COMPUTER SYNTHESIS OF LINE DRAWINGS USING SEMANTIC NETS

By

Raymond D. Giustini

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Engineering

Department of Electrical Engineering,

McGill University,

Montreal, Quebec

August, 1975.

COMPUTER SYNTHESIS OF LINE DRAWINGS USING SEMANTIC NETS

Raymond D. Giustini

Abstract

This thesis describes a program which employs a semantic memory for the generation of simple line drawings. Input to the program is by means of a Picture Language (PL) whose syntax is also defined in this thesis. The semantic memory used is modeled after Quillian's. Compilation of a PL input by the program results in the creation of a Short-Term Memory (STM) which partially defines the objects to be drawn. Any missing information which is necessary to draw the objects is supplied by the semantic memory.

A number of examples which demonstrate some features and shortcomings of the system are described. The results of these examples, together with proposed changes which would improve the versatility and efficiency of the program, demonstrate the feasibility of incorporating a semantic memory into the software of an interactive graphics display system.

SYNTHÈSE DE DESSINS GRAPHIQUES PAR L'EMPLOIE DE LA MEMOIRE SEMANTIQUE

Raymond D. Giustini

Résumé

Cette thèse décrit la programmation faisant emploi de la mémoire sémantique pour générer des tracés simples de dessins. La programmation est controllée au moyen d'un langage spécial dont la syntaxe est aussi définie dans cette thèse. La mémoire sémantique employée est basée sur le principe de Quillian. La compilation du langage spécial par la programmation résulte en la création d'une mémoire à court terme, laquelle définit partiellement les objets qui doivent être tracés. Toute autre information non transmise à l'entrée mais nécessaire à la génération du tracé est continuée dans la mémoire sémantique.

Les exemples démontrent les characteristiques et les déficiences du système y sont décrits. Le résult de ces exemples et les changements proposés pour améliorer la versatilité et le rendement de la programmation, démontrent la possibilité d'incorporer une mémoire sémantique dans le logiciel d'un système à interaction pour la génération de graphiques sur écran.

ACKNOWLEDGEMENTS

The author is indebted to his thesis supervisor, Dr. M.D. Levine, for providing him with the opportunity of doing this work, for his guidance throughout the project, and especially for his comments and corrections to preliminary drafts of this thesis. Without his kind help and cooperation, the completion of this project would not have been possible. Dr. A. Malowany, who acted as the author's councillor during Dr. Levine's sabbatical, provided numerous comments and suggestions which influenced the scope of the work. Appreciation is also extended to Mr. C. Durand who translated the abstract into French and to Miss Irene Lewicky who typed the manuscript.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Overview

A semantic memory is a data structure in which factual assertions representing a computer's knowledge of the universe are stored. This thesis describes a procedure whereby a semantic memory and an input Picture Language (PL) are used by a compiler to synthesize line drawings.

It must be stressed at the outset that the purpose of this thesis is not to present a practical graphics language. Rather, its purpose is to formulate the structure of a semantic memory which could be incorporated into a graphics system and to demonstrate the inherent advantages of such a memory. The following is an outline of the topics which will be discussed.

The first chapter summarizes pertinent previous work in picture synthesis (computer graphics) and pattern recognition. Although this thesis is concerned with the former, the latter is discussed for two reasons. First, it shares many concepts with picture synthesis. Indeed, some previous work concerns systems which can be used for both pattern recognition and picture synthesis. Second, a discussion of some developments in pattern recognition techniques yields an insight into some of the concepts in-

volved in this thesis. The brief summary of research in semantic memories in the latter part of the chapter is especially relevant.

The second chapter describes the author's semantic memory and how its concepts and structure compare with those of earlier models. It also discusses the feature set and primitives of objects which comprise the line drawings and formalizes the syntax of the PL.

The third chapter discusses the operation of the compiler and the way in which it interacts with the PL and the semantic memory to produce line drawings. The features of the present implementation of the compiler are dealt with as are its shortcomings.

The fourth chapter deals with the results obtained from some picture synthesis problems posed to the semantic memory in the PL. Each problem is chosen to demonstrate certain aspects of the operation of the program. The chapter also discusses improvements which could be implemented on the semantic memory and compiler to eliminate some of their shortcomings.

1.2 Picture Synthesis

The following is a brief summary of the historical development of computer graphics software systems. This summary is restricted to systems which deal with the synthesis of line drawings representing two-dimensional objects. The synthesis of grey-level images, the two-dimensional rendering of three-dimensional solids and related areas of research are not considered here because these have no bearing on the scope of the work discussed in this thesis.

1.2.1 Early Graphics Systems

In recent years an abundance of hardware has been developed for the graphical input-output (I/O) of information. This hardware includes CRT displays, light pens, joysticks and tablets (for a description of these devices see Poore et al. (1969), and also Newman and Sproull (1973)). Such hardware packages often contain supporting software for the generation of primitive objects such as points, lines and arcs. Unfortunately, this is inadequate for the generation, modification and storage of complex pictures on an interactive basis.

Early attempts to circumvent this difficulty have involved the extension of programming languages such as FORTRAN and ALGOL to include an interactive graphics handling capability. Languages resulting from such an approach have found applications in the

generation of patterns composed of a number of "template" sub-patterns. Examples of such applications are the generation of integrated-circuit masks, circuit diagrams and architectural drawings. The visible drawback of these languages is that they do not possess the facility for storing a structural description of the patterns they create. This shortcoming restricts their applicability to the generation and modification of pictures having an unnecessarily rigid description of their subcomponents and constrains the user to low-level methods of picture synthesis. For example, the crudest method of picture representation involves using an array to store a digitized picture point by point. Such a representation is too cumbersome to modify because it does not explicitly define the elements of the picture and their interrelationships. The Graphical Data Structure (GDS) defined in the next section constitutes an attempt to surmount this difficulty.

1.2.2 The Graphical Data Structure

A desirable feature of a graphics language is its ability to easily modify the components of a picture by modifying a structural description of it. This can be accomplished by incorporating a Graphical Data Structure (GDS) into the language. Such a structure is composed of interconnected lists which consist of strings of nodes. These nodes are structures in the PL/1 sense[1] and contain

---

[1] See footnote on next page.

information about the picture components and their interrelationships. Pioneering work on the specification of GDS's was done with the SKETCHPAD system (Sutherland, 1963). In SKETCHPAD, the defined GDS is called a ring structure because the last node of each list is connected to its first node. To speed accessing of information, the nodes contain forward and backward pointers to their nearest neighbors in the list. A picture represented by a SKETCHPAD-GDS can be altered by the insertion or deletion of nodes or by altering the information content of the nodes. Because of their compact nature, many such ring structures can be stored in the secondary storage facilities of a computer.

A modification of the SKETCHPAD ring structures is defined in CORAL (Class Oriented Ring Associated Language) (Roberts, 1964). The nodes in a CORAL-GDS contain a forward pointer as in SKETCHPAD, but the backward pointer of every alternate node is replaced by a pointer to the header node of the list.

It is possible to define many variants of the GDS depending on the types of problems with which it must interact. The GDS defined by the author in this thesis is called a Short-Term Memory (STM). A discussion of its structure and capabilities is deferred until

---

[1] PL/1 structure: a tree whose leaves (terminal elements) represent variables. Specification of these variables is by name and type (e.g. pointer, arithmetic real, character string, etc).

Chapter 3,

As is true with any system, the GDS has advantages and disadvantages. One of its favorable aspects is its high information density. Furthermore, the manipulation of pictures represented by a GDS is at a higher level than that of pictures not represented by a GDS.

A disadvantage is that the information tends to be "buried" in the structure and subsequently can take appreciable time to access. Indeed, accessing and modification of information may require rather complex manipulation procedures. However, it is generally agreed that the advantages of a GDS greatly outweigh its disadvantages. The next section discusses some graphics languages which employ a GDS.

1.2.3 Graphics Languages Which Employ a GDS

A requirement for a graphics language-GDS combination is that the language must be capable of handling the GDS in addition to controlling the I/O peripherals. Kulsrud (1968) discusses some requisite features of such a language and presents his own version. He also proposes the use of a metacompiler or a compiler-compiler to generate an object program from the source language input. The metacompiler accepts inputs representing the syntax and semantics of the graphics language and generates a compiler for the language.

Although Kulsrud's language presents improvements over languages which do not employ a GDS, it still suffers some of their

shortcomings, but to a lesser extent. The problem is that the interaction between man and computer is still at a low level.

In general, this language and those like it are essentially extensions of programming languages with three added features. First, they possess a subroutine package for the implementation of graphical operations. Second, they can interact with the additional I/O devices associated with the graphics. Third, they are capable of handling the GDS (albeit at a low level). Unfortunately, the subroutine calls required to implement these features often involve the specification of lengthy parameter lists. Such cumbersome procedures constrain the solution of problems to an unnecessarily low level and increase the possibility of programmer errors.

An attempt to rectify some of the problems associated with the abovementioned languages is provided by GPL/1 (Smith, 1971) which is an extension of PL/1. Smith chooses PL/1 because of its built-in ability to generate and act on interrupts (a feature useful for handling I/O devices), because of its many data types and structures, and because of its list processing capability. GPL/1 is a higher-level language than those already discussed because it manipulates the GDS in such a way as to mask its existence from the programmer. The GPL/1 language is of particular interest to the author because the compiler presented in this thesis was implemented in PL/1. The compilation of the author's PL by GPL/1

would facilitate the interaction of the PL with a graphics system, a capability which it does not now possess.

Lecarne (1971) defines a graphics extension of FORTRAN called EUPHEMIE. Its advantage is that the subroutine calls with lengthy parameter lists are replaced by graphics instructions which generate the subroutine calls. The syntax of these instructions specifies most of the parameters associated with the subroutines. The remaining parameters (e.g. numerical coordinates) are specified by the position of a light pen on a CRT display. This effective use of the keyboard-light pen combination produces a high level interactive graphics language.

A problem associated with GDS's is that for all but the most simple pictures their storage necessitates the usage of large segments of computer memory. French and Teger (1972) have developed GOLD (Graphical On-Line Design System) which uses GDS's but partitions them for secondary disk storage so that the graphics system can be operated from a minicomputer. The obvious advantage of such a system is that it allows the implementation of high-level graphics languages in relatively small computer installations with but a small increase in processing time of programs. French and Teger have applied GOLD to the design of integrated-circuit masks.

Stack and Walker (1971) have proposed a sophisticated multi-terminal graphics display system called AIDS (Advanced Interactive

Display System) whose GDS is similar to Bellgraph's (Christensen and Pinson, 1967). The AIDS language retains the algebraic features of FORTRAN and appends to them graphical and interactive capabilities. In brief, objects in pictures are defined by trees whose elements are images, instances, sets and labels. Images are the points, vectors or characters which define the object. Instances are specific occurrences of an image. Sets are collections of instances, and labels are non-graphical data associated with a graphical element. These trees comprise the GDS's of AIDS.

A basic difference between this language and previous ones is that the execution of an AIDS program is described by a sequence of states and their transitions. During each transition between states, any number of operations which alter subsequent execution of the program may be performed.

More recent work in the specification of graphics languages which employ a GDS has been performed by Gonzales and Vidal (1975), and Williams and Giddings (1975).

A shortcoming of languages like AIDS is that subpatterns are still essentially specified by templates. Such an approach to picture synthesis is useful where fixed patterns recur frequently, but is too rigid for general line drawings. The PL described in this thesis constitutes an attempt to surmount this difficulty by using a semantic memory in conjunction with the GDS.

This section completes the historical discussion of computer graphics. The following sections will contain a short history of pattern recognition techniques.

1.3 Pattern Recognition

A brief historical review of some aspects of the field of pattern recognition or picture analysis now follows. While some of the techniques discussed do not deal primarily with the recognition of line drawings, their description is included in order to illustrate both the features and the shortcomings of techniques pertinent to this thesis. The development of the field is demonstrated by a discussion of the classification model, the descriptive model, and the semantic memory model.

1.3.1 The Classification Model

The classification model represents a procedure by which pictures can be classified as instances of objects from a given allowable set. Its objective is to minimize the probability of error for picture classification while also minimizing the processing time required.

Evans (1968, 1969) discusses the classification model (Fig. 1-1). Its operation can be described as follows. The transducer acts as an interface between the input picture and the feature extractor. It may consist of a scanning device and a software or hardware filter which alters the input to a form suitable for further processing. The output of this stage is passed on to the feature extractor which uses property filters $p_1,\ldots,p_n$ to obtain the feature vector $[f_1\ f_2\ \ldots\ f_n]^T$ from the (preprocessed) input.

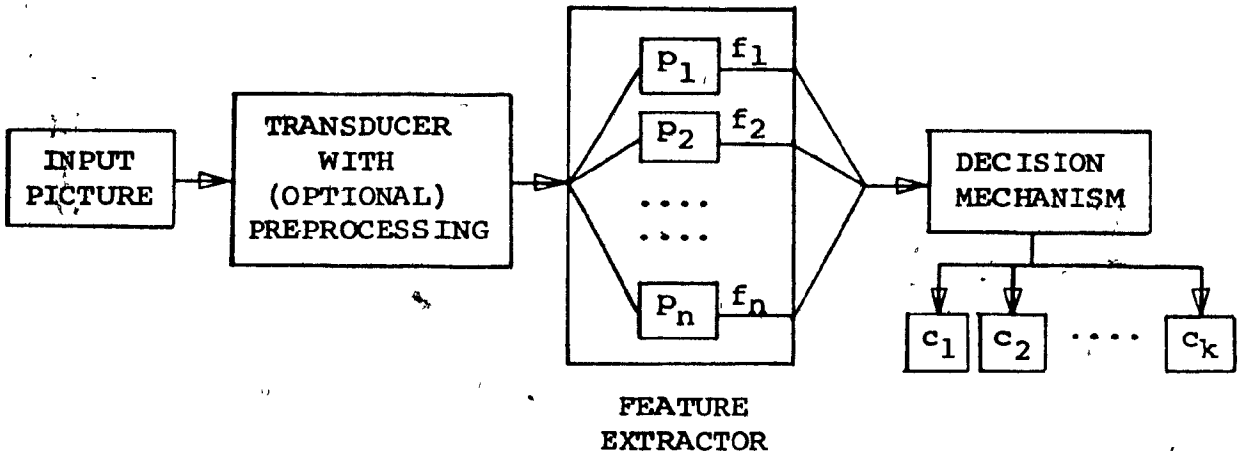


Fig. 1-1: The Classification Model.
$p_1,\ldots,p_n$ are property filters while $c_1,\ldots,c_k$ are possible classifications of the input picture.

The decision mechanism then maps the vector into the n-dimensional feature space and, in accordance with a previous partitioning of the space, classifies the input as a member of the set of possible alternatives $c_1,\ldots,c_k$. Fig. 1-2 shows the feature space with a partitioning scheme for the case of two property filters and three alternative classifications (i.e. n=2, k=3).

According to Duda and Hart (1973), the distinction between the concept of a feature extractor and that of a decision mechanism is fluid. A sufficiently powerful version of one renders the task of the other trivial. However, it is obvious from the nature of the feature extractor that its operation is more problem-dependent than that of the decision mechanism. The former may vary

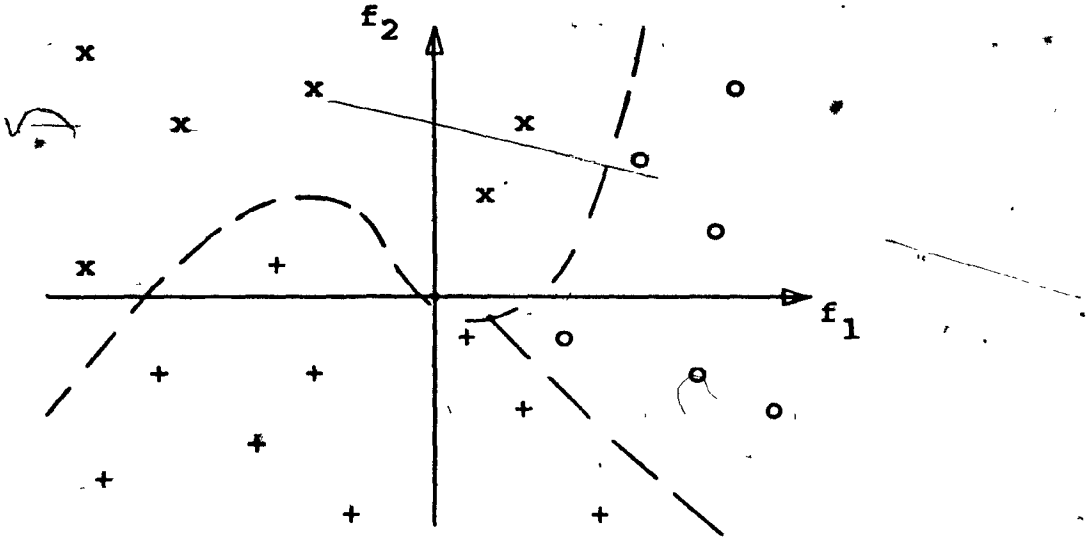


Fig. 1-2: A pictorial representation of a two-dimensional partitioned feature space in which three types of inputs (x, o and +) are defined.

for each type of input and for each feature while the latter can be a general algorithm to partition the n-dimensional feature space into k regions and then to classify an input according to the value of its feature vector.

As was stated earlier, an objective of the classification model is to minimize the probability of error in pattern classification by choosing optimal partitions of the feature space. This model has been successful in the analysis of simple pictures such as alphanumeric characters.

Unfortunately, the model has several limitations. One is that its operation does not parallel the way that humans assimilate and handle pictures. Therefore, the model is not aesthetically pleasing; its methodology appears highly artificial to a human user.

A more severe limitation of the model becomes obvious with increasing complexity of input pictures. If the probability of error in classification is to remain acceptable for these complex pictures, an increasingly large feature set as well as more complex property filters and decision mechanisms become necessary. As this feature set grows, the model becomes increasingly cumbersome. Consequently, it becomes necessary to question the wisdom of arbitrarily attempting to force the information content of a complex picture into a rigid feature vector description.

An equally severe limitation of the classification model is that it is oriented toward a single task. Specifically, it does not consider the structural content of pictures. This limitation severely limits its versatility in the performance of other tasks. For example, if such a model is to be incorporated into a question-answering system, then the questions which may be asked of it must be of a limited nature (e.g. What object is this? or, What is the value of its feature vector?). If, for example, the picture contains several triangles, it is not possible to ask the model to find all the triangles unless that number is represented by a member of the feature vector.

What is needed, then, is a more general approach to picture analysis which incorporates the structural elements of a picture in its operation. The descriptive approach discussed next fulfils this objective.

1.3.2 The Descriptive Model

As its name implies, the descriptive model entails the description of the structure of a scene in terms of its primitives (or some more complex subcomponents of the picture) and the geometrical relations between them. The details of such a description are dictated by the nature of the problem.

Firschein and Fischler (1971, 1972) have made a strong case for the use of picture-description techniques in question-answering systems. Referring to Fig. 1-3, it should be noted that the crucial step in the implementation of a workable descriptive scheme is the preparation of the picture description. Specifically, the picture representation in symbolic form must be amenable to formal use in the question-answering system.

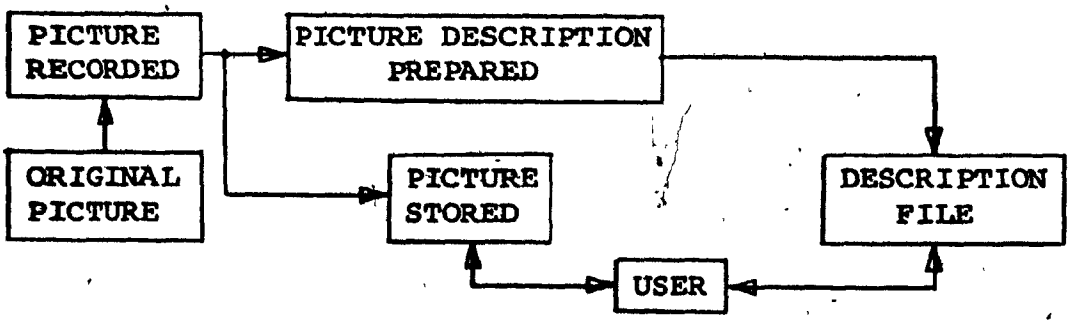


Fig. 1-3: Question-Answering System using a Picture Description File. Picture taken from Firschein and Fischler (1971).

Subsequent to the pioneering work of Grimsdale (1959), Kirsch (1964) and others, the descriptive approach has largely replaced the classification model in complex picture analysis problems.

Furthermore, as shall soon become apparent, a suitably designed descriptive scheme is useful for both picture analysis and picture synthesis. In the pages that follow three descriptive models will be discussed: the grammar-based model, the descriptor-based model and the procedure-based model. In addition, the advantages and drawbacks of these models will be considered.

1.3.2.1 Grammar-Based Models

Because of the conceptual parallels between some early implementations of the descriptive model and the theory of formal languages, these implementations have been called grammar-based (also syntax-based and linguistic). In such schemes the picture, viewed as a statement in the picture language, is translated into a linear string which describes the picture. The grammar of the language then parses the string yielding the recognition of the picture. This parsing is accomplished using a set of rewriting rules which constitute the grammar. Note that the parse of the string is obtained when the grammar is used in the analytic mode. When the generative mode (picture synthesis) is employed, new statements (i.e. pictures) are synthesized using existing primitives and rewriting rules.

Early attempts to implement linguistic methods are exemplified by the work of Grimsdale et al. (1959). Their research involves obtaining descriptions of simple hand-drawn line figures.

These descriptions consist of the primitive curves in the figures, their properties, and their connectivity.

An interesting approach is that of Kirsch (1964). He proposes a two-dimensional grammar instead of one which produces linear string representations of pictures. As an example, he considers the class of forty-five degree right-angled triangles. His grammar consists of an alphabet and ten rewriting rules. Elements of the alphabet are line segments and are represented by capital letters. For example, the three rules shown in Fig. 1-4 can be used to describe the hypotenuse of this class of triangles. Unfortunately, Kirsch's grammar appears to have severe limitations: for more complex objects, the rules and alphabet must undergo radical expansion and would probably become too cumbersome to be practical.

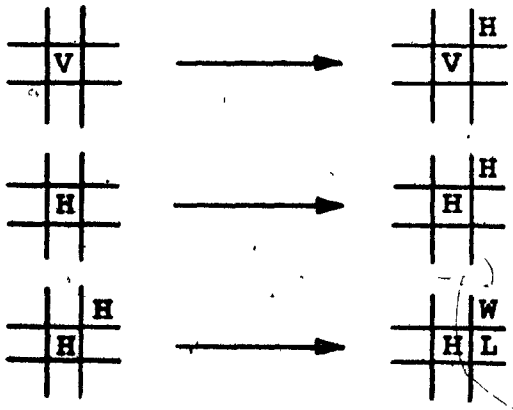


Fig. 1-4: Three of Kirsch's rewriting rules for the description of the hypotenuse of a 45° right-angled triangle. V and W are two vertices, H is a hypotenuse segment, and L is a vertical line segment.

More recently, Pfaltz (1972) and Rosenfeld (1973, 1975) have proposed the usage of multidimensional array grammars (or "web" grammars) for the representation of pictures. The elements of these versatile grammars are sets of arrays or graphs rather than linear strings. It can be seen that the language proposed by Kirsch in the previous paragraph defines a primitive array grammar.

Breeding (1965) and Amoss and Breeding (1970a, 1970b) have proposed the PADEL (Pattern Description Language) language for both the generation and synthesis of line drawings. In PADEL, an object is defined by a set of nodes and paths. A path is a set of line segments joined end-to-end while a node is any point where three line segments meet. Unfortunately, picture descriptions generated by PADEL are not unique for all but the simplest line drawings. This, together with the highly artificial nature of the descriptions generated by PADEL, appear to be its major drawbacks.

Among the more versatile schemes which have been developed, Shaw's syntax-based Picture Description Language (PDL) (1968a, 1968b, 1969, 1972), is suitable for the generation and analysis of line drawings. The primitives of PDL are two-dimensional objects having both a head and a tail. Concatenation of the primitives may occur only at these endpoints. For example, if the + operator concatenates head to tail, then ↗ + ↘ is equivalent to ↗↘ .

In such a language, a picture becomes a directed graph with primitives forming the directed edges.

The work of other notable proponents in the field such as Fu and Rosenfeld is referenced in the bibliography. For a somewhat dated but nevertheless excellent survey paper, see Anderson (1971).

Grammar-based description schemes have some seribus disadvantages, however. First, there is the rigidity and artificiality of these approaches. Although to a lesser extent than the classification model, the methods by which these schemes synthesize and analyze pictures are in contrast to the methods by which humans accomplish these tasks. The rigidity of existing schemes renders them unable to cope with grey-level pictures. Also, these approaches inherently contain a lack of semantic capability. That is, they do not possess knowledge of a universe to which a particular scene relates and which can facilitate its processing.

The models which will be discussed in the next pages represent attempts to circumvent these difficulties.

1.3.2.2 Descriptor-Based and Procedure-Based Models

Of the three categories of descriptive techniques considered in this thesis, the first which is a grammar-based model has already been discussed. This section briefly considers the remaining two techniques.

One of these is the class of descriptor-based models. The primitives of such descriptions are individual phrases which describe the picture. Firschein and Fischler (1971) discuss the applications of such schemes to still photography, movie film and videotape, and indexer aids. The problem with such schemes is to obtain a consistent terminology which can be used by a human for interaction on a question-answering basis.

The last category of descriptive techniques to be considered is the class of goal-directed or knowledge-based models. These consist of a system, either grammar-based or descriptor-based, which is capable of producing a large number of descriptions of a picture. The system also contains a control mechanism which selects a specific description according to the context in which the description is desired. Fischler (1969), who has used such an approach in the description of Sanskrit characters, concludes that it may be able to surmount the problems inherent in the grammar-based approach.

Preparata and Ray (1972) and Yakimovsky and Feldman (1973) have applied goal-oriented models to the computer recognition of color photographs. More recent work in the same vein has been performed by Bajcsy and Lieberman (1974). The procedure used by Yakimovsky and Feldman entails the generation of regions from a picture and the subsequent use of contextual information defined by relationships between neighboring regions to decide on what

the picture represents. The contextual information required is stored in a semantic memory (the concept of such a memory is discussed in the next section). The control mechanism is a probability function whose members are the probabilities that given regions can be assigned a defined interpretation given the results of the contextual information applied to the regions and the measurements on the features of the regions. The goal is to maximize the probability function which is the product of these members. It can be perceived from this example that the procedure-based model represents a powerful technique for the analysis of scenes by computer.

This section completes the brief survey of the development of the descriptive model for picture analysis. It has already been stated that a failing of both grammar-based and descriptor-based models is that they lack semantic capability. On the other hand, a knowledge-based model used in conjunction with a semantic memory can overcome these failings. The semantic memory model necessary for the implementation of such an approach is the next topic of discussion.

1.3.3 The Semantic Memory Model

Except for the procedure-based models, a shortcoming of the picture manipulation schemes described so far is that they do not interact with information at the same level that humans do. One

reason for this is that the structures they employ for information storage and retrieval are not as flexible and powerful as those employed by humans. What is therefore required is a memory capable of performing tasks by methods analogous to those used by a human memory.

Frijda (1972) has specified four properties of a human memory which must be possessed by an artificial intelligence if it is to be capable of human-like interaction with information. First, the memory structure must be associative. In such a structure, the input data which requests information is associated with an abbreviated map of the memory. A match between the input data and the map implies that the desired information can be localized to regions of the memory specified by the matching map entry. With each additional match of input data with map entries the location of the required information within the memory is further localized. Information may then be located and retrieved by entry and search of the appropriate portions of the memory structure. Second, the memory must be teachable; it must be capable of receiving information, of determining the truth or falseness of that information and of incorporating the information within itself if it is recognized as a truth. Third, the memory must be inferential; it must be able to deduce facts not directly contained within it by inference on facts which are. Finally, it should possess the ability to retrieve information given input data whose structure differs

from that by which the memory previously learned the information. Such a quality adds to the flexibility of the intelligence employing the memory structure.

The determination of the format of a memory structure suitable for information storage and retrieval as specified by the above requirements is an appropriate starting point in the realization of a system capable of displaying human-like intelligence. As a result, an increasing amount of research has been dedicated to the determination of the attributes of such a class of structures called semantic memory networks or semantic nets. The primitive elements of such structures are nodes and links. Each node defines a concept in the net while each link defines a relationship between two nodes.

The LEAP language developed by Rovner and Feldman (1969) is especially suitable for the manipulation of semantic nets. It defines operations on the relational structure (O A V) where O is an object, V is its value and A is the attribute linking the object to the value. For example, in the relational structure (SHOE COLOR BROWN) the object is SHOE, the value is BROWN and the attribute is COLOR. Each relational structure may itself be an element of another such structure as in (CLOTHING EXAMPLE (SHOE COLOR BROWN)).

In relating these ideas to those of a semantic net it can be seen that O and V are nodes (concepts) while A is a link (relation-

ship). The link may itself be considered as a concept in other relational structures as in (COLOR INTENSITY BRIGHT). In general, a relational structure as applied to a semantic net has the form (node link node).

The discussion has thus far centered on the concept of a semantic net. However, the net is only one of the two components of a semantic memory. The other component is the semantic map. This map is a list of all the concepts defined in the net together with the appropriate net entry points for each concept. It allows the semantic memory to fulfil the requirement that it be an associative structure.

Considerable work has been performed in the implementation of semantic nets in various applications. Quillian (1968, 1969a, 1969b) is the first to have applied them to the computer processing of English text. As defined by Quillian's net, a concept may be a word, a phrase or even a paragraph or more of text. Each concept is represented by a semantic plane. A semantic map called a dictionary contains a list of all the concepts which are defined by the net. Each dictionary entry contains pointers to at least one plane in the net which defines that entry. For example, the word PLANT may have three pointers from the dictionary to the net corresponding to three different meanings of the word (i.e. (1) the verb to plant, (2) a living plant, or (3) an industrial plant). The method by which this is accomplished is shown on the next page

(Fig. 1-5).

In Quillian's semantic net, concepts are defined by superset modification. For example, a horse can be defined as a four-legged animal. In terms of the net structure, horse is the concept, animal is the superset and four-legged is the modifier. With reference to the LEAP relational structures, it can be seen that for this simple example the net contains information of the type (SUPERSET MODIFIER CONCEPT). Unfortunately, the structure of the net is far more complex than this as will be seen shortly. Thus it is difficult to model Quillian's net in terms of a sequence of nested relational structures. Instead, a general description of the net will be used to demonstrate some of its features.

