

https://dspace.org/

A paradigm for general scene analysis

Forsyth, David Alexander

2015-01-21
http://hdl.handle.net/10539/16664

Downloaded from a DSpace repository. https://dspace.org/

A Paradigm for General Scene Analysis.

1st October, 1985

David Alexander Forsyth.

DEDICATION.

To Tess, who did so much.

DECLARATION.

I declare that this dissertation is the result of my own unaided work, which has not before been submitted to any university or institution for any degree or diploma, save that work clearly described as such, which has been inserted for the purpose of continuity. It has been proofread and corrected, and is free of errors.

SIGNED:

DATE:

A dissertation submitted to the University of the Witwatersrand in fulfilment of the requirements for the degree of Master of Science in Electrical Engineering.

A farmer, wanting an automatic chicken plucking machine, spoke to an engineer, who swore that it couldn't be built. The disappointed farmer then went to a theoretician, who said "It's easy! Assume a spherical chicken, arbitrary diameter d, ..."

(Modern Folktale, of unknown origin.)

ACKNOWLEDGEMENTS.

The following people and companies have helped me, and I thank them for their time, patience, and suggestions.

Advanced Semiconducter Devices lent me an image acquisition and display system, without which I could have done very little.

The Council for Mineral Technology provided generous financial support.

Hewlett-Packard (S.A.) lent me a minicomputer system, on which all the work described here, and a lot more, was done.

Mitsui Corporation provided an excellent software package, SPIDER, for image processing applications, which helped me over an immense learning curve.

The staff and students at Uwtech provided a very pleasant working environment, which made my work easier. Gunther Sommer and Professor Bruce Stewart of the Council for Mineral Technology gave me their support and interest, and were very tolerant of my wilder ideas.

Dave Penckler and Adrian Solomon of Hewlett-Packard helped me to use the minicomputer, and rescued me on a number of occasions.

Gerald Bloch and Steve Meyer of Uwtech provided stimulating discussion, and much support.

ACKNOWLEDGEMENTS.

The following people and companies have helped me, and I thank them for their time, patience, and suggestions.

Advanced Semiconducter Devices lent me an image acquisition and display system, without which I could have done very little.

The Council for Mineral Technology provided generous financial support.

Hewlett-Packard (S.A.) lent me a minicomputer system, on which all the work described here, and a lot more, was done.

Mitsui Corporation provided an excellent software package, SPIDER, for image processing applications, which helped me over an immense learning curve.

The staff and students at Uwtech provided a very pleasant working environment, which made my work easier. Gunther Sommer and Professor Bruce Stewart of the Council for Mineral Technology gave me their support and interest, and were very tolerant of my wilder ideas.

Dave Penckler and Adrian Solomon of Hewlett-Packard helped me to use the minicomputer, and rescued me on a number of occasions.

Gerald Bloch and Steve Meyer of Uwtech provided stimulating discussion, and much support.

Dave Smith and Gunther Berger, who worked with me in the image processing group, generated a stimulating working environment and allowed me to mature my ideas through long discussions. My debt to Gunther is very large, as the reader will see, and he has been a consistently stimulating research partner since we started working together in early 1984.

Mrs. Sue Rodd very kirdly proofread this work in a very short time, and did so very well. Any errors which remain in the text are my fault entirely.

Professor Mike Rodd, my supervisor, has given me tremendous support and encouragement, and I owe him a debt that is difficult to express, let alone repay, for his advice, his patience, his encouragement, and the support he gave me.

My family has tolerated me when the work made me intolerable, and I owe them great thanks for this.

Finally, Tess von Willich drew the pictures, made those changes in the text that I was suggested, cooked, kept house, remained tolerant and good natured, and beleived I was doing worthwhile work when I was doubting it. For this help, I owe her a tremendous amount.

ABSTRACT

A new paradigm is described for knowledge-based scene analysis. The paradigm requires firstly that images be understood essentially in terms of two-dimensional information about how objects in pictures look, and secondly that features be considered as collections of regions. A system for the analysis of general scenes is described and the results of an attempt to prototype this system are given. Although the prototype is a simplified version of the system described, it appears to be capable of great power. The paradigm is further supported by promising results obtained in work on an instrument to analyse the size distribution of particles on a conveyor belt.

TABLE OF CONTENTS

INTRODUCTION. 1

BACKGROUND 3
The origins of this work. 4
Knowledge-based systems. 6
Scene-analysis paradigms. 10
Image representation. 13
Knowledge representation 16
Knowledge manipulation. 18
Reasoning about control. 20
Implementation issues. 22
Conclusion. 22

NEIGHBOUR CODES. 23
Properties of the neighbour code. 24
Forming the neighbour code. 25
Conclusion. 31

THE PARADIGM AND ITS USE. 32
Guiding concepts in the paradigm itself. 32
A system designed to analyse scenes 35
Potential problems 40
Conclusion 41

THE DESIGN OF THE PROTOTYPE SYSTEM. 42

The block design of the prototype system. 43
The design of the data structures. 48
The syntax and structure of the knowledge sources. 50
The condition field. 51
The action field. 54
Examples of rule use. 54
The algorithms used within the blocks. 57
How the prototype evaluates a KS. 58
How the prototype factors a goal. 67
How the system generates and tests merges. 69
Finding the goal. 71
Conclusion. 73

THE IMPLEMENTATION OF THE PROTOTYPE. 75
The data structures. 75
Error handling. 76
Coding techniques used for building the prototype system. . . . 78
The code. 79
Conclusion. 80

RESULTS. 81
Results obtained within the paradigm. 81
Results obtained by prototyping the system. 83
conclusion. 86

CONCLUSION 87

REFERENCES. 88

READINGS. 91

APPENDICES 92

APPENDIX A. THE SUBROUTINES INVOLVED 93

Program RINT 93

Program INITS 94

Program START 94

Program FGOAL 95

Program GTMER 96

Subroutine AGET 97

Subroutine AHEAD 97

Subroutine ANDSTACK 98

Subroutine APSETUP 98

Subroutine ASETUP 98

Subroutine ATORSTACK 99

Subroutine BACKE 99

Subroutine BTS 100

Subroutine CANDSTACK 100

Subroutine CHCTXT 100

Subroutine CLEAN 101

Subroutine CLFIND 101

Subroutine CMERGE 101

Subroutine COMPLIST 102

Subroutine CONHEAD 102

Subroutine CONMAT 103

Subroutine CSSETUP 103

Subroutine CVARFCL 104

Subroutine CVARFIX 105

Subroutine EDGE 106
Subroutine GTMERCON 106
Subroutine INSTNO 108
Subroutine LGBUF 108
Subroutine LISTPGOALS 108
Subroutine MATCHP 109
Subroutine MEROK 110
Subroutine MRECORD 110
Subroutine NAMELIST 110
Subroutine NAMFIND 111
Subroutine NCFUL 111
Subroutine NCMERG 112
Subroutine NEWSTATS 112
Subroutine *NNUME* 113
Subroutine NUMEVAL 113
Subroutine PRMLEXP 114
Subroutine PRMLIST 115
Subroutine PUTIT 115
Subroutine QUALEVAL 116
Subroutine REVAL 116
Subroutine REVERSE 117
Subroutine RMPAR 117
Subroutine RNUMEVAL 117
Subroutine ROCKE 118
Subroutine ROTATE 119
Subroutine ROUND 119
Subroutine RQUEUE 119
Subroutine SEARCHI 120
Subroutine SEGLD 120

Subroutine SEGRT 121
Subroutine SIMGE 121
Subroutine SINTE 122
Subroutine STRLIST 122
Subroutine TRUEE 123
Subroutine VECREVAL 123

APPENDIX B. THE BACKUS NAUR FORM SYNTAX OF THE KS'S. 125

APPENDIX C. A COMPLETE LISTING OF THE CODE. 126

LIST OF FIGURES.

Figure 1: The Rite of Spring.

Figure 2: A Neighbour Code of the Form 1 2 3 4 3 4 5.

Figure 3: The Reason the Direction of Pixel Traversal in Forming Neighbour Codes is Important.

Figure 4: A Code That Contacts in N+1 Points Encloses N Sets of Regions.

Figure 5: To Illustrate Merging Regions

Figure 6: The Block Diagram of a System Designed Under the New Paradigm.

Figure 7: The Block Diagram of the Prototype System.

Figure 8: To Illustrate Evaluating KS's.

Figure 9: A Picture Taken at Dealkraal Mine, and its Segmentation by the PSDA.

Figure 10: The Knowledge Base Used in the Example.

Figure 11: The Picture Used in the Example, and its Representation.

Figure 12: The Output of the Prototype System for the Example.

INTRODUCTION.

This dissertation offers a new paradigm for scene analysis, the basic tenets of which are that a scene should be analysed using explicit knowledge, and that the scenes should be understood and represented in terms of regions.

We will discuss the origins and implications of these tenets, and show some results which indicate that they allow a system to be built which can indeed analyse scenes. The advantages of the use of regions to interpret the scene will be shown to be considerable, as they allow knowledge to be actively deployed. We will discuss a representation developed to handle the picture as a set of regions, and show how the [illegible]ions may be handled using this representation, which is known as [illegible]'s neighbour code.

A system that is capable of general scene analysis is described, and we discuss the prototyping of a simple version of such a system. The design of the prototype is exhaustively presented, and its implementation is discussed. The code of the prototype system is presented as appendices.

