNER ?CORNER) T)
        ;VERIFY THIS FACE USING TOP-DOWN SEARCH
        (COMMENTARY '"VERIFY" (: ?SELF NAME) '"USING TOP-DOWN
          SEARCH")
```

```
    (VERIFY-FACE ?SELF (ADD (: CW-CORNER ANGLE)
                            (: CCW-CORNER ANGLE)))))

(DEFUN CCW-FACE-RECR QEXPR <!:EXPECT !:FIRST-VERTEX>
  (: ;GET THIS SCHEMA.
    (ITEM SELF (SELF))
    ;SUSPEND THIS PROCESS TO CCWISE EXPECTATIONS.
    (SUSPEND <FACE-RECR
               CCWISE
               ?EXPECT
               ?FIRST-VERTEX
               !:VERTEX
               !:NEXT-EXPECT
               !:ANGLE
               !:SECTOR>
             CONSTRAINTS)
    (COMMENTARY (: ?SELF NAME) '"HAS BEEN MATCHED BY"
      (: ?VERTEX NAME))
    ;CREATE A NEW CORNER FOR THIS FACE.
    (ITEM CORNER (OBJECT 'SNET
                   PART-OF ?SELF
                   VERTEX (: ?VERTEX NAME)
                   R-EDGE ?NEXT-EXPECT
                   L-EDGE ?EXPECT
                   ANGLE ?ANGLE
                   NEXT-CW CCW-CORNER))
    ;VERTEX SECTOR COMPRISES THIS NEW CORNER.
    (: ?VERTEX (PUT* ?SECTOR ?CORNER) T)
    ;ADD TO LIST OF CORNERS FOR THIS FACE.
    ?SELF
    (PUT* 'CORNERS (CONS ?CORNER CORNERS))
    ;UPDATE LIST OF EDGES FOR THIS FACE.
    (PUT* 'EDGES (CONS ?EXPECT EDGES))
    ;EXCHANGE CWISE AND CCWISE POINTERS.
    (: CCW-CORNER (PUT* 'NEXT-CCW ?CORNER) T)
    (: ?CORNER (PUT* 'NEXT-CW CCW-CORNER) T)
    ;UPDATE CCWISE CORNER.
    (: (PUT* 'CCW-CORNER ?CORNER) T)
    ;VERIFY THIS FACE USING TOP-DOWN SEARCH.
    (COMMENTARY '"METHOD: VERIFY" (: ?SELF NAME)
      '"USING TOP-DOWN SEARCH")
    (VERIFY-FACE ?SELF (ADD (: CW-CORNER ANGLE)
                            (: CCW-CORNER ANGLE)))))

;SEARCH TOP-DOWN FOR THE REMAINDER OF THIS FACE.
(DEFUN VERIFY-FACE @EXPR (THIS-FACE EX-ANGLE)
  (: ;GET NEIGHBORING CWISE VERTEX.
    (ITEM NEIGHBOR-VERTEX (: ?THIS-FACE
                             CW-CORNER
                             (NEIGHBOR-VERTEX L-EDGE VERTEX)))
    (COMMENTARY '"GET NEXT CLOCKWISE NEIGHBOR VERTEX:"
   (: ?NEIGHBOR-VERTEX NAME)
   '"FROM" (: ?THIS-FACE CW-CORNER L-EDGE))
 ;GET FACE RECR THAT THIS CWISE SECTOR IS PART OF.
 ; ELSE NIL.
 (ITEM OTHER-FACE (: ?THIS-FACE
                     CW-CORNER
                     ?NEIGHBOR-VERTEX
                     (GET* (:(CWISE-SECTOR L-EDGE))
                           '(RETURN NIL ':))
                     PART-OF))
 (COND ;IS THIS VERTEX PART OF THIS FACE RECR?
       ((EQ ?THIS-FACE ?OTHER-FACE)
          (COMMENTARY '"THIS VERTEX IS ALREADY CONTAINED IN"
            (: ?THIS-FACE NAME))
          (: ;COMPLETE THE DESCRIPTION OF THIS FACE.
             ?THIS-FACE
             ;UPDATE LIST OF EDGES.
             (PUT* 'EDGES (CONS (: CW-CORNER L-EDGE) EDGES))
             ;MAKE CWISE AND CCWISE RINGS.
             (: CW-CORNER (PUT* 'NEXT-CW CCW-CORNER) T)
             (: CCW-CORNER (PUT* 'NEXT-CCW CW-CORNER) T)
             ;TEST THE COMPLETED FACE.
             (COMMENTARY '"COMPARE" (: ?THIS-FACE NAME)
               '"TO POLYGON MODEL")
             (TEST-COMPLETED-FACE ?THIS-FACE ?EX-ANGLE)
             ;MATCH THIS FACE TO POLYHEDRON SUPERGOALS.
             (COMPLETE-FACE ?THIS-FACE)))
       ;IS THIS SECTOR PART OF SOME OTHER RECR?
       (?OTHER-FACE
          (COMMENTARY '"ITS CWISE SECTOR IS ALREADY PART
             OF A CORNER OF" (: ?OTHER-FACE NAME))
          ;DELETE THE OTHER FACE FROM DNET OF GENERIC FACE.
          (: FACE (PUT* 'DNET (DELQ ?OTHER-FACE DNET)) T)
          ;INCORPORATE ITS CORNERS INTO THIS FACE.
          ;EXCHANGE CWISE AND CCWISE POINTERS.
          (: ?OTHER-FACE
             CCW-CORNER
             (PUT* 'NEXT-CCW (: ?THIS-FACE CW-CORNER)))
          (: ?THIS-FACE
             CW-CORNER
             (PUT* 'NEXT-CW (: ?OTHER-FACE CCW-CORNER)))
          ;FOLLOW THE EDGES OF THE OTHER FACE CWISE.
          (COMMENTARY '"INCORPORATE THE CORNERS OF"
            (: ?OTHER-FACE NAME) '"INTO"
            (: ?THIS-FACE NAME))
          (:(FOLLOW-FACE ?THIS-FACE ?EX-ANGLE)))
       ;ELSE THIS VERTEX HAS NOT BEEN OBSERVED BEFORE.
       (T (COMMENTARY '"INCORPORATE THIS VERTEX INTO A
            NEW CORNER OF" (: ?THIS-FACE NAME))
          (: ;CREATE A NEW CORNER FOR THIS VERTEX.
             ?THIS-FACE
```