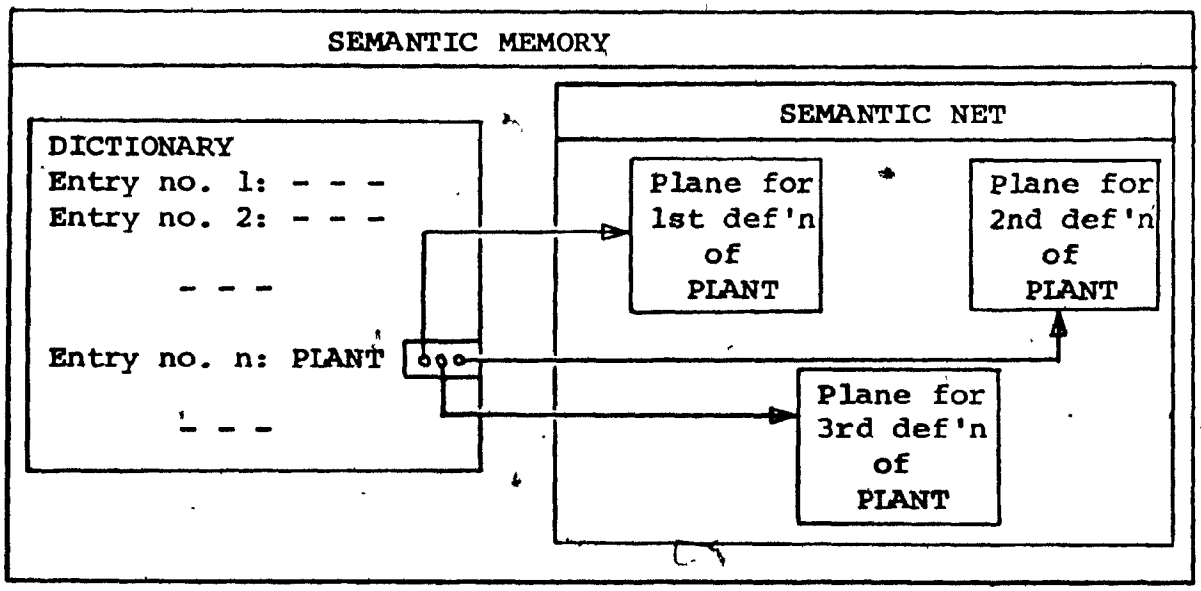


Fig. 1-5: The Semantic Memory showing links from the Dictionary to the Semantic Net for the definitions of PLANT.

Each plane of the semantic net consists of both unit and property nodes. These in turn contain pointers to other unit and property nodes both in their own semantic plane as well as in other semantic planes. A unit node represents a concept. Its first element is a pointer to the unit representing the concept's superset. The second and subsequent elements (as many as are desired) are pointers to property nodes in the semantic plane. Property nodes represent predication of concepts. The first element of a property node is a pointer to a unit node representing the property's attribute; the second element is a pointer to a unit node representing its value. These two elements represent an attribute-value pair. Subsequent elements, if any, are pointers to subproperty nodes of the property. The property node may thus represent a noun clause, an adverb clause, a true attribute-value pair such as (COLOR WHITE), or any other grammatical construct entailing predication.

From the above, it may be inferred that Quillian's semantic net is not a hierarchical structure; instead it is a general graph of interconnected nodes having no patriarchal node but rather many points of entry specified by the dictionary list. An example of the net's structure is given by Quillian (1969a) for the definition of "client" (Fig. 1-6).

One advantage of such a memory structure is that each concept

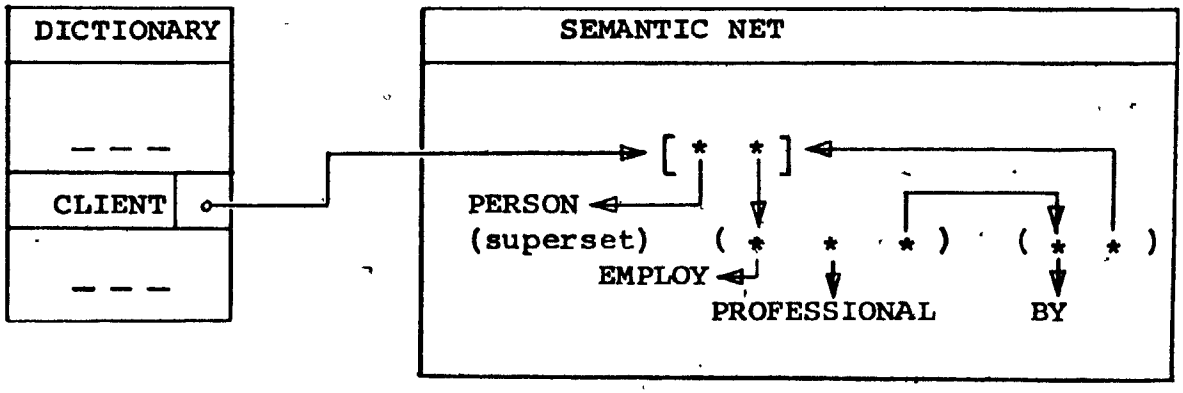


Fig. 1-6: The semantic plane for the concept CLIENT where [ ] is a unit node and ( ) is a property node. Pointers to words in block letters are pointers to planes defining these words.

need be defined only once. For example, assume that the concept of a triangle has been defined and one wishes to define an isosceles triangle. Fig. 1-7 illustrates the required semantic memory.

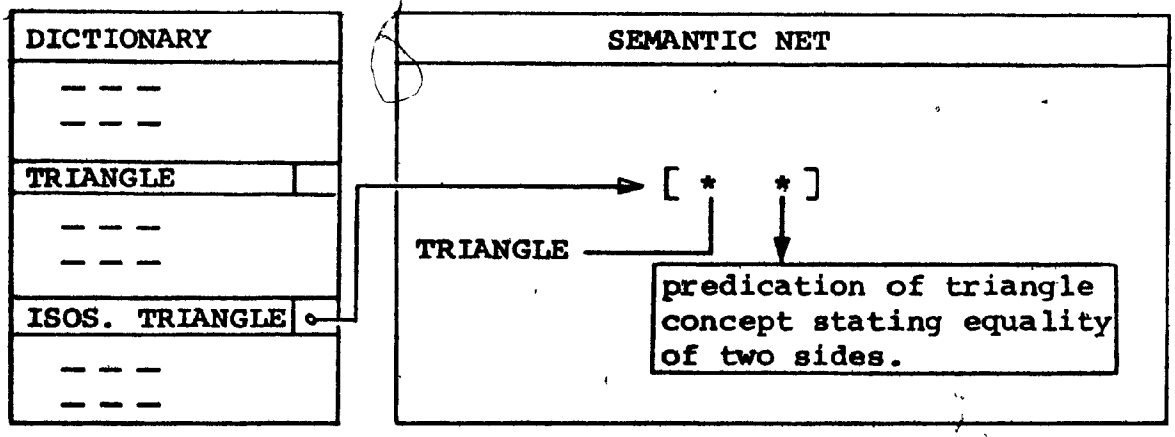


Fig. 1-7: Representation of the semantic plane for the concept of an isosceles triangle.

It is important to note that the extent of a concept is not limited to its semantic plane because each node in the plane contains pointers to concepts outside the plane. Indeed, a concept's full meaning consists of all the nodes accessible either directly or indirectly from its semantic plane. This feature permits the usage of the semantic net for the inference of the common meaning between two concepts. For example, let the definition of CRY be TO MAKE A SAD SOUND; let the definition of COMFORT be TO MAKE LESS SAD. The net can find the common meaning between these two concepts by searching outwards from the entry points specified by the dictionary until the search graphs intersect at a concept. In this case, the concept of intersection is SAD.

The above paragraphs have constituted a brief discussion of some of the features of Quillian's semantic memory structure. The rest of this section discusses some of the work performed on semantic nets for picture analysis. Preparata and Ray (1972) have defined a semantic net containing objects of a universe, binary relations and an algorithm which, under the guidance of the net, iterprets and categorizes simple color photographs. Their method, like that of Yakimovsky and Feldman (Section 1.3.2.2) is a knowledge-based descriptive model for scene analysis. In both cases, the semantic net is the control mechanism which determines the path of the solution.

Firschein and Fischler (1972) have also employed a semantic net for picture descriptions. Specifically, they have employed the descriptions of photographs by humans to construct networks in which concept kernels (phrases) and relations are represented by nodes and pointers respectively. Thus, from the description "it is an aerial photograph whose most arresting feature is a waterway, possibly an estuary or part of a harbour, with many ship docks" they obtain the network of Fig. 1-8.

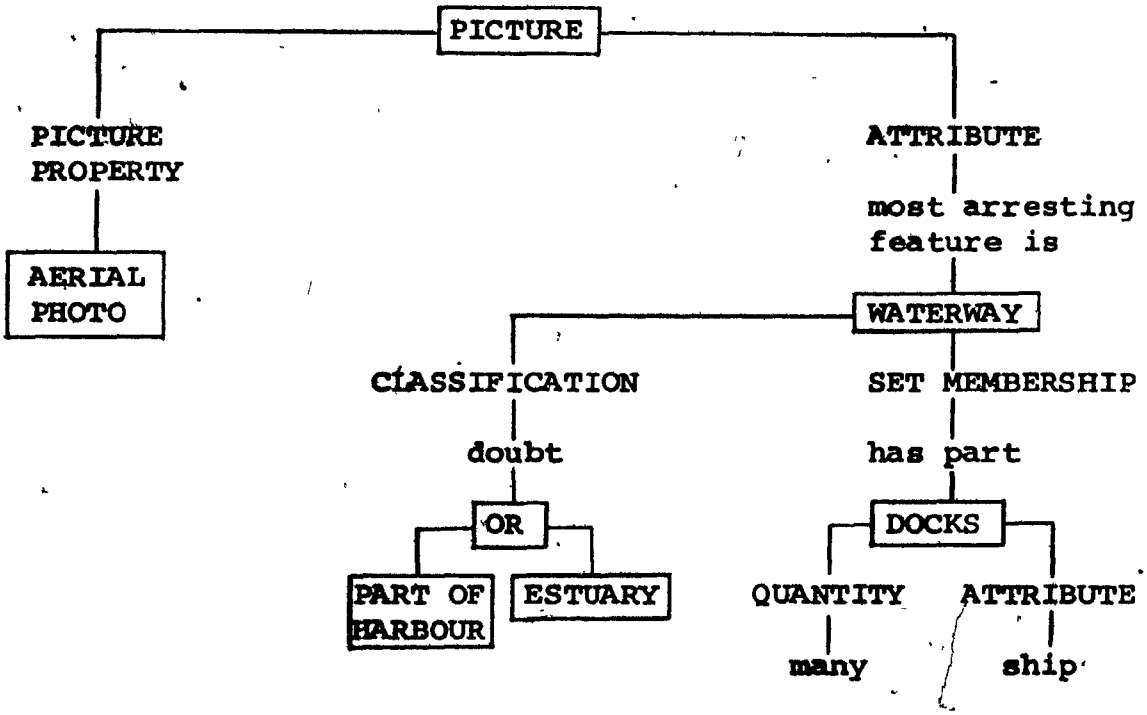


Fig. 1-8: A Semantic Network based on a Description.

This chapter has discussed the historical aspects of picture synthesis and pattern recognition. The concepts of a GDS and a semantic memory have been presented in some detail. The following

chapter will present the semantic memory developed by the author and Chapter 3 will discuss the Short-Term Memory (STM) and the associated compiler. In both chapters, comparisons will be made between these structures and those already discussed.

Chapter 2

The Semantic Memory and PL

2.1 Overview

The next two chapters will describe the system implemented by the author for the synthesis of line drawings. An overview of the method is now presented.

If the system did not possess a semantic memory, its structure could be modeled by the block diagram of Fig. 2-1. As such

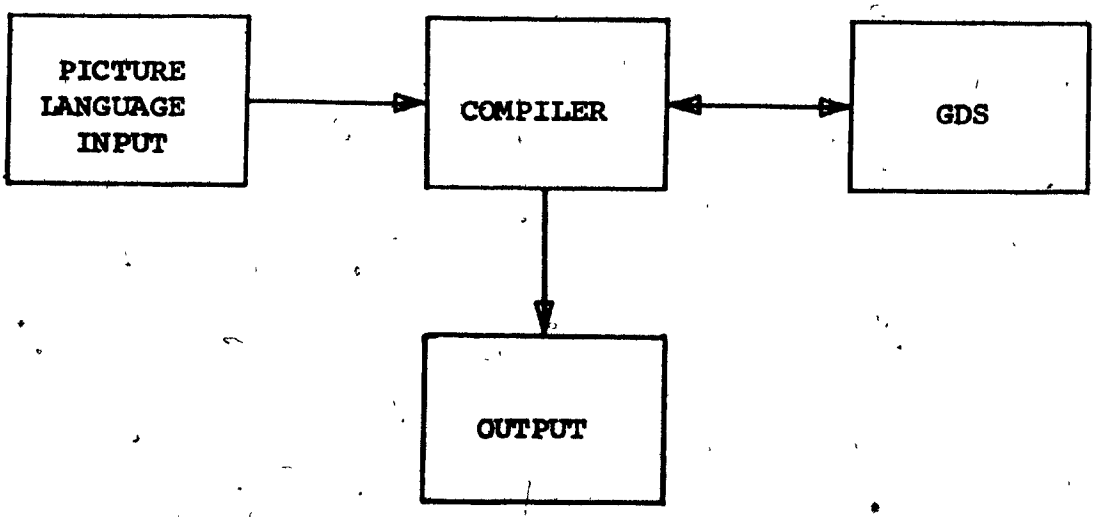


Fig. 2-1: Picture synthesis plan which does not employ a semantic memory.

the system would not be materially different from those described in the first part of Chapter 1. The compiler would accept statements in the picture language and would incorporate their information into the GDS using the language's syntax. When all the statements would have been processed the compiler would, in conjunction with the GDS, output the resulting pictures. With the addition of a semantic memory, the block diagram becomes that shown in Fig. 2-2. As was the case for the system depicted by Fig. 2-1, the compiler accepts statements in the picture language

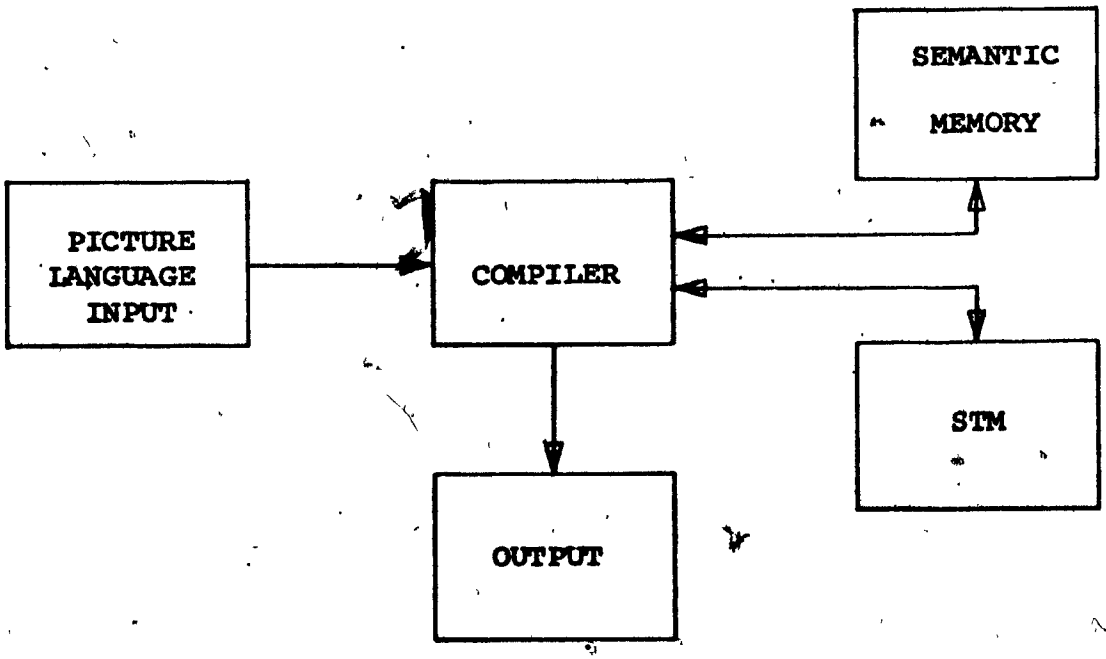


Fig. 2-2: Picture synthesis plan employing a semantic memory.

and processes them into the GDS (called the Short-Term Memory or STM) using the language's syntax. The difference with this method is that the picture language statements need not provide all the information about the objects to be drawn since that information resides within the semantic memory. Indeed, the existence of the semantic memory allows the programmer to specify as much or as little information as is desired about the objects to be drawn since any missing information is supplied by the memory.

In this scheme the picture language provides the syntax while the semantic memory provides the semantics. As will be seen in the rest of this thesis, it is this feature which allows a high level of interaction between man and computer. Another feature of interest is the nature of this model. It will be seen later in this chapter that it is descriptive because of the structure of the picture language. However, because the semantic memory "guides" the solution path for picture synthesis, it is also knowledge-based or goal-directed. It is therefore in the same category as the work of Preparata and Ray (1972) and Yakimovsky and Feldman (1973). The author's system differs from theirs in that it is primarily aimed at picture synthesis while theirs are employed in scene analysis.

This chapter discusses the structure of the semantic net and the syntax of the picture language. Chapter 3 will deal with the compiler and the STM.

2.2 Constraints on the Structure of the Semantic Memory

The semantic memory presented in this thesis is modeled after Quillian's. However, its primary task which is the synthesis of simple line drawings is different if not simpler than those which Quillian's memory must perform. It is therefore reasonable to assume that simplifications to Quillian's memory may be incorporated into the present implementation. The example which follows will demonstrate not only the possibility but also the necessity of such simplifications.

Consider the definition of a triangle. According to the Encyclopedia Britannica (1963), a triangle is "the geometrical figure composed of three points called the vertices (not lying in one straight line) and three straight lines joining these called the sides". Using the concepts of units and properties described in Section 1.3.3, a realization of the semantic plane for the triangle definition might be as shown in Fig. 2-3. Unfortunately, the linguistic definition of a triangle is not immediately useful for the synthesis of a drawing. The reason for this is that such concepts as GEOMETRICAL FIGURE, CONSISTING OF and NONCOLLINEAR have not yet been defined. Indeed, their definition would be in terms of other undefined concepts and so on. However rich and esthetically pleasing such a description of a triangle might eventually be, it does not immediately permit drawing the object.

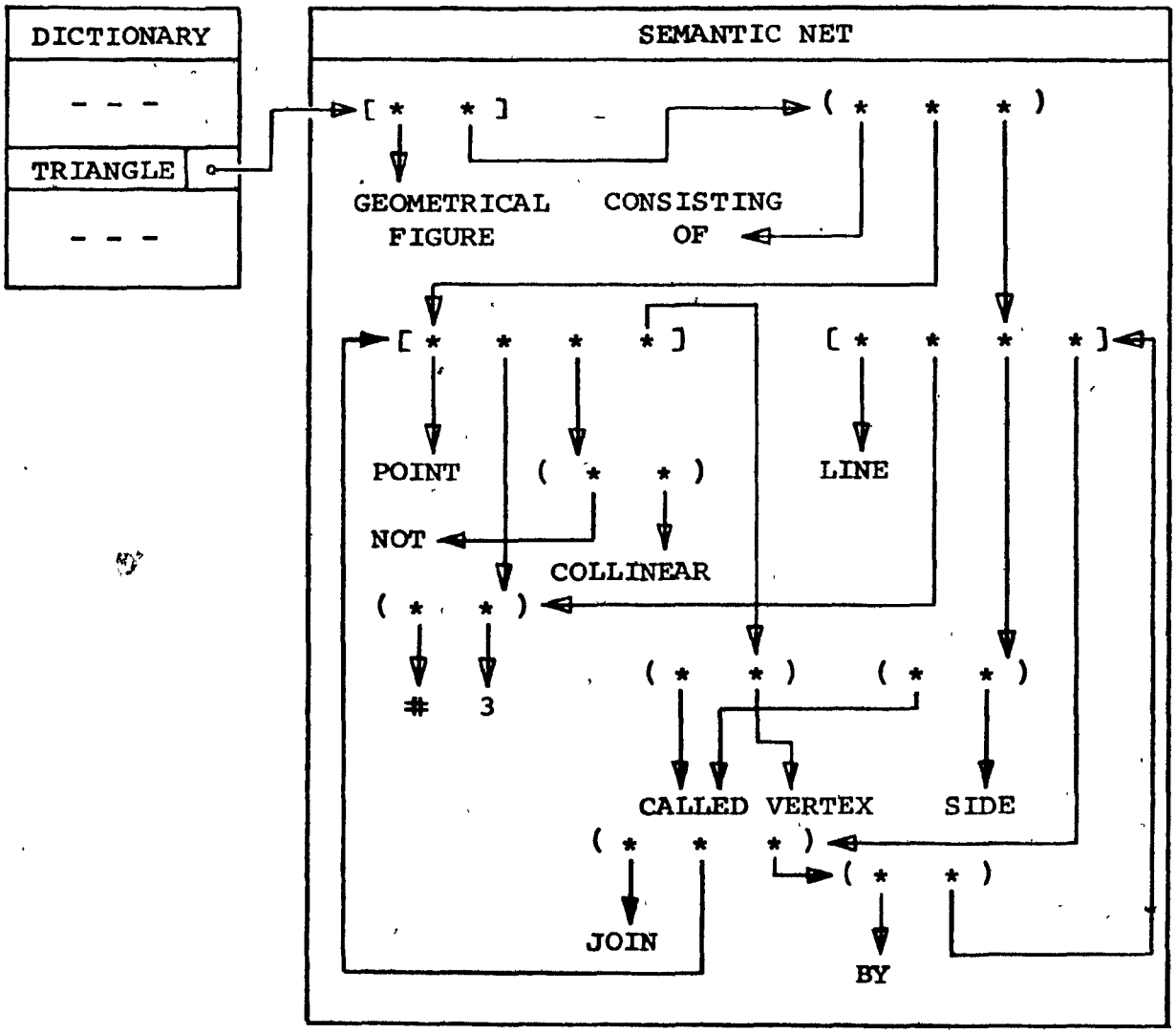


Fig. 2-3: A possible realization of the semantic plane for the concept of a triangle in Quillian's semantic net.
An English-like description of the above plane might be:

triangle: geometrical figure consisting of
point number three
point not collinear
point called vertex
line number three
line called side
line join point
join point by line

Certain simplifications have been incorporated into the definition of the plane in order to facilitate its understanding.

The solution to this problem is the creation of a semantic memory employing mathematical rather than linguistic descriptions of objects. Such mathematical descriptions involve the specification of objects in terms of their primitives and the relationships between them. It will be seen later in this chapter that the resulting structure is hierarchical, that is, it is a tree rather than a network. While it is a practical structure for the tasks it must perform, it is more restricted and therefore less versatile than Quillian's. Specifically, it can only "understand" a limited set of English words while Quillian's could theoretically be built up to encompass the entire English language.

This restriction is not as serious as it might at first appear to be. The semantic memory may be thought of as part of a larger memory which contains linguistic definitions of concepts. These two memories would be interlinked. Whenever the linguistic memory would receive queries or commands which required computations of feature values of an object, the mathematical memory would be accessed.

Given the restrictions on the capabilities of the semantic memory, it is now possible to describe the syntax of the picture language. This is the topic of the next section. A discussion of the structure of the semantic memory is deferred until later in this chapter.

2.3 The Picture Language (PL)

Before a formal definition of the syntax of the PL can be given, it is necessary to define both the primitives and the feature set of the geometrical objects considered in this thesis.

2.3.1 The Primitives

While a versatile set of primitives for the synthesis of line drawings consists of a point, a straight line and a conic section, it was decided at the outset that the primitive of the present realization would be the straight line only. Such a decision is of considerable import because it affects both the syntax of the PL and the structure of the memory. It was felt that this choice would allow for sufficient versatility to demonstrate the concepts involved in the semantic memory while reducing the programming of such a system to manageable levels.

2.3.2 The Feature Set

The impact of the choice of a primitive set is immediately apparent on the resultant feature set necessary to completely describe the objects to be drawn. In the present framework a sufficient and in fact redundant set consists of size, location of the contour vertices and orientation. A redundant set has been chosen for the purpose of allowing sufficient flexibility in the system. The components of the feature set are now described.

The defintion of the size feature is dependent upon the topology of the object to be described. If a closed object is being described, size is defined by the perimeter of the object. If an open object is being described, size is the length of those lines which define its boundary (Fig. 2-4). A problem arises if

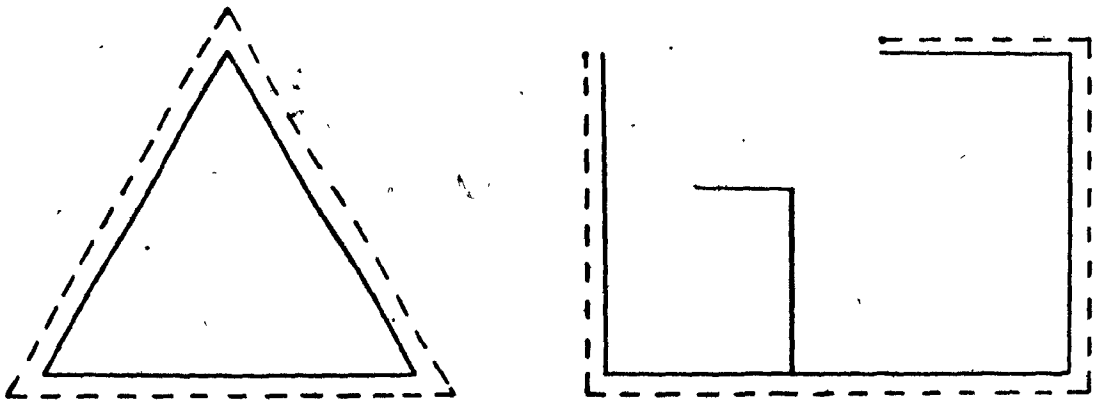


Fig. 2-4: Examples of the definition of size for a closed object (a) and an open object (b). In both cases the size is equal to the length of the lines paralleled by the broken line.

objects which are not simply connected are allowed (Fig. 2-5). For this reason, this implementation of the semantic memory is restricted to simply-connected objects. If one wished to incorporate multiply-connected objects, one could define another feature for lines which would determine whether they were visible or

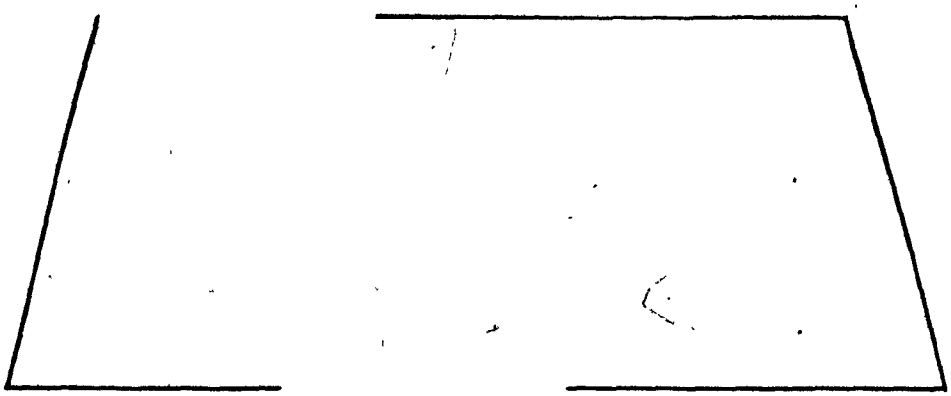


Fig. 2-5: An object which is not simply-connected and for which the present definition of size is not applicable. Such an object is not allowed in the present memory implementation.

invisible. Then if size were made applicable to both visible and invisible lines its present definition would remain valid (Fig. 2-6). It was felt that the addition of this capability would not sufficiently increase the versatility of the system to justify the additional programming effort required. For this reason, this feature was not incorporated into the system. The size feature is represented by the PL string SIZE.

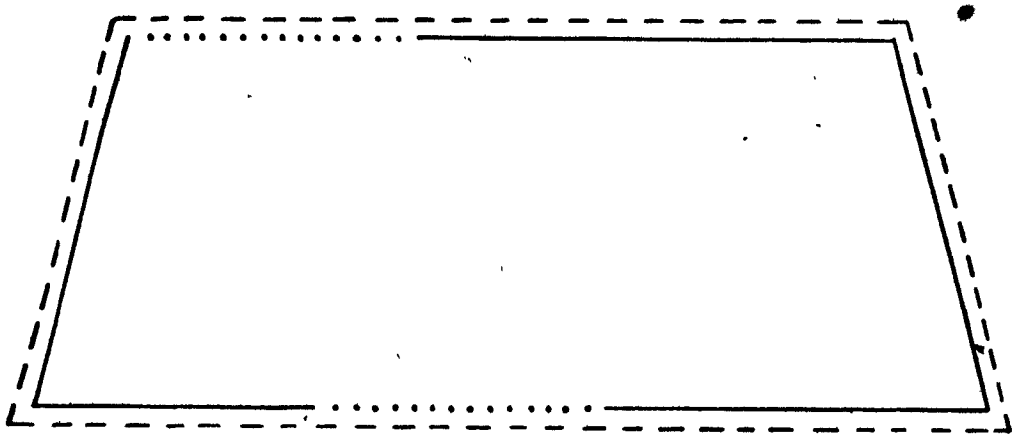


Fig. 2-6: The addition of invisible (dotted) lines renders the definition of size consistent.

The contour vertices of an object represent those points at the object's boundary where two or more lines meet. Because the semantic memory is defined for two-dimensional objects only, each contour vertex $\vec{V}$ is represented by two features which specify the X and Y coordinates of the vertex relative to some arbitrary frame of reference. The features of the i th contour vertex $\vec{V}_i$ are therefore represented in the PL by VXi and VYi. The contour vertices of an object are labeled sequentially in a counterclockwise

manner with the first vertex being the lowest left vertex of the object when it is in the horizontal orientation (Fig. 2-7). From the above discussion, it is obvious that the number of contour vertex features for a given object is twice the number of vertices since each vertex is described by two coordinates.

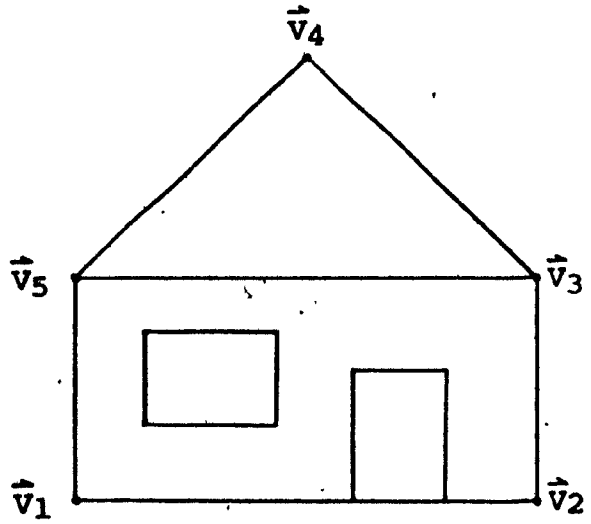


Fig. 2-7: The contour vertices of a house. In this case there are ten contour vertex features since there are five vertices each of which is described by two coordinates.

The orientation feature of an object is represented in the PL by ORIENT and is defined as the inclination to the horizontal of the line specified by the contour vertices $\vec{v}_1$ and $\vec{v}_2$ of that object. Orientation is measured in degrees. Its value increases as the directed line specified by $\vec{v}_1$ to $\vec{v}_2$ is rotated counterclockwise.

The features sufficient to describe the primitive of the PL (i.e. a line) are SIZE, ORIENT, VX1, VX2, VY1 and VY2. These features are common to all objects in the PL. More complex objects such as the house of Fig. 2-7 require the additional features VX3 and VY3, VX4 and VY4, ... which represent third and subsequent contour vertices. For reasons which will become apparent in Chapter 3, it is convenient to label the members of the former set as primary features while the members of the latter set are called secondary features.