The approach proposed cannot be shown to be the best one, nor can it yet be shown that it will *definitely work*, but we *have* managed to show that a system that works can be developed within the paradigm, although it is not yet analysing real scenes. We will produce convincing arguments, however, that it is a good approach, and that it can work. Results both of the prototyping of a general scene analysis system and of work on an-

other system which is claimed to have been developed within this paradigm, indicate that working systems may be built within these tenets.

BACKGROUND

Computer vision presents a fascinating problem, with many potential engineering benefits presented by even a partial solution. The aspect of the problem that we shall address is that of scene analysis, which can fairly be described as attempting to solve the problem:

"Given a picture, describe what is happening in it."

or the rather simpler version:

"Given a picture, find the ------- in it."

These can be seen to be subtle problems. We shall address the latter, as it is a simpler version of the former, and has not yet been solved with any degree or promise of generality.

Many techniques have been proposed to solve this problem, the essence of most solutions being the finding of appropriate parts of the picture, and then hopefully exploiting some information to classify the right set of pieces as the object sought. We do not depart in any great measure from this strategy, save to insist that the information exploited be expressed explicitly rather than in the structure of the scene analysis program. This does not sound unreasonable: we intend to show just how reasonable it in fact is.

THE ORIGINS OF THIS WORK.

The work that is described in this dissertation originated in an instrumentation problem. A popular milling technique on many mines is run-of-mine milling, where ore is milled as it arrives from underground, without being crushed. It is desirable to mill this ore autogenously and it is becoming more and more desirable to instrument the process (Sommer, 1985). Work is being done on instrumenting the processes occurring within the mill during milling, and it was felt important that an attempt be made to instrument the size distribution of the ore offered to the mill. Instruments are available to perform this function, but are extremely primitive at this stage (see the Armco Autometrics manual,1984). This author and Berger (1984) were required to investigate the feasibility of building a particle-size distribution analyser using-image analysis techniques, and managed to produce a design, which is at present being implemented by Berger and a third worker, Smith.

The research raised many issues, an important one being the robustness of present image-processing methods, as the problem requires for its solution techniques able to handle complex pictures of simple objects, taken under adverse conditions. It is not possible to use structural clues to find rocks in a picture, and simply binarising the picture is not helpful (i.e. calling all grey areas rocks, all black, background) as all rocks in the picture are covered with a thin layer of adsorbed fine particles, as is the belt to a large extent, and it is quite possible to classify whole pictures incorrectly as representing single large rocks. The technique developed, and used with some considerable success, was segmenting the picture into regions, classifying those regions, and then

merging them on the basis of merging rules, which were developed ad hoc. The merge rules were based largely on knowledge acquired by looking at a running conveyor belt with ore on it: for example, it was realised at an early stage that, as a result of the processes acting to produce the ore, few rocks would appear with holes in them, and that the holes were unimportant. Thus, a region, whatever its classification, that was entirely surrounded by just one rock region could be merged to that rock region and forgotten as an entity on its own.

This technique was based on a number of assumptions, the major ones being that the rocks on the belt would be nice ones (i.e. is not extraordinarily long and thin or unreasonably curved), that a two-dimensional picture of a rock lying on a vibrating surface would give a good idea of its size (We do not even consider the real meaning of size for a rock, lest we become embroiled in unpleasant issues. See, for example, Mandelbrot, 1982.) and that sufficient merge rules of the type described exist to find a reasonable interpretation of the picture. The validity of these assumptions could be debated: what is certain is that the system based on them seems to work very well.

From this work some important ideas originated.

- Firstly, it was seen that regions are an extremely useful feature to extract and work with, particularly for real-world pictures taken under poor conditions.

- Secondly, it was inferred that one could work very well with pictures in two dimensions only.

- Thirdly, it became apparent that when one is dealing with a region, the most important information available is the nature of its immediate neighbours, and that a picture representation that would carry this information could be most useful. (Notice that if we represent a picture properly in this way, if we require information about a *region's neighbours' neighbours*, this is also obtainable. We will expand on this point later.)

- Fourthly and finally, it was soon seen that a system that allows experimentation with explicit knowledge would be most desirable, as if the knowledge used by the system were implicit, system modifications or updates would be complex. As a result the system would have to be committed to one task. With a system where the knowledge is explicit, however, we may change the targeting of the system by changing the knowledge available to it.

This worker then started to use the above results and ideas from the work on particle size distribution analysis, in an attempt to build a vision system able to handle a large class of scenes using knowledge supplied explicitly to it.

KNOWLEDGE-BASED SYSTEMS.

In a sense, all software systems that solve problems are knowledge-based; we use this term to refer to systems that use explicit knowledge in some form. This type of system has the advantage that one may improve the

performance of the system or change the problem it can solve, up to a point, by improving the quality, or by changing the subject of the knowledge available to it. Systems based on implicit knowledge, on the other hand, will require the overhead of system changes to improve their performance or change the problem that they solve. Implicit knowledge systems make up for their inflexibility by their efficiency, as they do not require the huge overhead of knowledge management, and are popular in engineering image processing applications as a result. Generality will however be difficult to achieve with an implicit knowledge system.

We can see that, when we have to analyse a scene in a purely engineering context, it is desirable to embed the description of the objects we are *seeking as efficiently as possible* in the system. When, however, our system is likely to be faced with a large number of different objects, it becomes desirable to be able to provide the system with a description of each object, without having to change the system software. A rock finder can then, at a pinch, become an elephant-finder. At some stage, the hardware technology may become such that this may well be the best approach, from an engineering point of view: at present, the approach does not promise results, save as a training, development and research tool.

The best-known class of knowledge-based system is the expert system. As Brachman (1983) points out, it is not correct to call a system that analyses scenes, an expert system, knowledge-based or rule-driven though it may be. The task itself does not require expertise. The literature is not consistent in this nomenclature. We use the term knowledge-based system instead, but regard the expert-system literature as being relevant. This is by no means merely a semantic quibble; expertise in vision is not easily described, and the problem is not one of simply determining the

right rules for an expert-system kernel. An expert system typically solves an expert problem, that is, a complex problem requiring highly specifi knowledge, which is generally used in a specific fashion. We require our system to use poor knowledg n a poorly-defined field, in whichever way it chooses, to solve a problem that is not well understood-by human experts.

The expert system as described in the literature has two segments: a knowledge base and an inference engine. The knowledge base typically contains only knowledge through which the problems posed to the system may be solved. The inference engine then exploits this knowledge to solve these problems. There are visible disadvantages to this structure for our purposes. The system cannot reason out a strategy to solve its problem; the strategy is that of the system's author. Typically, the domain of expertise of an expert system is limited, generally by the sheer logistics of marshalling a large quantity of knowledge. Such a system is often restricted to having in its knowledge base only knowledge that is consistent with the knowledge base as a whole. The knowledge base can too easily be considered purely as a rule base, which can cause the designer to restrict the power of the knowledge. (One often sees systems described as "forward chaining" - for a powerful system, surely the system itself should decide in which direction it should chain ?) We therefore follow Davis (1980) in referring to the knowledge base as containing knowledge sources (KS's): in this case each KS will be of a similar format to and discharge the same function as, a rule in the conventional expert system.

We require of a knowledge-based system that it be able to reason about how it should manage knowledge, that it be able to choose between con-

tradictory KS's on the grounds of other evidence, and that it be limited as little as possible in the ways that it handles knowledge. We hasten to add that the prototype described below has not solved the problems presented by such requirements: rather, as many issues as can be sidestepped have been, to allow the development of a prototype. The prototype has not been developed as yet another expert system, it is intended to demonstrate the feasibility of knowledge based scene analysis and of a new paradigm with which we have attempted to simplify the problem of analysing real scenes.

Knowledge-based systems are beginning to penetrate the image-processing literature, and precedents do exist for using these systems for this type of work. Nazif and Levine (1984) describe an expert (sic) system for low level-image segmentation, which appears to give good results, although according to the point raised above, it is in fact a knowledge-based system. Niemann and his group (1985) describe another system which uses knowledge to interpret results from a segmented picture: it appears, however, that it uses implicit knowledge to segment the picture and extract the information it needs for its analysis. We would like our system to use knowledge at all levels, so that high-level knowledge may aid low-level actions.

Our system is thought of in a fashion different from the conventional "inference engine - knowledge base" representation; rather, it has been developed as an interpreter operating on a picture-knowledge language. The term "interpreter" is used advisedly; different reasoning facilities are available to different bits of knowledge, as at this stage the facility of instructing the system, through the knowledge base, as to how it should manage knowledge has not been implemented. This interpreter

model is to avoid the implicit assumption of author-imbedded reasoning techniques in the knowledge-management section.

SCENE-ANALYSIS PARADIGMS.

Typical of the state of scene analysis at present is the system proposed by this author (Forsyth, 1984) to solve the instrumentation problem described above, which exploits as many problem constraints as possible to produce a workable design. In itself, this approach is not incorrect, but the investment in using a system built to this type of design to solve a problem differing even slightly from the problem first envisaged, is considerable. We would like therefore to build a scene analyser that is as general as possible - how do we do this ?

We must first decide in what form we wish to handle the scene. It is obviously important to be able to exploit the structure of the object we seek, yet we must also decide how to handle the appearance of its parts in the picture.

We may handle the picture in terms of its edges, or of its regions. Its edges will lead us to a form of line drawing, or wire-frame representation: its regions, to a representation in terms of picture subsets. We have chosen the latter, and will explain and justify the choice both by reference to work in terms of the line-drawing representation, and by experimental results.