```
                    ?NEIGHBOR-VERTEX
                    (ITEM CORNER (OBJECT 'SNET
                                   PART-OF ?THIS-FACE
                                   VERTEX NAME
                                   L-EDGE
                                    (: CW-CORNER (CW-LINE L-EDGE))
                                   R-EDGE L-EDGE
                                   ANGLE
                                    (: CW-CORNER
                                       (CW-ANGLE L-EDGE))))
                    ;MAKE VERTEX POINT TO CORNER.
                    (PUT* (:(CWISE-SECTOR (: CW-CORNER L-EDGE)))
                          ?CORNER)
                    ;ADD TO THIS FACE.
                    ?THIS-FACE
                    (PUT* 'CORNERS (CONS ?CORNER CORNERS))
                    ;UPDATE LIST OF EDGES FOR THIS FACE.
                    (PUT* 'EDGES (CONS (: CW-CORNER L-EDGE) EDGES))
                    ;EXCHANGE CWISE AND CCWISE POINTERS.
                    (: CW-CORNER (PUT* 'NEXT-CW ?CORNER) T)
                    (: ?CORNER (PUT* 'NEXT-CCW CW-CORNER) T)
                    ;UPDATE CWISE CORNER
                    (: (PUT* 'CW-CORNER ?CORNER) T)
                    ;RECURSE ON THE NEXT CWISE VERTEX.
                    (VERIFY-FACE ?THIS-FACE (ADD ?EX-ANGLE
                                    (: ?CORNER ANGLE)))))))

;INCORPORATE THE NEW CORNERS INTO THIS FACE.
(DEFUN FOLLOW-FACE @EXPR (?THIS-FACE ?EX-ANGLE)
  (: ;GOTO CWISE CORNER OF THIS FACE.
     ?THIS-FACE
     CW-CORNER
     (COND ;IS THERE A NEXT CWISE CORNER?
           ((GET* 'NEXT-CW)
              (COMMENTARY '"INCLUDE THE CORNER FOR"
                (: NEXT-CW VERTEX))
              ;INCLUDE IN CORNERS LIST.
              (: ?THIS-FACE
                 (PUT* 'CORNERS (CONS NEXT-CW CORNERS))
                 (PUT* 'EDGES (CONS L-EDGE EDGES)))
              ;ADVANCE POINTER TO THE NEXT CWISE CORNER.
              (: ?THIS-FACE (PUT* 'CW-CORNER NEXT-CW))
              ;MAKE THIS CORNER POINT TO THIS FACE.
              (: CW-CORNER (PUT* 'PART-OF ?THIS-FACE))
              ;RECURSE ON THE NEXT CORNER.
              (:(FOLLOW-FACE ?THIS-FACE (ADD ANGLE ?EX-ANGLE))))
           ;VERIFY EACH CWISE VERTEX.
           (T(COMMENTARY '"ALL CORNERS HAVE BEEN INCLUDED")
             (:(VERIFY-FACE ?THIS-FACE
                         (ADD ANGLE ?EX-ANGLE)))))))
```

```
;TEST THIS FACE AND ASSIGN ITS TYPE.
(DEFUN TEST-COMPLETED-FACE @EXPR (FACE EX-ANGLE)
  (: ?FACE
     ;CHECK THE SUM OF THE FACE'S EXTERIOR ANGLES.
     (COND ((EPSILON 10 360 ?EX-ANGLE)
            (PUT* 'TYPE 'INSIDE-CLOSURE))
           ((EPSILON 10 -360 ?EX-ANGLE)
            ;DECIDE FACE'S TYPE.
            (SELECTQ (LENGTH EDGES)
              (3 (PUT* 'TYPE 'TRIANGLE))
              (4 (COND ;IS FACE A PARALLELOGRAM?
                   ((AND (EPSILON 10 (MINUS (: CW-CORNER ANGLE))
                                     (: CCW-CORNER NEXT-CW ANGLE))
                         (EPSILON 10 (MINUS (: CCW-CORNER ANGLE))
                                     (: CW-CORNER NEXT-CCW ANGLE)))
                    (PUT* 'TYPE 'PARALLELOGRAM))
                   (T(PUT* 'TYPE 'QUADRALATERAL))))
              (PUT* 'TYPE 'MULTILATERAL))))
     (COMMENTARY '"COMPLETED" NAME '"IS A" TYPE)))