This completes the definition of the feature set. The next section describes the syntax of the PL.

2.3.3 The PL Syntax

As was stated earlier in this chapter, input to the program is accomplished using the PL. Analogous to some previous work (Chapter 1), a linear string language similar to LISP has been chosen. However, because some of the information necessary to draw pictures is supplied by the semantic memory, the PL is simpler than those previously discussed. The statement types presently accepted by the language are the ⟨draw⟩ command and the ⟨logic⟩ and ⟨topol⟩ statements. These will now be defined.

The ⟨draw⟩ command specifies an object which is to be drawn by the program. The ⟨logic⟩ statement specifies predication on an object. In effect it yields information about the value of one of the object's features. The ⟨topol⟩ statement yields information about the topological relationship between two objects. The syntax of the PL in Backus-Normal Form is given below:

(1) ⟨numeral*⟩ ::= 1|2|3|4|5|6|7|8|9 [1]

(2) ⟨numeral⟩ ::= ⟨numeral*⟩|0

(3) ⟨integer⟩ ::= ⟨numeral⟩|⟨numeral⟩⟨integer⟩

(4) ⟨number⟩ ::= ⟨integer⟩{·|"}|{⟨integer⟩|"}·⟨integer⟩ [2]

---

[1] Except for the symbols "|" (the "or" delimiter), "{" and "}" ("{" and "}" are delimiting brackets), all strings not enclosed by the ⟨,⟩ brackets represent valid names, numbers or characters in the PL.

[2] The symbol " is the null string.

(5) ⟨object*⟩ ::= LINE|TRIANGLE|ISOSTRI|EQUILTRI|RECTANGLE|
SQUARE|GABLE|WINDOW|DOOR|FRAME|HOUSE [1]

(6) ⟨object⟩ ::= ⟨object*⟩{⟨numeral*⟩|"}{((⟨object⟩)|"} [2]

(7) ⟨feature*⟩ ::= SIZE|ORIENT|VX⟨numeral*⟩|VY⟨numeral*⟩ [3]

(8) ⟨feature⟩ ::= ⟨feature*⟩{(⟨object⟩)|"}

(9) ⟨function⟩ ::= SIN|COS|ARCCOS|SQR|SQRT [4]

(10) ⟨operator⟩ ::= SUM|DIFF|PROD|QUOT [5]

(11) ⟨logical⟩ ::= EQ|NE|GT|GE|LT|LE [6]

(12) ⟨term⟩ ::= ⟨feature⟩|⟨number⟩|
OP⟨operator⟩(⟨term⟩,⟨term⟩)|
FCN⟨function⟩(⟨term⟩) [7]

---

[1] This list specifies the objects presently defined in the semantic memory. To ease the programming effort, the maximum number of characters in a name has been specified to be ten. For this reason the objects "isosceles triangle" and "equilateral triangle" are represented by the strings ISOSTRI and EQUILTRI respectively.

[2] It will later become apparent that ⟨object*⟩ represents generic objects in the semantic memory while ⟨object*⟩⟨numeral*⟩ represents specific instances of these objects defined by the PL and incorporated into the Short-Term Memory (STM).

[3] These are the strings representing the feature set previously discussed.

[4] These are the strings representing the functions recognized by the compiler. SQR is the function specifying the square of its argument; the remaining functions are self-explanatory.

[5] These are the strings representing the four arithmetic operations.

[6] These are the strings representing the equality/inequality constraints recognized by the compiler.

[7] Terms are the equivalents of mathematical expressions (e.g. sin(a + bx)).

(13) <predicate> ::= <feature>b<term>

(14) <toprel> ::= LEFTOF|RIGHTOF|ABOVE|BELOW[1]

(15) <draw> ::= DRAW(<object*>{<numeral*>|"})

(16) <logic> ::= LOGIC(<logical>(<predicate>))

(17) <topol> ::= TOPOL(<toprel>(<object*>{<numeral*>|"}b
<object*>{<numeral*>|"}))

The PL input consists of a stream of characters on punched cards.[2] Statements may begin or end in any column and may be continued on as many cards as is desired. Blanks may be inserted anywhere except within a name or a number.

A PL statement may contain errors in either syntax or semantics. While the present implementation of the compiler checks the syntactic validity of PL statements, it only partially checks their semantic validity. By using a sufficiently powerful compiler in conjunction with the semantic memory, it would be possible to fully determine semantic validity. The availability of such a compiler would permit simplifications to the structure of the semantic memory implemented in this thesis. More will be said about this in Chapters 3 and 4.

---

[1] These are the strings representing the four topological relationships recognized by the compiler.

[2] Obviously, a practical system would primarily input this information in an interactive fashion via a graphics terminal.

As an example of the capabilities of the PL, consider the problem of drawing a square and a triangle subject to the following constraints:

size(square) $\leq (5 + \sin 52)^2$ (i)

VX_1(square) = 63 (ii)

triangle is to left of square (iii)

A PL program coding of this problem is:

```
DRAW(SQUARE)
LOGIC(LE(SIZE FCNSQR(OPSUM(5 FCNSIN(52)))))
LOGIC(EQ (VX1 63))
DRAW(TRIANGLE)
TOPOL(LEFTOF(TRIANGLE SQUARE))
```

This example demonstrates that the DRAW command possesses a "sphere of influence" which extends to statements between it and the next DRAW command. For statements within this "sphere" the object being predicated is implicitly defined. Thus the third line of the program really means

```
LOGIC(EQ(VX1(SQUARE)63)).
```

Note that the order of statements between two DRAW commands is unimportant.

This section completes the discussion of the PL for the moment. It will be resumed in Chapter 3 where the translation of PL statements into the STM by the compiler will be considered.

The remainder of this chapter is concerned with a description of the author's semantic memory.

2.4 The Semantic Memory

As was stated earlier, the semantic memory consists of both a semantic map and a semantic net. The structure of each of these will be detailed later in this chapter. For the moment it suffices to make the following observation. Analogous to Quillian's work, objects in the author's semantic net are represented by planes. Each object is a concept and each concept need be defined only once. Complex objects are defined by specifying a predication of their constituent objects. For example, if RECTANGLE has been defined then SQUARE is represented by a plane which contains the information necessary to modify the RECTANGLE concept so as to produce the definition of SQUARE. The way in which this is done will become apparent in due course.

The following section describes the types of nodes present in the semantic memory.

2.4.1 Node Types

The semantic memory contains three types of nodes: OBJECT, MODIFIER and LIST. These nodes consist of several fields each of which contains either a character string or a pointer to another node.

The first of these, the OBJECT node, is a "header" node; there is only one for each semantic plane (Fig. 2-8). The NAME field of the OBJECT node specifies the object to be defined, the SUBOBJ

field is a pointer to the subobject list, the ATTS field points to the predicates on the object's features and the PARENT field points to the object's parents. This last field is unused in the present implementation of the program; it has been included in the node for possible future applications of the semantic memory to picture analysis. It also demonstrates a difference between Quillian's memory and the author's model. In the former each concept had only one parent whereas in the latter many parents are permitted. For example, two parents of LINE are RECTANGLE and TRIANGLE.

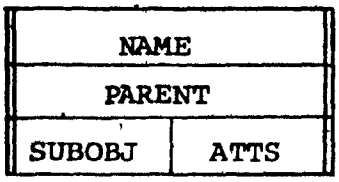


Fig. 2-8: The OBJECT node. The NAME field contains a character string while the other fields contain pointers.

The primary function of the MODIFIER node is the predication of the features of an object. Its TYPE and FCN fields describe the type of predication to be performed while its PTR1 and PTR2 fields contain pointers to the left and right operands involved (Fig. 2-9).[1] Some applications of this versatile node will be

[1] Note that in drawings, the OBJECT and MODIFIER nodes are distinguishable because of double bars on the former and single bars on the latter.

described in the examples of the next section.

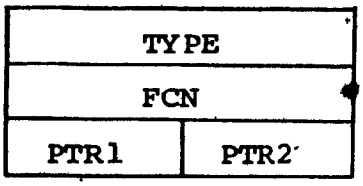


Fig. 2-9: The MODIFIER node. The TYPE and FCN fields contain character strings while the PTR1 and PTR2 fields contain pointers.

The LIST node is a general-purpose link for stringing together the other two types of nodes. Its THIS field is a pointer to the node being referenced while its NEXT field is a pointer to the next LIST node on the chain (Fig. 2-10).

Fig. 2-10: The LIST node. Both the THIS and NEXT fields contain pointers

This concludes the discussion of the node types found in the semantic memory. The next sections detail the memory's structure.

2.4.2 Structure of the Semantic Map

The structure of the semantic map is very simple: it forms a chain consisting of LIST nodes. Each member of the chain represents a concept by pointing to a plane in the semantic net which defines the concept (Fig. 2-11). The DICTIONARY pointer locates the header node of the semantic map. Furthermore, the LIST nodes are ordered so that the concepts they access are in lexicographical order.

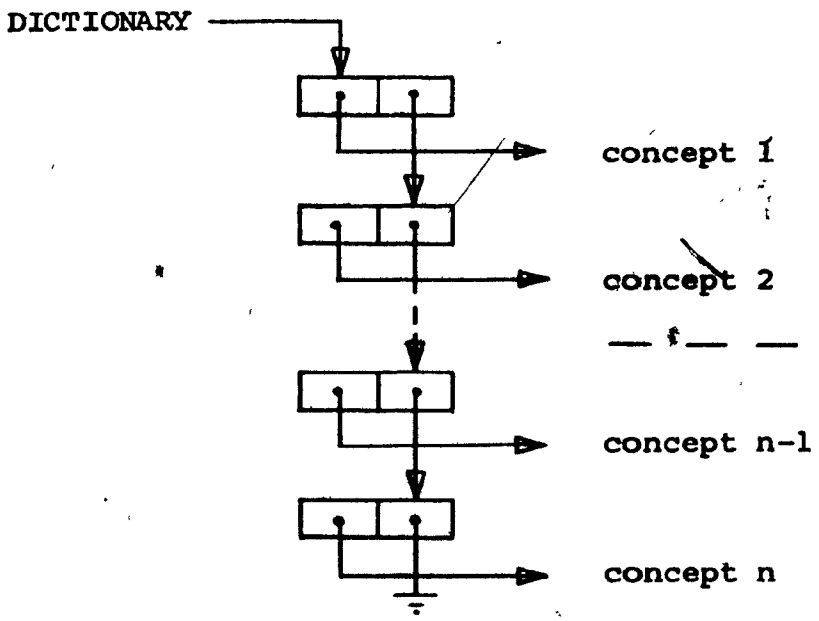


Fig. 2-11: The structure of the Semantic Map. The names of $concept_1$, $concept_2$,..., $concept_{n-1}$, $concept_n$ are in lexicographical order. Note that the NEXT field of the last node of the map contains the NULL pointer.

2.4.3 Structure of the Semantic Net

The semantic net consists of a collection of interlinked planes each of which is accessed by the semantic map. A suitable starting point for the specification of its structure is the description of the way in which the planes are interlinked. Consider the concept of a square. Because it is a specific type of rectangle and because a rectangle contains lines, it is expected that the SQUARE, RECTANGLE and LINE planes should be somehow linked. This is indeed the case as is shown in Fig. 2-12.

Fig. 2-12 also demonstrates the two types of links which are used between semantic planes. The first type, the subobject link, yields the subobjects of an object. It is the one which links the headers of the SQUARE and RECTANGLE planes to the header of the LINE plane. The second type, the predicate link, is the one which allows a concept to be defined as the predication of another concept. It is the usage of this link which yields a compact structure for the semantic net. An illustrative example of its function is given in Fig. 2-13. The examples that follow will demonstrate the utility of the predicate link.

The next step in the description of the system is an analysis of the structure of the semantic planes. It was decided at the outset that planes should be defined for both objects and features. The former are predicated by the values of the features and their

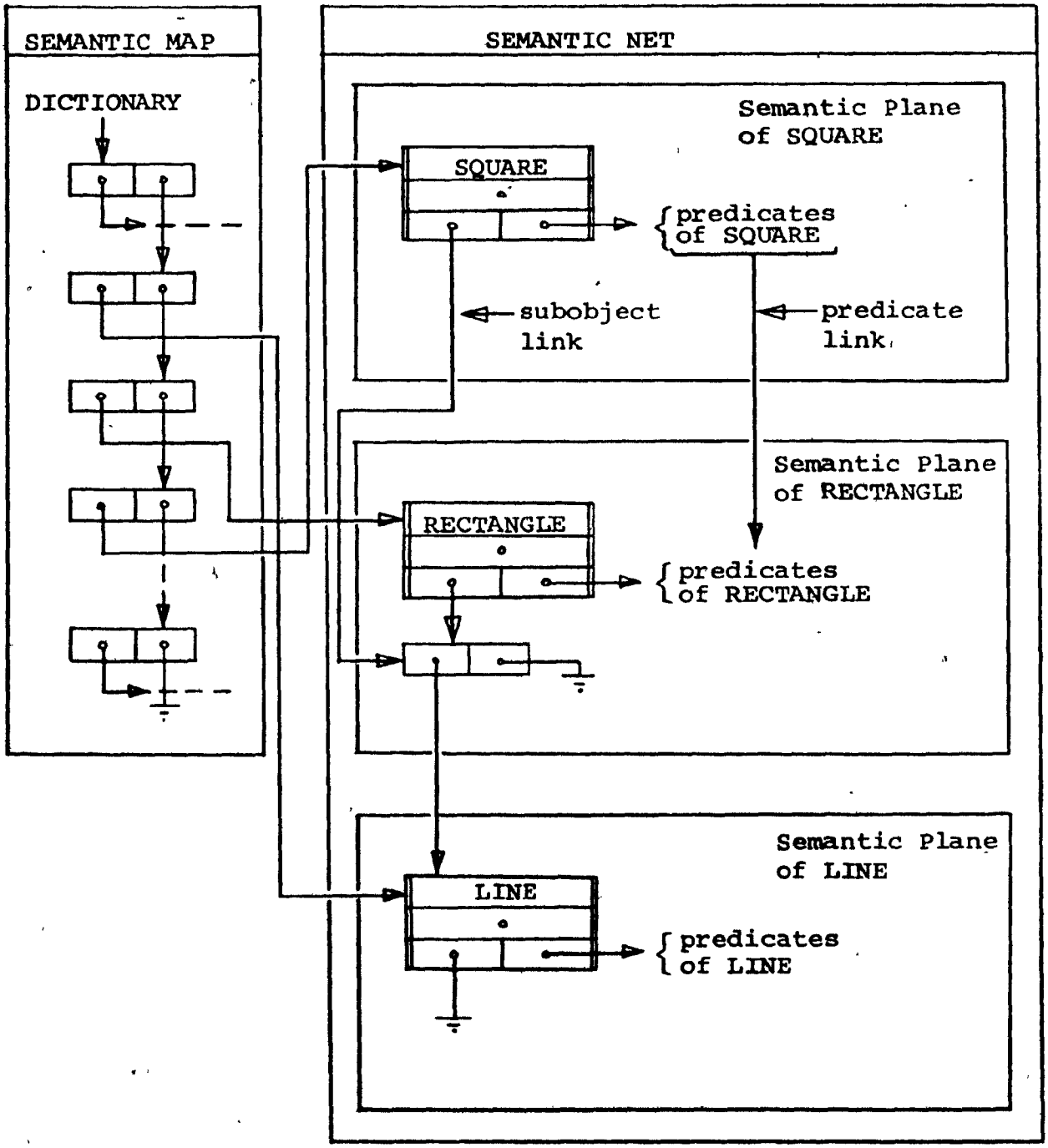


Fig: 2-12: The linking of the SQUARE, RECTANGLE and LINE semantic planes. The nodes are structured as described in Section 2.4.1.

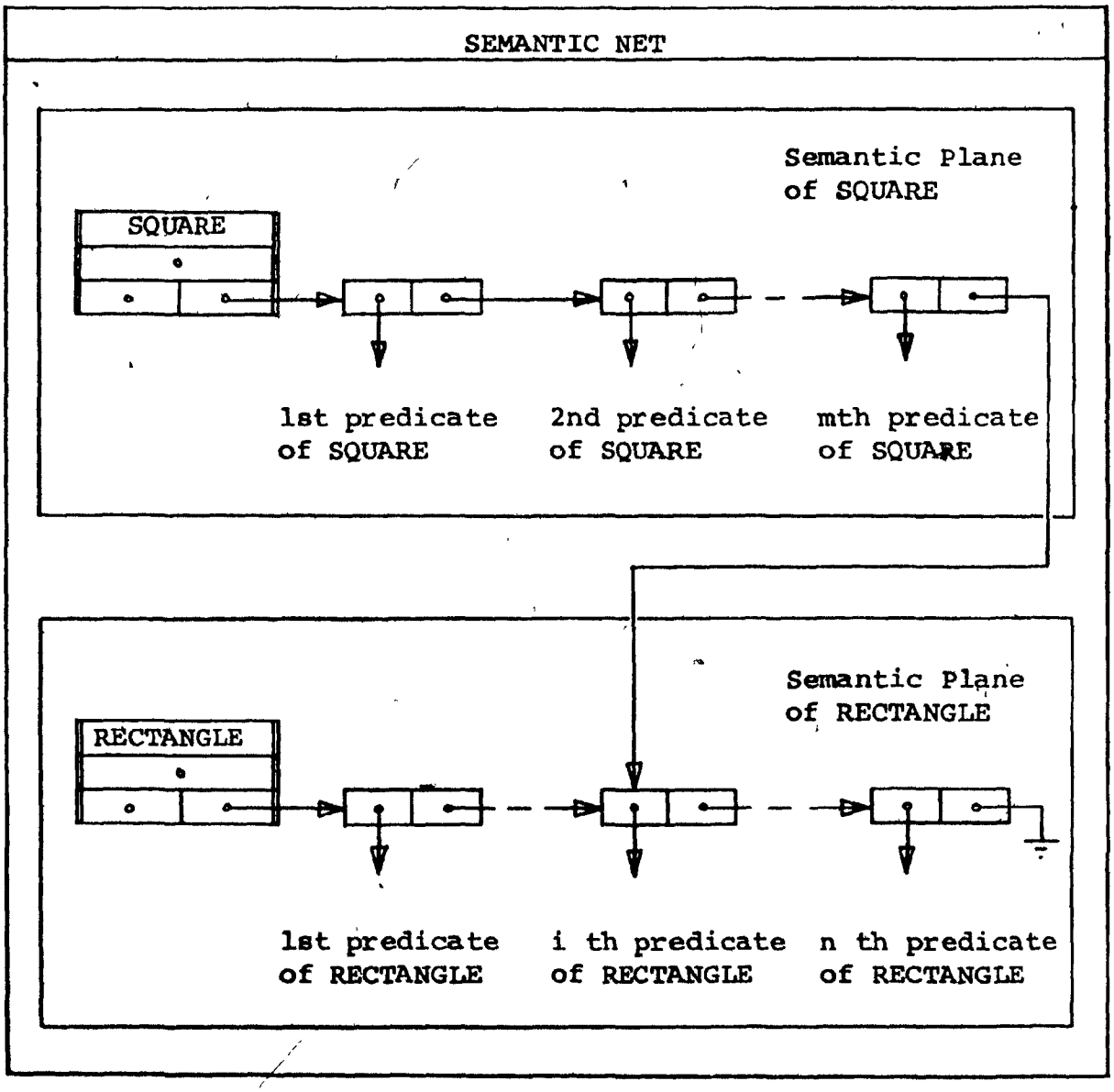


Fig. 2-13: Example of the function of the predicate link.

planes may be quite complex. On the other hand the latter, being themselves features, have no predicates; their semantic planes consist of a sole OBJECT node with all pointers set to NULL (Fig. 2-14). The features which are assigned semantic planes

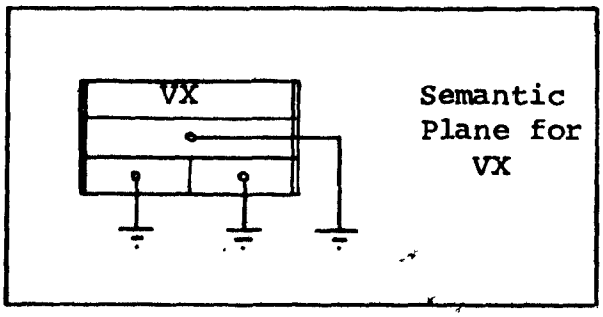


Fig. 2-14: The semantic plane for the VX feature. Similar planes exist for VY, SIZE and ORIENT.

are SIZE, ORIENT, VX and VY. Note that the contour vertices VX and VY have no suffixes. The next example explains the reason for this supposed oversight.

Consider the LINE plane. One of its predicates may be $vx_2 \geq 0$, where vx_2 is the x-coordinate of the second vertex of the line. Its representation in the semantic net is shown in Fig. 2-15. Three applications of the usage of the MODIFIER node as specified by its TYPE field are evident from this example. When the field contains the string LOGIC then the FCN field consists of a string naming the binary relation between the two operands accessed by

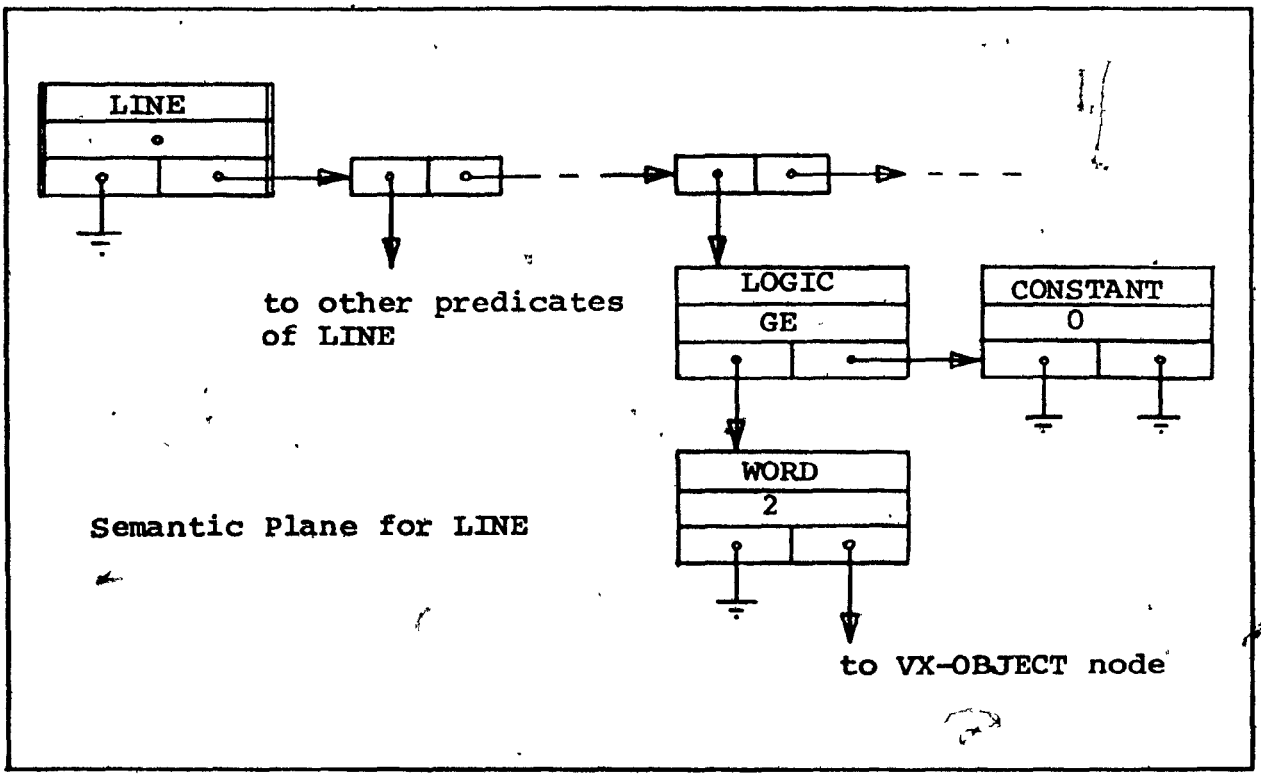


Fig. 2-15: The semantic plane for LINE showing the structure of the predicate LOGIC(GE(VX2 0)).

the PTR1 and PTR2 fields. When the field contains the string CONSTANT then the FCN field consists of a string specifying a numeric constant. The usage of the node when the TYPE field contains the string WORD pertains to the discussion of the previous paragraph and requires some explanation. According to the syntax of the PL discussed in Section 2.3.3, VX2 is a member of ⟨feature⟩. Although the concept VX has been defined (as was shown in the previous example), VX2 has not. What is then required is a predication of VX. This is accomplished by the cre-

ation of a modifier node whose TYPE field contains the string WORD and whose function field contains a string which is to be appended onto the concept specified by the node's PTR2 field. In the example of Fig. 2-15, this procedure yields VX || 2[1] or VX2. The node's PTR1 field is set to NULL because no further qualification of the feature is required. Fig. 2-16 considers a more complex example of the usage of WORD. In this case the feature as specified by the PL is VX2(LINE1(TRIANGLE)).

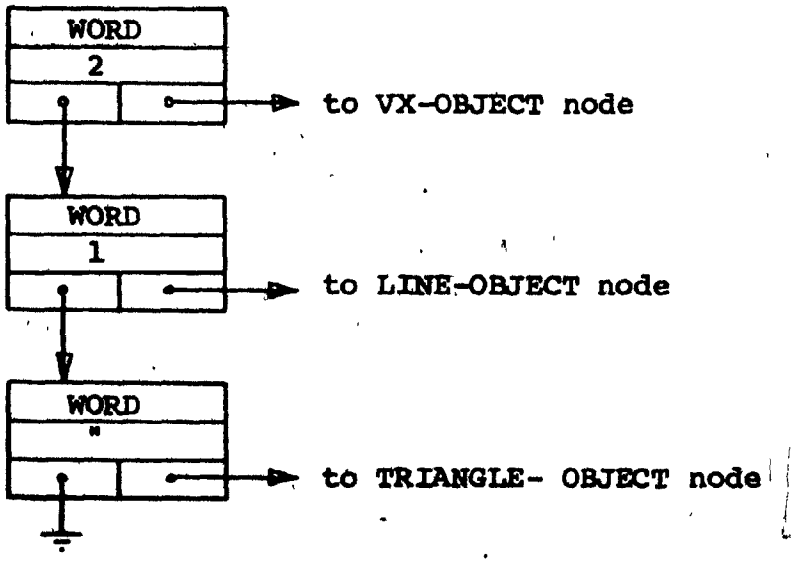


Fig. 2-16: Semantic net representation of VX2(LINE1(TRIANGLE)). Note that in the bottom node the null string is to be appended to TRIANGLE as required.

---

[1] || means "concatenated with".

It can now be explained why the contour vertex features were defined without a suffix. If this were not done then a plane would have to be defined in the semantic net for each of VXi and VYi corresponding to the i th contour vertex. But with 2n such planes, the semantic net would be unable to describe objects having more than n contour vertices since the character strings for the (n + 1)th vertex would not exist. One could, of course, work within such a restriction assuming that one were prepared to make n sufficiently large so as to provide for an object having a large number of contour vertices. Besides being inelegant, such a procedure would clutter the semantic map with pointers to 2n VX and VY planes. The solution to the problem implemented in the last example allows the specification of an unlimited number of contour vertices by defining semantic planes for only one. It can also be seen from the last example that this scheme allows the names of objects to be defined by an unsuffixed character string with the same resultant simplicity of the semantic map.

The next example in this section further illustrates the complexity of predicates which can be incorporated into the semantic net. Consider the PL string LOGIC(EQ(SIZE FCNSQRT(OPSUM(VX2 VX2)))). Its representation in the semantic net is shown in Fig. 2-17. It can be seen from an examination of the figure that

the MODIFIER node is being employed using both the OP and FCN strings in its TYPE field. When the OP mode is used PTR1 and PTR2 point to the left and right operands while the FCN field specifies the operation. When the FCN mode is used the PTR1 field is set to NULL while the FCN field specifies the function and PTR2 points to the argument.

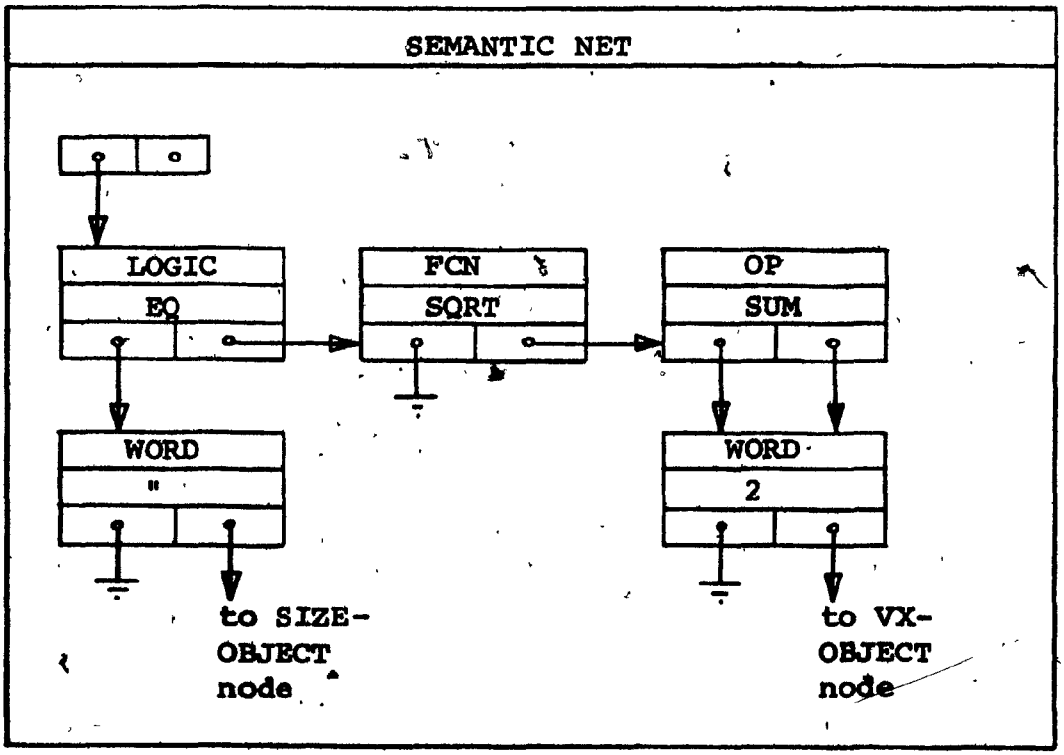


Fig. 2-17: Semantic net representation of the predicate LOGIC(EQ(SIZE FCNSQRT(OPSUM(VX2 VX2)))).

One usage of the modifier node not yet discussed is the KNOWN mode. It will not be described in this chapter because its usage arises not out of a conceptual necessity but rather to compensate for a shortcoming of the compiler. The details of its usage will be given in Chapter 3.

The next example of this section illustrates a method by which the predicate link yields a compact structure for the semantic net. Consider the semantic plane for LINE. Certainly one of LINE's predicates is LOGIC(GT(SIZE 0)) since every line must have a nonzero length. Consider now the semantic plane for TRIANGLE. One of its predicates must also be LOGIC(GT(SIZE 0)) since every triangle must have a nonzero perimeter. Since these predicates will have the same structure in the semantic net, it would be wasteful to specify both separately. The predicate link allows the specification of both while using only one structure. The way in which this is done is shown in Fig. 2-18.