The major line-drawing-type paradigm appears to be that of Marr (1979), although much other work has been done in this field, where pictures have been interpreted in terms of their edges. (We are using the term "line drawing" rather loosely here: if one examines a picture's edges, one is working in this type of paradigm. Thus, the term includes generalised cylinder representations and the like.) The great problem with this paradigm is that one is required to understand the picture in terms of models that one has built of it. Knowledge, not of what objects look like, but of what they should look like in pictures, is used. The difficult question of shape can be avoided in this fashion, for example by describing a rocket as looking like a cone on top of a cylinder, but a system designed like this is at the mercy of its own modelling techniques. Too much knowledge is deployed in modelling the scene, and not enough in handling it. Thus, a system built under this paradigm cannot deploy knowledge to find the most likely position of an object which it knows should be there, but which it cannot find for shadows. The paradigm we propose here is intended to allow marshalling of knowledge to make it effective at all levels.

Another feature with present line-drawing vision paradigms is the effort put into techniques to interpret a two-dimensional image by using clues to its shape in three dimensions contained, for example, in the shading of the picture. It is felt that these techniques are likely to produce misleading results when applied to a real picture, particularly one taken under poor lighting conditions. It is surely better to hold information in one's knowledge base as to the likely appearance of the objects described, *in a picture*, than to try to obtain a potentially misleading model of the picture in three dimensions, and then to interpret this probably defective model.

Marr's paradigm limits the power of the knowledge used in systems built under it. It is laborious for such a system to find, as Freuder's (1976) system can, an object it knows should be present by *lenient classification* of regions in that object's most likely location. If the line drawing has been poorly produced as a result of bad lighting, for example, the system will have difficulty in making another to conform to other knowledge that becomes available at some later stage. It cannot use the implications of its high-level knowledge to improve its handling of low-level decisions. We are proposing, then, that pictures be understood in terms of pictures.

Look at figure 1. Marr (Marr,1979) claims that this picture is perceived

> "in terms of very particular three-dimensional shapes, some familiar, some less so."

This author disputes his claim; surely the name of the picture is used to provide clues as to the identity of those objects not clearly classifiable, and those that are understood without these clues, are understood as two-dimensional blotches. Then, once everything has been satisfactorily classified, all objects inherit from the viewer a shape in three dimensions. While it is unfair to criticise the paradigm on the basis of a complex piece of artwork, where background knowledge becomes important by the nature of the picture, the claim holds that three-dimensional shape is of little importance in the early stages of classifying a scene. We claim that shape is often inherited by objects in a picture after they have been classified; this effect is particularly noticeable when one considers pictures of poor quality. In the rock pictures shown, the reader should notice that the rocks acquire shape only after he has puz-



FIG. 1 The rite of spring.

zled out where they are, not while he is doing so; thus information about the two-dimensional appearance of objects is being exploited.

The amount of information involved in understanding scenes is well illustrated by the rock pictures. Most people who have seen these pictures, did not know what they represented until they were told. Even then they were not sure whereabouts in the pictures the rocks were, and could interpret the pictures correctly only after having several pictures explained to them. From this stage on, they were able to interpret these pictures for themselves. Thus the process is learnt, and in stages. Only once these stages have been traversed, is the picture understood in terms of the three-dimensional shape of the objects represented.

We have laboured the relative unimportance of three-dimensional shape information in the early stages of scene analysis, because the system described here ignores this information completely. We feel that the results obtained in the particle-size distribution analysis work justify the use of the techniques described, although they make the later exploitation of three-dimensional shape information difficult. The relative unimportance of this information means that their inability to handle this information does not militate against them.

IMAGE REPRESENTATION.

It is important to consider how images may be represented, as the choice of image representation will often influence the abilities of the system

in question. We wish to represent the image in as powerful a form as possible, while conserving as much space as possible. We wish to store the picture in a fashion compatible with the system we are describing. that is, we wish to hold information pertaining to the regions within the picture in as tractable a form as possible.

For this purpose, it must be assumed that the region segmenter is trustworthy, that is, that it will not be necessary at some later stage to ask it to revise its decisions. This is the same as assuming that each primitive is represented by at least one region in the segmented picture, and that no two primitives share a region, where we define a primitive as a piece of picture that can be understood by a single piece of knowledge in the system we describe. (That is, we do not need to inspect structure to find a primitive: it is a piece of picture that stands in its own right.) So, once we have segmented a picture, we may discard the picture itself in favour of a representation of the segmented picture that is more compact than the picture itself, if it retains the information we require. The question is, what do we wish to represent ?

This issue is not yet fully settled, as the system itself is still a prototype; however, we shall describe what was represented and how, and we claim that the system allows expansion of the representation. As has been seen above, the most important information that one may have about a region appears to be the list, held in some form, of its neighbours. We wish also to hold some representation of its size, its average grey level, its texture, and its shape in two-dimensions.

It can be stated clearly at this point that the issue of two-dimensional shape has been largely avoided; in its present state, the prototype system

neither needs nor uses the information. It is accepted that this will become an important issue, and that it is crucial to the representation of pictures at a high level to find a representation of shape that will allow one to decide what the shape will be of a region created by merging two other regions of known shape, without reference to the picture itself. The issue has simply caused insufficient problems up to this stage to warrant more than brief attention; it is felt that the region representation adopted has sufficient power, with some attention, to allow a solution of this problem without modification to the representation.

The issue of representing the region structure of the picture has, however, received much attention, dating from the early days of the particle-size distribution analyser work. At that stage, a representation was required that indicated how a region related to its outside neighbours, the inside neighbours being ignored for reasons we shall detail later. The chain-code approach, as detailed in, for example, Rosenfeld (1982), was unappealing on the grounds of its potential intractability. In the rock pictures obtained, regions tended to have long boundaries, with many changes of direction in them, but with few neighbours represented. Thus, a vast chain code could be obtained with much largely-redundant information. The neighbour-code representation was then suggested by Berger (1984 (a)) and then refined by this author. This representation will be described in detail in the body of the dissertation.

The standard deviation of the grey level over the region has been chosen to represent texture. This choice is open to discussion: it is certainly not the only way in which one may represent texture. However, we have again not addressed this issue in detail, as the main thrust of the work

was intended to establish the feasibility and usefulness of the type of system described.

KNOWLEDGE REPRESENTATION

Representing knowledge is not the same problem as representing picture data: however, we require from our solution to the former problem, the same characteristics as we require from the solution to the latter. A simple, tractable technique is required at this stage, and power may be sacrificed for simplicity. Thus, although the issue will be discussed, our knowledge format will not in any way represent the state of the art.

There are several knowledge representations proposed at present: the main themes seem to be semantic nets, frames, rules and the predicate calculus, while other representations have been proposed. (See for example, Charniak (1981), a paper which proposes a knowledge representation that has the flavour both of frames and the predicate calculus.) Considering a piece of knowledge as a rule, and thinking of it solely in those terms appears to restrict one's perception of the potential of the knowledge, as it is too easy to get distracted by the implications of the "if - then" construct. Thus, although the knowledge representation we have adopted is in a rule-like format, we have avoided saying "if this then that". The representation is rather as a "this relates to that" form.

Barr and Feigenbaum (1981) present a discussion of these techniques which we shall summarise here. Logical representations such as the predicate

calculus are precise, flexible and modular, and they appear to be a natural representation for certain notions. However, it is often the case that logics need to be non-monotonic[1] and this renders them, for our purpose at least, intractable.

A semantic network is a structure in which knowledge is represented by links of different types between nodes. Nodes usually represent situations, objects or concepts, and the links the relations that exist between them. Semantic networks, and an elaboration of the network idea into a structure, the frame, are the subjects of much research.

The representational structure with which we are concerned, the rule, has been much in vogue recently, with the success of expert-system technology in commercial applications. The idea acting here is that a piece of knowledge may be expressed in an "IF (conditions) THEN (actions)" form. This representation has been used very successfully for expert domain knowledge, and seems to lend itself very well to representing the type of knowledge required to solve this kind of problem. It has disadvantages: present expert systems face potential problems with saturation, where the rule base becomes so large that the system has real difficulty getting anything done.

Recent work has shown that systems may be built that change or update rules to allow them to handle situations that they have not previously

---

[1] A monotonic logic is a logical system in which, if it is possible to deduce a theorem with a set of assertions, additions to that set will not later negate the theorem.

encountered (Lenat,1980). Brachman et al (1983) quote Hart as saying in a 1980 paper that current expert systems typically show up badly, as they are unable to recognise or deal with problems for which their own knowledge is inapplicable or insufficient: this situation would appear to be changing, but the criticism is still largely valid. These problems are to an extent inherited from the rule representation of knowledge.

We have reviewed here the major ideas in knowledge representation. The criterion by which a technique was chosen for the prototype system was again simplicity of implementation, recalling that a representation was desired, less for the maintenance of problem knowledge, than for providing a simple explicit knowledge facility in the system. A rule-type representation was chosen for its potential simplicity of implementation in FORTRAN (which language choice will be explained) and because it appeared that updating the the knowledge base could be simplified by using this representation. It will be seen later that the rules have been used in different ways, depending on their terms of reference; there is a strong similarity to an interpreted language where statement order is not important.

KNOWLEDGE MANIPULATION.