   ;MATCH THIS FACE TO POLYHEDRON RECRS.
   (DEFUN COMPLETE-FACE @EXPR (THIS-FACE)
     ;DELETE THIS RECR FROM DNET OF FACE RECRS.
     (: FACE (PUT* 'DNET (DELQ ?THIS-FACE DNET)))
     (COMMENTARY '"MATCH THIS FACE TO THE EXPECTATIONS OF
       POLYHEDRON RECOGNIZERS")
     (: POLYHEDRON
        (OR ;MATCH "CONNECT" EXPECTATIONS OF SOME POLYHEDRON RECR.
            (MAPC '(LAMBDA (RECR)
                     (COMMENTARY '"MATCH THE CONNECT EXPECTATIONS
                       OF" (: ?RECR NAME))
                     (: ?RECR
                        (ELSE '(RETURN NIL ':))
                        ;GENERATE ALL CONNECT EDGES FOR THIS FACE.
                        (GEN-EDGES '<?THIS-FACE
                                     !:EXPECT
                                     !:EDGE-TYPE>
                                   '(FAIL))
                        (OR (EQ ?EDGE-TYPE 'CONNECT)(FAIL))
                        (COMMENTARY '"TRY" ?EXPECT)
                        ;MATCH THIS RECR'S EXPECTATIONS.
                        (COMPLETE '<POLYHEDRON-RECR
                                    CONNECT
                                    ?EXPECT
                                    !:NODE
                                    ?THIS-FACE>
                                  CONSTRAINTS
                                  '(FAIL))
                        ;DONE.
                        (RETURN T 'MAPC)))
```

```
                RECRS)
          ;ELSE MATCH "MAYBE-CONNECT" EXPECTATIONS OF ALL RECRS.
          (PROGN
            (MAPC '(LAMBDA (RECR)
                     (COMMENTARY '"MATCH THE MAYBE-CONNECT
                       EXPECTATIONS OF" (: ?RECR NAME))
                     (: ?RECR
                        (ELSE '(RETURN NIL ':))
                        ;GENERATE MAYBE CONNECT EXPECTATIONS.
                        (GEN-EDGES '<?THIS-FACE
                                     !:EXPECT
                                     !:EDGE-TYPE>
                                   '(FAIL))
                        (OR (EQ ?EDGE-TYPE 'MAYBE-CONNECT)(FAIL))
                        (COMMENTARY '"TRY" ?EXPECT)
                        ;MATCH THIS RECR'S EXPECTATIONS.
                        (COMPLETE '<POLYHEDRON-RECR
                                     MAYBE-CONNECT
                                     ?EXPECT
                                     !:NODE
                                     ?THIS-FACE>
                                   CONSTRAINTS
                                   '(FAIL))
                        ;DONE WITH THIS RECR.
                        (RETURN NIL ':)))
                  RECRS)
            ;AND CREATE A NEW RECR FOR THIS FACE.
            (:(NEW-POLYHEDRON-RECR ?THIS-FACE))))))

;GENERATES THE EDGES AND EDGE TYPES FOR A GIVEN FACE.
(DEFUN GEN-EDGES QEXPR <!:FACE !¬EDGE !¬EDGE-TYPE>
  (: ?FACE
     ;FOR EACH EDGE IN THIS FACE...
     (MAPC '(LAMBDA (E)
             (: (AVAL ?E)
                (COND ((OR (: (AVAL V1)(AND (EQ TYPE 'ARROW)
                                             (EQ L2 ?E)))
                           (: (AVAL V2)(AND (EQ TYPE 'ARROW)
                                             (EQ L2 ?E))))
                       ;THEN EDGE IS CONNECT.
                       (SETQ EDGE ?E EDGE-TYPE 'CONNECT))
                      (T ;ELSE EDGE IS MAYBE-CONNECT.
                       (SETQ EDGE ?E EDGE-TYPE 'MAYBE-CONNECT)))
                (POST)))
           EDGES)
     (FAIL)))


) ;END OF FACE
```

```
(: (PUT* 'POLYHEDRON (OBJECT 'SCHEMA
                             NAME 'POLYHEDRON
                             COMPOSITION 'FACE
                             RECRS NIL
                             NRECRS 0))