The predicate link may also be used to link subcomponents of predicates. Consider the two PL strings LOGIC(GE(VX3 0)) and LOGIC(LE(VX3 5)). The representation of these in the semantic net is shown in Fig. 2-19. Note that the node symbolizing VX3 is shared between the two predicates.

It is obvious that the usage of the predicate link yields a compact semantic network possessing a minimal amount of dupli-

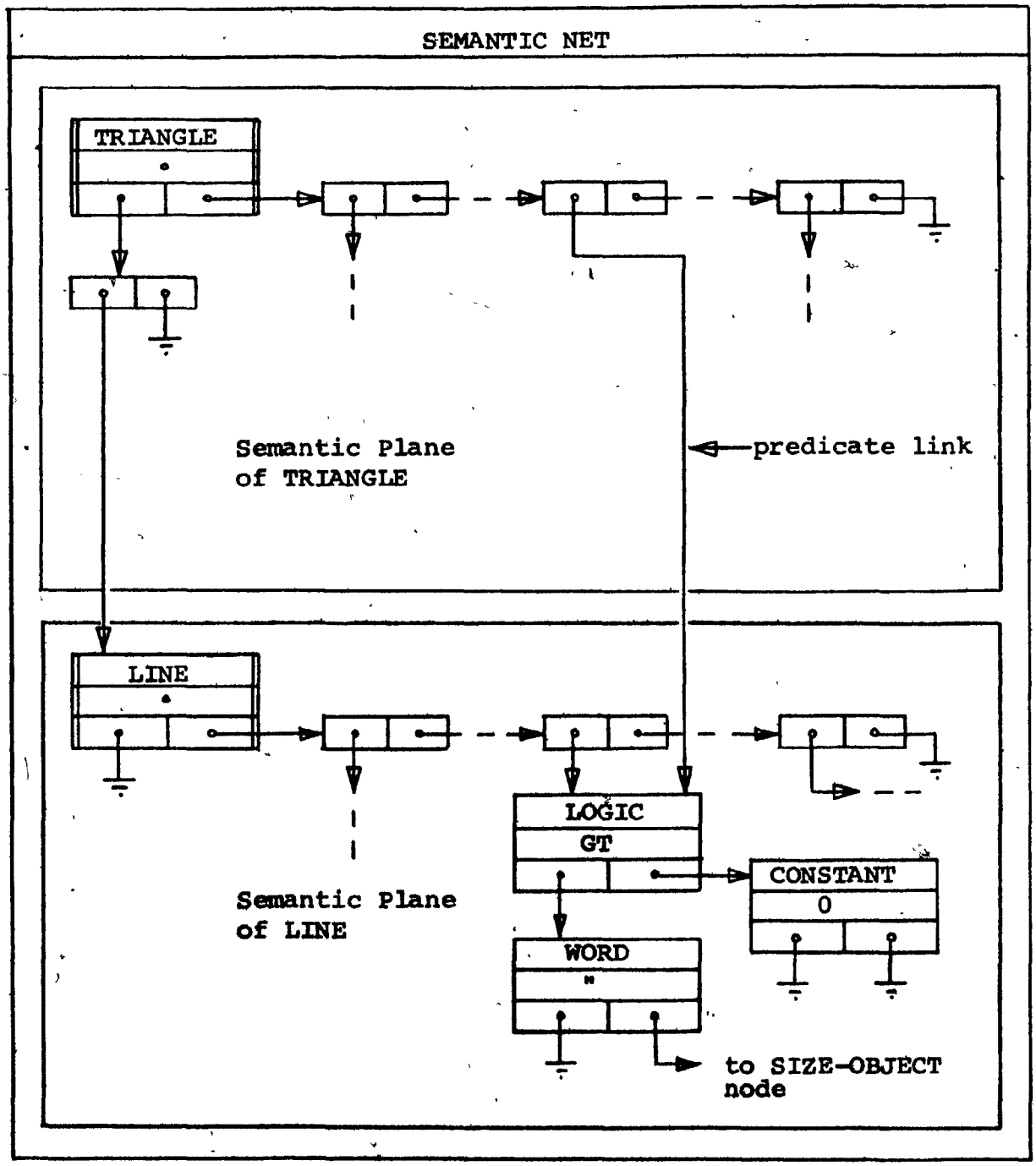


Fig. 2-18: The semantic planes for TRIANGLE and LINE showing the usage of the predicate link.

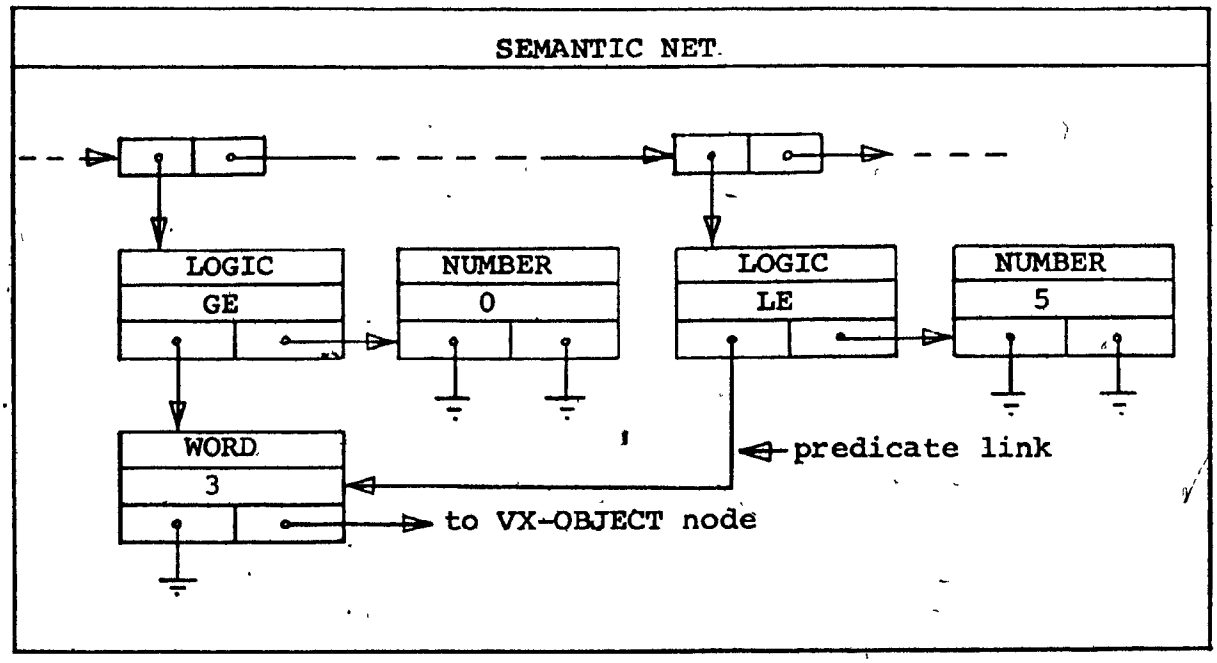


Fig. 2-19: The use of the predicate link in linking together subcomponents of the predicates LOGIC(GE(VX3 0)) and LOGIC(LE(VX3 5)).

cated information.

The example of Fig. 2-18 illustrates another interesting feature of the semantic net. As many already been noticed by the alert reader, the PL description of the predicates were incomplete in that the features were not sufficiently qualified. To fully qualify the SIZE feature the first predicate should have been LOGIC(GT(SIZE(LINE) 0)) and the second predicate should

have been LOGIC(GT(SIZE(TRIANGLE) 0)). This was not done because, analogous to the PL language structure, each plane in the semantic net possesses a "sphere of influence". Thus the name of the object of the plane which accesses a predicate is automatically appended to each of the conditions on features specified by the predicate.

It is now possible to generalize the structure of a predicate if one omits from this discussion the structures which result from the use of the KNOWN mode of the MODIFIER node. Specifically, any predicate can be modeled by a binary tree[1] whose patriarch is a MODIFIER node of the LOGIC type. For example the predicate specified by the example of Fig. 2-17 can be modeled by the binary tree of Fig. 2-20. Note that the leaves of the tree are either OBJECT nodes, NULL pointers or MODIFIER nodes containing the CONSTANT string in their TYPE field.

The examples of this chapter have served to explain and highlight representations of various concepts within the semantic memory. Fig. 2-21 depicts a pictorial description of the memory in general.

---

[1] Binary tree: "a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left and right subtrees of the root" (Knuth, 1968, p309).

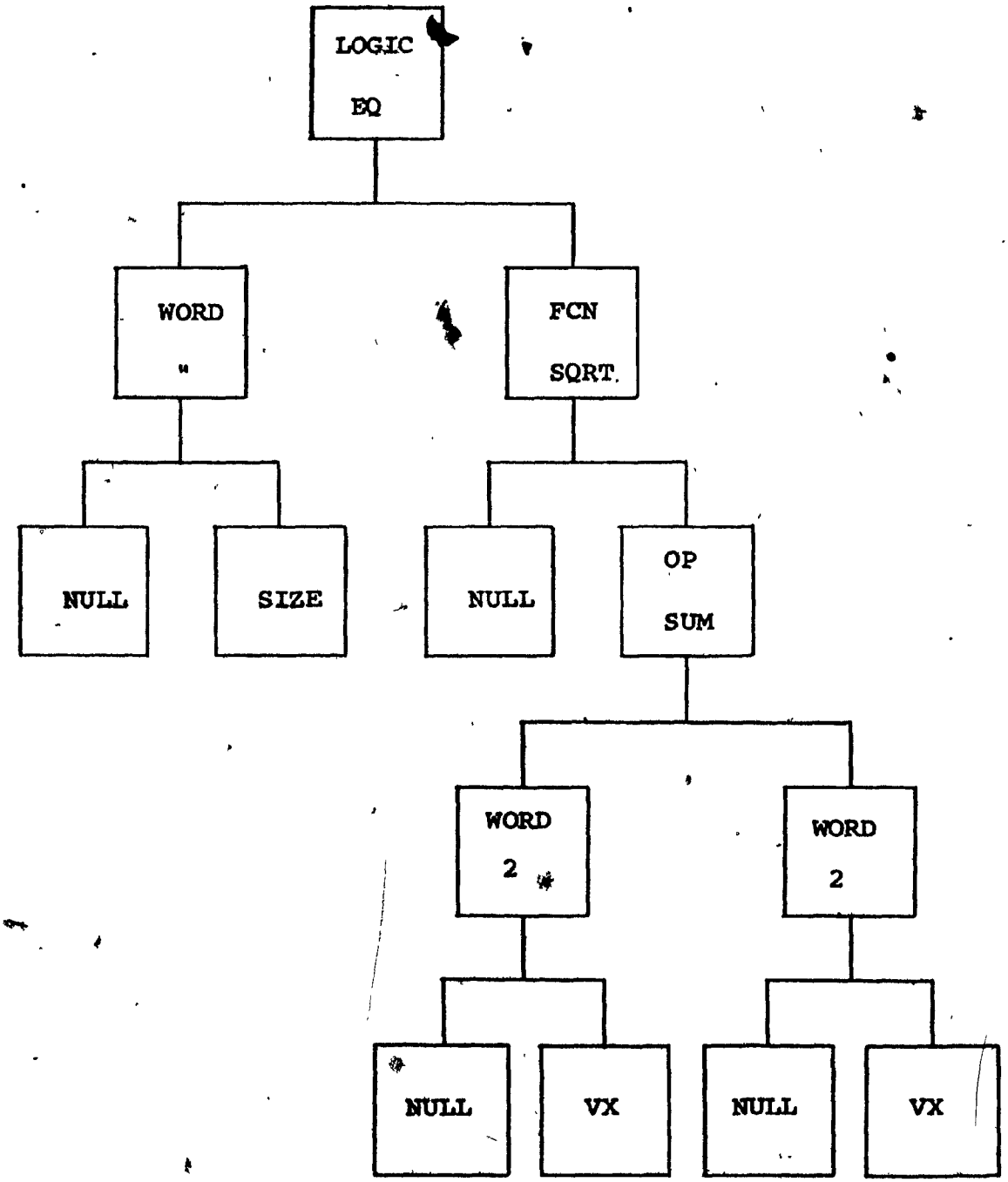


Fig. 2-20: The binary tree representation of the predicate of Fig. 2-17.

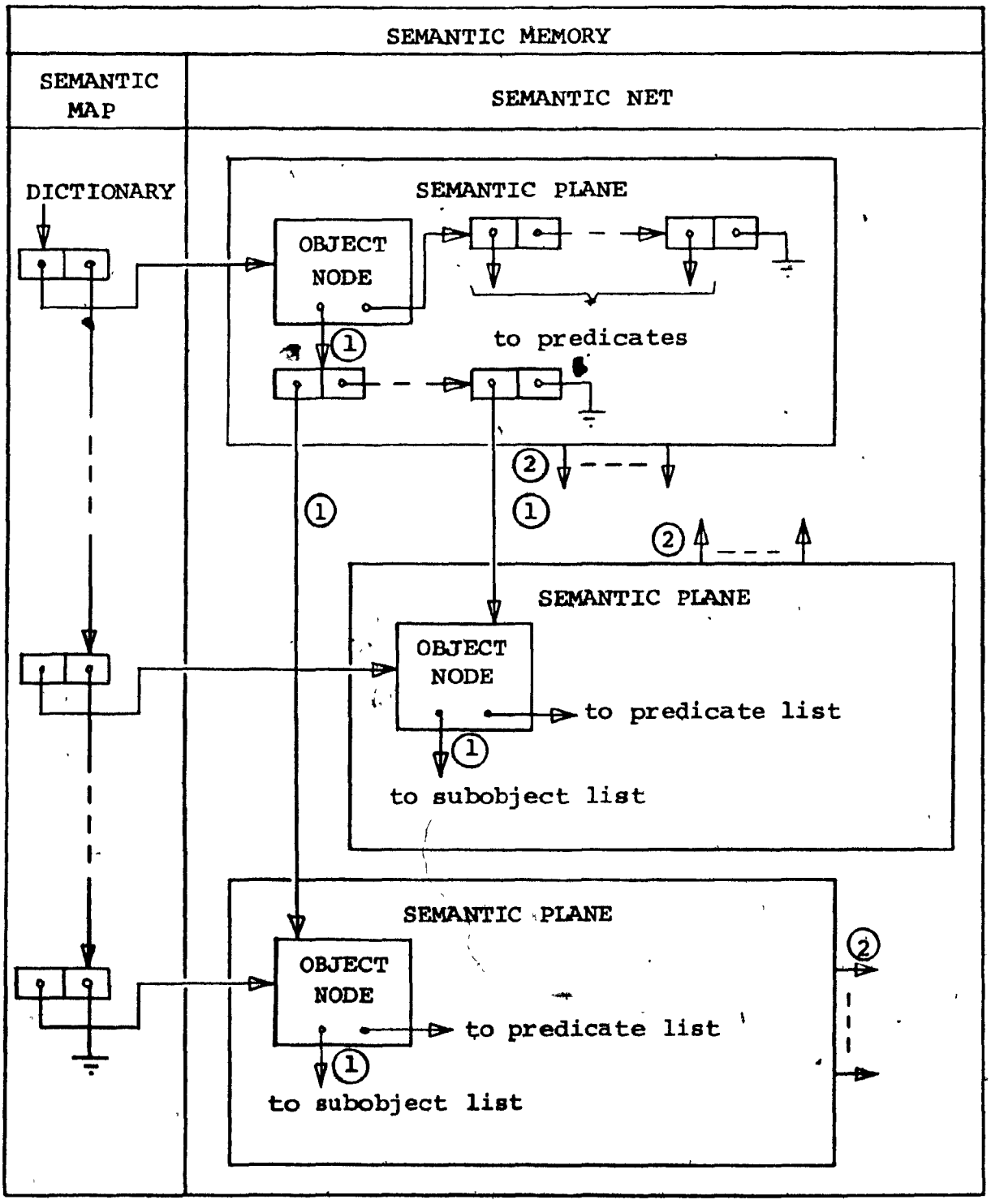


Fig. 2-21: Generalized form of a Semantic Memory; (1) and (2) refer to subobject links and predicate links respectively. While the predicate link is shown external to semantic planes only, it also exists within individual planes (see Fig. 2-19).

Given the details of the structure of the semantic memory, it is possible to justify the claim that it can supply all the information necessary to draw an object. The memory has been defined so that it contains enough information to limit the value of each feature (f_v) of an object as follows:

either (i) $f_v = a$

or (ii) $f_v \in [b,c]$[1] and $f_v \neq d_i$ $\quad i = 1,2,\ldots$

where a,b,c and d_i's are constants and the d_i's are members of the (possibly null) set of forbidden values of f_v. If the constraint on f_v is type (i), then the feature value is fixed. Otherwise, a random selection of f_v subject to the constraints of (ii) will yield an acceptable value. Any constraints supplied by PL statements serve to further limit the range of acceptable values for f_v.

The way in which the semantic memory's information is accessed and used is the subject of the next chapter. The next section of this chapter formalizes some of the concepts described to this point.

---

[1] This notation means: "the value of f_v falls within the interval bounded by the points b from below and c from above".

2.4.4 Concepts

This section formalizes the structure of the semantic net. Its objective is the definition of what constitutes a concept and the determination of whether it is possible to represent all concepts by the relational LEAP structures of Section 1.3.3.

A characteristic of a semantic net is that it contains structures called concepts which need be defined only once. The converse of this statement is now assumed. That is, any structure which need be defined only once is a concept of the semantic net. By definition, a structure consists of a node called the patriarch and all the nodes accessed by it, either directly or indirectly. Because of the manner in which the semantic net had been structured, the above definitions imply that choosing any node in the net as patriarch will isolate a structure which represents a concept. It is contended that each structure may be represented by one or more LEAP triples or by a member of a triple.

The specification of a relational triple in the LEAP language may follow either the (O A V) format discussed in Chapter 1 or the more general (A R B) format. The former is especially suited to linguistic descriptions while the latter simply states that two concepts A and B are linked by a relation R. Because the author's net is mathematical rather than linguistic, its concepts are more amenable to representation by (A R B) triples

than by (O A V) triples.

Consider the set of nodes in each plane of the author's net which may be patriarchs. Reference to Fig. 2-21 yields the following members:

(i) Any MODIFIER node. This node accesses either all or part of a predicate within a plane.

(ii) Any one of the LIST nodes accessed either directly or indirectly by a pointer from the ATTS field of an OBJECT node. The LIST node accesses both a predicate as well as the next LIST node (if any) in the list of predicates (see Fig. 2-21).

(iii) The OBJECT (i.e. header) node of a semantic plane. This node accesses a set of subobjects through its SUBOBJ field and a set of predicates through its ATTS field.

(iv) Any one of the LIST nodes accessed either directly or indirectly by a pointer from the SUBOBJ field of an OBJECT node. The LIST node accesses both a subobject (i.e. the header of another plane) as well as the next LIST node (if any) in the list of subobjects (see Fig. 2-21).

The concepts described by (i) can easily be shown to be representable by either a relational triple or by a member of such a triple. It was shown in the previous section that any

predicate can be modeled by a binary tree. Because any portion of a binary tree is either a leaf or another binary tree, any part of a predicate can be also so modeled.

A binary tree can be modeled by a relational triple as can be seen from Fig. 2-22. Therefore, any part of a predicate described by (i) can either be modeled as a relational triple or

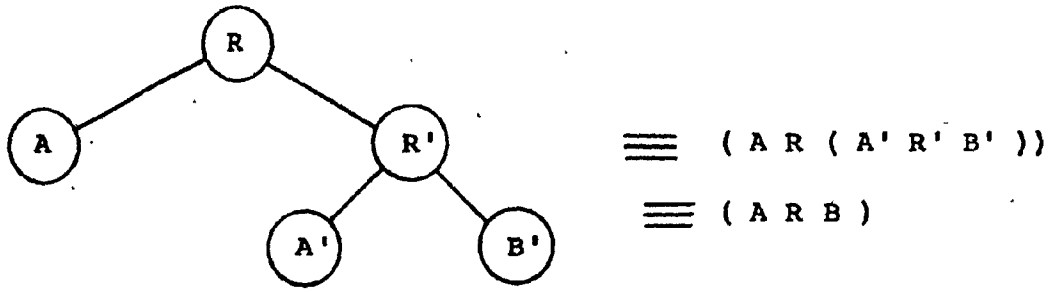
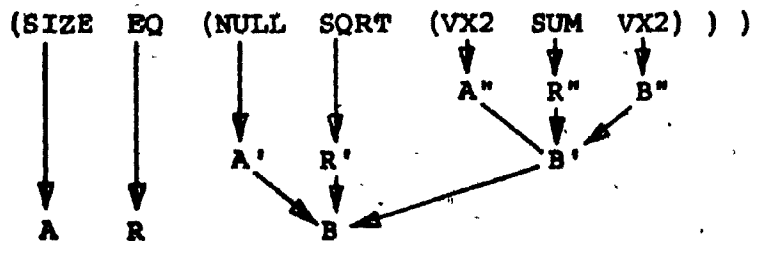


Fig. 2-22: Representation of a binary tree by (A R B) where B (A' R' B').

as a member of one. The only exception to this occurs when the patriarch node is a MODIFIER node containing the string WORD. In this case the structure represents a feature and is represented by a terminal member of a triple. For example, the predicate of Fig. 2-20 is represented by the triple:



Because the function of a LIST node is to AND together groups of OBJECT or MODIFIER nodes, the concepts described by (ii) are representable by an AND'ed set of relational triples each of which symbolizes a predicate in the semantic plane. Thus type (ii) concepts become $(A_1\ R_1\ B_1) \cap (A_2\ R_2\ B_2) \cap \ldots \cap (A_n\ R_n\ B_n)$.

The concepts described by (iii) are representable by AND'ing two classes of relational triples. The first class describes the predication of an object by its subobjects and can be constructed by observing that the SUBOBJ pointer from the OBJECT node can be represented by the relation R $\equiv$ "has-as-subobject". The subobjects of the patriarch node which are linked by LIST nodes become the "B" members of the triples and the class of triples is modeled by:

$$\begin{aligned} &(\text{OBJECT has-as-subobject OBJECT}_1) \\ \cap\ &(\text{OBJECT has-as-subobject OBJECT}_2) \\ &\quad \ldots \\ \cap\ &(\text{OBJECT has-as-subobject OBJECT}_n) \end{aligned}$$

In a similar manner, the second class of relational triples describes predication of an object by restrictions on values of its features. It can be constructed by observing that the ATTS pointer from the OBJECT node can be represented by the relation R $\equiv$ "has-as-predicate". The predicates of the patriarch node which are linked by LIST nodes become the "B" members of the triples and the class of triples is modeled by:

$$\begin{aligned} &(\text{OBJECT has-as-modifier MODIFIER}_1) \\ \cap\ &(\text{OBJECT has-as-modifier MODIFIER}_2) \\ &\quad \ldots \\ \cap\ &(\text{OBJECT has-as-modifier MODIFIER}_n) \end{aligned}$$

In summary , the concepts described by (iii) can be represented by AND'ing the two sets of relational triples described above.

In a manner similar to that used for type (ii) concepts, those of type (iv) can be represented by an AND'ed set of sub-objects each of which can be represented by an AND'ed set of relational triples. Thus type (iv) concepts are modeled as:

$$(\text{SUBOBJECT}_1) \cap (\text{SUBOBJECT}_2) \cap \ldots \cap (\text{SUBOBJECT}_n)$$

where each of the SUBOBJECT_i is representable by the set of relational triples outlined in the preceding paragraph.

This section has formalized the structure of the author's semantic net in two ways. First it has defined a concept as any structure in the semantic net. Second it has shown that any concept can be represented by an AND'ed set of relational triples or by a member of a relational triple. The next section briefly considers whether the semantic memory is a viable model of a human long-term memory in accordance with the precepts outlined in Section 1.3.3.

2.4.5 A Brief Comparison with Human Long-Term Memory

According to Chapter 1, four features of a human long-term memory are that it is associative, teachable, inferential and that it have a flexible retrieval mechanism. This section will briefly consider whether the author's semantic memory possesses these features.

That the memory is associative follows by the definition of a semantic memory. First, the existence of the semantic map means that entry into the net is on a content-addressable rather than location-addressable basis. Second, because the semantic net consists of linked sets of relational triples, accessing of information within the net is also on a content-addressable basis. These two features define the author's memory to be associative.

While the memory is theoretically teachable, this feature has not been implemented. Indeed, the techniques required to make the memory teachable constitute a complex procedure which is beyond the scope of this thesis. Some attributes of such a procedure will be discussed in Chapter 4.

That the memory is inferential is demonstrated by the following example. Consider the semantic plane of a triangle. One of its predicates is LOGIC(LT(SIZE(LINE1) OPQUOT(SIZE 2))) which states that the length of line number one is less than half the perimeter of the triangle. If the size of the triangle is given, then the compiler will be able to infer a limit to the size of line one.

If the memory is to have a flexible retrieval mechanism, it must be able to retrieve information given data whose format differs from that by which the memory previously learned the information. This feature is not present in the author's semantic memory because of its specialized (mathematical) nature. However,

it can be assumed that the more general linguistic memory of which it would be a part would possess this capability insofar as it must be able to translate input data to a form recognizable by the mathematical memory.

This section completes Chapter 2 which has been concerned with the PL and the semantic memory. The next chapter will discuss the compiler and the method by which it incorporates information from the PL and the semantic memory into the STM.

Chapter 3

The Compiler and STM

3.1 Overview

This chapter describes the compiler whose function is the translation of information from both the PL and the semantic memory into the STM so that the information can be used in the synthesis of line drawings. A flowchart of the operation of the compiler appears in Fig. 3-1.

As can be seen from the flowchart, compilation occurs in three steps. In the first step, the compiler accepts a PL program, checks its syntax and uses it to create the STM. The next step, which is repeated for each object to be drawn, consists of locating the semantic plane representing the object, creating a Subobject Tree (see Section 3.4.2), forming all predicates of the object and translating these into the STM using information stored in the Subobject Tree. In the third step, the Feature Value Selection Algorithm (see Section 3.5.3) is executed. This algorithm yields a numerical value for every feature of each object to be drawn consistent with the constraints specified by the STM. The feature values are then used to produce a line drawing.

The remainder of this chapter presents a detailed description of the above procedure and points out the limitations of the pre-

sent implementation of the compiler.

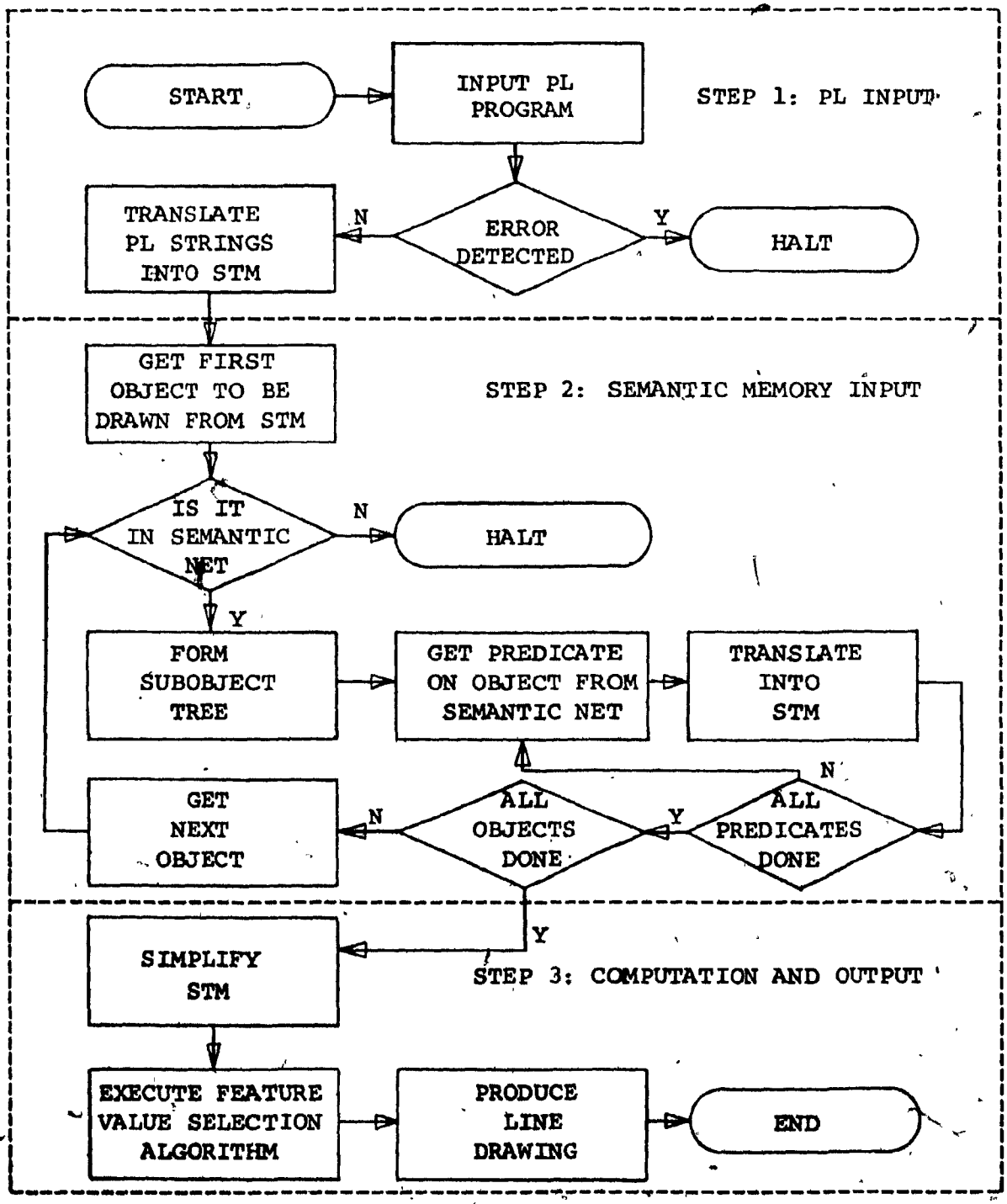


Fig. 3-1: General Flowchart of the operation of the compiler.

3.2 Structure of the STM

The operation of the compiler on the PL and the semantic memory produces the STM. Before the procedure by which this is accomplished is described, it is instructive to consider the structure of the STM in general.

The information stored within the STM represents specific instances of concepts defined in the semantic memory. It is therefore reasonable to assume that the structures of the STM and the semantic memory will be similar. A comparison of Fig. 3-2 with Fig. 2-21 shows this to be the case.[1]

These similarities in structure permit a carryover of many ideas evolved in Chapter 2 to the present discussion. For example, it will be shown that the STM contains LIST, OBJECT and MODIFIER nodes as does the semantic memory. Furthermore, the usage of these nodes will often be the same for the two structures.

A description of the methods by which the compiler creates the STM from a PL program and the semantic memory is the topic of the following sections.

---

[1] The TOPOLOGICAL RELATIONSHIP block of Fig. 3-2 is a temporary structure whose function will be described in Section 3.3.3.