Another important question which arises is, "How we manipulate knowledge that is inaccurate: how do we reach a decision based on poor evidence ?" This is again a research issue in itself, and the system we have adopted does not claim to be optimal. As the techniques developed so far appear

to be *in a large measure empirical*, and as there appears to be no strong unifying idea at present in this field, we have again chosen the simplest technique that will meet our present needs in this matter.

The problem of reaching a decision based on poor evidence is a considerable one, and although not an immediate problem in the case of the present prototype, which will simply make the wrong decision when faced with poor evidence, it is one that requires addressing, as the system is intended to be a prototype of a device that will operate on real pictures, and hence will require a certain degree of noise immunity.We will therefore briefly summarise the techniques available.

One may use a Bayesian technique, where a combination of a priori probabilities leads to a probability on which a decision may be based. Shortliffe et al (1984) quote Duda as using these methods in the PROSPECTOR system, with considerable success. The problem with Bayesian techniques is that the a priori probabilities are not always easy to find, and the technique seems to be unpopular as a result.

DuBois et al (1980) present Zadeh's concept of the fuzzy set, where a set's members are assigned a degree of membership, and where functions may be defined to describe the degree of membership of an object in the union or intersection of the sets. The theory is deep and rather appealing, as fuzzy functions and fuzzy numbers may be defined. There is, however, at least one considerable problem to be considered in using this theory in an implemented system.

Consider the set of students in a class; we wish to define for each student his degree of membership in the set of tall people. A seven-foot

student is a member to degree 1 (the scale is conventionally 0 - 1); but to what degree is a six-foot student a member ? The problem is simply what the membership function should look like. There are other problems: generally, with the functions described in DuBois et al (1980), the truth value of the proposition does not necessarily increase as evidence mounts, and a "strong" piece of evidence may be outweighed by a weak one. This does not easily lead to a good model of human evidence combination.

For the MYCIN system, Shortliffe (1983) developed a system that appears to work rather well. It does however, appear to have been developed rather empirically with the MYCIN issues in mind, and as such would not suit our purpose, as it cannot be lifted straight out of the MYCIN context and used. (Shortliffe (1983) points out weaknesses in the system, with the proviso that MYCIN itself worked very well.)

As in all the other fields we have discussed, we wished for our prototype the simplest possible technique that would serve our purposes. Thus the fuzzy set representation was used, both because it defaults in the case of definite evidence to the Boolean model, and because we did not require extensive work with poor evidence.

REASONING ABOUT CONTROL.

We have stated how important it is that a knowledge-based system be able to reason about how it will use its knowledge. Although the literature seems to reflect no clear trends at this stage, this author feels that

the solution to this problem lies both in using the technique of diverting the interpreter's attention to certain KS's at the expense of others, and in providing the interpreter with different techniques of knowledge exploitation, and allowing the knowledge base to force switches between them, generally as a result of the context in which knowledge appears.

For example, the first technique would result in a scene analysis system which when asked to find an elephant in a scene, would prevent knowledge about different animals from reaching the interpreter. Davis (1980) discusses a technique where plausibly-useful knowledge sources (PKS's) are found, by first rejecting those KS's which are definitely not useful, and then ordering by utility the rest. These processes are themselves knowledge-driven. *It is not easy, however,* to see a reliable technique whereby knowledge may be suppressed without losing relevant knowledge, whose relevance is not immediately obvious. In our prototype system all pieces of knowledge are brought to the attention of the interpreter, and those which do not fulfil a relevance criterion are ignored. This is a primitive implementation of the above technique.

The second technique is illustrated in our prototype when we consider the knowledge that describes structure. If a KS relevant to the structural description we are seeking is loaded into the interpreter, it is inspected to ascertain what other knowledge may be relevant, and then held for future reference; later, when we are actually looking for that structure in the scene, the knowledge will be evaluated as a rule to indicate what value we associate with a given set of regions belonging to that structure.

IMPLEMENTATION ISSUES.

The system described was implemented in HP FORTRAN-77, on an HP-1000 mini-computer. FORTRAN was used because of the author's considerable familiarity with the language and because the only other languages available were an untested subset of LISP, and PASCAL. To avoid interpreter-testing issues, and because the PASCAL compiler available at the time was extremely slow, FORTRAN was the obvious choice. The code was designed, written, and tested from May to August, 1985, inclusive.

As the HP-1000 used has a maximum partition-size (code and data) of 32K it became necessary to segment the code, which had not originally been written with this in mind. The word segment will refer below to the code segments, which unfortunately correspond to the system block, leading to potential confusion unless one's nomenclature is careful.

CONCLUSION.

We have introduced the ideas that appear in the rest of this dissertation and attempted to show their origins. The work that is described below departs in many ways from the mainstream of current vision work, but is rooted in respectable concepts. The prototype built is still a fragile curiosity, rather than an engineering solution, yet it represents an attempt to [illegible] the ideas presented, with techniques developed by this author and two other workers, to produce a system that can see.

NEIGHBOUR CODES.

(Some of the work described here was submitted in a research report to the Wits Department of Electrical Engineering for a B.Sc.(Elec.Eng.), in which the concept of neighbour codes, and the way that they could be merged when no enclosed regions existed, were described. The algorithm to deal with this situation was developed after this report was submitted, and the material is presented as a whole for simplicity's sake.)

We have said above that when one considers a picture in terms of regions, their neighbours' qualities become the most important information that we can provide in a representation at a higher level than the picture itself. However, it also became apparent that representations such as the chain code were not ideal. These facts spurred the development of the idea of the neighbour code, which has so far proved itself useful as a representation of a segmented picture.

If one may merge and analyse regions on a basis of their neighbours, can we not then develop a representation, less bulky than that of the entire picture, that will hold this information, and in which one may represent merges without reverting to the original picture ? It turns out that we can indeed do so. The representation is formed simply by listing the neighbours in the order in which they occur in a traversal of the outside boundary of the region, and is called the neighbour code.

PROPERTIES OF THE NEIGHBOUR CODE.

The first question that occurs is, "Why only the outside neighbours?" The answer is simple: "If a region is entirely contained by another, the information is preserved in the contained region's neighbour code, and need not be entered into the other's code, where it could cause considerable book-keeping problems at some later stage." We ignore internal regions when forming the code; that does not mean that the regions on the inside receive no consideration, nor that their relationship with their enclosing neighbour is not recorded, but that such consideration is performed when we inspect the enclosed regions' outside neighbours. For a consistent representation of the region space, it is, however, important to traverse each region's boundary in the same direction (i.e. either all clockwise or all anti-clockwise). The code is cyclic; it is not important to start the traverse with any particular neighbour. The edge of the picture must be regarded as a region that surrounds all the regions in the picture, and its occurrence in the neighbour code must be recorded; in this work, we have assigned the edge the region number -1. (Recording the edge is important, as we shall show.) The code is normally formed of the region numbers of neighbouring regions, but may equally well be formed of their types, textures, etc. It appears to be possible to construct an unambiguous representation of the segmented picture with this representation, but I cannot prove this.

FORMING THE NEIGHBOUR CODE.

We assign each connected region in the segmented picture a unique symbol, conventionally a positive integer, by which we identify it, which we use here, we refer to this symbol as the region number. The code is structured as follows: a neighbour that appears on the boundary of the region in question is represented once in the neighbour code for each contact between it and the central region, however long the boundary on which they contact . Thus the following are not legal neighbour codes:

- 1 2 3 3 3 4 5 6
- 1 2 3 4 5 6 1

(recalling that the code is cyclic). The following codes are,however, legal:

- 1 2 3 4 3 5 6
- 1 2 3 4 1 5 6 7

It should be noted that a code of the form:

- 1 2 3 4 3 4 5

is legal, but potentially confusing, and not meaningful (see figure 2). The illegal codes occur occasionally the merging process, and require only a simple cleaning-up process: the above meaningless code misrepresents

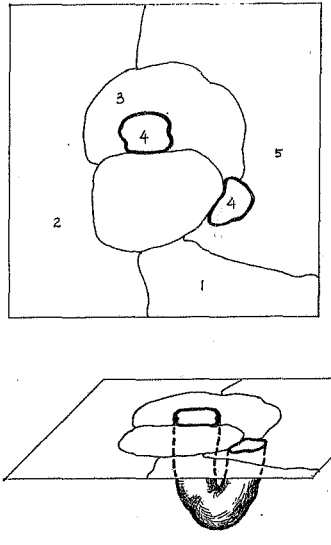


FIG. 2 Codes of the form:
1 2 3 4 3 4 5

the space, and can arise through incorrect formation of the code. Such codes cannot, however, arise when the code is correctly formed, or the representation itself would not be useful.

We have not seen before the confusion that can arise from the question of whether the segmented picture is four- or eight-connected. Figure 3 will illustrate the point. The problem is simple; if we define a region as eight-connected, its neighbours must then be four-connected. The problem then arises that it is not easy to say in certain cases whether a pixel is an outside neighbour or a small enclosed region. Fortunately, we can say that the issue will be dealt with by the labeller, by requiring that it label connected regions with a single label per region (allowing it to decide whether a pixel belongs on its own, or with a connected region) and by then informing it whether it should define eight-connected regions or four-connected regions. The latter decision we have made arbitrarily: Berger (1985) has been able to produce meaningful neighbour codes from the segmented picture produced by a package routine (SPIDER,1984), and there appears to be no good reason why his routines should not work for four-connected regions.