   ;CREATES A NEW POLYHEDRON RECR.
   (DEFUN NEW-POLYHEDRON-RECR @EXPR (FACE1)
     ;IS THIS FIRST FACE COMPATIBLE WITH THIS RECR'S MODEL
     ;  OF POLYHEDRA?
     (OR (: POLYHEDRON (TEST-FACE-TYPE ?FACE1))
         (RETURN NIL 'NEW-POLYHEDRON-RECR))
     (: ;CREATE A CONNECT NODE TO REPRESENT THIS FACE.
        (ITEM NODE (OBJECT 'SNET
                    NFACE ?FACE1
                    CONNECT NIL))
        ;CREATE A POLYHEDRON SCHEMA INSTANCE.
        (NEW POLYHEDRON
          NAME (MKATOM '"POLYHEDRON-" (: POLYHEDRON (ADD1 NRECRS)))
          CONSTRAINTS (OBJECT 0)
          NODES (LIST ?NODE)
          MAYBE-NODES NIL)
        ;SAVE POINTER TO THIS RECR.
        (ITEM RECR (SELF))
        (COMMENTARY '"CREATE A NEW RECR:" NAME '"CONTAINING"
          (: ?FACE1 NAME))
        (COMMENTARY '"COMPUTE EXPECTATIONS ABOUT OTHER FACES OF"
          NAME)
        ;RECORD IN THE GENERIC POLYHEDRON SCHEMA.
        (: POLYHEDRON
           (PUT* 'RECRS (CONS ?RECR RECRS))
           (PUT* 'NRECRS (ADD1 NRECRS)))
        ;CREATE A PROCESS BOUND TO AN EXPECTATION FOR EACH EDGE
        ; IN THIS FACE.
        (: (ELSE '(RETURN NIL ':))
           ;GENERATE EACH EDGE OF THIS FACE AND ITS TYPE.
           (: FACE (GEN-EDGES '<?FACE1 !:EDGE !:EDGE-TYPE> '(FAIL)))
           (COMMENTARY '"FOR" ?EDGE '"OF" (: ?FACE1 NAME))
           ;EITHER A CONNECT OR MAYBE-CONNECT PROCESS.
           (COND ((EQ ?EDGE-TYPE 'CONNECT)
                  (PROCESS ?RECR
                           'ACCEPT-CONNECT-FACE
                           '<?RECR ?EDGE ?NODE>))
                 ((PROCESS ?RECR
                           'ACCEPT-MAYBE-CONNECT-FACE
                           '<?RECR ?EDGE ?NODE>)))
           ;GO BACK FOR NEXT EDGE.
           (FAIL))
        ;SEARCH FOR OTHER FACES OF THIS POLYHEDRON BY OBSERVING
        ; 3-LINE VERTICES OF THIS FACE.
        (COMMENTARY '"METHOD: SEARCH FOR OTHER FACES OF"
          (: ?RECR NAME) '"BY OBSERVING VERTICES OF")
        (COMMENTARY '"      " (: ?FACE1 NAME)
          '"THAT MAY BE PART OF MORE THAN ONE FACE")
        ?FACE1
        (MAPC '(LAMBDA (CORNER)
                 (: ?CORNER
                    (AVAL VERTEX)
                    (COND ((MEMQ TYPE '(ARROW FORK T))
                           (COMMENTARY (: ?RECR NAME) '"OBSERVES"
                             NAME)
                           (RESUME '<OBSERVE-VERTEX> PROC)))))
              CORNERS)))

;ACCEPTS MATCHING CONNECT FACE INTO THIS POLYHEDRAL DESCRIPTION.
(DEFUN ACCEPT-CONNECT-FACE QEXPR <!:RECR !:EXPECT !:EXPECT-NODE>
  (COMMENTARY '"CREATE A PROCESS BOUND TO THE CONNECT
    EXPECTATION:" ?EXPECT)
  (: ?RECR
     ;SUSPEND THIS PROCESS TO THIS EXPECTATION.
     (SUSPEND <POLYHEDRON-RECR CONNECT ?EXPECT ?EXPECT-NODE
               !:FACE> CONSTRAINTS)
     (COMMENTARY '"A CONNECT EXPECTATION OF" (: ?RECR NAME)
       '"HAS BEEN MATCHED BY" (: ?FACE NAME))
     ;IS NEW FACE COMPATIBLE WITH THIS MODEL OF POLYHEDRA?
     (OR (:(TEST-FACE-TYPE ?FACE))(FAIL))
     ;CREATE A NODE IN THIS RECR'S DESCRIPTION FOR THIS FACE.
     (ITEM NODE (OBJECT 'SNET
                 NFACE ?FACE
                 MAYBE-CONNECT NIL
                 CONNECT (LIST ?EXPECT-NODE)))
     (PUT* 'NODES (CONS ?NODE NODES))
     (: ?EXPECT-NODE (PUT* 'CONNECT (CONS ?NODE CONNECT)))
     ;COMPUTE TRANSITIVE CLOSURE OF EDGES FOR THIS FACE.
     (COMMENTARY '"COMPUTE TRANSITIVE EDGE CLOSURE FOR THIS FACE")
     (: (ELSE '(RETURN NIL ':))
        ;FOR EACH OTHER EDGE IN THIS FACE...
        (: FACE (GEN-EDGES '<?FACE !:EDGE !:EDGE-TYPE> '(FAIL)))
        ;IS IT THE PREVIOUS EXPECTATION?
        (AND (EQ ?EDGE ?EXPECT)(FAIL))
        (COMMENTARY '"FOR" ?EDGE '"OF" (: ?FACE NAME))
        ;ELSE COMPUTE CLOSURE FOR THIS EDGE.
        (EDGE-CLOSURE ?RECR ?EDGE ?EDGE-TYPE ?NODE)
        ;GET NEXT EDGE.
        (FAIL))
     ;MATCH MAYBE CONNECT FACES TO THESE NEW EXPECTATIONS.
     (MAPC '(LAMBDA (MNODE)
              (: (ITEM MFACE (: ?MNODE NFACE))
                 (COMMENTARY '"ATTEMPT TO MATCH MAYBE-CONNECT
                   FACE:" (: ?MFACE NAME) '"TO THESE NEW
                   EXPECTATIONS")
```