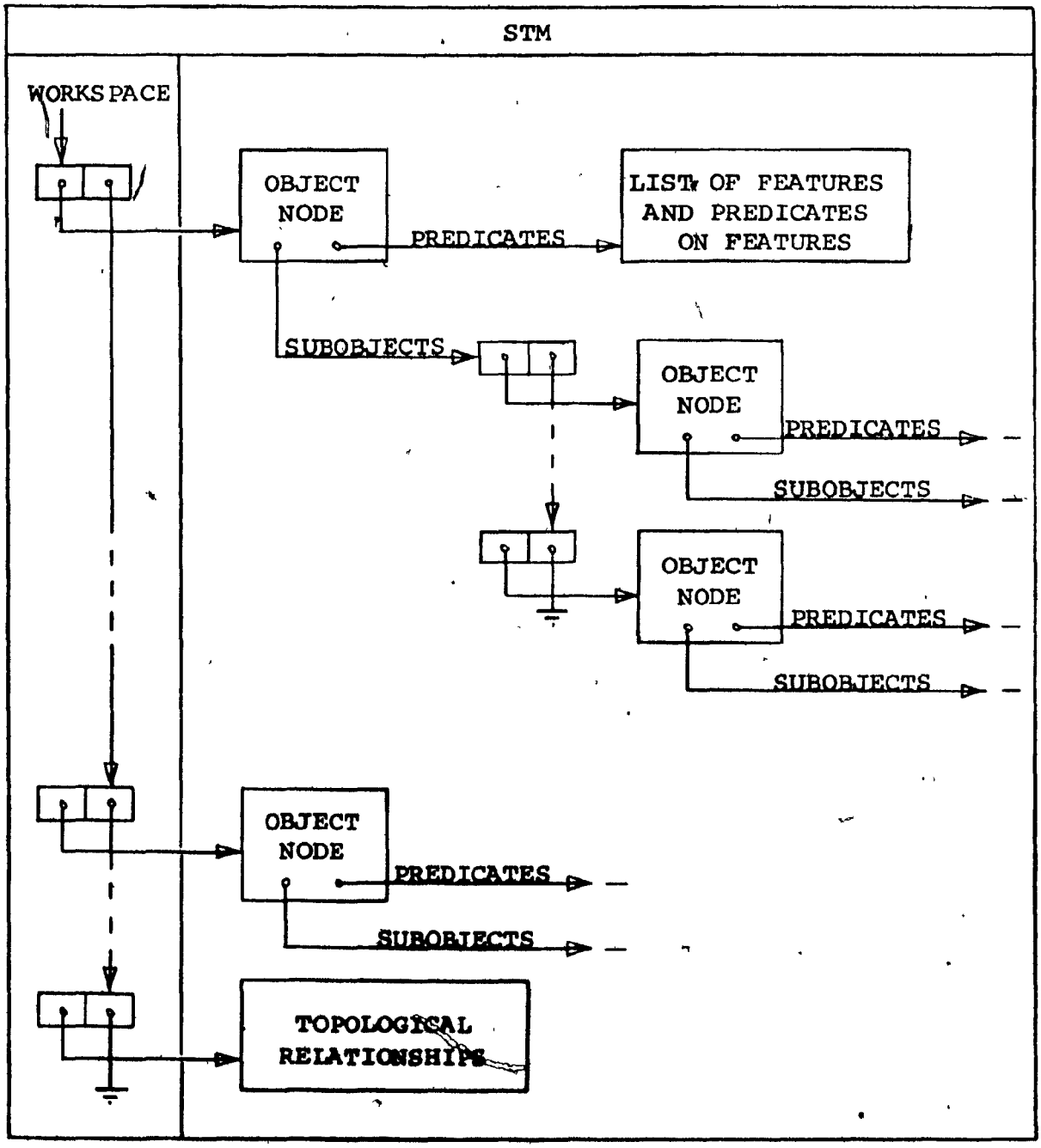


Fig. 3-2: Overview of the structure of the STM. The function of the TOPOLOGICAL RELATIONSHIPS block is described in a later section. WORKSPACE is a header pointer which locates the STM.

3.3 Translation of PL Strings

The first task which must be performed by the compiler is the translation of PL strings into the STM. The algorithm used by the compiler to check for syntax errors will not be described. Instead, it is assumed that the algorithm functions properly and that any syntax errors are detected and cause termination of compilation.

It was shown in Chapter 2 that the PL contains three statement types: DRAW, LOGIC and TOPOL. The way in which these are incorporated into the STM is discussed next.

3.3.1 PL Statements of the DRAW Type

The DRAW statement defines an instance of an object. Compilation of the statement creates a structure within the STM which represents the object. The details of this structure are now considered.

According to the syntax of the PL (Section 2.3.3), the DRAW statement has the form DRAW(<object*>{<numeral*>|"}).[1] Given such an input the compiler can deduce that an object having a feature set is to be drawn. It was shown in Chapter 2 that each object possesses six primary features. These are SIZE, ORIENT,

---

[1] If no numerical suffix is attached to the name of the object, a "1" is appended by the compiler. Thus, DRAW(LINE) becomes DRAW(LINE1).

VX1, VY1, VX2 and VY2. Depending on its complexity, it may also possess secondary features. Because the compiler cannot a priori know about the existence of the latter, the DRAW statement results in the creation of the STM depicted in Fig. 3-3.

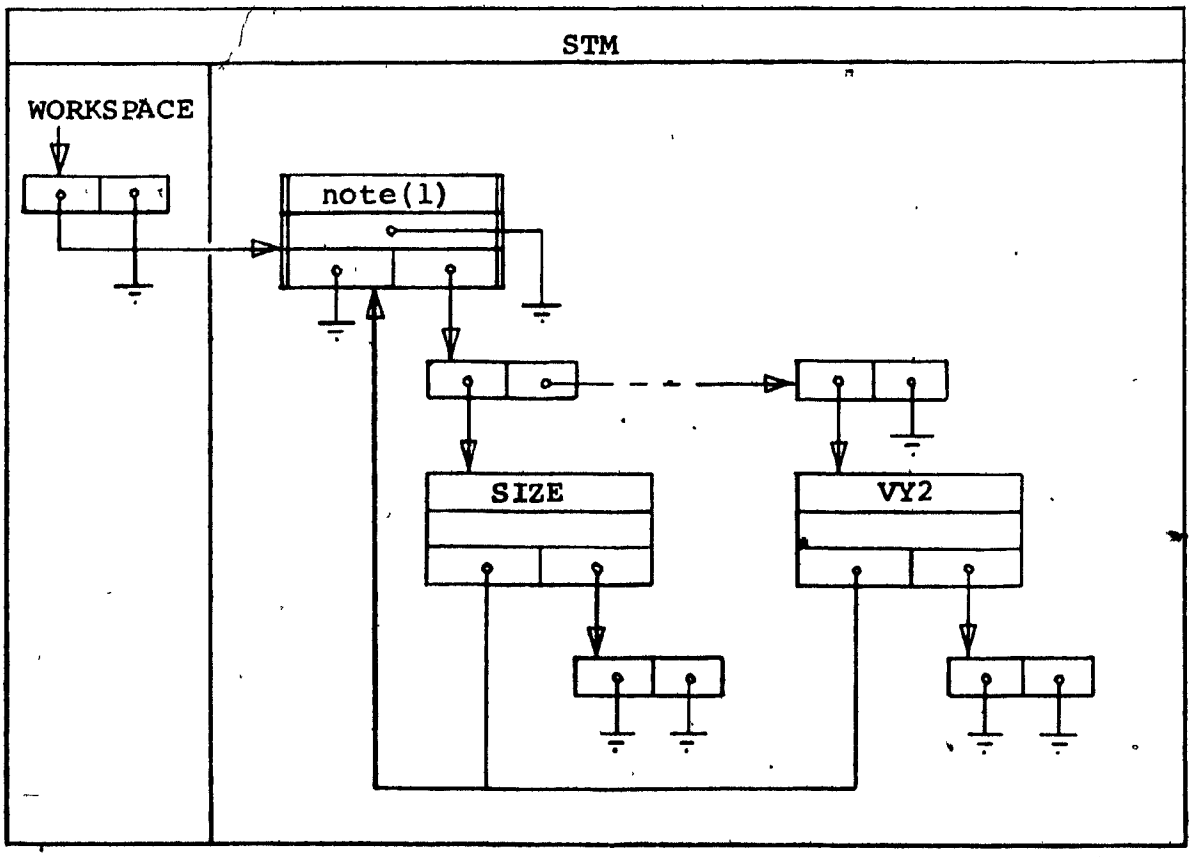


Fig. 3-3: The STM which results from a single DRAW statement. Conventions for drawing LIST, OBJECT and MODIFIER nodes are the same as those used in Chapter 2. The string represented by "note(1)" is ⟨object*⟩⟨numeral*⟩ where ⟨object*⟩ is as defined in the DRAW statement.

It can be seen from this figure that the OBJECT node identifies the object to be drawn. Its ATTS field points to the list of primary features each of which is represented by a modifier node. A backward pointer from each of these MODIFIER nodes permits the identification of the OBJECT node to which the feature pertains. Each MODIFIER node also contains a pointer to a LIST node. This node can be used to set up a list of Value Dependence Pointers (VDP's) whose function will be described in this chapter.

This completes the description of the compilation of the DRAW statement. Each time one of these is encountered by the compiler, another structure like that described above is created within the STM.

The next section describes the compilation of statements of the LOGIC type.

3.3.2 PL Statements of the LOGIC Type

The LOGIC statement specifies predication on the features of an object. By its very nature it may also reference secondary features[1] and subobjects which have not yet been encoded into the STM. By using two examples, this section will describe the compilation of the LOGIC statement into the STM. The first example will also demonstrate the incorporation of secondary features into the STM while the second will illustrate the incorporation of subobjects.

Consider the PL statements

```
DRAW(TRIANGLE2)
LOGIC(LE(VX3 OPSUM(VY2 5))).
```

The DRAW statement causes a structure similar to that of Fig. 3-3 to be set up in the STM. In compiling the LOGIC statement the compiler recognizes VX3 as a feature because of its position in the statement[2]. But because VX3 is not a primary feature in that it contains the suffix "3", it is recognized as a secondary feature. A new node representing it is therefore appended to the feature set of TRIANGLE2. Furthermore, the LOGIC statement specifies a predication on VX3 and this information is also encoded into the STM (Fig. 3-4).

---

[1] See Section 2.3.2.

[2] Refer to the BNF description of the PL (see Section 2.3.3).

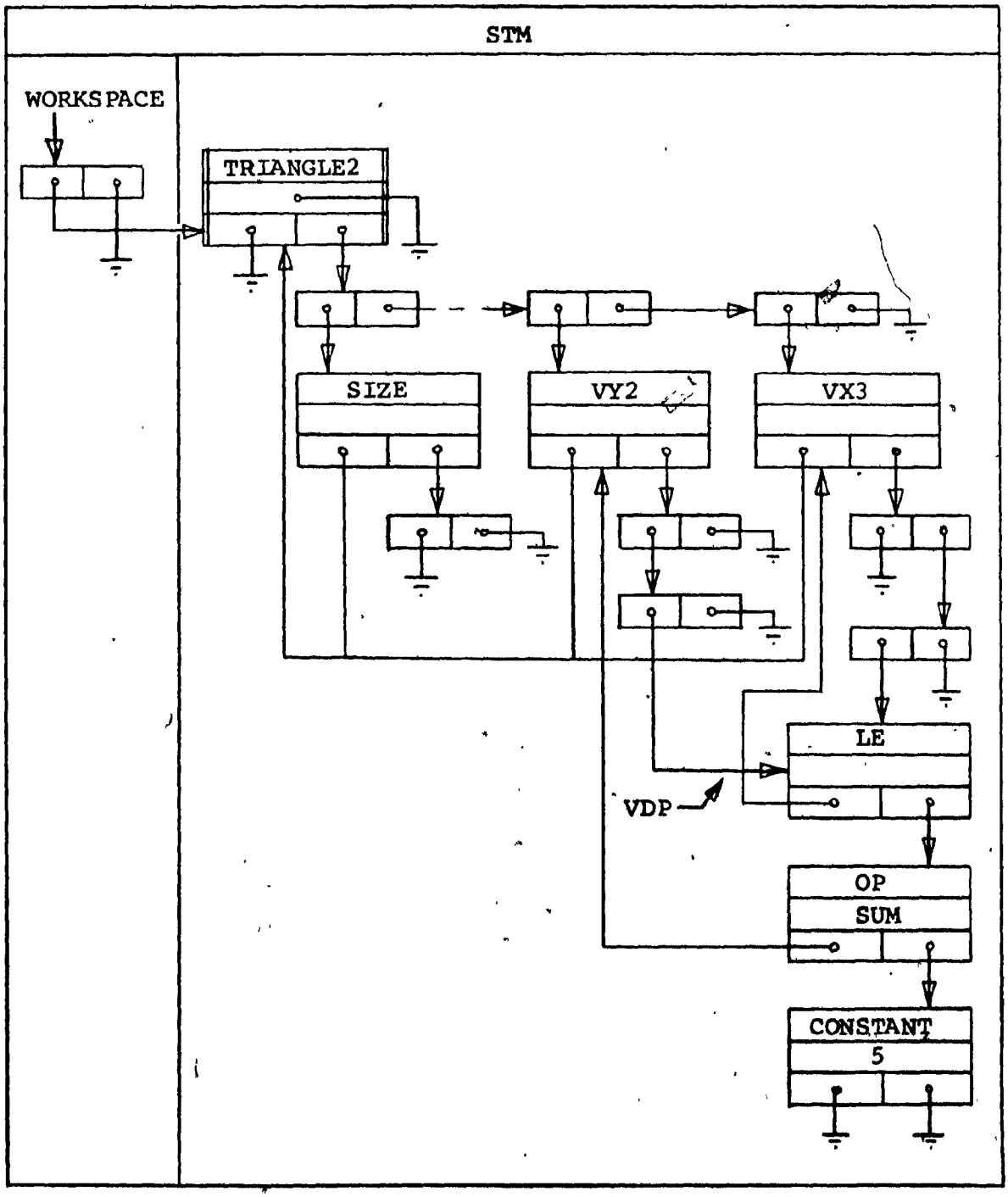


Fig. 3-4: STM representation of
DRAW(TRIANGLE2)
LOGIC(LE(VX3 OPSUM(VY2 5)))
VDP is the Value Dependence Pointer described in the text.

The LOGIC statement of the above example stipulates that VX3 is a function of VY2. As a result of this functional dependence, a backward pointer called a Value Dependence Pointer (VDP) is set up from the independent variable (VY2) to the predicate (see Fig. 3-4). The purpose of this pointer will be explained later in this chapter during the discussion of the Feature Value Selection Algorithm.

It can be seen from Fig. 3-4 that the structures of the STM and the semantic memory are similar in that MODIFIER nodes play a dominant role in the encoding of information. Their structures differ to the extent that the STM's is tailored to the specification of instances of objects defined by the memory. The resultant STM structure represents these instances in a manner which facilitates the computation of feature values.

The second example of this section illustrates the usage of the LOGIC statement in the specification of subobjects of an object. Consider the PL statements

```
DRAW(RECTANGLE1)
LOGIC(NE(SIZE(LINE2) SIZE)).
```

Because of its location within the LOGIC statement[1], the compiler recognizes LINE2 as an object. Furthermore, because the LOGIC statement is within the "sphere of influence" of RECTANGLE1,

---

[1] Refer to the BNF description of the PL (see Section 2.3.3).

LINE2 is recognized as a subobject of RECTANGLE1. The STM which results form the compilation of these statements is depicted in Fig. 3-5.

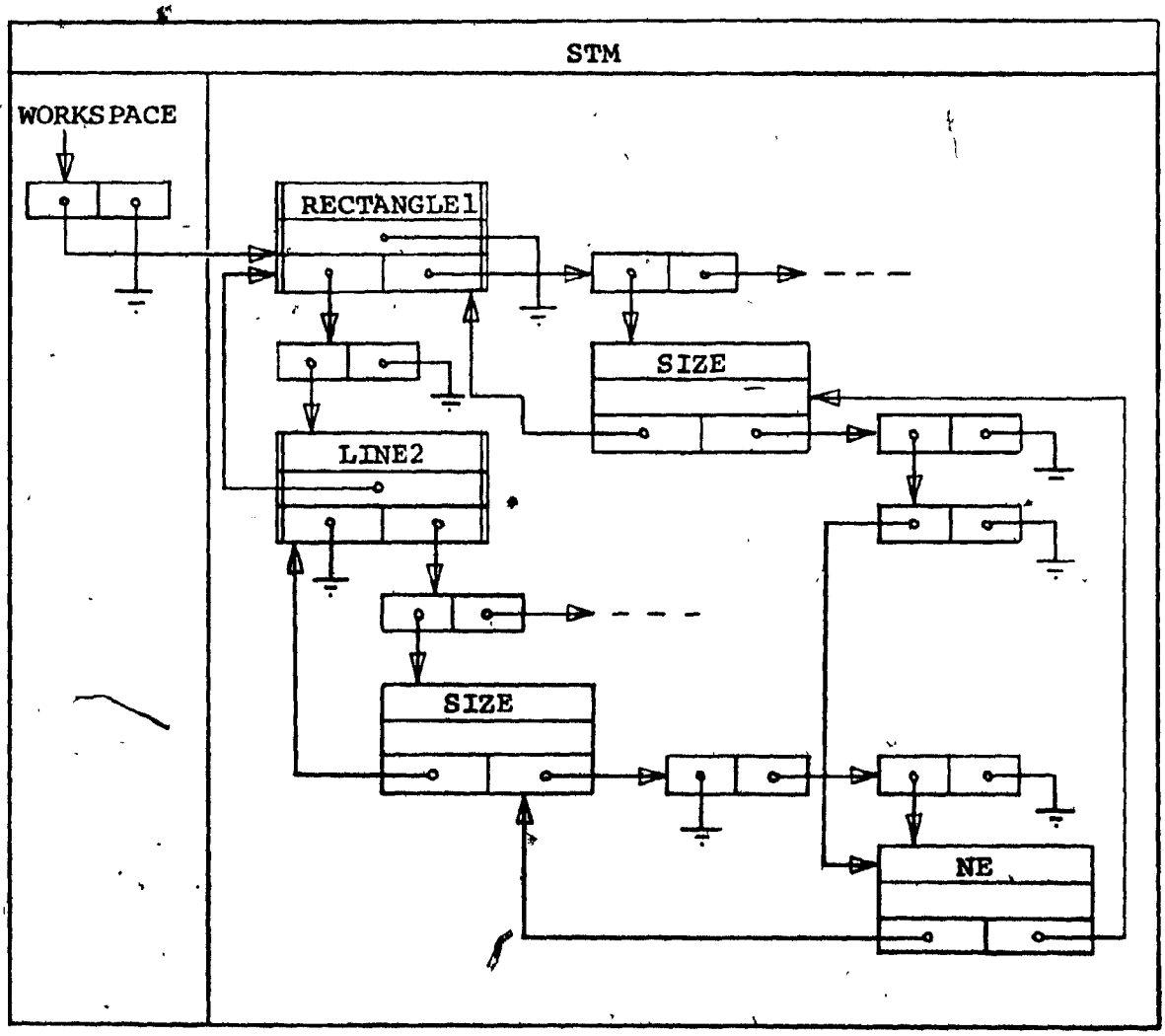


Fig. 3-5: STM resulting from the PL statements
DRAW(RECTANGLE1)
LOGIC(NE(SIZE(LINE2)SIZE))

It can be seen from Fig. 3-5 that the PARENT field of LINE2 contains a pointer to RECTANGLE1. The purpose of this pointer is to allow the identification of the parent of an object given the object's location in the STM. This usage is in contrast to that in the semantic memory where the PARENT field is presently undefined.

This example concludes the description of the compilation of LOGIC statements. The next section considers the compilation of the last type of PL input, the TOPOL statement.

3.3.3 PL Statements of the TOPOL Type

The TOPOL statement defines a topological relationship between the two objects which are its arguments. Its position in a PL program must be such that its arguments have been previously defined by appropriate DRAW commands.

As was seen in Chapter 2, its syntax is specified by the expression TOPOL(⟨toprel⟩(arg_1 arg_2)).[1] In this form, the TOPOL statement does not yield useful information on the features of its arguments. What is required, then, is that the statement be translated into a set of predicates on these features. The method by which this is done is now described.

---

[1] arg_1, arg_2 are the objects for which the topological relationship is being defined. In BNF form, arg_1 and arg_2 are members of the set of strings defined by ⟨object*⟩{⟨numeral*⟩|"}.

A sufficient (but not necessary) condition for the TOPOL statement to be valid is that <u>each</u> of the contour vertices of its arguments also satisfy the topological relationship specified by the statement. Fortunately, it is an easy matter to determine appropriate constraints for the contour vertices. Thus, for the general statement of the previous paragraph, a sufficient set of constraints are as follows:

if ⟨toprel⟩ ::= LEFTOF	then $vx_i < vx_j$	$\vec{v}_i = (vx_i, vy_i)$ and $\vec{v}_j = (vx_j, vy_j)$ where $\vec{v}_i$ are the contour vertices of arg_1 and $\vec{v}_j$ are the contour vertices of arg_2.
if ⟨toprel⟩ ::= RIGHTOF	then $vx_i > vx_j$	
if ⟨toprel⟩ ::= ABOVE	then $vy_i > vy_j$	
if ⟨toprel⟩ ::= BELOW	then $vy_i < vy_j$	

In order to reduce the programming effort, this sufficient set of constraints is the one employed in the present implementation of the compiler.

It would therefore appear that when a TOPOL statement would be encountered, the compiler need only generate a suitable set of predicates on the contour vertices of the appropriate objects. These predicates could then be incorporated into the STM using the techniques described in the previous section.

Unfortunately, this is not possible because some of the contour vertices of the objects may be secondary features whose existence is unknown to the compiler. The processing of TOPOL

statements into the STM must therefore be delayed until all the pertinent information of the semantic net which includes predicates on all secondary features has been accessed and compiled. So that the compiler may "remember" the TOPOL statement, a temporary representation of its information is set up in the STM. For example, the PL program

```
DRAW(LINE1)
DRAW(LINE2)
TOPOL(ABOVE(LINE1 LINE2))
```

results in the structure of Fig. 3-6.

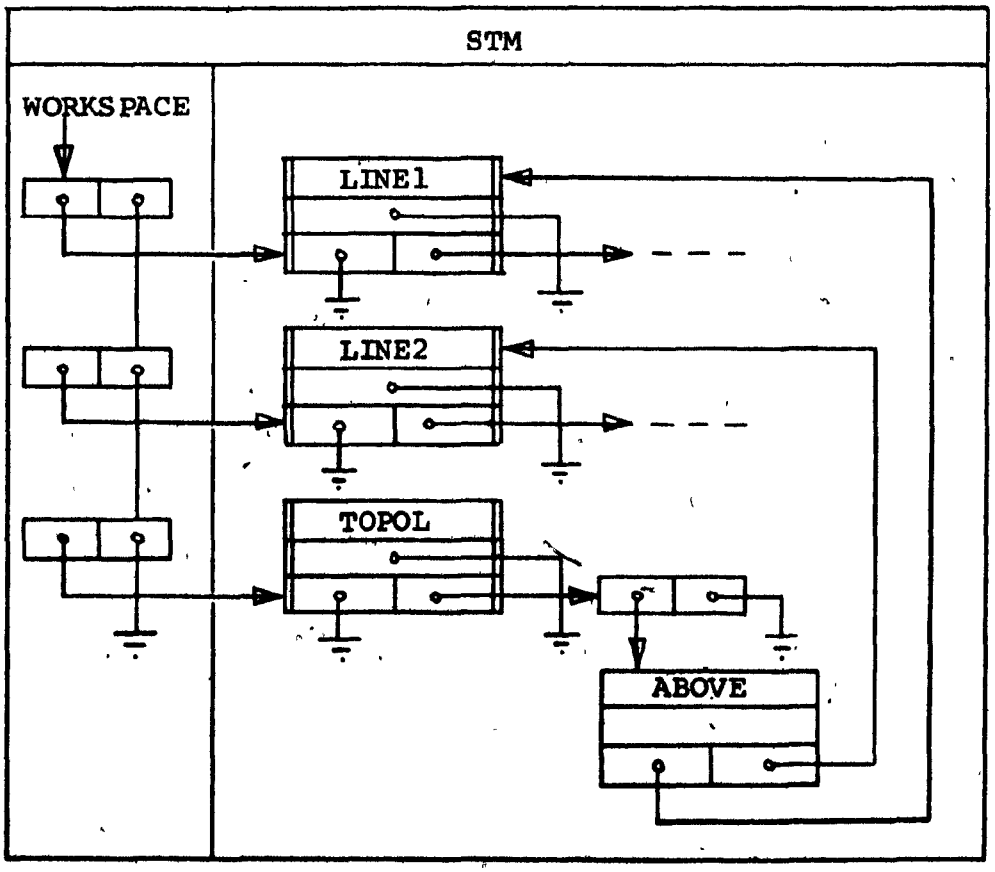


Fig. 3-6: Temporary representation in the STM of the statement TOPOL(ABOVE(LINE1 LINE2)).

Once all the pertinent predicates of the semantic net have been accessed and incorporated into the STM, the information of the TOPOL statement can be translated using the methods described. The resulting LOGIC statements which represent predicates on the contour vertex features of the arguments can then be easily integrated into the STM. Upon completion of this procedure, the topological relationships block is deleted from the structure of Fig. 3-2.

This section completes the description of the compilation of PL statements. The next sections discuss the algorithms for the retrieval of predicates from the semantic net and for their subsequent compilation into the STM.

3.4 Translation of Semantic Net Predicates into the STM

The following sections will describe the algorithms required for the transfer of information from the semantic memory to the STM. It is assumed that a PL input program has been successfully compiled. What is required is that the compiler access the pertinent predicates of the semantic net and incorporate their information into the STM. Upon completion of this task, the STM will contain all the information necessary to draw the objects specified by the PL program.

It was seen in Chapter 2 that all semantic net predicates can be represented by LOGIC statements. Furthermore, Section 3.3.2 described a procedure for the translation of such statements into the STM. Therefore, the procedure which shall be adopted will be the translation of semantic net predicates into LOGIC statements and then the translation of these into the STM using the methods of Section 3.3.2.

It can be seen from the flowchart of Fig. 3-1 that the translation procedure which will be described is repeated for each object defined in the STM by a DRAW command. The first step in the procedure is the localization of the object to be drawn in the semantic net. The syntax of the PL states that the name of an object in the STM is defined by the BNF string ⟨object*⟩⟨numeral*⟩ while its counterpart in the semantic net is defined by ⟨object*⟩.

Therefore, the compiler truncates ⟨numeral*⟩ from the object's name and compares the remaining string to that contained in the OBJECT node of each semantic plane. If a match is not found, the compiler assumes that the object is undefined and execution of the program is terminated.

Assuming that a match has been found, one might assume that the compiler need only generate PL strings from the semantic net predicates pertaining to the object to be drawn and incorporate these into the STM. Unfortunately, this is untrue because the strings generated in this manner do not sufficiently qualify the features of the object. Nevertheless, the translation of semantic net predicates into PL statements does constitute part of the solution. The next section describes the methodology of this translation procedure as well as its shortcomings.

3.4.1 Synthesizing PL Strings from Semantic Net Predicates

The translation of a semantic net predicate into a PL string must involve accessing its information by traversing its nodes in accordance with an algorithm. It was stated in Chapter 2 that each predicate may be modeled by a binary tree. Therefore, it is worth considering whether an algorithm for the traversal of binary trees may be of use. Knuth (1968) considers three such algorithms: preorder traversal, postorder traversal and endorder

traversal. Of these, preorder traversal is the most useful for the required application. It stipulates that the traversal of a binary tree is to be accomplished recursively in the following sequence:

(i) visit the root node

(ii) traverse the left subtree

(iii) traverse the right subtree

Thus the sequence for visiting the nodes of the tree of Fig. 3-7 is a b d h i e c f j k g.

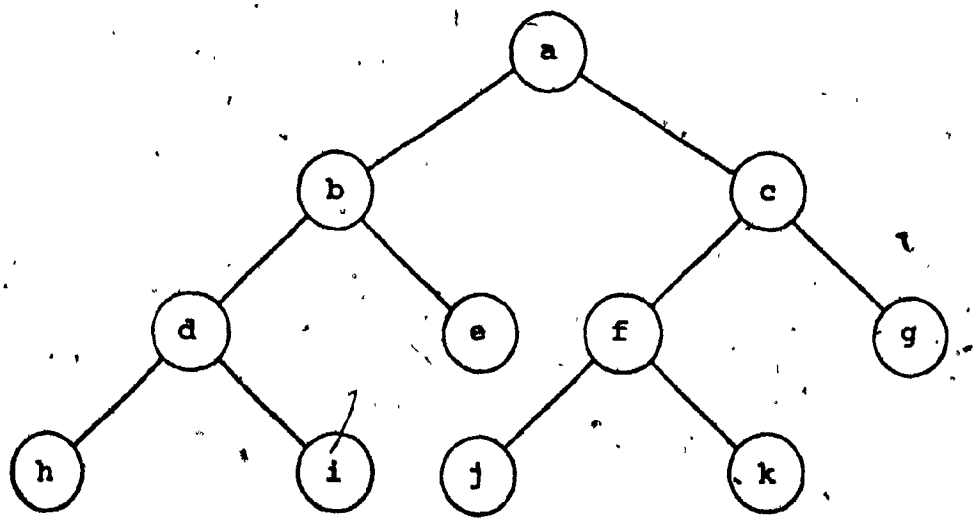


Fig. 3-7: A typical binary tree

The algorithm employed for the traversal of semantic net predicates uses preorder traversal as its basis. It will be derived using an illustrative example. Consider the predicate of Fig. 3-8. The traversal algorithm for this predicate must be capable of producing its PL representation. That is, it must be

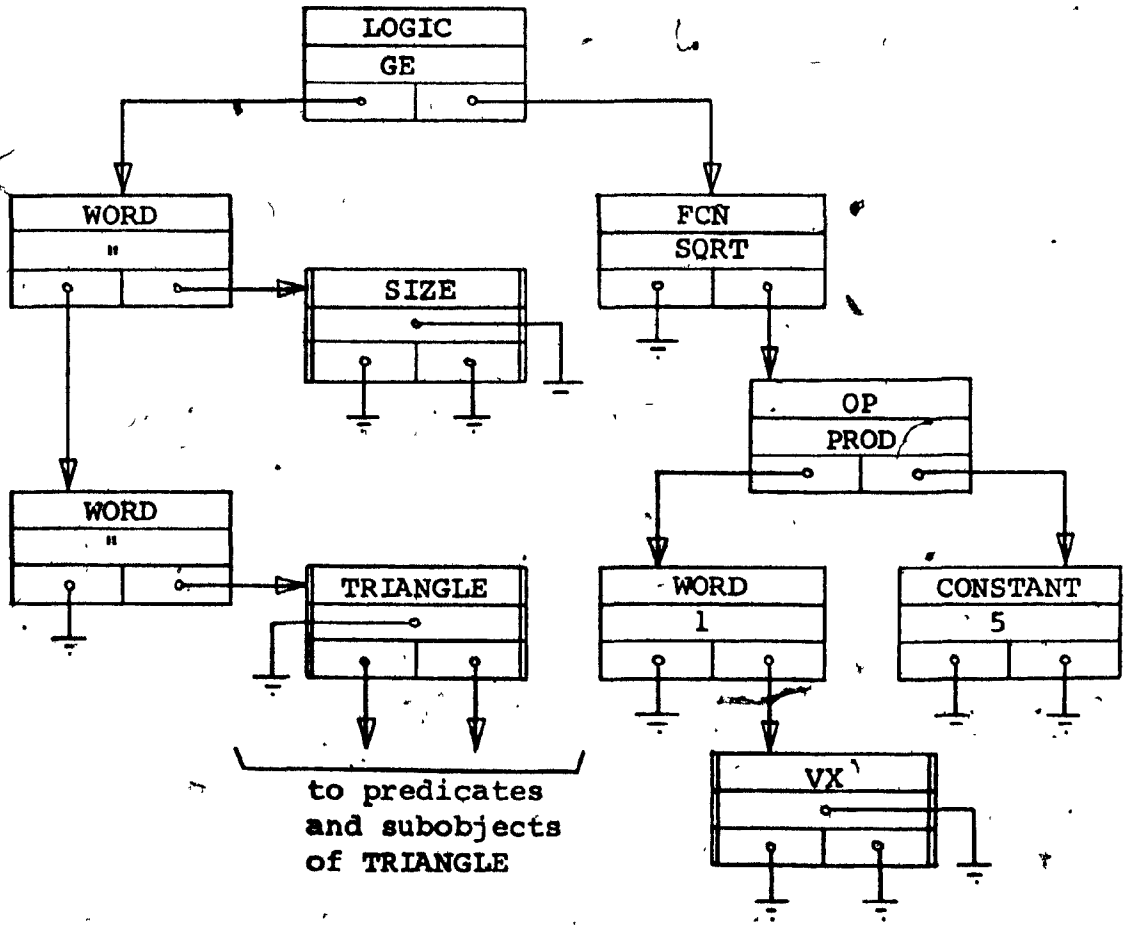


Fig. 3-8: A semantic net predicate.

capable of producing the string LOGIC(GE(SIZE(TRIANGLE)FCNSQRT (OPPROD(VX1 5)))).