It will be seen when figure 3 is consulted that it is also important in which direction each boundary pixel's neighbours are traversed when we are forming the code. We form the code by traversing the boundary in a given direction: if we have defined the region as eight-connected, its complement (the rest of the picture) is then four-connected, so we must then inspect each boundary pixel's four neighbours. The traverse of these four neighbours must be performed in the correct direction, starting in the region whose neighbour code we are forming, as failing to do this will lead to spurious reversals in the code. For example:

- *in forming a code*, we can include each neighbour pixel and then clean the code up, by compressing each run of region numbers to a single member, and suppressing the region number at the end of the code if it is the same as that at the beginning, so that

  – 1 1 1 2 2 1 3 3 3 4 4 4 5 5 5 6 6 6 1 1 1

  becomes

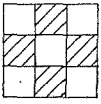
  – 1 2 3 4 5 6

- however, if we have mistraversed pixels, we may get

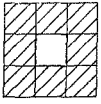
  – 1 1 1 2 1 2 2 3 2 3 3 4 3 4 4 5 4 5 5 5 5 6 5 6 6 6

  which is unmanageable and certainly even with cleaning-up could not represent a set of connected regions on a plane. Reference to figures 2 and 3 should remove any confusion.

If, however, we *cannot merge two regions using their neighbour codes* alone to find the code for the composite region, the representation is of little use. For some time it appeared that this was indeed the case, until a sufficiently powerful algorithm was developed. The merging of codes is not complex unless the two candidate regions will form a composite that encloses another set of regions, which were not previously enclosed, in which case the problem appears that *it is not easy to tell which of the groups represented in the codes* will be enclosed. We shall examine the simple case first.

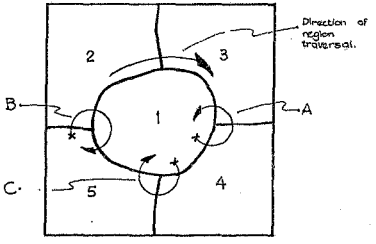


A pixel's 4-neighbours



A pixel's 8-neighbours

A region is n-connected if one may reach any one point in it from any other, by a path whose next member is an n-neighbour of the current member.



If we traverse pixels as A, we will see 3, 4 and then 3 again.
If we traverse pixels as B, we will see 5, 2 and then 5 again.
If we traverse pixels as C, the code will be correct.

FIG.3 The importance of the direction of pixel traversal.

- Given that we wish to merge regions 1 and 2 with codes 2 3 4 5 6 and 7 8 1 9 respectively, and that 1 is the senior region (the composite will be called 1, and we are merging 2 to 1), to produce the code for the composite, we rotate each region's code such that the other merge candidate appears in the front of the code, so that we get

  - for 1 : 2 3 4 5 6

  - for 2 : 1 9 7 8 .

- We then remove from each code the first member, and write one code after the other (it is not important which code is written first) and this becomes the code for the composite. So we get:

  - 3 4 5 6 9 7 8

We now consider the more interesting case where the merge candidates will enclose one or more sets of regions when merged. On reflection, or on consultation of figure 4, it will be seen that when the regions will enclose n sets of regions, they will contact each other n+1 times. Thus, each candidate will refer to the other in its code n+1 times, and only one of the groups of regions delimited by these references will refer to regions that will be on the outside of the composite region: the other groups refer to regions that will be enclosed and hence should not appear in the code. How do we differentiate between these groups ?

It is in fact fairly simple to find which groups of regions are enclosed when we notice that only a finite number of regions can be enclosed in each set. Also important is the fact that we may 'chain' across the representation to find a region which has the edge of the picture as a neighbour, simply by recording all the neighbours of our given region, and their neighbours, and so on, until we find the edge. The algorithm to merge neighbour codes with enclosed regions can then be expressed as:

- Rotate both codes so that their first member is the other merge candidate. We then have a group of regions between the first and the second instance of the codes, which we record separately for each code.

- Now we wish to decide for each candidate whether this group represents an enclosed region or not. Notice that the groups must be considered separately for each candidate as we have no way of knowing whether the groups correspond to the same set of regions. We then, for each group, look for the edge of the picture in the manner described above: if we find it, then the group must represent an outside group; if we do not, but rather enumerate a finite set of regions (where no new regions are found by looking at new regions' neighbour codes, and after a while there are only regions whose neighbour codes have been inspected) then we have an inside group.

- If the group was an inside group, we discard it by rotating the code so that the second instance of the other candidate is in

the first place. If it was an outside group, we discard the rest of the code, and that code is ready for merging.

- When both codes are ready for merging, (each will have only one instance of the other candidate's region number) they are merged in the same way as simple codes.

Figures 4 and 5 may clarify this explanation. We give an example below.

Our picture has produced the following neighbour codes: (we use -1 to indicate the edge)

region	code
1	-1 2 4 3 2
2	1 3 4 1 -1
3	4 5 6 4 2 1
4	3 1 2 3 6 5
5	3 4 6
6	4 3 5

and we wish to merge regions 3 and 4.

The first group for 4 is 1 2 (3 is not included; we are merging to 3 and to regard 3's code would confuse the issue as all groups would then be outside groups). Now for 1, we obtain as new neighbours -1 3 and 2: (4 is a candidate, and is ignored) b t -1 indicates the edge, and hence this group is the one we want, and the code for 4 becomes:

3 1 2

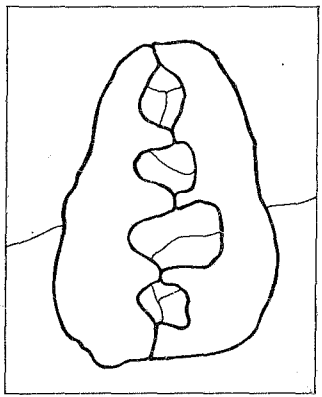
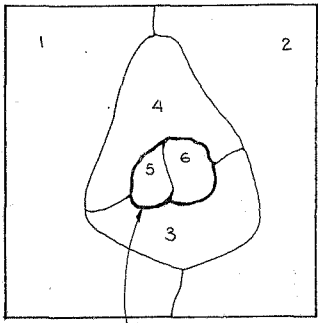


FIG. 4 Two regions contacting in n+1 points enclosing n sets of regions.



The inside group: as we don't consider 3 or 4 when obtaining new neighbours, this is an enclosed set containing only 5 and 6.

FIG.5 The neighbour code merging example.

The first group for 3 is 5 6 and no new neighbours are found from these regions' codes, (ignoring 3 and 4 for the same reasons as above) so that this group is an enclosed group and we may remove it from the code and get for 3:

4 2 1

Now we may merge 4 and 3 conventionally.

This representation is implemented fairly simply by keeping a table of the codes, and a cross-reference table in which we may store information as to which region has been merged to which.

CONCLUSION.

The power of the neighbour code representation is the simplification it provides in the representation of the segmented picture. The codes form a representation in which it is simple to merge regions, and which is in general better-suited to the needs of a knowledge-intensive system than is the conventional representation of the segmented picture.

THE PARADIGM AND ITS USE.

In this author's opinion, vision is a function very deeply embedded in living systems, and is not correctly described as an intelligent act. The problem of general scene analysis does however seem to require for its solution the employment of some form of knowledge, which appears intrinsic to most seeing life-forms. Although we claim that knowledge will have to be employed to produce a general scene-analysis system, we cannot claim that this knowledge will be easy to find, or to express. It seems likely too that the KS's will proliferate, and there seems to be no good reason for expecting that the knowledge base will not hold internal contradictions. With this in mind, we still feel it productive to propose a new paradigm, the motivation for which has been given above, in which scenes may be analysed in a knowledge-intensive fashion.

GUIDING CONCEPTS: THE PARADIGM ITSELF.

The major tenet, and the theme of this dissertation, is the belief that pictures should be analysed in terms of explicit knowledge about pictures, and about what objects look like on pictures, rather than in terms of models made of the picture. A model of the world must be used, to have sufficient knowledge to interpret the picture, but the picture need not be modelled, so that picture-level operations may be guided by knowledge both about the picture and about the world.

The second tenet is that the picture must be handled in terms of regions: this is important, as the processes of edge detection and of outline growing do not easily allow backtracking to incorporate world knowledge. This tenet assumes that a segmenter is available, that is able to segment the picture such that each defined primitive is represented by at least one region. We envisage backtracking to the level of undoing incorrect merges; we do not at present see backtracking to a level of a resegmentation of the picture (although this could be achieved, more simply than by the process which would apply if the picture had been interpreted in terms of its edges).

The employment of knowledge should occur at many levels, from planning the process by which primitives are merged, to using the context in which the picture is presented to reduce the quantity of knowledge required to interpret it.

How then do we intend a system to find a given object in a scene? Firstly, it must reduce the description of the object to primitives which it understands, using the knowledge available to it. It must then find these primitives, with the relationships between primitives that it requires. If it cannot find all of the required primitives, it should inspect the set which it has, and decide whether it can still reasonably find the object by lenient reclassification of some regions. Thus, the system has a defined goal structure, which will be as follows:

- Find out what should be found in the picture (i.e. find the main system goal).

- Factor this goal into a list of primitives and a description of structure, using available knowledge. If this is impossible, the system is entitled to fail. (How would a human respond when asked to find an object in a scene, when he does not know what that object looks like, and there are more than one unknown objects in the scene?)

- Now find as many of these primitives in the scene as possible, by using knowledge to generate and test merges of possible candidate regions, and to classify these regions or groups of regions.