```
                (ELSE '(RETURN NIL ':))
                ;FOR ANY CONNECT EDGE IN THIS FACE...
                (: FACE (GEN-EDGES '<?MFACE !:EDGE !:EDGE-TYPE>
                                   '(FAIL)))
                (OR (EQ ?EDGE-TYPE 'CONNECT)(FAIL))
                (COMMENTARY '"TRY" ?EDGE)
                ;IS THIS EDGE NOW CONNECT WITH THIS RECR?
                (MATCH '<POLYHEDRON-RECR
                          CONNECT
                          ?EDGE
                          !:EXPECT-NODE
                          XX>
                       CONSTRAINTS
                       '(FAIL))
                (COMMENTARY (: ?MFACE NAME) '"IS CONNECT WITH"
                  (: ?RECR NAME))
                ;THEN MOVE THIS NODE TO CONNECT LIST.
                (PUT* 'NODES (CONS ?MNODE NODES))
                (PUT* 'MAYBE-NODES (DELQ ?MNODE MAYBE-NODES))
                ;ADD CONNECT LINK TO PREVIOUS MAYBE CONNECT.
                (: ?EXPECT-NODE (PUT* 'CONNECT
                                      (CONS ?MNODE CONNECT)))
                ;COMPUTE TRANSITIVE CLOSURE FOR EDGES
                ; OF THIS NODE.
                (COMMENTARY '"COMPUTE TRANSITIVE EDGE CLOSURE
                  FOR" (: ?MFACE NAME))
                (: (ELSE '(RETURN NIL ':))
                   ;FOR EACH EDGE IN THIS FACE...
                   (: FACE (GEN-EDGES '<?MFACE
                                        !:EDGE
                                        !:EDGE-TYPE>
                                      '(FAIL)))
                   ;EXCEPT EDGE OF PREVIOUS MAYBE-CONNECT?
                   (AND (EQ ?EDGE (: ?MNODE MEDGE))(FAIL))
                   (COMMENTARY '"FOR" ?EDGE '"OF THIS FACE")
                   ;COMPUTE CLOSURE FOR THIS EDGE.
                   (EDGE-CLOSURE ?RECR ?EDGE ?EDGE-TYPE ?MNODE)
                   ;GET NEXT EDGE.
                   (FAIL))
                ;FINISHED WITH THIS NODE.
                (RETURN NIL 'LAMBDA)))
          MAYBE-NODES)
   (COND ;DOES DESCRIPTION SATISFY POLYHEDRON MODEL?
         ((:(TEST-POLY ?RECR))(:(COMPLETE-POLYHEDRON ?RECR)))
         ;ELSE SEARCH FOR OTHER FACES OF THIS VERTEX BY
         ; OBSERVING 3-LINE VERTICES THAT ARE AT THE
         ; PERIPHERY OF THIS INCOMPLETE POLYHEDRON.
         (T(COMMENTARY '"METHOD: OBSERVE VERTICES THAT WILL
              DRIVE THE RECOGNITION OF NEIGHBORING FACES")
           (: (ELSE '(RETURN NIL ':))
              ;FOR EACH EXPECTATION IN THIS RECR...
```

```
                (MATCH '<POLYHEDRON-RECR
                         !:EDGE-TYPE
                         !:EDGE
                         !:EXPECT-NODE
                         XX>
                       CONSTRAINTS
                       '(FAIL))
                (AVAL ?EDGE)
                ;OBSERVE EITHER VERTEX OF EDGE IF A 3-LINE VERTEX.
                (: (AVAL V1)(COND ((MEMQ TYPE '(ARROW FORK T))
                                   (COMMENTARY (: ?RECR NAME)
                                     '"OBSERVES" NAME)
                                   (RESUME '<OBSERVE-VERTEX>
                                           PROC))))
                (: (AVAL V2)(COND ((MEMQ TYPE '(ARROW FORK T))
                                   (COMMENTARY (: ?RECR NAME)
                                     '"OBSERVES" NAME)
                                   (RESUME '<OBSERVE-VERTEX>
                                           PROC))))
                ;GO BACK FOR NEXT EXPECTATION.
                (FAIL))))))

;ACCEPTS MAYBE-CONNECT FACES INTO THIS RECR.
(DEFUN ACCEPT-MAYBE-CONNECT-FACE QEXPR
  <!:RECR !:EXPECT !:EXPECT-NODE>
  (COMMENTARY '"CREATE A PROCESS BOUND TO THE MAYBE-CONNECT
    EXPECTATION:" ?EXPECT)
  (: ?RECR
     ;SUSPEND THIS PROCESS TO THIS EXPECTATION.
     (SUSPEND <POLYHEDRON-RECR
               MAYBE-CONNECT
               ?EXPECT
               ?EXPECT-NODE
               !:FACE>
             CONSTRAINTS)
     (COMMENTARY '"A MAYBE-CONNECT EXPECTATION OF"
       (: ?RECR NAME) '"HAS BEEN MATCHED BY" (: ?FACE NAME))
     ;IS NEW NODE COMPATIBLE WITH MODEL OF POLYHEDRA?
     (OR (:(TEST-FACE-TYPE ?FACE))(FAIL))
     ;CREATE A NEW NODE FOR THIS FACE.
     (ITEM NODE (OBJECT 'SNET
                 NFACE ?FACE
                 MEDGE ?EXPECT
                 CONNECT (LIST ?EXPECT-NODE)))
     ;INCORPORATE THIS NODE INTO THIS POLYHEDRON DESCRIPTION.
     (PUT* 'MAYBE-NODES (CONS ?NODE MAYBE-NODES))))