Application of preorder traversal to the structure of Fig. 3-8 yields the following strings:

LOGIC, GE, WORD, ", WORD, ", SIZE, TRIANGLE, FCN, SQRT, OP, PROD, WORD, 1, VX, CONSTANT and 5.

In order to convert these strings to that which is required, the following changes must be made to the traversal algorithm:

1. Subtrees whose patriarch is a MODIFIER node containing the string WORD are to be traversed recursively using a "mirror image" postorder algorithm[1] defined by the following steps:

   (i) visit the right subtree

   (ii) visit the root node

   (iii) visit the left subtree

   Usage of this algorithm and removal of the strings WORD and CONSTANT results in:

   LOGIC, GE, SIZE, ", TRIANGLE, ", FCN, SQRT, OP, PROD, VX, 1 and 5.

2. Concatenation of the strings in OP-and FCN-MODIFIER nodes as well as the concatenation of those in OBJECT nodes to their immediate successor from the traversal algorithm yields LOGIC, GE, SIZE, TRIANGLE, FCNSQRT, OPPROD, VX1 and 5.

---

[1] For a more detailed discussion on binary tree traversal, see Knuth (1968), pp. 315-328.

3. Merging these strings and using brackets where appropriate[1] results in LOGIC(GE(SIZE(TRIANGLE)FCNSQRT(OPPROD(VX1 5)))) which is the required string.

The algorithm which has been described is the one used by the compiler in the synthesis of PL strings from semantic net predicates. Unfortunately, the strings generated in this manner are unsuitable for direct translation into the STM in the sense that they do not contain a sufficient amount of information. To see why this is so, consider the semantic net representation of the concept TRIANGLE. It can be seen from Fig. 2-18 that this concept is defined by the two semantic planes TRIANGLE and LINE and by the subobject link[2] between them. In incorporating the concept TRIANGLE into the STM, the algorithm of this section would first traverse the predicates of TRIANGLE and then those of LINE. However, a typical PL string resulting from the traversal of LINE would be

LOGIC(GT(SIZE 0)) - - - iii-(1)

Such a string is clearly inadequate for usage by the compiler because of its ambiguity. Indeed, the required set of strings is

---

[1] While it is not described here, a procedure for the insertion of brackets does exist and has been implemented in the present compiler.

[2] See Section 2.4.2.

LOGIC(GT(SIZE(LINE1(TRIANGLE) 0)) - - - iii-(2)

LOGIC(GT(SIZE(LINE2(TRIANGLE) 0)) - - - iii-(3)

LOGIC(GT(SIZE(LINE3(TRIANGLE) 0)) - - - iii-(4)

A comparison of these expressions demonstrates two shortcomings of the strings which are generated by the traversal algorithm. First, the qualification of features is insufficiently deep. For example, the PL string of iii-(1) is ambiguous insofar as it does not specify whether the feature SIZE pertains to TRIANGLE or to LINE. Second, the quantity of predicates specified is insufficient.

To resolve these inadequacies, an algorithm which resolves a string like iii-(1) into the strings iii-(2) to iii-(4) is required. Such an algorithm must employ hierarchical and quantitative information gleaned from the semantic memory. The hierarchical information would alleviate the first shortcoming discussed above while the quantitative information would alleviate the second.

A structure which contains the required information is the Subobject Tree. Its structure and the way it is used by the compiler are the topics of the next section.

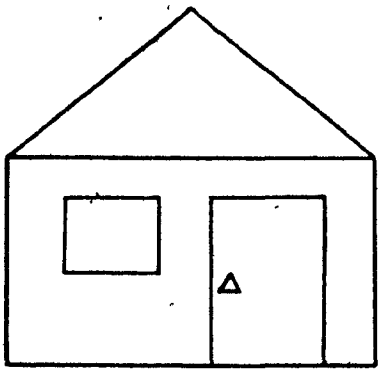
3.4.2 The Subobject Tree

The Subobject Tree is a temporary structure created in the "scratchpad" memory of the computer. Its purpose is to determine the extent to which a PL representation of a semantic net predicate must be modified so that it can be translated into the STM.

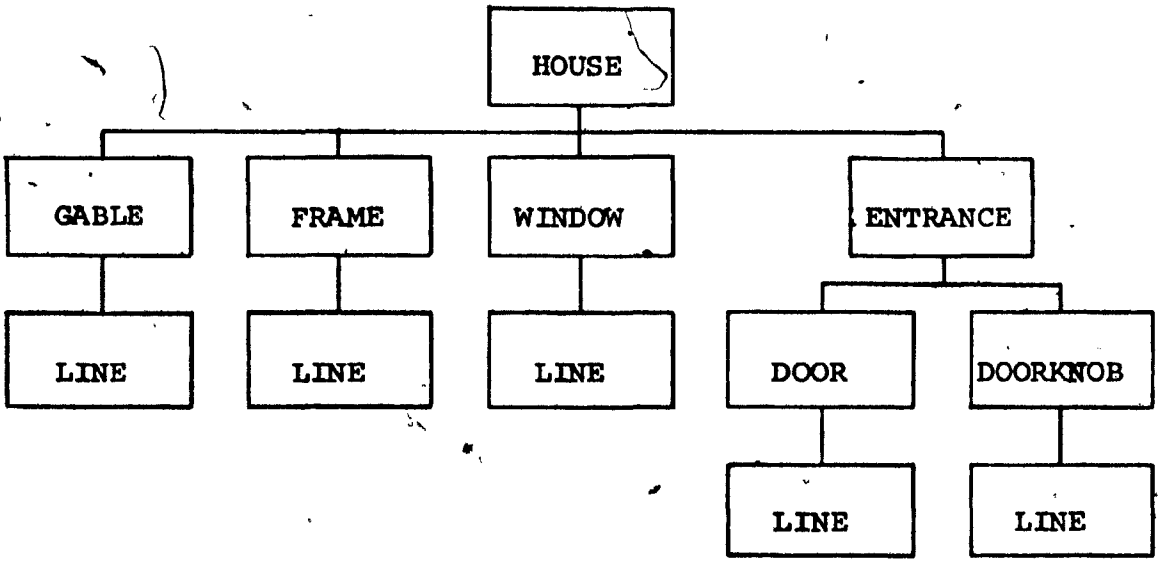
As can be seen from the flowchart in Fig. 3-1, a new Subobject Tree is created each time that the compiler first enters the semantic net to retrieve the predicates of an object to be drawn. Its structure is derived as a result of a breadth-first[1] traversal of the OBJECT nodes of all the semantic planes accessed by the subobject links in the plane of entry. This traversal proceeds outwards from plane to plane until the terminal planes which represent the primitive of the semantic net have been accessed. The resultant tree consists of nodes which represent the semantic planes visited and branches which represent the subobject links between them. Thus, the patriarch node of the tree represents the object to be drawn while the remaining nodes represent both its direct and indirect subobjects. For example, the Subobject Tree for a house might be structured as shown in Fig. 3-9.

---

[1] Breadth-first traversal involves visiting the nodes of a tree in order of their distance from the root node. Thus the root node is visited first. All nodes which are direct descendants of the root node are visited next. All nodes which are direct descendants of these nodes are visited next, and so on. For a discussion of this and other traversal techniques for trees, see Nilsson (1971), Chapter 3.



(a)



(b)

Fig. 3-9: Conceptual representation of a particular Subobject Tree.
(a) A line drawing for a house.
(b) The resulting Subobject Tree.
Note that a GABLE is in reality an isosceles triangle, FRAME, WINDOW and DOOR are rectangles and DOORKNOB is an equilateral triangle. This implies that the definition of, say, GABLE in the semantic memory requires the addition of only two nodes: an OBJECT node in the semantic net which contains the string GABLE as well as a set of pointers accessing the semantic plane for ISOSTRI, and a LIST node in the semantic map which accesses the OBJECT node.

The Subobject Tree defined by the author consists of nodes each of which contains four pieces of information. The first elements of each node is a string which represents the name of the object represented by the node. The second element is an integer which indicates the required quantity of the object[1,2]. The third and fourth elements of each node are pointers which link the node to the other nodes in the Subobject Tree. Thus, after all pertinent predicates have been accessed, the tree for the house of Fig. 3-9 is as shown in Fig. 3-10.

Let us examine the method by which the compiler creates the Subobject Tree. It was stated earlier in this section that a new Subobject Tree is formed each time that the compiler first enters the semantic net to retrieve the predicates of an object to be drawn. Upon entry, the compiler threads through the set of OBJECT nodes representing all the subobjects accessed either directly or indirectly by the plane of entry. In doing this, it creates the nodes of the Subobject Tree, assigns a name to each node, and links them together in the manner shown by the example of Fig. 3-10.

---

[1] Referring to Fig. 3-9(b), the LINE node beneath GABLE would contain the integer 3 to signify that three lines are required to draw a gable.

[2] The patriarch node of the Subobject Tree is an exception. Its second element contains an integer which references the instance of the object specified by the STM. Thus, if the object currently referenced in the STM is HOUSE6, then the node will contain the integer "6".

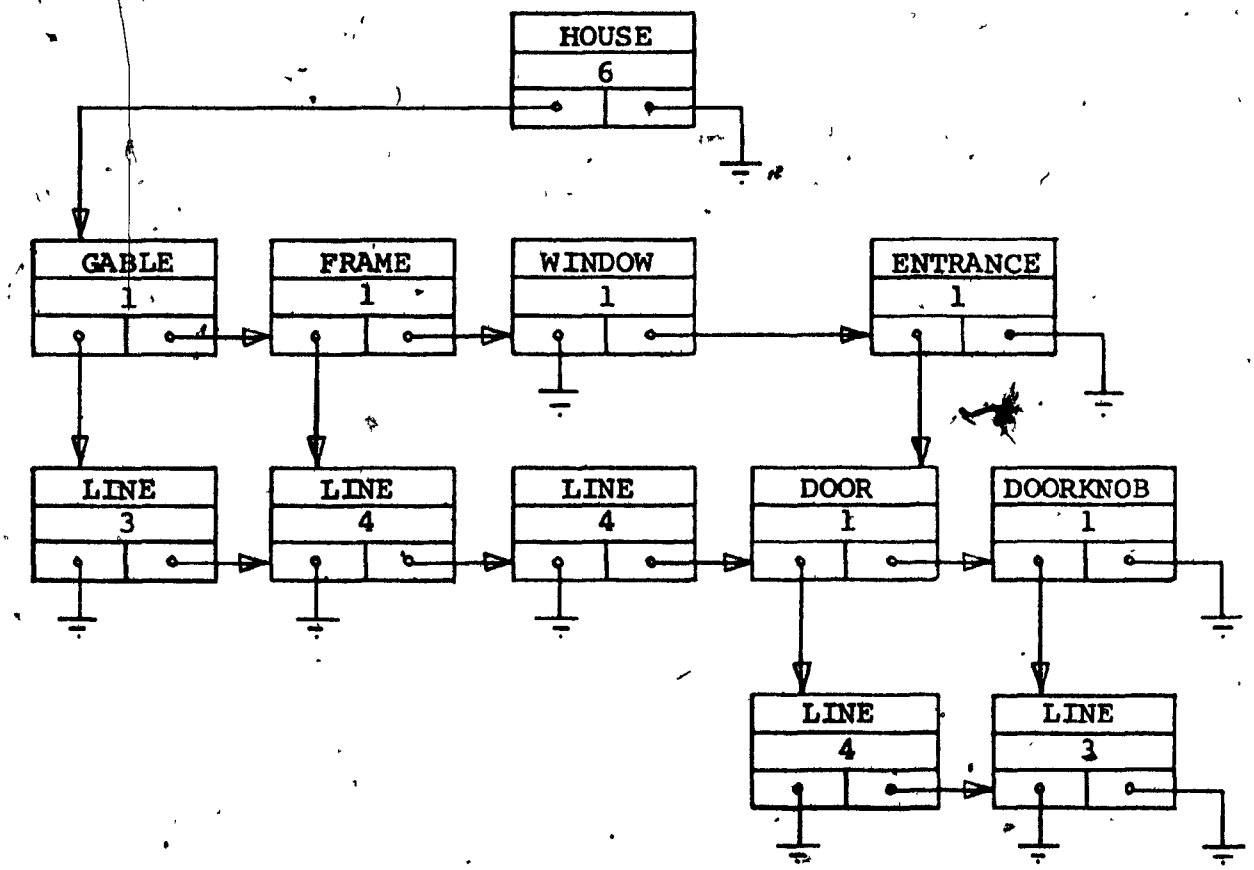


Fig. 3-10: The Subobject Tree for HOUSE6. The house itself is drawn in Fig. 3-9(a).

But the OBJECT nodes of the semantic memory contain no information about the required quantity of the object. Therefore, the second element of each node of the tree is initially undefined. This missing information must be obtained from the predicates of the semantic net.

To prove that the missing information resides within the pre-

dicates, consider any semantic plane that represents an object which is not a primitive. This plane will contain two types of predicates. The first type will yield conditions on the features of the object and as such is not pertinent here. However, the second type will yield conditions on the features of its subobjects. This is useful because it informs the compiler about the existence of these subobjects. Now the predicates of each semantic plane must mention each direct subobject at least once. If this were not true, it would mean that the compiler would have no way of recognizing some subobject and its existence would remain unknown. Such a subobject would be useless and could therefore be disregarded. This argument implies that once a semantic plane has been traversed and its predicates incorporated into the STM, the compiler can obtain the required quantity of each direct subobject by scanning the contents of the STM. Furthermore, this procedure can be repeated for each node of the Subobject Tree. In this manner, the second element of each node of the tree can be defined before the compiler enters the semantic plane of that subobject.

Now that the Subobject Tree has been defined, its utility can be explained. It is employed to resolve PL strings like that of iii-(1) into the strings iii-(2) to iii-(4) and so complete the translation of semantic net predicates into strings suitable for incorporation into the STM. Thus, each feature of a PL string can be qualified by the name of the object to which that feature

pertains as well as by the names of the object's parents as determined in the Subobject Tree. The second element of each pertinent node of the tree (except for the patriarch) then determines the number of strings which must be generated for the object represented by that node. Thus, in the example of the previous section, the string LOGIC(GE(SIZE 0)) becomes LOGIC(GE(SIZE(LINE) 0)). Furthermore, because there are three lines in a triangle, the LINE node of the Subobject Tree will contain the integer "3" and the above PL statement becomes the required strings iii-(2) to iii-(4).

This section has shown how the Subobject Tree is formed and how it is used to modify predicates from the semantic net to a form amenable for translation into the STM. It can now be assumed that the STM contains all the information necessary to draw a set of objects. The sections that follow will discuss the algorithms used for the computation of the feature values of the objects to be drawn.

3.5 Computation of Feature Values

The compiler implemented in this thesis has been shown to be adequate for the performance of the tasks described in the previous sections of this chapter. The sections that follow will discuss the algorithms used for both the simplification of the STM and the selection of feature values. In explaining these algorithms, the sections will also demonstrate the inadequacies of the compiler.

3.5.1 Simplification of the STM

It has been shown in Chapter 2 that the translation of pertinent semantic net predicates into the STM produces a structure which contains all the information necessary to draw an object. It therefore follows that the additional incorporation of the predicates of an input PL program results in a structure which contains redundant information. This section presents an algorithm which partially checks the semantic validity[1] of all information encoded into the STM and which also eliminates some redundancies inherent in that information.

Before the simplification algorithm for the STM can be de-

---

[1] A statement is semantically valid if its meaning is consistent with that implied by any other statements which have already been defined. Thus, the PL statement LOGIC(GT(VX2 5)) is syntactically and semantically valid if taken by itself. If, however, it is stated in conjunction with the statement LOGIC(LT(VX2 3)), an inconsistency arises and one of the statements is assumed to be semantically invalid.

scribed , it is necessary to explain why it is only partially effective in the tasks it must perform.

The STM contains two types of predicates. The first type, called the numeric predicate, imposes a numeric bound on the value of a feature. The second type, called the symbolic predicate, stipulates that the value of a feature is functionally dependent upon the values of other features. The algorithm described in this section is only capable of checking the semantic validity of numeric predicates and eliminating those which are redundant. A compiler for an algorithm which would be capable of handling symbolic predicates would have to be able to perform symbolic manipulations on strings. For example, such a compiler would have to be capable of accepting conditions like

$$vx_2 < (vx_3)^2$$

$$vx_3 = 5$$

and subsequently deducing that

$$vx_2 < 25.$$

Because the purpose of this thesis is to demonstrate the inherent advantages of including a semantic memory in a graphics system, it was felt that the effort required for the incorporation of a symbolic compiler was unwarranted. The implications of using a symbolic compiler in conjunction with the semantic memory will be considered in Chapter 4.

Now that the class of allowable predicates has been defined, it is possible to describe the simplification algorithm. For each feature in the STM, the algorithm performs the following tasks:

(1) If no more than one numeric predicate is defined, then no action is performed on the predicates of that feature.

(2) If more than one numeric predicate is defined, then the following steps are performed for all possible pairs of numeric predicates[1]:

   (a) The semantic validity of each pair of predicates is tested (see Fig. 3-11).

   (b) If any inconsistency is detected, then an error message listing the inconsistent predicates is printed and program execution is terminated.

   (c) If no inconsistency is detected then any redundant predicate is eliminated from the STM.

Thus, the comparison of two numeric predicates as outlined in step (2) above causes one of three effects: termination of compilation due to inconsistency (T), simplification of the STM due to redundancy (S), or no action (N). The table presented in Fig. 3-11 yields a summary of the possible outcomes resulting from the comparison of any two numeric predicates.

---

[1] If n is the number of numeric predicates pertaining to a feature, then the possible number of pairs of these is ${}^{n}C_{2} = n(n-1)/2$.

a \ b	EQ	NE	LT	LE	GT	GE
EQ	T,Sb,T	Sb,T,Sb	Sb,T,T	Sb,Sb,T	T,T,Sb	T,Sb,Sb
NE	Sa,T,Sa	N,Sb,N	N,Sa,Sa	N,Sa*,Sa	Sa,Sa,N	Sa,Sa*,N
LT	T,T,Sa	Sb,Sb,N	Sb,Sb,Sa	Sb,Sb,Sa	T,T,N	T,T,N
LE	T,Sa,Sa	Sb,Sb*,N	Sb,Sa,Sa	Sb,Sb,Sa	T,T,N	T,Sb*,N
GT	Sa,T,T	N,Sb,Sb	N,T,T	N,T,T	Sa,Sb,Sb	Sa,Sb,Sb
GE	Sa,Sa,T	N,Sb*,Sb	N,T,T	N,Sb*,T	Sa,Sa,Sb	Sa,Sb,Sb

Fig. 3-11: Result matrix for the simplification algorithm. The rows represent the operand for the condition f (op) a while the columns represent the operand for the condition f (op) b (i.e. f is a feature, (op) is one of EQ,NE,LT,LE,GT,GE and a,b are numbers). For each element, the three entries represent the results of a comparison of the two conditions given that $a < b$, $a=b$ and $a > b$, respectively.
The letters N, T and S signify no action, termination and simplification respectively. The letter appended to each S-entry stipulates the predicate which is to be deleted from the STM as a result of a simplification. Thus, Sa states that the predicate symbolized by f (op) a is to be deleted, while Sb states that the predicate f (op) b is to be deleted. For example, the first entry of the element in the fourth row, third column specifies that: if the conditions $f \leq a$ and $f < b$ where $a < b$ exist, then $f < b$ is redundant and can be eliminated. The S-entries with an asterisk specify that besides the deletion of a predicate, further simplifications to the surviving predicate are possible. Thus, for $f \geq a$, $f \leq b$ and $a=b$, then $f \leq b$ can be eliminated and $f \geq a$ can be simplified to $f=a$. In such cases, all numeric predicates for the affected feature must be retested.

The simplification algorithm presented in this section has been used for the simplification of numeric predicates within the STM. It has been shown that the lack of a symbolic manipulation capability within the compiler restricts the versatility of the algorithm. The next section will discuss how this shortcoming in the compiler also affects the resultant structure of the semantic memory. It will also outline restrictions on the types of allowable statements in PL input programs.

3.5.2 Restrictions on the Structure of the Semantic Memory

This section discusses the implications to the semantic memory of the inability of the compiler to perform symbolic manipulations on strings. It will be shown that, as a result of this shortcoming, the resultant structure of the semantic memory becomes unnecessarily restricted.

The basic restriction is that the translation of both a PL input and the pertinent semantic net predicates into the STM should not result in the creation of a self-loop within the STM. To understand the meaning of the term "self-loop", consider a set of predicates (f_1, f_2, f_3) and the following constraints on them:

$$f_1 = f(f_2) = (f_2)^2 \qquad \text{- - - iii-(5)}$$

$$f_2 = g(f_3) = \cos f_3 \qquad \text{- - - iii-(6)}$$

$$f_3 = h(f_1) = f_1/2 \qquad \text{- - - iii-(7)}$$

This set of predicates contains a self-loop in the sense that no one variable can be interpreted as the independent variable. Thus,

$$f_1 = f(f_2) = f(g(f_3)) = f(g(h(f_1)))^1$$

The way in which these self-loops arise is now described.

Consider the semantic plane for a line. The predicates of this plane can be represented symbolically by the following conditions:

$0 < s \leq (100^2 + 50^2)^{\frac{1}{2}}$ - - - iii-(8)

$0 \leq \theta < 360$ (θ in degrees) - - - iii-(9)

$0 \leq vx_1, vx_2 \leq 100$ - - - iii-(10)

$0 \leq vy_1, vy_2 \leq 50$ - - - iii-(11)

$vx_2 = vx_1 + s \cos \theta$ - - - iii-(12)

$vy_2 = vy_1 + s \sin \theta$ - - - iii-(13)

The numbers 100 and 50 evident in iii-(8), (10) and (11) have been chosen so that the resultant line can fit inside a 100 point by 50 point raster field. It can be seen from the above that s, θ, vx_1 and vy_1 are the independent variables since their values are independent of the values of other features. On the other hand, vx_2 and vy_2 are dependent variables.

---

[1] In fact, this set of equations can be simplified to $f_1 = (\cos f_1/2)^2$ using symbolic manipulation. The further use of an iterative solution technique yields the values $f_1 = .835$, $f_2 = .914$ and $f_3 = .418$.

[2] Where $(s, \theta, vx_1, vx_2, vy_1, vy_2)$ represent (SIZE,ORIENT,VX1,VX2,VY1,VY2) respectively.

A PL input consisting of the command DRAW(LINE1) would result in the translation of the above predicates into the STM. Because of the careful manner in which these predicates have been stated in the semantic memory, no self-loops would be incorporated into the STM.

However, the addition of the PL statement LOGIC(EQ(VX1 VX2)) would result in the creation of a self-loop because vx_1 would then become a function of itself (i.e. $vx_1 = vx_2 = vx_1 + s \cos \theta$).

A possible solution to this problem in the absence of symbolic manipulation is to store many alternate sets of constraints within each plane. Each set of constraints would then represent the predicates necessary to specify an object given a particular set of independent variables. For example, the branch within the LINE plane of Fig. 3-12 would allow the choice of either vx_1 or vx_2 as the independent variable. Then, depending on whether a PL input program had set the value for vx_2 or not, the appropriate branch could be taken in the traversal of semantic net predicates to forestall the creation of a self-loop within the STM.

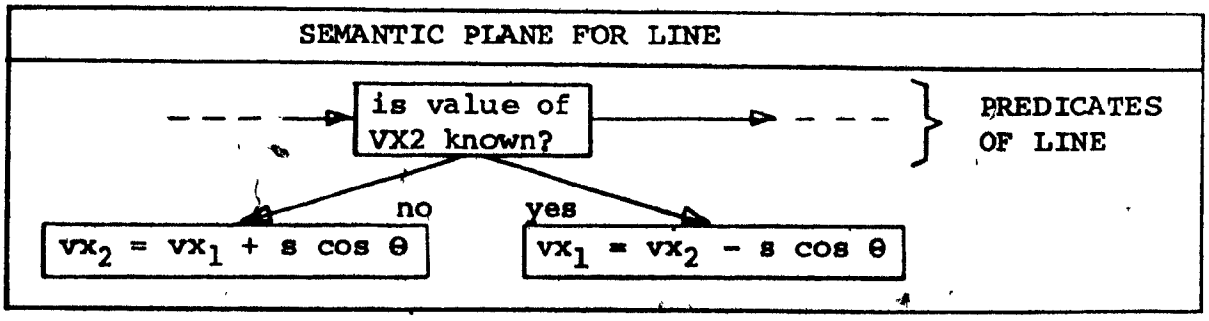


Fig. 3-13: Portion of semantic plane for LINE showing alternate paths for the stipulation of predicates.

Except for very limited applications, this is not a feasible solution since the resulting semantic memory would soon become too unwieldy to be useful. To see why this is so, consider a semantic plane which represents an object having n features $f_1, f_2, \ldots, f_n$. Of these features, some will represent independent variables whose value can be chosen at random from a range of acceptable values while others will represent dependent variables. If, however, PL inputs which change the set of independent variables are allowed, then the semantic plane must contain alternate paths which allow the set of independent variables to be as small or as large as is required.

For even a small number of features, the number of these "decision paths" required is too large. Thus, two paths would exist for the feature f_1. Each of these would then split into two paths for f_2 resulting in 2 x 2 = 4 paths. For n features, then, the resulting "decision paths" would produce a binary tree having 2^n leaves[1] each of which could represent a set of predicates on the object.

Even for LINE which is a primitive, this methodology would create a tree having $2^6 = 32$ alternate paths. Clearly, this is an unwieldy structure which would destroy the concept that the semantic memory is an inherently compact structure.

---

[1] A leaf is a terminal node of a tree.

It was therefore decided that, unless a portion of an "alternate path" structure was required by the semantic memory itself, such a structure would not be implemented. In fact, one such "alternate path" is required in the semantic plane for RECTANGLE. The details of the self-loop which would otherwise occur in this plane will not be discussed. However, the implementation of this "alternate path" within the semantic plane results in the usage of the KNOWN mode of the MODIFIER node mentioned in Section 2.4.3. The type of structure required in the semantic memory for the definition of an alternate path is shown in Fig. 3-13.

The decision to restrict alternate paths in the semantic memory also results in a restriction on the types of allowable PL inputs to the compiler in that only those which specify a predicate on an independent feature are allowed.

This section completes the discussion of the restrictions imposed on the structure of the semantic memory and the content of the PL due to the inadequacies of the compiler. The next section will discuss the Feature Value Selection Algorithm.

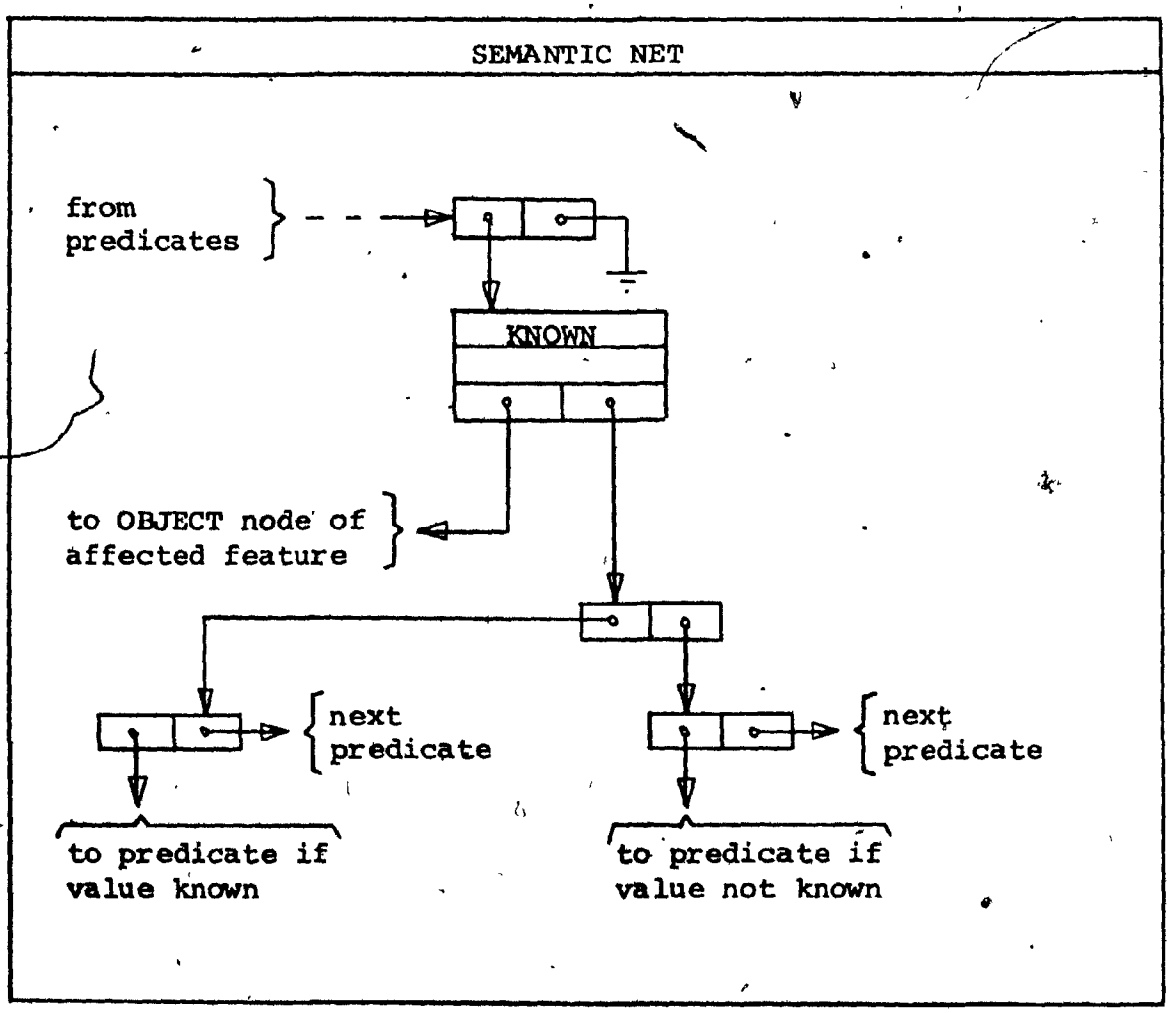


Fig. 3-13: Usage of the KNOWN mode of the MODIFIER node.