- Finally, find a group of primitives the structure of which corresponds to the structural description stored; if a promising group is missing a primitive, attempt to find it by reclassifying regions where the structure of the object indicates that the primitive is likely to occur. If this group can be found, the system goal has been found, and the next goal may be requested.

We have seen that the use of regions should simplify the process, as backtracking is simplified, and the system uses knowledge; thus, it has been produced under the new paradigm. But what is the structure of such a system ?

A SYSTEM DESIGNED TO ANALYSE SCENES

The task of this system will be to find a given object in a picture, or to say that the object is not in the picture, by examining regions and groups of regions with the intention of finding a group that best corresponds to the object required. (We will refer to this object as the main system goal.) The system should be able to evaluate its attempts to merge regions to form primitives, and to plan a best course of action by which it should be able to find the main system goal. To achieve these aims, the system will use knowledge.

The system will, then, have a knowledge base, which will need to contain both world knowledge and picture-level knowledge. The system will consist of sections, which will be self-contained, and which we shall call Knowledge-Application Blocks or KAB's. Each KAB will refer to and use a different form of knowledge, and will not in general require access to the entire knowledge base. However the KAB which uses world knowledge to factor the main system goal may need at some stage to have a form of control exercised on its knowledge base, to prevent it from saturating. And we have discussed the available strategies above; in this case, it would appear that the best strategy is to order the knowledge base according to its relevance to the main system goal. Thus, in a search for a telephone, knowledge about handsets will be made more accessible than knowledge about hands. We refer to the knowledge conveyed by this type of strategy as "contextual knowledge", as it is knowledge about the context in which the picture appears. As we have said, it does not appear easy to implement this form of knowledge management. A block diagram of the system is shown in figure 6. Knowledge-application blocks, with their

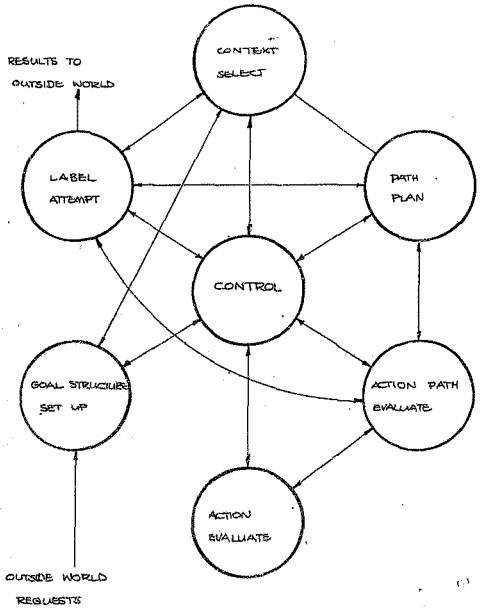


FIG. 6 A system designed under the new paradigm.

relevant knowledge bases, are shown as circles, and the flow of information is shown by arrows. The duties of each KAB will be described below.

KAB	DUTIES
control	This block will recommend changes of context and will manage the process by which control is passed from KAB to KAB.
context select	This block will provide contextual information by ordering the knowledge base.
goal structure setup	This block will put together a goal structure to allow the finding of complex structures by finding their components, using available world knowledge, and will inform the action evaluator as to which structures should be found where, if necessary by lenient classification.
action evaluate	This block will evaluate any merge suggested in terms of the properties of the merged region and of what is being sought.
action path evaluate	This block will evaluate a series of merging decisions in terms of picture knowledge and world knowledge, and will recommend or penalize a proposed merge in terms of the other merges arising from it. It should be able to decide between two

possible merge decisions in terms of their overall effect, and then report to the path planner.

label attempt Given a region with given physical properties, this block will decide what it is likely to represent. This block will also attempt to label sets of regions, given appearance and structural information.

path plan This block will provide a merging plan to meet the goal structure, and will offer alternative merge paths to the action path evaluator. Thus, it will interact with the action path evaluator to provide an overall strategy.

actor This block will take the actions recommended by the path evaluator.

A system structured as described above should be able to analyse fairly general scenes; however, it will require sophisticated techniques to construct, and would appear to be beyond the capabilities of the techniques described in the present literature. Useful information should be obtained by prototyping a system with some of the features of this system. To illustrate the intention of this design, we shall describe the functioning of the system when required to find a telephone in a scene. We assume a black dial telephone for argument's sake.

- The goal structure setup KAB will receive the main system goal, and factor it, for argument's sake, into "body", "body cord", "handset", and "handset cord". "Body" will be factored again

into "dial" and "main body"; the other sub-goals will be regarded as primitives. The primitives are described as follows, for simplicity:

PRIMITIVE	DESCRIPTION
body cord	dark, long and thin, small, low reflectance, low texture.
handset	dark, banana-shaped, medium-sized, medium reflectance, low texture.
handset cord	dark, long and coiled, small, medium reflectance, medium texture.
dial	*light, round with holes, medium-sized, medium* reflectance, low texture.
main body	dark, square, large, low reflectance, low texture.

- The picture is then segmented, and the control KAB will pass control to the path planner KAB, which will provide a path of this type: find any regions which are primitives in themselves; then generate merges between dark neighbouring regions. Test these merges with the action path evaluate KAB, which will test individual cases with the action evaluate KAB. The intention is to merge towards at least one primitive.

- Assuming that a primitive has been found, say the handset cord, the path planner KAB will pass the action path evaluate KAB this information, by informing it that the handset itself is likely to be found as a neighbour, or as a group containing a neighbour, of this primitive. Similar information will be passed for the "main body" primitive. Assume that with this new information, the "main body" primitive is now found.

- The path plan KAB now may pass to the action path evaluate block a new strategy, where the likely position of the dial is assumed, and the likely position of the cord is offered. Assume that the body cord is now found: the handset is still lacking.

- The control KAB will now pass control to the goal structure setup KAB, which will inspect its world knowledge and inform the path planner KAB, through the control KAB, that the handset may be either on the main body, at the other end of the handset cord, or absent, with likelihoods for each situation.

- Finally, the path planner, having had a merge path accepted by the action path evaluate KAB, and with the label attempt KAB not having found the handset using this path, will pass the entire set of regions to the label attempt KAB, as it knows that it need not find a handset. The label attempt KAB will then use knowledge left to it by the goal structure setup block KAB to label the set of regions (which are also primitives: we merge regions in the representation in the process of merging up to the primitive level, and only in a scratchpad form when we attempt to find structures) as a telephone. If this labelling attempt

fails, the process starts again, with a different set of dark regions being tried as a start to the process.

It can be seen that to a large extent the control KAB acts as a scheduler: in fact, in the prototype, the control KAB has been implemented by structuring the way that the KAB's are invoked, rather than by configuring a separate KAB.

The system as described should be able to analyse scenes with considerable generality, if it can be implemented effectively. Certainly, the structure of the system is such that if it is able to analyse a scene, it will be able to analyse at least a class of scenes. There are, however, potential problems with this design.

POTENTIAL PROBLEMS

It must be realised at once that the design described is a pencil and paper design. It is the product of very recent thought, and has not been proven, implemented, or deeply tested, and as such should be expected to have errors in it. A very simple version of this system has been prototyped, but, although it shows promising characteristics, it does not prove the validity of the design. Vision is a complex activity, and this author feels it unlikely that the design presented has solved the problem of scene analysis; it is presented as a new approach, which promises good results. I am confident that the tenets of the paradigm presented are sound, however, and that they are potentially productive.

The knowledge base is likely to be huge, and it is not easy to see how its growth may be controlled. The knowledge which drives human vision systems is not known, and it may be so deeply-embedded that to discover it at all will require a roundabout approach involving an operator grading the response of the system to scenes. It is not a forgone conclusion that this knowledge can be discovered at all, although it does not appear impossible.

The results of the prototyping exercise are described below, and it will be shown there that hope exists for the production of a general scene-analysis system.

CONCLUSION

A paradigm for the analysis of general scenes has been described, and we have shown the design of a system that will perform this task under this paradigm. Criticisms of the paradigm and of the design have been presented, but it is felt that further development of this system is a worthwhile goal.

THE DESIGN OF THE PROTOTYPE SYSTEM.

The system described above has been prototyped in a very limited form, in an attempt to demonstrate the validity of the paradigm and to show that useful systems may be designed within it. The prototype has been constructed with every simplification possible, to allow its construction; the reader will have gathered from the introduction that many complex issues have been sidestepped. The prototype, although not yet analysing scenes, is showing some promise, and the neighbour code representation has been effectively implemented in it. We will discuss the design of this prototype here, and its implementation and the results that support the validity of the paradigm in a later chapter.

The prototype was intended, not to display the more advanced of the concepts discussed above, nor to test the whole of the structure of the system described above, but to demonstrate the desirability of continuing with work of this type in this field and, more particularly, in this paradigm with the intention of showing the validity of the design described above. As a result, not all the KAB's were implemented, and those that were, were implemented in a limited form. The KAB's implemented were: goal structure setup (called fgoal), action path evaluate, action evaluate, actor and a form of label attempt (gtmer). The design of the prototype will be discussed under four headings: the block design of the system, the design of the data structures, the design of the rules, and the algorithms used in the KAB's.