;COMPUTES THE TRANSITIVE CLOSURE OF CONNECT EDGES.
(DEFUN EDGE-CLOSURE @EXPR (THIS-RECR EDGE EDGE-TYPE NEW-NODE)
  (: ?THIS-RECR
     (COND ;DOES EDGE MATCH EXPECTATIONS OF THIS RECR?
           ((: (PDELETE '<POLYHEDRON-RECR
                              ?EDGE-TYPE
                              ?EDGE
                              !:THIS-NODE
                              XX>
                          CONSTRAINTS
                          '(RETURN NIL ':))
               (COMMENTARY ?EDGE '"MATCHES A" ?EDGE-TYPE
                 '"EXPECTATION OF" (: ?THIS-RECR NAME))
               ;EXCHANGE CONNECT LINKS BETWEEN NODES.
               ?THIS-NODE
               (PUT* 'CONNECT (CONS ?NEW-NODE CONNECT))
               ?NEW-NODE
               (PUT* 'CONNECT (CONS ?THIS-NODE CONNECT)))
           ;ELSE CREATE A CONNECT OR MAYBE-CONNECT PROCESS.
           ((EQ ?EDGE-TYPE 'CONNECT)
              (PROCESS ?THIS-RECR
                       'ACCEPT-CONNECT-FACE
                       '<?THIS-RECR ?EDGE ?NEW-NODE>))
           ((PROCESS ?THIS-RECR
                     'ACCEPT-MAYBE-CONNECT-FACE
                     '<?THIS-RECR ?EDGE ?NEW-NODE>)))))

;MATCH COMPLETED POLYHEDRON TO EXPECTATIONS OF SCENE SCHEMA.
(DEFUN COMPLETE-POLYHEDRON @EXPR (RECR)
  (COMMENTARY '"MATCH COMPLETED" (: ?RECR NAME)
    '"TO THE EXPECTATIONS OF THE SCENE RECR")
  (: SCENE (COMPLETE '<SCENE-RECR ?RECR> PROC '(FAIL))))

;TEST FOR COMPLETED POLYHEDRON.
(DEFUN TEST-POLY @EXPR (RECR)
  (COMMENTARY '"DOES DESCRIPTION OF" (: ?RECR NAME)
    '"SATISFY THE CRITERIA FOR A COMPLETE POLYHEDRAL OBJECT?")
  (: ?RECR
     ;ARE THERE NO MORE CONNECT EXPECTATIONS FOR THIS POLYHEDRON?
     (COND ((MATCH '<POLYHEDRON-RECR CONNECT !:XX !:XX XX>
                   CONSTRAINTS)
              (COMMENTARY '"NO")
              NIL)
           ((PROG (NTRI)
              (SETQ NTRI 0)
              (MAPC '(LAMBDA (NODE)
                       (: ?NODE
                          NFACE
                          (AND (EQ TYPE 'TRIANGLE)
                               (SETQ NTRI (ADD1 ?NTRI)))))
                    NODES)
              (SELECTQ ?NTRI
                (0 (PUT* 'TYPE 'CUBE))
                (1 (PUT* 'TYPE 'WEDGE))
                (PUT* 'TYPE 'PYRAMID))
               (COMMENTARY '"YES:" NAME '"IS A" TYPE)
               T)))))

   ;IS PROPOSED FACE COMPATIBLE WITH MODEL OF POLYHEDRA?
   (DEFUN TEST-FACE-TYPE @EXPR (FACE)
     (COMMENTARY '"IS" (: ?FACE NAME)
       '"COMPATIBLE WITH THIS CLASS OF POLYHEDRA?")
     (COND ((MEMQ (: ?FACE TYPE) '(TRIANGLE PARALLELOGRAM))
            (COMMENTARY '"YES")
            T)
          (T(COMMENTARY '"NO: REJECT THIS FACE")
            NIL)))


 ) ;END OF POLYHEDRON.

;AUXILIARY FUNCTIONS.

(: VERTEX
   ;GENERATES SECTORS OF A 3-LINE VERTEX.
   (DEFUN GEN-SECTORS QEXPR
     <!:VERTEX !¬L-LINE !¬R-LINE !¬ANGLE !¬SECTOR>
     (: ?VERTEX
        (SETQ L-LINE L2
              R-LINE L1
              ANGLE (SUB 180 ANGLE-L1-L2)
              SECTOR 'SECTOR-L1-L2)
        (POST)
        (SETQ L-LINE L3
              R-LINE L2
              ANGLE (SUB 180 ANGLE-L2-L3)
              SECTOR 'SECTOR-L2-L3)
        (POST)
        (SETQ L-LINE L1
              R-LINE L3
              ANGLE (SUB (ADD ANGLE-L1-L2 ANGLE-L2-L3) 180)
              SECTOR 'SECTOR-L3-L1)))

   ;RETURNS THE CWISE SECTOR NAME OF THE ENCLOSING VERTEX
   ; WHOSE R-LINE IS ?LINE.
   ;DEFAULT IS FOR 3-LINE VERTICES.
   (DEFUN CWISE-SECTOR @EXPR (LINE)
     (COND ((EQ ?LINE L1) 'SECTOR-L1-L2)
           ((EQ ?LINE L2) 'SECTOR-L2-L3)
           ((EQ ?LINE L3) 'SECTOR-L3-L1)
           ((BREAK '"*** ERROR: LINE IS NOT PART OF THIS VERTEX"
             ?LINE))))