3.5.3 The Feature Value Selection Algorithm

The Feature Value Selection Algorithm (FVSA) implemented in this thesis was selected with due consideration to the inadequacies of the compiler. This section constitutes a brief description of the operation of the algorithm. Chapter 4 will discuss a more powerful algorithm which could have been implemented had a symbolic compiler been available.

For the discussion that follows, it is assumed that the structure of the STM has been simplified to the extent possible by the simplification algorithm of Section 3.5.1. The remaining tasks are that a consistent set of feature values be selected for the objects to be drawn and that a picture of these objects be produced. The latter task is a trivial one which will be discussed in Chapter 4. On the other hand, the former task which is executed by the FVSA is not so trivial. This algorithm is now described.

The first step in the FVSA is the traversal of the STM and the subsequent creation of a stack, each element of which represents the feature of an object in the STM. The method of traversal of the STM is unimportant as is the ordering of the elements in the stack. What is important is that all the features of the objects in the STM be represented in the stack. Each stack element does not name the affected feature but rather contains a pointer to the MODIFIER node in the STM which identifies that feature. In this way, the stack not only accesses the feature but also all the pre-

dicates on it as well as all the Value Dependence Pointers (VDP's)[1] which emanate from it.

The next step in the FVSA is the ordering of the features in the stack so that those which represent independent variables are placed "higher up" in the stack than those which represent dependent variables. If the features in the STM contain any self-loops which were discussed in the previous section, then this step of the algorithm detects these and execution is terminated.

The feature ordering is accomplished by using the VDP's appended to each feature. It was stated earlier in this chapter that such pointers are set up in the STM from the independent variables to the dependent variables. As a result, each VDP from a feature x to a feature y stipulates that x is an independent variable relative to y or that $x \prec y$ (where "$x \prec y$" is read as "x precedes y"). Because of this, each element in the stack can be used to access the VDP's which emanate from the feature x_i represented by that element. A set of precedence relations $x_i \prec y_1$, $x_i \prec y_2$,..., $x_i \prec y_n$ can then be defined. Once these precedence relations have been defined, it is a relatively simple matter to order the elements in the stack in the desired manner. Knuth (1968)[2] defines a generalized "Topological Sort" algorithm which accomplishes this

---

[1] The VDP was discussed in Section 3.3.2.

[2] See Knuth (1968), Section 2.3.3 on Linked Allocation.

task. This algorithm will not be described here. The interested reader is referred to the pertinent section of Knuth's text for a description.

For the last step of the FVSA, it is assumed that an ordered feature stack as described above has been successfully produced. What is required, then, is that suitable values be determined for all features. If the intent of this work were to produce a practical interactive graphics language, then a generalized algorithm for the performance of this task would have been defined. Because the intent of this project was not so ambitious, a more modest solution to this problem was implemented.

It was stated in Chapter 2 that the semantic memory provides sufficient information to limit the value of each feature so that its value may be chosen from a finite interval. The FVSA therefore performs the following task for each element in the feature stack in order of its "independence" as determined in the previous step. Each predicate on that feature is translated into a "software arithmetic unit" within the compiler. This arithmetic unit computes a numeric limit on the value of that feature from the predicate and stores this number in the TYPE field of the MODIFIER node in the STM which heads that predicate. Thus, for the predicate LOGIC(LT(VX2 5)) in the STM, the FVSA produces the result shown in Fig. 3-14. Once all the predicates on that feature have

been evaluated, a suitable feature value which satisfies all the predicates can be randomly chosen. This value can then be used in the evaluation of predicates of subsequent features which are dependent variables.

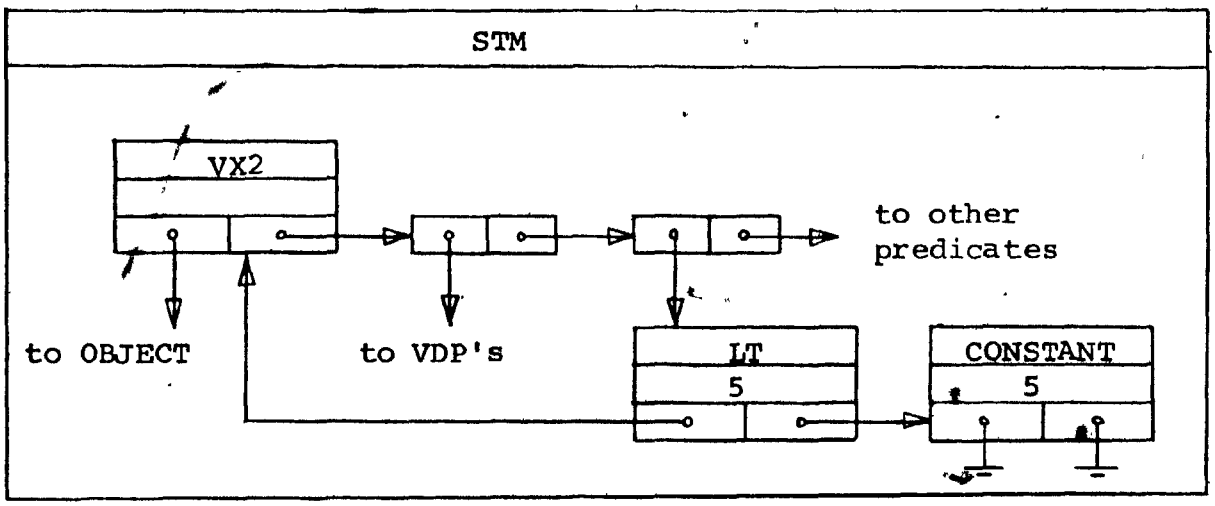


Fig. 3-14: Determination of numeric bound on a feature. Note the number "5" which has been encoded into the TYPE field of the MODIFIER-LT node.

The shortcoming of this approach is that if the value of some independent variable has been poorly chosen, then the calculated value for a subsequent dependent variable may be outside of the allowable limits for that variable. The solution to this problem as implemented in this thesis was to make the third step of the algorithm iterative. That is, if the computed value of some variable were outside acceptable limits, then the value of an independent variable of which it was a function could be chosen again. Hopefully, such a choice would produce an acceptable

value for the dependent variable. Such a procedure is clumsy at best since it is time consuming and since it can result in the creation of infinite loops during program execution. However, the procedure was found to work without too much difficulty for the simple objects defined in this thesis. A more powerful algorithm based on the usage of an optimization technique is discussed in Chapter 4.

It can now be assumed that the FVSA has computed appropriate values for the features of all the objects to be drawn. The next chapter discusses the method used to draw these objects as well as some sample PL programs which were run using the compiler. The chapter also discusses modifications which could have been implemented in the system described in these chapters.

Chapter 4

Results and Conclusions

4.1 Overview

The previous two chapters have described the methods by which the system[1] implemented by the author is used to determine a consistent set of feature values for objects specified by a PL program. The following is an outline of the topics discussed in this, the final chapter.

The first part of this chapter describes the method by which the system outputs line drawings. It also discusses the results obtained from the compilation of six selected PL programs. Each program demonstrates some features or shortcomings of the system.

The second part of this chapter discusses some possible modifications to the system. Emphasis is placed on those changes which would alleviate the shortcomings of the compiler discussed in the previous chapter.

---

[1] In this chapter, the word "system" will refer to the compiler, semantic memory, STM and all associated data structures implemented in this thesis.

4.2 Output Procedure and Results

4.2.1 Output Procedure

Chapter 3 has described a procedure for the creation of an STM which contains the predicates on the features of objects to be drawn. The chapter has also described the algorithms used by the compiler for the determination of a consistent set of values for these features. What is now required is a procedure which draws pictures of the objects defined in the STM.

It was stated in Chapter 2 that the features of an object define a redundant set. In fact, only the contour vertex features of the straight line segments which comprise each object are required to draw that object. Therefore, the output procedure which has been implemented extracts the contour vertex pairs $(\vec{V}_1, \vec{V}_2)$[1] of each line segment defined within the STM and uses these pairs to produce a line drawing on a digital plotter.

The next section describes some results which were obtained using the system discussed in this thesis.

---

[1] Recall that each line consists of the six features SIZE, ORIENT, VX1, VY1, VX2 and VY2. The notation $\vec{V}_i$ represents the features VXi and VYi.

4.2.2 Results

The semantic memory, compiler and STM discussed in this thesis were implemented in PL/1 on an IBM 360/75 Digital Computer and a CALCOMP 663 Digital Incremental Plotter. This section describes six PL programs which were executed using this system. For each example, a description of the PL program and the resultant picture are given as are the program execution time and core requirements.[1] It will become apparent in these examples that the execution time and core requirements of each program are quite large. However, it must be remembered that the system implemented was not meant to be a practical interactive graphics system. Consequently, little effort was expended in improving its operating efficiency. The modifications described later in this chapter would serve to alleviate this problem. Nevertheless, the execution time and core requirements obtained are useful in that they illustrate the relative complexity of the selected programs. The results of the six examples are now described.

The first two examples were selected to demonstrate the effort required to draw the simplest and the most complex objects defined in the semantic memory. In the first example, the PL

---

[1] Core requirements include the storage area required for the compiler, the semantic memory, the STM, the PL program and all associated temporary structures such as the Subobject Tree.

program consisted of the single command DRAW(LINE). Both the input program and the PL representation of the derived semantic net predicates are shown in Fig. 4-1 while the feature stack[1] both before and after ordering is shown in Fig. 4-2. The resultant STM after execution of the Feature Value Selection Algorithm is depicted in Fig. 4-3. Execution of the program required 4.1 seconds of Central Processing Unit (CPU) time and 162K bytes[2] of core. The resultant line drawn by the plotter is shown in Fig. 4-4(a).

The PL program of the second example consisted of the single command DRAW(HOUSE). Execution of the program required 45.1 seconds of CPU time and 196K bytes of core. The resultant house drawn by the plotter is shown in Fig. 4-4(b).

A comparison of these examples shows that the amount of core required by the system is quite large and does not appear to be very dependent on the complexity of the object to be drawn. This is due to the fact that a large portion of core required is expended for "fixed overhead" items such as the storage of the compiler and the semantic memory. This problem is not serious because core requirements could be substantially reduced by pro-

---

[1] The feature stack was defined in Section 3.5.3.

[2] A byte of core is equal to eight bits (binary digits).

```
INPUT PROGRAM

   1  DRAW(LINE)

SEMANTIC NET CONDITIONS

   1  LOGIC(GT(SIZE(LINE1) 0))
   2  LOGIC(LE(SIZE(LINE1) FCNSQRT(OPSUM(FCNSQR(100)FCNSQR(50)))))
   3  LOGIC(GE(ORIENT(LINE1) 0))
   4  LOGIC(LT(ORIENT(LINE1) 360))
   5  LOGIC(EQ(VX2(LINE1) OPSUM(VX1(LINE1) OPPROD(SIZE(LINE1) FCNCOS(ORIENT(LINE1)))))
      )
   6  LOGIC(EQ(VY2(LINE1) OPSUM(VY1(LINE1) OPPROD(SIZE(LINE1) FCNSIN(ORIENT(LINE1)))))
      )
   7  LOGIC(GE(VX2(LINE1) 0))
   8  LOGIC(LE(VX2(LINE1) 100))
   9  LOGIC(GE(VY2(LINE1) 0))
  10  LOGIC(LE(VY2(LINE1) 50))
  11  LOGIC(GE(VX1(LINE1) 0))
  12  LOGIC(LE(VX1(LINE1) 100))
  13  LOGIC(GE(VY1(LINE1) 0))
  14  LOGIC(LE(VY1(LINE1) 50))
```

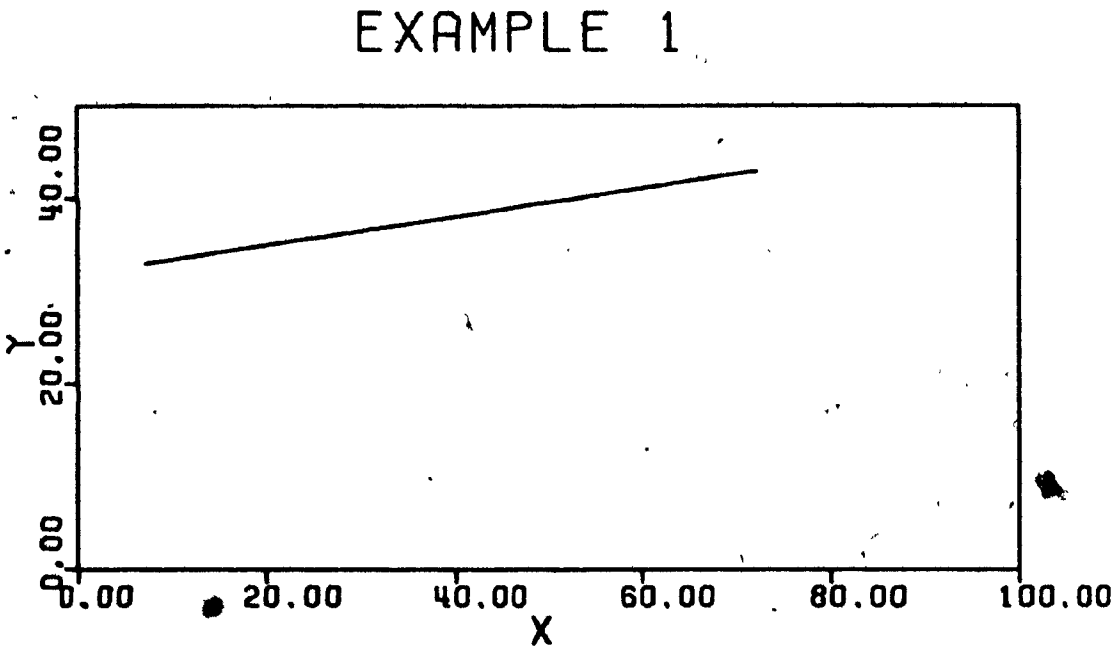
Fig. 4-1: A listing of both the PL input program and the pertinent semantic net predicates of Example 1.

	INITIAL ORDER			FINAL ORDER	
1	VY2	LINE1	1	SIZE	LINE1
2	VX2	LINE1	2	ORIENT	LINE1
3	SIZE	LINE1	3	VY1	LINE1
4	ORIENT	LINE1	4	VY2	LINE1
5	VX1	LINE1	5	VX1	LINE1
6	VY1	LINE1	6	VX2	LINE1

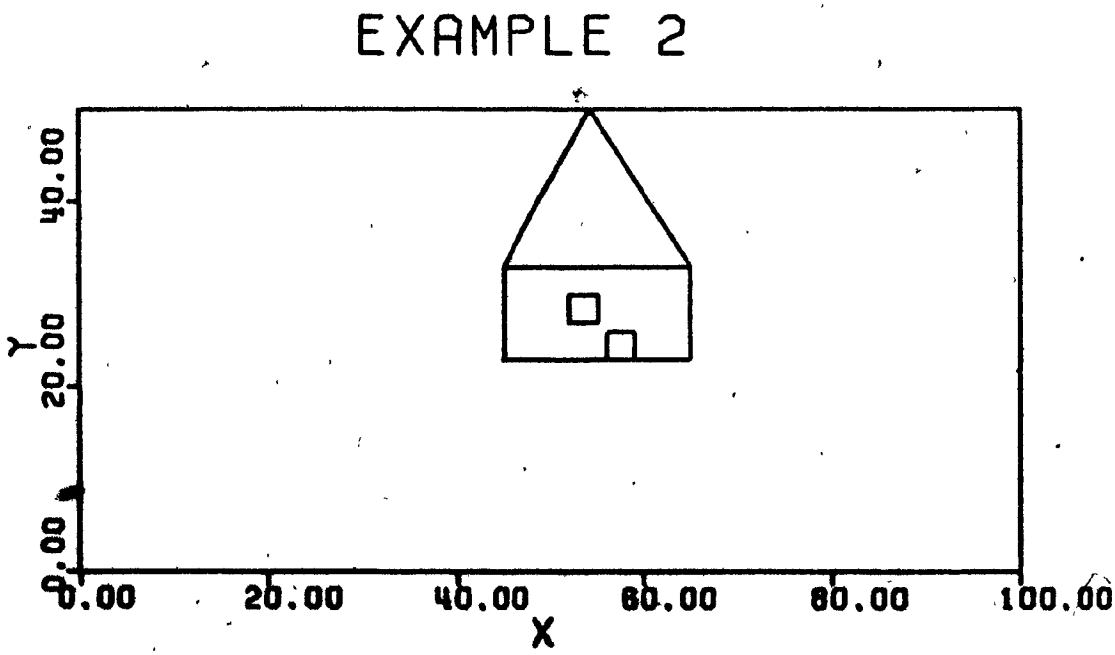
Fig. 4-2: The initial and final order of the feature stack of Example 1.

```
OBJECT=LINE1        PARENT=NULL          SUBFIG=NULL

        FEATURE=VY2          PARENT=LINE1        CONDS=LIST          VALUE=33
                LOGIC=LE             LEFT=VY2            RIGHT=50
                LOGIC=GE             LEFT=VY2            RIGHT=0
                LOGIC=EQ             LEFT=VY2            RIGHT=33

        FEATURE=VX2          PARENT=LINE1        CONDS=LIST          VALUE=7
                LOGIC=LE             LEFT=VX2            RIGHT=100
                LOGIC=GE             LEFT=VX2            RIGHT=0
                LOGIC=EQ             LEFT=VX2            RIGHT=7

        FEATURE=SIZE         PARENT=LINE1        CONDS=LIST          VALUE=65
             BACKWARD POINTER TO->VX2            LINE1
             BACKWARD POINTER TO->VY2            LINE1
                LOGIC=LE             LEFT=SIZE           RIGHT=111
                LOGIC=GT             LEFT=SIZE           RIGHT=0

        FEATURE=ORIENT       PARENT=LINE1        CONDS=LIST          VALUE=188
             BACKWARD POINTER TO->VX2            LINE1
             BACKWARD POINTER TO->VY2            LINE1
                LOGIC=LT             LEFT=ORIENT         RIGHT=360
                LOGIC=GE             LEFT=ORIENT         RIGHT=0

        FEATURE=VX1          PARENT=LINE1        CONDS=LIST          VALUE=72
             BACKWARD POINTER TO->VX2            LINE1
                LOGIC=LE             LEFT=VX1            RIGHT=100
                LOGIC=GE             LEFT=VX1            RIGHT=0

        FEATURE=VY1          PARENT=LINE1        CONDS=LIST          VALUE=43
             BACKWARD POINTER TO->VY2            LINE1
                LOGIC=LE             LEFT=VY1            RIGHT=50
                LOGIC=GE             LEFT=VY1            RIGHT=0
```

Fig. 4-3: The structure of the STM of Example 1 after execution of the FVSA.



(a)



(b)

Fig. 4-4: Digital Plotter output for Examples 1 and 2.

gramming the system in a low-level assembler language rather than in PL/1 and by eliminating the usage of character strings to encode information into the semantic memory and the STM.

The above examples also show that the increase in the execution time required to draw a house instead of a line is more than tenfold. In part, this occurs because the greater complexity of a house in relation to a line requires the retrieval of more predicates from the semantic memory, the creation of a more complex STM, and the determination of a greater number of features. However, part of this increase can be attributed to inefficiencies of the Feature Value Selection Algorithm. An alternative approach to feature value selection which would result in a more efficient algorithm will be described later in this chapter.

The third and fourth examples were chosen to demonstrate the effort required to draw an object given two different ways of specifying it. In both cases, the object drawn was an equilateral triangle. The "high-level" specification of the object in Example 3 resulted in the following PL program:

```
DRAW(EQUILTRI)-
LOGIC(EQ(ORIENT 0))
```

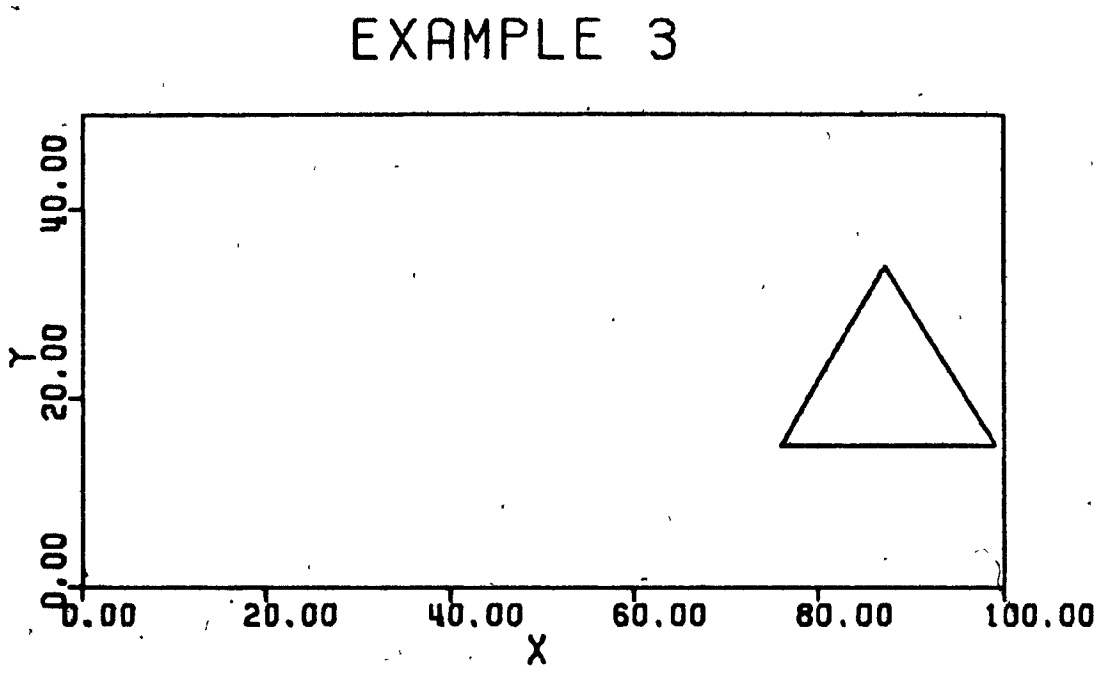
Execution of this program required 10.3 seconds of CPU time and 168K bytes of core. The resultant triangle drawn by the plotter is shown in Fig. 4-5(a).

The "low-level" specification of the object in Example 4 resulted in the following PL program:

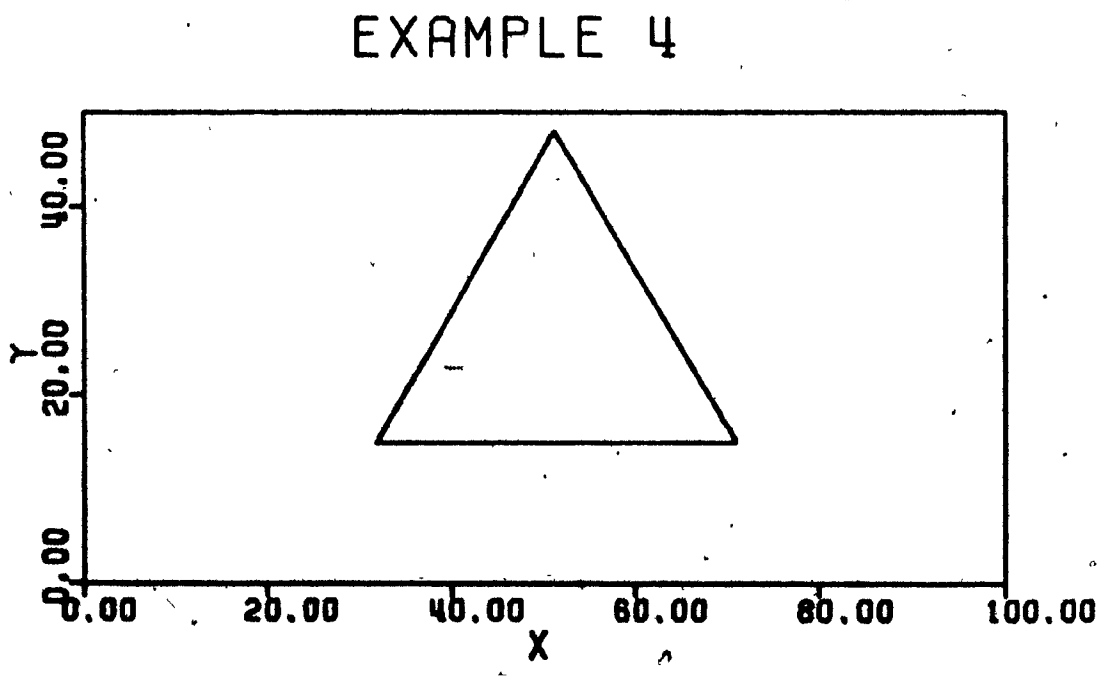
```
DRAW(LINE1)
LOGIC(EQ(ORIENT 0))
DRAW(LINE2)
LOGIC(EQ(SIZE SIZE(LINE1)))
LOGIC(EQ(ORIENT OPSUM(ORIENT(LINE1) 60)))
LOGIC(EQ(VX1 VX1(LINE1)))
LOGIC(EQ(VY1 VY1(LINE1)))
DRAW(LINE3)
LOGIC(EQ(SIZE SIZE(LINE1)))
LOGIC(EQ(ORIENT OPSUM(ORIENT(LINE1) 120)))
LOGIC(EQ(VX1 VX2(LINE1)))
LOGIC(EQ(VY1 VY2(LINE1)))
```

Execution of this program required 7.4 seconds of CPU time and 166K bytes of core. The resultant triangle drawn by the plotter is shown in Fig. 4-5(b).

It can be seen from these two examples that a "low-level" specification of an object results in a slightly more efficient program in terms of execution time and core requirements than a "high-level" specification of that object. Unfortunately, this increased efficiency can only be obtained by specifying an object in terms of its subobjects and the relationships between them. For even the simple case of a triangle, the above examples show that this is too great a price to pay. The extra effort required to specify a PL program for an object using a "low-level" approach more than offsets any advantage gained by improved efficiency of execution.



(a)



(b)

Fig. 4-5: Digital Plotter output for Examples 3 and 4.

The last two examples were selected to demonstrate the types of pictures which can easily be drawn by the present implementation of the system. Fig. 4-6(a) depicts two equilateral triangles of equal size and orientation joined at one vertex. The PL program required to draw these was:

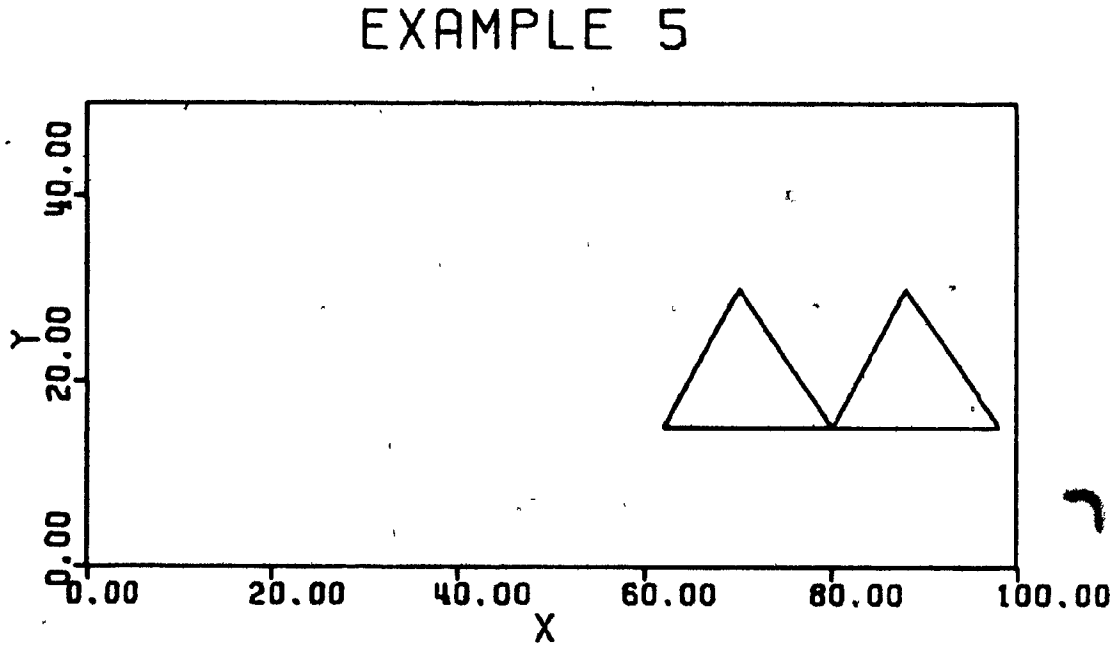
```
DRAW(EQUILTRI1)
LOGIC(EQ(ORIENT 0))
DRAW(EQUILTRI2)
LOGIC(EQ(ORIENT 0))
LOGIC(EQ(SIZE SIZE(EQUILTRI1)))
LOGIC(EQ(VX1 VX2(EQUILTRI1)))
LOGIC(EQ(VY1 VY1(EQUILTRI1)))
```

Execution of this program required 19.1 seconds of CPU time and 174K bytes of core.