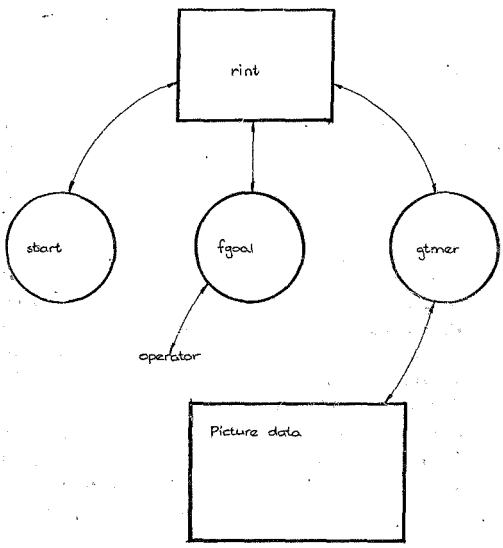


FIG. 7 Block diagram of the prototype system.

THE BLOCK DESIGN OF THE PROTOTYPE SYSTEM.

We have simplified the construction of the system described above by dividing the duties of the KAB's into two clear blocks, whose duties are respectively: find the description of the main system goal in terms of primitives, and find the goal described in the picture. The first block we call fgoal, the second gtmer. We will describe here the duties and the techniques used in these blocks, as well as the two other blocks the system has inherited from its implementation: start, inits, and the loader, mrint.

Refer to figure 7, which shows the block diagram of the prototype system; the implementation has required the introduction of a further block, not shown on the figure, for initialising data structures. Note that we do not have here a situation where each block acts on data, and then passes this data to a further block (hence the form of the block diagram). Instead, the loader, known as mrint (the program name), loads each block as a segment until the block terminates, informing the loader which block is to be used next. Each block must then terminate, having changed the system variable which informs mrint which block should be loaded next (we call this the system block context, or SBC). Thus the operation of the system is as follows:

- To start the system, mrint is invoked. In turn mrint retrieves from disk, without consulting the SBC, the picture representation left there by the representer program, and then loads the inits blc k (not shown in figure 7), which initialises the system data structures. This block will always terminate "happy", unless there are hardware

errors, and will not affect the SBC. The SBC is then set to start by mrint.

- Mrint then retrieves the first KS from disk. Each KS has an SBC field, and if this field does not agree with the current SBC, the KS is discarded, and the next KS retrieved. The SBC at this stage will be "start", and when a KS is found with this value in its SBC field, the start block is invoked, and the KS passed to it. When this block has finished with the KS, it returns control to mrint. The process continues until the start block changes the SBC, which it will always change to "fgoal".

- The SBC is now "fgoal"; mrint will continue to invoke KS's, discard them if their SBC field does not agree with the current SBC, and load the appropriate block if it does. After it has been passed a number of KS's, fgoal will terminate. If it has been able to discharge its duties, it will change the SBC to "gtmer"; if it hasn't, it will not touch the SBC, but will inform mrint that it has failed. Mrint will, in turn, fail.

- If the SBC is now "gtmer", gtmer will be loaded and will run each time a KS with this SBC field is loaded. When it in turn has discharged its duties, gtmer will set the SBC to "finish", and the system will halt.

Thus, each time a KS is retrieved with its SBC field equal to the current SBC, the appropriate block will be loaded, will operate on the KS, and will then terminate, changing the current SBC if appropriate. The next KS is then retrieved.

It can be seen that this strategy is equivalent to using a number of independent knowledge based systems which intercommunicate. It is, however, slower as, if we use the latter strategy because we require certain operations to be performed before others, we may implement a form of pipelining to increase the system's speed. With this strategy, communication issues are simplified, as all system data are held in common blocks, and are then available to any loaded block. Note that the loader approach is not ideally simple; the original intention was to construct a single program, with blocks implemented as procedures, and the present loader a form of case block. This was not possible, as it led to code that exceeded the available code space in the computer used. The duties of the blocks will now be described.

The first block invoked by the main system is the inits block, which exists entirely as a result of the implementation of the system. The minicomputer on which the system was implemented has an address range of only 32K words, which means that any program that compiles down to greater than 32k words (program and data) cannot be run. The system was implemented using active data structures, where each data structure is implemented as a procedure with operators defined as parameters; the data structures require initialisation, which was implemented as an operator defined on the structure. Towards the end of the prototyping phase, the code began to compile down to more than 32k, with the result that the code had to be segmented, so that separate segments of code were loaded into the code space by mrint, and run. It was felt that it would be simpler to collect all the data-structure procedures into a segment whose single duty was to initialise them (the data themselves being kept in common) than to add code that would initialise the structures to the main program.

This segment is inits, which is used only to initialise those system data structures that require it.

The second block that arises from the implementation of the system is the start block, which would in fact be a part of the goal structure setup KAB, albeit a minor one. At this stage, the start block is used only to recover the main system goal; it seemed possible that it would be required for other functions, and was thus implemented as a complete knowledge based block, as the infrastructure existed. Thus the only two rules (in this case they are more like rules or language statements than KS's) that exist in start are:

- Always recover the main system goal, and
- always change SBC to fgoal.

It will be seen that, in this case, the order of their execution is important.

We have discussed mrint above: the duties of this block are simple, being another block that has arisen from the implementation of the system. Mrint is required to retrieve from disk the representation of the picture, and to set up the system data structures that contain this representation. It then loads inits and, when that block has terminated, begins the process of loading KS's. If an SBC field matches the current SBC, it loads the block indicated by the current SBC and waits for it to complete. Mrint steps through the [illegible] one, and continues to do so until the SBC has changed to "finish". At this stage, it proceeds to step through

the KS's without loading any block, but this is as a result of messy coding - it could as easily terminate.

We come now to the interesting blocks; we will discuss fgoal first. The duty of this block is to factor the main system goal into primitives, and to keep a record of the structure of the main system goal in terms of these primitives, using the techniques we shall describe below. The primitive and structural descriptions are kept in system data structures for reference by gtmer. The process by which this block is invoked makes its execution particularly slow, as it is unable to search the knowledge base actively, but must accept or reject KS's as they are passed to it. This is unfortunate, but the segment was simply easier to implement in this fashion.

The second major block, gtmer, is required to use the primitive descriptions provided by fgoal and to merge regions to form these primitives, by generating and testing merges (hence its name). This block will then generate groups of regions as candidates for structure, and will test them against the structural descriptions stored by fgoal. Generate and test has been used here as a technique, as the path plan/action path evaluate structure is intended to use a form of that technique, albeit with greater sophistication than this block. The block contains all the neighbour code merging infrastructure, as well as the KS evaluation infrastructure.

By now the reader should understand the high-level design of the prototype, and we shall begin to discuss the more solid design issues.

THE DESIGN OF THE DATA STRUCTURES.

In a system of this kind, it is important that data are correctly employed. The system requires two major categories of data: Simple data are employed within a block while it is running, and will not be needed the next time the block is invoked. System data are required either by more than one block, or by one block on more than one occasion. We will describe only the latter data structures: the former are described where necessary where the blocks are described.

We require the following system data structures:

- a structure to store a compact description of the structure of the main system goal,
- a structure into which sub-goals that have been found may be demoted, and where a record of sets of regions that match them may be kept, and finally,
- structures for the fgoal block to use as a scratchpad while expanding a goal.

We use the term "system sub-goal" to refer to a name (of either a composite or a primitive) that is a part of the structure of the main system goal, and has not been expanded. We will discuss the latter structures first, as they are important to the discussion of the others.

There are two structures used by fgoal as a scratch pad: the first, the system composite list, is used to keep track of the system sub-goals, and the second, the system primitive list, is used to keep track of the system primitives. The system composite list has been implemented as a forward-linked list; this structure has been used as it is simple to remove an element from a list in a way that does not impede searching. (No inspection of flags, or reshuffling of records, is necessary.) The structure contains the names of all unexpanded names in the factoring of the main system goal. Thus, it contains a list of the objects whose definitions the system still requires, and if a name does not appear in this list, a KS describing an object of that name is not of interest to fgoal. The system primitive list contains the description of all primitives which have been encountered in factoring the main system goal, and whose names were part of the system composite list when they were encountered. This list is not in fact a linked list, as it is not searched in this form, but is a simple random access structure.

The structure of the main system goal is stored in the structural list. It is implemented as a forward linked list, each element being a set of three stacks, and a linked list being chosen for the same reasons given for the system primitive list. The elements take the form described because of the structure of the KS's, whose syntax is such that their evaluation is simplified by the use of stacks. Thus, when a structure that has been recorded must be evaluated for a set of regions, the appropriate record from the list is loaded into the evaluating procedure's stacks, and the KS may be evaluated.

The need for a structure into which sub-goals that have been found may be demoted, and where a record of sets of regions that match them may be

kept will become apparent when the algorithms by which the blocks operate are described; we will call it the structural primitive list. It is a simple array of records, with a "next record" pointer being maintained, as records are not deleted, and garbage collection overheads will not become apparent. Each record contains a field in which the name of the new primitive (that is, either a true primitive, or a composite to which a match has been obtained) is stored, and a field in which the numbers of the regions which match this new primitive's description, are stored.

The system data structures are important to the functioning of the block algorithms. We have described these structures, and will describe the algorithms when we have discussed another subject important to understanding the algorithms, the syntax of the KS's.

THE SYNTAX AND STRUCTURE OF THE KNOWLEDGE SOURCES.