   ;RETURNS THE CCWISE SECTOR NAME WHOSE L-LINE IS ?LINE OF
```

```
  ; THE ENCLOSING VERTEX.
  (DEFUN CCWISE-SECTOR @EXPR (LINE)
    (COND ((EQ ?LINE L1) 'SECTOR-L3-L1)
          ((EQ ?LINE L2) 'SECTOR-L1-L2)
          ((EQ ?LINE L3) 'SECTOR-L2-L3)
          ((BREAK '"*** ERROR: LINE IS NOT PART OF THIS VERTEX"
             ?LINE))))


 ;RETURNS THE CCW NEXT LINE OF AN ENCLOSING 3-LINE VERTEX.
 (DEFUN CCW-LINE @EXPR (LINE)
   (COND ((EQ ?LINE L1) L3)
         ((EQ ?LINE L2) L1)
         ((EQ ?LINE L3) L2)
         ((BREAK '"*** LINE IS NOT PART OF THIS VERTEX" ?LINE))))

 ;RETURNS THE CW NEXT LINE OF AN ENCLOSING 3-LINE VERTEX.
 (DEFUN CW-LINE @EXPR (LINE)
   (COND ((EQ ?LINE L1) L2)
         ((EQ ?LINE L2) L3)
         ((EQ ?LINE L3) L1)
         ((BREAK '"*** LINE IS NOT PART OF THIS VERTEX" ?LINE))))

 (DEFUN CW-ANGLE @EXPR (LINE)
   (COND ((EQ ?LINE L1) (SUB 180 ANGLE-L1-L2))
         ((EQ ?LINE L2) (SUB 180 ANGLE-L2-L3))
         ((EQ ?LINE L3)
            (SUB (ADD ANGLE-L1-L2 ANGLE-L2-L3) 180))
         ((BREAK '"*** LINE IS NOT PART OF THIS VERTEX" ?LINE))))

 (DEFUN CCW-ANGLE @EXPR (LINE)
   (COND ((EQ ?LINE L3) (SUB 180 ANGLE-L2-L3))
         ((EQ ?LINE L2) (SUB 180 ANGLE-L1-L2))
         ((EQ ?LINE L1)
            (SUB (ADD ANGLE-L2-L3 ANGLE-L1-L2) 180))
         ((BREAK '"*** LINE IS NOT PART OF THIS VERTEX" ?LINE))))


) ;END OF VERTEX

(: L-VERTEX
  ;GENERATES SECTORS OF A 2-LINE VERTEX.
  (DEFUN GEN-SECTORS QEXPR
    <!:VERTEX !¬L-LINE !¬R-LINE !¬ANGLE !¬SECTOR>
    (: ?VERTEX
       (SETQ L-LINE L2
             R-LINE L1
             ANGLE (SUB 180 ANGLE-L1-L2)
             SECTOR 'SECTOR-L1-L2)
       (POST)
       (SETQ L-LINE L1
           R-LINE L2
           ANGLE (SUB ANGLE-L1-L2 180)
           SECTOR 'SECTOR-L2-L1)))

  ;RETURNS THE CWISE SECTOR NAME WHOSE R-LINE IS ?LINE OF THE
  ; ENCLOSING VERTEX.
  ;FOR 2-LINE VERTICES.
  (DEFUN CWISE-SECTOR @EXPR (LINE)
    (COND ((EQ ?LINE L1) 'SECTOR-L1-L2)
          ((EQ ?LINE L2) 'SECTOR-L2-L1)
          ((BREAK '"*** ERROR: LINE IS NOT PART OF THIS VERTEX"
             ?LINE))))

  ;RETURNS THE CCWISE SECTOR NAME WHOSE L-LINE IS ?LINE
  ; OF THE ENCLOSING VERTEX.
  ;FOR 2-LINE VERTICES.
  (DEFUN CCWISE-SECTOR @EXPR (LINE)
    (COND ((EQ ?LINE L1) 'SECTOR-L2-L1)
          ((EQ ?LINE L2) 'SECTOR-L1-L2)
          ((BREAK '"*** ERROR: LINE IS NOT PART OF THIS VERTEX"
             ?LINE))))


 ;RETURNS THE CCW OR CW NEXT LINE OF AN ENCLOSING 2-LINE VERTEX.
 (DEFUN CW-LINE @EXPR (LINE)
   (COND ((EQ ?LINE L1) L2)
         ((EQ ?LINE L2) L1)
         ((BREAK '"*** LINE IS NOT PART OF THIS VERTEX" ?LINE))))

 (: (PUT* 'CCW-LINE CW-LINE) T)

 (DEFUN CW-ANGLE @EXPR (LINE)
   (COND ((EQ ?LINE L1) (SUB 180 ANGLE-L1-L2))
         ((EQ ?LINE L2)
            (SUB ANGLE-L1-L2 180))
         ((BREAK '"*** LINE IS NOT PART OF THIS VERTEX" ?LINE))))

 (DEFUN CCW-ANGLE @EXPR (LINE)
   (COND ((EQ ?LINE L2)(SUB 180 ANGLE-L1-L2))
         ((EQ ?LINE L1)(SUB ANGLE-L1-L2 180))
         ((BREAK '"*** LINE IS NOT PART OF THIS VERTEX" ?LINE))))


) ;END OF L-VERTEX.