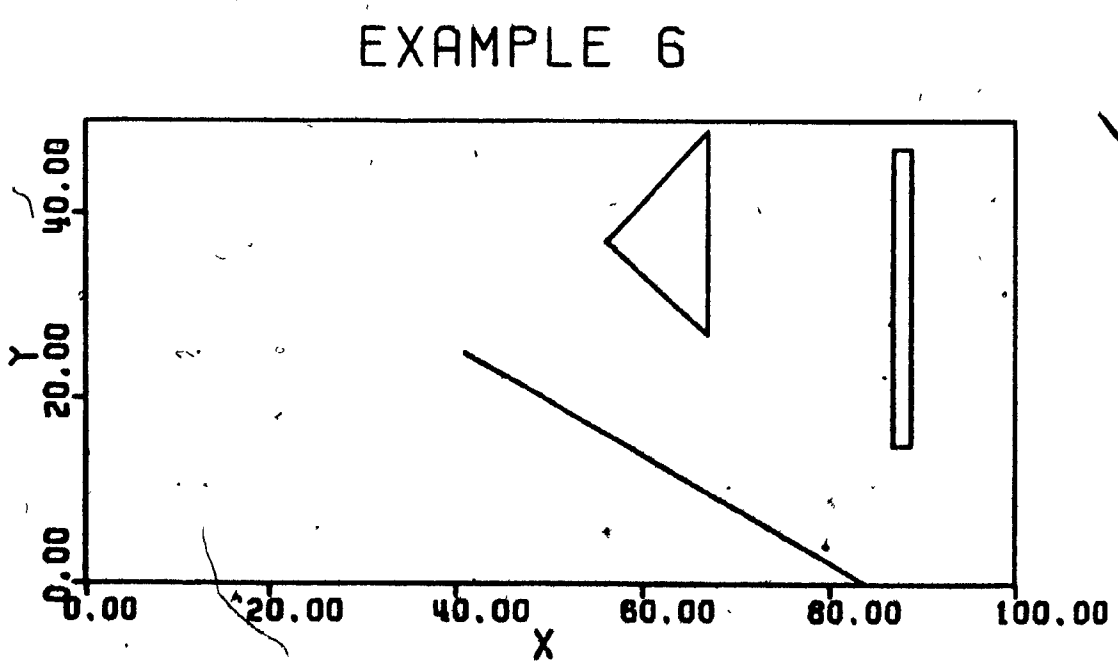
The final example demonstrates the usage of the TOPOL statement in defining a set of topological relationships between objects. The PL program executed consisted of the following statements:

```
DRAW(ISOSTRI1)
LOGIC(LT(SIZE 80))
LOGIC(GT(SIZE 30))
LOGIC(EQ(ORIENT 90))
DRAW(LINE1)
TOPOL(BELOW(LINE1 ISOSTRI1))
DRAW(RECTANGLE1)
LOGIC(GT(SIZE 25))
LOGIC(LT(SIZE 100))
LOGIC(EQ(ORIENT 0))
TOPOL(RIGHTOF(RECTANGLE1 ISOSTRI1))
```

Execution of this program required 22.3 seconds of CPU time and 178K bytes of core. The resultant picture drawn by the plotter is shown in Fig. 4-6(b).



(a)



(b)

Fig. 4-6: Digital Plotter output for Examples 5 and 6.

The above six examples have demonstrated both the flexibility and drawbacks of the present system. The remaining sections of this chapter will briefly propose modifications which would greatly enhance its flexibility and eliminate its drawbacks.

4.3 System Modifications

The discussions of the previous chapters have detailed the structure and operation of the author's system. In so doing, they have also drawn attention to its shortcomings. In the pages that follow, several improvements to the system will be proposed. The practicality of some of these improvements is dependent upon the availability of a symbolic compiler, while other improvements could be implemented on the present system. The aim in proposing these changes will be to demonstrate the feasibility of incorporating a modified version of the present system into the software of an interactive graphics display system.

4.3.1 Implications of a Symbolic Compiler

A symbolic compiler is one which is capable of accepting expressions containing variables and operators, and of subsequently manipulating those variables in accordance with the constraints defined by the operators. One of the capabilities of such a compiler would be to solve a group of expressions for any variable or variables. For example, given the expressions

$$ax + by = c$$

and $y < d + 5$

the compiler could solve for x and produce the expression

$$x > (c - b(d + 5)) / a$$

The implications of including a symbolic compiler in the

present system will be discussed below. It will be shown that the usage of such a compiler permits a greater versatility in the types of statements allowed in a PL program, simplifies the structure of the semantic memory and STM, affords a check for the semantic validity of predicates, and allows the usage of a more powerful Feature Value Selection Algorithm (FVSA).

It was stated in Chapter 3 that the method of representing predicates within the STM partitions the features into two types: independent and dependent. By definition, an independent feature is one whose value is restricted or determined only by numeric predicates, while a dependent feature is one whose value is restricted or determined by both symbolic and numeric predicates.[1] To state this concept mathematically, let $y_1, y_2, \ldots, y_m$ be independent features and let $y_{m+1}, y_{m+2}, \ldots, y_n$ be dependent features. Furthermore, let $\langle op \rangle$ represent a member of the set of operators $\{=, \neq, >, \geq, <, \leq\}$. By definition, the predicates on the independent features are defined by the constraints

$$\left.\begin{array}{l} y_1 \langle op \rangle a_1 \\ \quad - - - \\ y_m \langle op \rangle a_r \end{array}\right\} \quad \text{where } a_1 \text{ to } a_r \text{ are numbers.}$$

On the other hand, the predicates on the dependent features are

---

[1] Numeric and symbolic predicates were defined in Section 3.5.1.

defined by the constraints

$$\left.\begin{array}{l} y_{m+1} \langle op \rangle f_1(y_1, y_2, \ldots, y_m) \\ y_{m+2} \langle op \rangle f_i(y_1, y_2, \ldots, y_m, y_{m+1}) \\ \quad - - - \\ y_n \langle op \rangle f_j(y_1, y_2, \ldots, y_m, \ldots, y_{n-1}) \end{array}\right\}$$

where j is the no. of constraints on the dependent features and $1 \leq i \leq j$.

Note that $j \geq n - m$ because there may be more than one constraint assigned to any feature.

It has already been shown that, in the absence of a symbolic compiler, such a partitioning scheme imposes severe restrictions on the capabilities of the system. This is because such a scheme precludes the migration of members of one class of feature into the other class. For example, the predicate $vx_2 = vx_1 + s \cos \theta$ of LINE makes vx_2 a dependent variable and vx_1, s and θ independent with respect to it. A PL input such as LOGIC(EQ(VX2(LINE)~5)) cannot therefore be allowed since it could produce a value for vx_2 inconsistent with that derived from the predicate.

A method of circumventing this difficulty by using "alternate paths" was outlined in Section 3.5.2. Besides being clumsy and inefficient, the unrestricted usage of such a method would quickly destroy the effectiveness of the semantic memory.

The availability of a symbolic compiler, however, forestalls this difficulty without increasing the complexity of the semantic memory. Indeed, the inclusion of such a compiler into the

present system would eliminate the need for any feature partitioning. This is because all features would then become variables without regard to their being either dependent or independent. The constraints on the feature set $\{y_1, y_2, \ldots, y_n\}$ of an object would then become

$$\left.\begin{array}{l} f_1(y_1, y_2, \ldots, y_n) \langle op \rangle 0 \\ f_2(y_1, y_2, \ldots, y_n) \langle op \rangle 0 \\ \quad - \quad - \quad - \\ f_j(y_1, y_2, \ldots, y_n) \langle op \rangle 0 \end{array}\right\} \begin{array}{l} \text{where } y_1, \ldots, y_n \text{ are features} \\ f_1, \ldots, f_n \text{ are predicates} \\ \text{and } \langle op \rangle \text{ is a member of the} \\ \text{set } \{=, \neq, >, \geq, <, \leq\}. \end{array}$$

Unlike the compiler of the present system, a symbolic compiler would be capable of manipulating both symbolic and numeric predicates. It could therefore be programmed to delete redundant information from the predicates f_1 to f_j. In so doing, it could also check their semantic validity. For example, let the predicates on an object be

$$f_1 = y_1 + y_2 = 0 \qquad \text{iv-(1)}$$

$$f_2 = y_1 - y_2 = 0 \qquad \text{iv-(2)}$$

$$f_3 = y_2 - 5 = 0 \qquad \text{iv-(3)}$$

In simplifying these constraints, the compiler would substitute 5 for y_2 in iv-(1) and iv-(2) and would subsequently obtain the inconsistent conditions

$$f_1 = y_1 + 5 = 0$$

and $f_2 = y_1 - 5 = 0$.

Thus, one implication of a symbolic compiler is that it could check the semantic validity of a set of predicates.

Furthermore, if a symbolic compiler were available, the inclusion of a PL input into a set of predicates would serve to modify them. For example, consider the predicates of LINE which were listed in Section 3.5.2. Translation of these into the present notation yields

$$f_1 = s > 0$$

$$f_2 = s - (100^2 + 50^2)^{1/2} \leq 0$$

$$f_3 = \theta \geq 0$$

$$f_4 = \theta - 360 < 0$$

$$f_5 = vx_1 \geq 0$$

$$f_6 = vx_1 - 100 \leq 0$$

$$f_7 = vy_1 \geq 0$$

$$f_8 = vy_1 - 50 \leq 0$$

$$f_9 = vx_2 \geq 0$$

$$f_{10} = vx_2 - 100 \leq 0$$

$$f_{11} = vy_2 \geq 0$$

$$f_{12} = vy_2 - 50 \leq 0$$

$$f_{13} = vx_2 - vx_1 - s \cos\theta = 0$$

$$f_{14} = vy_2 - vy_1 - s \sin\theta = 0.$$

The addition of the PL statement LOGIC(EQ(VY2 VX2)) would precipitate the following simplifications to the above constraints.

The PL input would be translated into the constraint

$$f_{15} = vy_2 - vx_2 = 0.$$

By using this equality constraint, the compiler could then eliminate vy_2 from f_1 to f_{14}. This would cause f_{11} and f_{12} to become

$$f_{11} = vx_2 \geq 0$$

$$f_{12} = vx_2 - 50 \leq 0.$$

Since f_{11} would then be identical to f_9, it could be deleted. A comparison of f_{10} and f_{12} using the table of Fig. 3-11 would result in the deletion of f_{10}. Next, f_{14} would be replaced by

$$f_{14} = vx_2 - vy_1 - s \sin \theta = 0.$$

Finally, evaluating for vx_2 in f_{14} and substituting for f_{13} would eliminate f_{14} and yield

$$f_{13} + vy_1 - vx_1 + s(\sin \theta - \cos \theta) = 0.$$

In the above paragraphs, it has been shown that the inclusion of a symbolic compiler would eliminate the necessity of differentiating between dependent and independent features. It would also permit PL inputs specifying restrictions on any feature without necessitating the usage of "decision paths" within the semantic memory. This would eliminate the usage of the KNOWN mode of the MODIFIER node described in Section 3.5.2. It will now be demonstrated that the structure of the STM and the nature of the FVSA would both be altered by the inclusion of a symbolic compiler.

The requirement that the compiler be able to create an ordered feature stack to be used by the FVSA results in the incorporation of VDP's into the STM. Since a symbolic compiler would eliminate the classification of features into either dependent or independent types, the VDP would become redundant and it could be deleted from the STM. In addition, a symbolic compiler would permit other changes to the structure of the STM. This would occur because each predicate would no longer pertain to only one feature but rather to all features. The structure of the STM which would result appears in Fig. 4-7 and can be compared to that of the present version which appears in Fig. 3-2. The main difference between the two structures is that in the new one, the predicates are grouped together in the PREDICATE block and do not contain VDP's; in the old STM, they were accessed through the OBJECT node to which they pertained.

As an example of the structural differences between the two structures, consider the PL input

```
DRAW(TRIANGLE2)
LOGIC(LE(VX3 OPSUM(VY2 5))).
```

In the absence of a symbolic compiler, the STM which would result from this input was drawn in Fig. 3-4. Using the modifications defined in this chapter, the predicate contained in the PL string can be represented by

$$f_1 = vx_3 - vy_2 - 5 \leq 0.$$

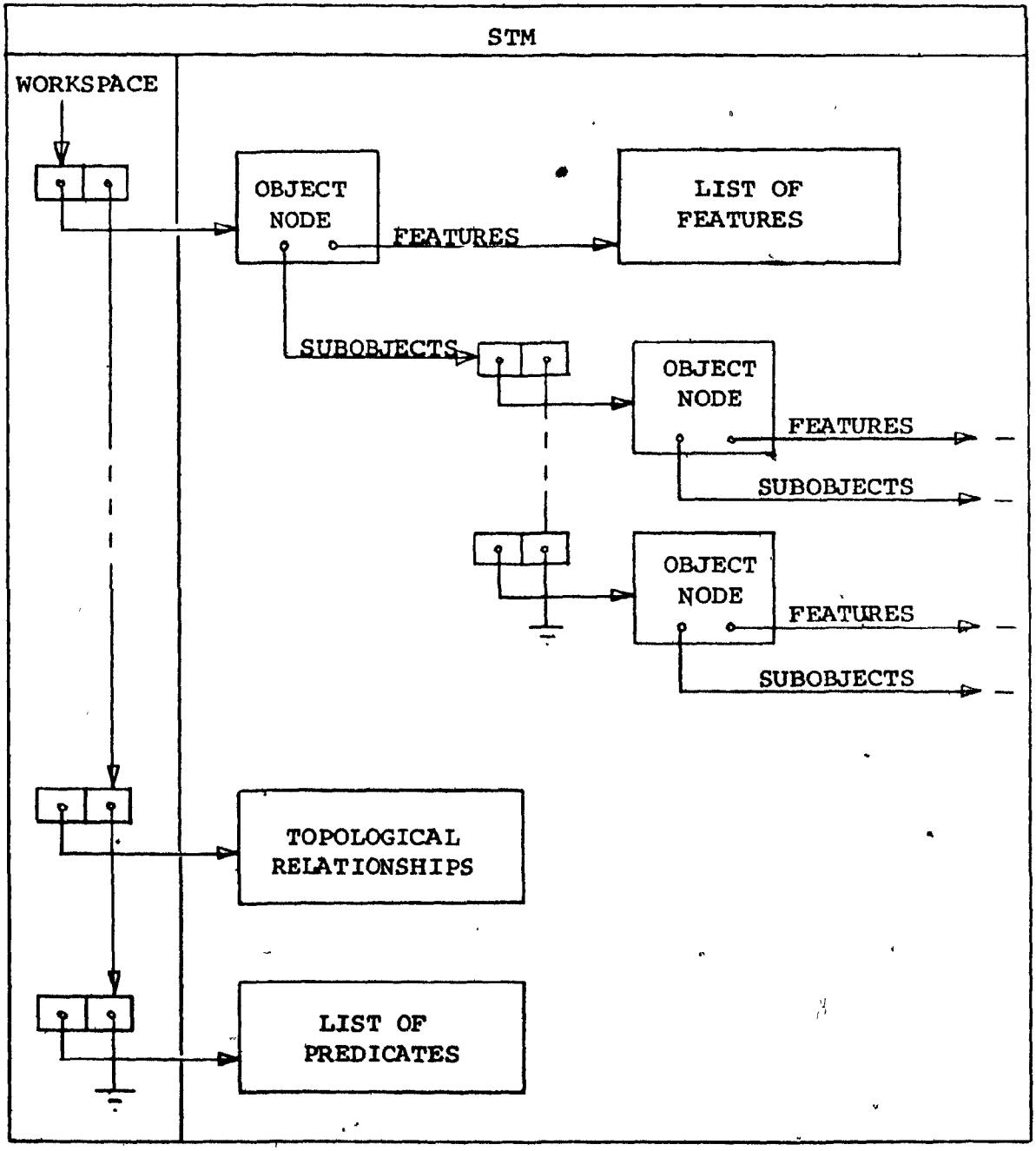


Fig. 4-7: Overview of the structure of the STM which would result if a symbolic compiler were available.

The translation of this input into the new STM would result in the structure of Fig. 4-8. Note that because the right hand side of the constraint f_i is always zero, the PTR2 field of the predicate's patriarch MODIFIER node is set to NULL. Note also the lack of a VDP in Fig. 4-8.

It has been shown that the usage of a symbolic compiler eliminates the need for a VDP and results in modifications to the structure of the STM. The remainder of this section discusses the type of FVSA which would be required by a symbolic compiler and how it would differ from that discussed in Chapter 3.

The discussions of this section have shown that the predicates on an unpartitioned set of features can be represented by the constraints

$$\left.\begin{array}{l} f_1(y_1,y_2,\ldots,y_n) \langle op \rangle 0 \\ \\ f_j(y_1,y_2,\ldots,y_n) \langle op \rangle 0 \end{array}\right\} \quad \text{iv-(4)}$$

What is required, then, is an algorithm which yields a solution vector for the feature set subject to the above constraints.

In his discussion of the Nonlinear Programming Problem, Pierre (1969) describes some techniques which are used for the solution of a similar problem. Specifically, he considers the problem of minimizing the nonlinear performance measure

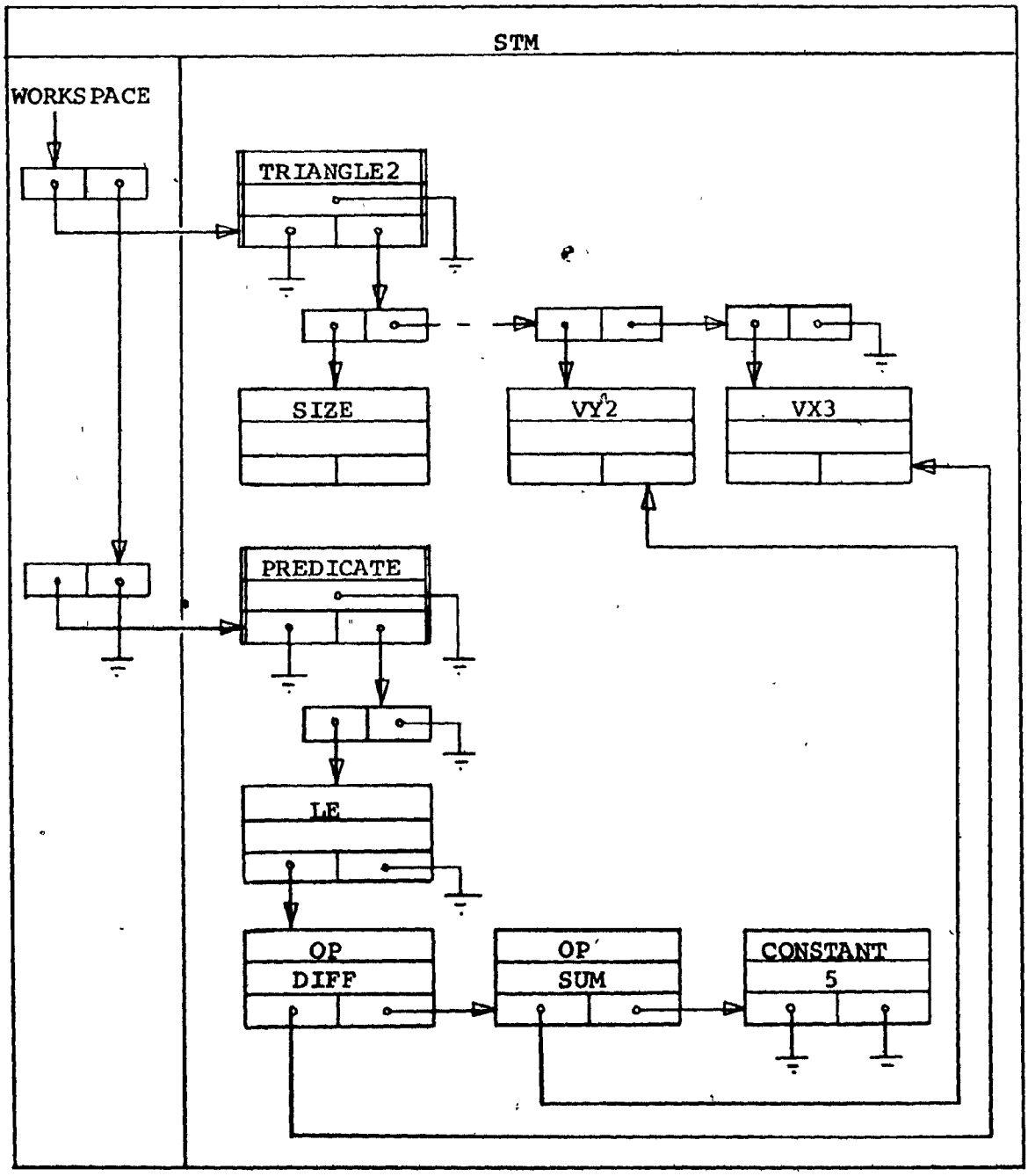


Fig. 4-8: STM representation of
DRAW(TRIANGLE2)
LOGIC(LE(VX3 OPSUM(VY2 5)))
assuming availability of a symbolic compiler.

$$P = f_o(\vec{X})$$

where $\vec{X}$ represents the n variables of the problem subject to the constraint equations

$$f_i(\vec{X}) = c_i \qquad i=1,2,\ldots,k \quad n \qquad \text{iv-(5)}$$

$$\text{and } f_i(\vec{X}) \leq c_i \qquad i=k+1,k+2,\ldots,m \qquad \text{iv-(6)}$$

According to Pierre, a powerful approach to the solution of such a problem is the addition of the penalty functions $p_i(\vec{X})$ to the performance measure P. For each constraint f_i, a new penalty function $p_i(\vec{X})$ is defined. This penalty function increases the value of P as a function of the amount by which the value of $\vec{X}$ is outside the limits specified by the constraint f_i. The solution to the Nonlinear Programming Problem then rests on the minimization of the Penalized Performance Measure

$$P_p = f_o(\vec{X}) + \sum_{i=1}^{m} w_i p_i(\vec{X})$$

where the w_i are suitably chosen weighting factors.

It is a simple matter to formulate the problem detailed by iv-(4) in terms of the Nonlinear Programming Problem. Since our objective is only to produce a set of feature values which satisfy the constraints $f_1,\ldots,f_j$, the performance measure $P = f_o(\vec{X})$ can be set equal to zero. What remains, then, is to transform the constraints of iv-(4) into those of iv-(5) and iv-(6).

For constraints where <op> is "=" or "$\leq$", a direct correspondence to those of iv-(5) and iv-(6) exists. Those constraints where <op> is "$\geq$" can be easily transformed by noting that if $f_i(y_1,\ldots,y_n) \geq 0$, then multiplying both sides by -1 results in $f_i' = -f_i \leq 0$ which is in the required form. Those constraints where <op> is "<" or ">" can also be easily transformed by noting that if $f_i < 0$, then $f_i + \delta \leq 0$ where δ is an arbitrarily small number. Finally, while the constraint $f_i \neq 0$ cannot be put into the required form, a penalty function for it can be incorporated into P_p by letting $p_i(y_1,\ldots,y_n) = 1 / f_i(y_1,\ldots,y_n)$.

It can be seen, then, that the formulation of iv-(4) as a Nonlinear Programming Problem is a simple matter. The subsequent solution of the constraint equations subject to the Penalized Performance Measure can be accomplished by any of a number of optimization techniques.[1]

The procedure outlined above represents a suitable FVSA for a system which includes a symbolic compiler. It differs from that described in Chapter 3 in that it seeks a viable "solution space" within the n-dimensional feature space without regard to which features are dependent variables or which are dependent. As such, it represents a much more powerful technique than that presented in Chapter 3.

---

[1] See Pierre (1969), Chapter 6 for several pertinent techniques.

The next section will briefly discuss an improvement to the system implemented in this thesis which would not require a symbolic compiler.

4.3.2 The DEFINE Statement

It has been stated in Chapter 1 that one of the properties of a human memory which must be possessed by an artificial intelligence if it is to be capable of human-like interaction with information is that it be teachable. This means that the artificial intelligence must be capable of receiving information, of determining the truth or falseness of that information, and of incorporating that information within itself if it is recognized as a truth.

It is obvious that the semantic memory presented in this thesis does not yet possess this capability. Instead, the a priori existence of the memory has been assumed and no procedures for augmenting or modifying its information by PL inputs have been provided. The incorporation of the DEFINE statement into the PL would be the first step in making the semantic memory teachable. Its function would be to inform the compiler that information regarding an object was to be incorporated into the semantic memory. The form of the DEFINE statement in the BNF notation would be

⟨define⟩ ::= DEFINE ⟨name⟩

where ⟨name⟩ can be any character string. Upon encountering a

DEFINE statement in a PL program, the compiler would search the semantic map to determine whether the object specified by <name> had already been defined in the net. If the object did not exist, the compiler would create appropriate nodes in the semantic memory to define it. If the object did exist, the compiler would access its semantic plane. In either case, any LOGIC statements following the DEFINE statement would describe possible inputs to the semantic memory. The techniques used for testing the semantic validity of these LOGIC statements and for incorporating them into the semantic memory constitute a complex procedure which will not be described here. Indeed, the implementation of such a procedure would be an appropriate subject for future research. It can be seen, however, that additional types of PL statements would be required in order to implement this procedure. For example, assume that the concept HOUSE were not yet included in the semantic memory. Then the incorporation of the predicates of HOUSE into the memory would require statements to inform the compiler that GABLE, say, was in fact an isosceles triangle, and that suitable links to reflect this fact should be set up between the GABLE and ISOSTRI semantic planes.

The incorporation of predicates into the semantic memory without regard to their semantic validity would not require a symbolic compiler. This assumes that the programmer would be

able to encode the predicates into the net without creating self-loops.[1] On the other hand, the availability of a symbolic compiler would allow a more flexible format for the PL statements which could be incorporated. Such a compiler would also permit checks on the semantic validity of these predicates.

This section has briefly considered some of the modifications which would be required to make the semantic memory teachable. The next section summarizes the work which has been described in this thesis.

---

[1] Self-loops were defined in Section 3.5.2.

4.4 Conclusions

The primary objective of this thesis has been to demonstrate the feasibility of incorporating a semantic memory into the software of an interactive graphics display system. It can be concluded that the system described in the previous chapters fulfils this objective. Specifically, it has been shown that the use of a semantic memory in conjunction with a compiler and a short-term memory results in a system which allows a high level of interaction between man and computer.

While the program which was implemented is limited in the types of problems which it can solve, it is contended that the concepts which it embodies would be applicable to the solution of more complex graphics problems. The modifications proposed in this chapter represent some steps which could be undertaken to improve the versatility of the system.

REFERENCES

Amoss, J.O., and Breeding, K.J., "A Syntactical Analysis of the Pattern Description Language PADEL", Technical Report 2768-2, Ohio State University, July 1970.

Amoss, J.O., and Breeding, K.J., "Topological Manipulation of Line Drawings Using a Pattern Description Language", Technical Report 2768-3, Ohio State University, August 1970.

Anderson, R.H., "An Introduction to Linguistic Pattern Recognition", Rand Corporation P-4669, July 1971.

Bajcsy, R., and Lieberman, L.I., "Computer Description of Real Outdoor Scenes", Second International Joint Conference on Pattern Recognition, August 1974, pp.174-179.

Breeding, K.J., "Grammar for a Pattern Description Language", Technical Report 177, University of Illinois, May 1965.

Christensen, C., and Pinson, E.N., "Multifunction Graphics for a Large Computer System", Proceedings of the AFIPS FJCC, Vol. 31, 1967, pp. 697-711.

Duda, R.O., and Hart, P.E., Pattern Classification and Scene Analysis, John Wiley and Sons, New York, 1973.

Encyclopedia Britannica, "Triangle", William Benton, Publisher, Toronto, 1963, pp. 459-460.

Evans, T.G., "Descriptive Pattern Analysis Techniques", Air Force Cambridge Research Laboratories, Bedford, Mass., 1968.

Evans, T.G., "A Grammar-Conrolled Pattern Analyzer", Information Processing 68, North Holland Publishing Company, Amsterdam, 1969.

Firschein, O., and Fischler, M.A., "Describing and Abstracting Pictorial Structures", Pattern Recognition, Vol. 3, 1971, pp. 421-443.

Firschein, O., and Fischler, M.A., "A Study in Descriptive Representation of Pictorial Data", Pattern Recognition, Vol. 4, 1972, pp. 361-377.

Fischler, M.A., "Machine Perception and Description of Pictorial Data", Proceedings of the International Joint Conference on Artificial Intelligence, May 1969.

French, L.J., and Teger, A.H., "GOLD - A Graphical On-Line Design System", Proceedings of the AFIPS SJCC, 1972, pp. 461-470.

Frijda, N.H., "Simulation of Human Long-Term Memory", Psychological Bulletin, Vol. 77, No. 1, 1972, pp. 1-31.

Gonzales, C., and Vidal, J.J., "GRAL - A Graphic Compiler Language for Intelligent Terminals", Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975, pp. 25-30.

Grimsdale, R.L., Sumner, F.H., Tunis, C.J., and Kilburn, T., "A System for the Automatic Recognition of Patterns", Proc. Inst. Elect. Eng., Vol 106 (Part B), 1959, pp.201-211.

Kirsch, R., "Computer Interpretation of English Text and Picture Patterns", IEEE Transactions on Electronic Computers, Vol. EC-13, No. 4, August 1964, pp. 363-376.

Knuth, D.E., The Art of Computer Programming: Volume 1/Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

Kulsrud, H.E., "A General Purpose Graphics Language", Communications of the ACM, April 1968, pp. 247-254.

Lecarne, O., "A System for Interactive Graphic Programming", Proceedings of IFIP Congress, Vol. 1, 1971, pp. 440-444.

Newman, W.M., and Sproull, R.F., Principles of Interactive Computer Graphics, McGraw-Hill, New York, 1973.

Nilsson, N.J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.

Pfaltz, J.L., "Web Grammars and Picture Description", Computer Graphics and Image Processing, August 1972, pp. 193-220.

Pierre, D.A., Optimization Theory With Applications, John Wiley and Sons, New York, 1969.

Poore, J.H., Davidson, J.E., and Kelley, D.P., "A Survey of Display Hardware and Software", Technical Report GITIS-69-03, Georgia Institute of Technology, 1969.

Preparata, F.S., and Ray, S.R., "An Approach to Artificial Non-symbolic Cognition", Information Sciences, Vol. 4, No. 1, January 1972, pp. 65-86.

Quillian, M.R., "Semantic Memory", from Semantic Information Processing, Minsky, M. (Ed.), MIT Press, Cambridge, Mass., 1968, pp. 227-270.

Quillian, M.R., "The Teachable Language Comprehender: A Simulation Program and Theory of Language", Communications of the ACM, August 1969, pp. 459-476.

Quillian, M.R., "Capturing Concepts in a Semantic Net", Technical Report 1885, Bolt Beranek and Newman Inc., Mass., 1969.

Roberts, L.G., "Graphical Communication and Control Languages", Second Congress on the Information System Sciences, Spartan Books Inc., 1964.

Rosenfeld, A., and Mercer, A., "An Array Grammar Programming System", Communications of the ACM, May 1973, pp. 299-304.

Rosenfeld, A., "Networks of Automata: Some Applications", IEEE Transactions on Systems, Man, and Cybernetics, May 1975, pp. 380-383.

Rovner, P.D., and Feldman, J.A., "The LEAP Language and Data Structure", Information Processing 68, North Holland Publishing Company, Amsterdam, 1969.

Shaw, A., "The Formal Description and Parsing of Pictures", Computer Science Report No. 94, Computer Science Department, Stanford University (Ph.D. Thesis), 1968.

Shaw, A., and Miller, W.F., "A Picture Calculus", from Emerging Concepts in Computer Graphics, Secrest, D., and Nievergelt, J. (Eds.), W. A. Benjamin Inc., New York, 1968, pp. 101-121.

Shaw, A., "A Formal Picture Description Scheme as a Basis for Picture Processing Systems", Information and Control, Vol. 14, 1969, pp. 9-52.

Shaw, A., "Picture Graphs, Grammars, and Parsing", from Frontiers of Pattern Recognition, Watanabe, S. (Ed.), Academic Press, New York, 1972, pp. 491-510.

Smith, D.N., "GPL/1 - A PL/1 Extension for Computer Graphics", Proceedings of the AFIPS SJCC, 1971, pp. 511-528.

Stack, T.R., and Walker, S.T., "AIDS - Advanced Interactive Display System", Proceedings of the AFIPS SJCC, 1971, pp. 113-121.

Sutherland, I.E., "Sketchpad - A Man-Machine Graphical Communication System", Lincoln Lab Technical Report 296, January 1963.

Williams, R., and Giddings, G.M., "A Picture-Building System", Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975, pp. 304-307.

Yakimovsky, Y., and Feldman, J.A., "A Semantics-Based Decision Theoretic Region Analyzer", Third International Joint Conference on Artificial Intelligence, Stanford, Cal., 1973.