; CREATE GIVEN NETWORK OF VERTICES AND LINES.
; DEFINES VERTEX OBJECTS AND BINDS THEM TO THEIR NAMES.
; CALLS ARE OF THE FORM: (DEFINE-VERTEX <NAME> <TYPE> <PAIR>*)
(DEFUN DEFINE-VERTEX @FEXPR (L)
  (: (PUT* (CAR ?L)
          (APPLY 'NEW (CDR ?L)))
```

```
      (PUT* 'NAME (CAR ?L)))
  (SETQ VERTEX-LIST (NCONC ?VERTEX-LIST (LIST (CAR ?L))))
  (CAR ?L))

(SETQ VERTEX-LIST NIL)

; DEFINES LINE OBJECTS AND BINDS THEM TO THEIR NAMES.
; CALLS ARE OF THE FORM: (DEFINE-LINE <NAME> <PAIR>*)
(DEFUN DEFINE-LINE @FEXPR (L)
  (: (PUT* (CAR ?L)
           (APPLY 'NEW (APPEND* '(LINE) (CDR ?L))))
     (PUT* 'NAME (CAR ?L)))
  (SETQ LINE-LIST (NCONC ?LINE-LIST (LIST (CAR ?L))))
  (CAR ?L))

(SETQ LINE-LIST NIL)

;GENERATES ITEMS FROM A LIST.
(DEFUN LGEN QEXPR <!:LIST !¬EL>
  (MAPC '(LAMBDA (N) (SETQ EL ?N) (POST)) ?LIST)
  (FAIL))


; RETURNS THE NEIGHBOR VERTEX OF SOME VERTEX NAME GIVEN A LINE.
(DEFUN NEIGHBOR-VERTEX @EXPR (LINE ME)
  (: (AVAL ?LINE)
     (COND ((EQ V1 ?ME) (AVAL V2))
           ((EQ V2 ?ME) (AVAL V1))
           ((BREAK '"***ERROR: LINE DOES NOT CONTAIN THIS VERTEX"
              ?ME)))))

; RETURNS EPSILON IF THE ABSOLUTE SUM OF ITS 1ST ARG IS LESS THAN
;   ITS 2ND THRU LAST ARG ELSE IT RETURNS NIL.
; CALLS ARE OF THE FORM: (EPSILON <EPS> <NUM>*)
(DEFUN EPSILON N
  (PROG (TALLY)
    (SETQ TALLY 0)
    LOOP
    (COND ((NEQ N 1) (SETQ TALLY (ADD TALLY (ARG N)) N (SUB1 N))
                     (GO LOOP))
          ((LESSP (ABS TALLY) (ARG 1)) TALLY))))

;PRINTS COMMENTARY.
(DEFUN COMMENTARY @FEXPR (LCOM)
  (OR ?COMMENTARY-SWITCH (RETURN NIL 'COMMENTARY))
  (TERPRI)
  (MAPC '(LAMBDA (COM) (PRIN1 (AVAL ?COM))) ?LCOM)
  (TERPRI))

(SETQ COMMENTARY-SWITCH T)
```

```
; DATA FOR THIS PROBLEM:
;
(DEFINE-VERTEX VERTEX-1
   ARROW-VERTEX
   L1 'LINE-1-5
   L2 'LINE-1-6
   L3 'LINE-1-2
   ANGLE-L1-L2 60
   ANGLE-L2-L3 45)

(DEFINE-VERTEX VERTEX-2
   L-VERTEX
   L1 'LINE-1-2
   L2 'LINE-2-3
   ANGLE-L1-L2 135)

(DEFINE-VERTEX VERTEX-3
   ARROW-VERTEX
   L1 'LINE-2-3
   L2 'LINE-3-6
   L3 'LINE-3-4
   ANGLE-L1-L2 45
   ANGLE-L2-L3 30)

(DEFINE-VERTEX VERTEX-4
   ARROW-VERTEX
   L1 'LINE-3-4
   L2 'LINE-4-6
   L3 'LINE-4-5
   ANGLE-L1-L2 40
   ANGLE-L2-L3 60)

(DEFINE-VERTEX VERTEX-5
   L-VERTEX
   L1 'LINE-4-5
   L2 'LINE-1-5
   ANGLE-L1-L2 120)

(DEFINE-VERTEX VERTEX-6
   FORK-VERTEX
   L1 'LINE-1-6
   L2 'LINE-4-6
   L3 'LINE-3-6
   ANGLE-L1-L2 120
   ANGLE-L2-L3 110)

(DEFINE-LINE LINE-1-5
   V1 'VERTEX-1
   V2 'VERTEX-5
   LENGTH 28)
```

```
(DEFINE-LINE LINE-1-2
   V1 'VERTEX-1
   V2 'VERTEX-2
   LENGTH 35)

(DEFINE-LINE LINE-2-3
   V1 'VERTEX-2
   V2 'VERTEX-3
   LENGTH 26)

(DEFINE-LINE LINE-3-4
   V1 'VERTEX-3
   V2 'VERTEX-4
   LENGTH 52)

(DEFINE-LINE LINE-4-5
   V1 'VERTEX-4
   V2 'VERTEX-5
   LENGTH 25)

(DEFINE-LINE LINE-4-6
   V1 'VERTEX-4
   V2 'VERTEX-6
   LENGTH 28)

(DEFINE-LINE LINE-3-6
   V1 'VERTEX-3
   V2 'VERTEX-6
   LENGTH 35)

(DEFINE-LINE LINE-1-6
   V1 'VERTEX-1
   V2 'VERTEX-6
   LENGTH 25)

; END OF DATA.
```