It is trite that a restrictive and carefully-chosen syntax will in general lead to a language which is simple to implement. The knowledge sources can be thought of as forming a simple language, by which the appearance of objects in pictures may be described, so we have used the above principle in their design to simplify the use of the KS's. Their syntax has been designed with simplicity of implementation being the prime consideration.

A KS contains four fields: a number field, an SBC field, a condition field, and an action field, in that order. We have used the

"condition-action" terminology here for simplicity, and are aware of its inconsistency with the dislike of the rules terminology expressed above. The use of the SbC field has been explained above: the number field is used to make the system error messages more meaningful, so that the user knows in which KS the system has failed. The uses of the condition and action fields will be explained below.

The condition field.

The condition field consists of a set of conditions linked by logical operators, forming a logical expression, which implies the action field. In general, if the condition field is true, then, depending on in which block the rule is defined, either the object named in the action field has been found, or the action named in the action field should be taken.

The common expert system practice (For example, in the *MYCIN system* described by Buchanan and Shortliffe (1984) is to provide only the logical operator AND in the KS language. Thus, if we wish to express that it will rain tomorrow if it is raining today, or if it is overcast today and the barometer is falling (I do not vouch for this KS), we would be obliged to use two rules, and say:

- IF it is raining today, THEN it will rain tomorrow.
- IF it is overcast today, AND the barometer is dropping, THEN it will rain tomorrow.

The OR operator is then achieved because either one, or both, of the KS's may fire. We have not used this technique because of the opacity it lends the rule base (It is not easy to see at a glance just what segment of the rule base relates to what conclusion.), and because provision of a full set of operators is not simple in the rule structure used.

We have provided as logical operators AND, OR, and NOT. AND and OR are entitled to only two operands, and NOT may have only one. The operands may be expressions, or simple conditions. The operators are expressed in post-fix notation, because it is extremely simple to construct an interpreter for this notation, oriented as it is toward a stack structure. Nesting is allowed to a depth of 20, the limit being imposed for practical reasons only.

We have mentioned above the important simplification that has been used in the construction of the condition list syntax: We do not allow a *condition to be defined by the condition* list of one KS (i.e. by appearing in its action field) and then to be used by another KS as a condition, if the block that uses the KS's attempts to evaluate them immediately rather than storing them for structural information. This is equivalent to forbidding the use of KS's which require backward chaining for their evaluation, if the block in which they appear evaluates KS's immediately, and is a simplification because when we evaluate KS's, we must define variables for condition evaluation. If we then wish to chain through the knowledge base, we must carry these variables. We will thus incur a considerable overhead to no good purpose, as, if a condition may be defined in the fashion we are forbidding, either it must be capable of being resolved into an expression, or, if it cannot be resolved, it is meaningless anyhow. Thus we gain a typing overhead in entering KS's by losing

a large coding overhead. An implementation in a language that allows recursion will not incur this overhead, as far as I can see.

The situation is very different when the block using the rules uses them to record structure: in this case, the rules are being loaded into data structures for future reference, and when they are being evaluated, they will be evaluated in a forward-chaining fashion, the system having already decided what conditions are primitive, so the problem of recording variables is not as severe. At the same time, in this case it is essential to allow backward references, and the overhead incurred is more acceptable.

Thus, for example, the system of KS's shown below is illegal (I do not vouch for these KS's either.):

- IF it is overcast AND the gardener's joints are aching, THEN it is likely to rain.

- IF it is likely to rain AND the barometer falls, THEN it is raining.

But we may easily resolve the above system into:

- IF it is overcast, AND the gardener's joints are aching

  AND the barometer falls, THEN it is raining.

Conditions may have up to seven operands (a result of the implementation; we have not required in practice more than three), none of which may it-

self be a condition. This results in a simplification of the code. We have not implemented variable-length parameter lists for the conditions, again for the sake of simplicity. New conditions are easily defined.

The action field.

The action field typically contains either the name of an action (with parameters), that must be performed, or the name of an object that is being described by the KS, with a parameter to say whether the object is composite or primitive. Multiple action names are not allowed (for implementation simplicity), and multiple object names would be meaningless.

Examples of rule use.

To this point it has been implied that the same structure has been used to describe objects and to imply actions; this faculty is achieved by a slightly different interpretation of the stacks generated by the KS. Thus (noting that condition and action fields are separated by a "*", and that a condition parameter list is enclosed in diamond brackets, to simplify parsing the KS):

```
(((((ON<LEG1,BODY> ON<LEG2,BODY>)A  ON<LEG3,BODY>)A ON<LEG4,BODY>)A
ON<BODY,HEAD>)A NEXT<HEAD,TRUNK>)A*ELEFNT<COMP>
```

will express a description of an elephant; the name is spelled elefnt because, again as a result of the implementation, names are required to have no more than six letters.

The KS may be described as saying: an object, with a single body supported by four distinct legs, and with a head on that body, and a trunk attached to the head, is an elephant. It says as well that an elephant is a composite: that is, the parts in terms of which the elephant is described must themselves be factored into primitives, which are described in terms of picture properties. Notice that the numerical tags appended to the names of the elephant parts describe these parts as being distinct. The elephant must by this description have four legs. The tags are stripped before the knowledge base is searched for the part's description, as it is not appealing to have to enter KS's describing leg1,leg2,leg3, and so on.

KS's that describe primitives, describe them in terms of picture level properties. Thus, a rock would be described as follows:

```
((((SHAPE<DK> SIZE<DK>)A REFLEC<MEDIUM>)A TEXTUR<MEDIUM>)A
CONCAV<HIGH>)A*ROCK<PRIM>
```

This says that a rock has an unknown shape and size, a high concavity (We use this term to mean a ragged outline.) and a medium texture and reflectance, and that it is a primitive. It is accepted that there are likely to be problems with the information given to represent the primitive, but at the present state of implementation it is sufficient, and it is open to expansion.

The following KS expresses an action:

```
(((((ROCK<R1,VERY> ROCK<CENTRE,Q+>)A (ROCK<R1,VERY> BACK<CENTRE,Q+>)A)O(.
ROCK<R1,VERY> ROCK<CENTRE,DK>)O RNNUM<CENTRE,1>)A*MERGE<CENTRE,R1>
```

This rule says that if the first region in a given region's neighbour code belongs strongly to the rock class, and the given region itself belongs to the rock class to a degree of quite or greater, or if the given region's first neighbour belongs strongly to the rock class, and the region itself belongs to the background class to a degree of quite or greater, or the given region's first neighbour belongs strongly to the rock class, and the given region belongs to the rock class to a degree of don't know, and the given region has only one neighbour, then that region and its neighbour should be merged. This is the same as saying that one may not have rocks with holes in them.

An alert reader will have noticed that as the neighbour code is cyclic, it is meaningless to refer to the first region in it. What R1 in fact refers to is all possible first regions in the code: thus, by referring to R1 and R3 we may refer to any two regions in the code offset by one region. We will discuss this point in greater depth below.

It is obvious that the KS format is extremely opaque; the system has not in any way been designed for an unskilled operator. It is of course possible to attach a natural language interface to the present simple rule editor, but this was felt to be far beyond the scope of this prototype.

We have discussed the system block diagram, the system data structures, and the syntax and structure of the KS's: we will now discuss the algorithms used by the blocks.

THE ALGORITHMS USED WITHIN THE BLOCKS.

The algorithms used by the blocks fgoal and gtmer to discharge their duties will not be presented formally, but will be described in detail. They are essentially fairly simple, but a formal presentation would involve a mass of unimportant detail, and would cloud the issue. We will describe how the system evaluates a KS, how it factors the main system goal and stores its structural description, how it generates and tests merges, and how it forward chains to the main system goal. The last algorithm has not yet been implemented, and as such is not definitely free of problems. The blocks based on the other algorithms work, but this does not guarantee the correctness of the algorithms.

These algorithms are not truly fail-safe, but errors encountered by the implemented code are trapped and reported; we will describe the error-handling facilities when we describe the implementation of the prototype.

How the prototype evaluates a KS.

We have discussed above the syntax and structure of the KS's and have said how they have been oriented to simplify evaluation of the KS. We will show here just how they simplify this process, and how the stack-oriented structure of the KS's simplifies their implementation.

The process of KS evaluation boils down to four steps:

- The KS's must be decomposed into their stack representation.
- The condition operands for each condition must be fixed.
- The conditions themselves must be evaluated.
- Finally, all the values of the conditions must be combined to form the value of the KS.

Decomposing KS's: This process is implemented by a simple state machine. The KS is read from back to front (this leads to the stacks being in a more convenient order, and the names are easily reversed) and if an operator ("(", ")", "A", "O" or "N") is encountered, it is pushed into the operator stack. The start of a condition field will be indicated by ">" (remember we are reading the KS backwards, and notice now the reason for using diamond brackets for delimiting condition parameter lists). After this we expect up to six letters, followed by a ", " or a "<": these letters form the name of the last of the condition operands, which is reversed and pushed into the condition operand stack. If the delimiter

encounte[illegible] it is a ", " then we continue to find and push condition operands until we encounter a "<". We do not store the diamond brackets or the commas, nor do we delimit the operands in the stack as belonging to separate conditions. Note that the limit of seven condition operands is not imposed at this point. A KS exceeding this limit, or a condition containing more or fewer operands than are defined for it, will cause an error based on the disarray of the condition operand stack, as there will be more, or fewer operands than required in the stack. As the system enforces typing on its operands, the error should be reported when a condition receives an operand not intended for it.

When a "<" has been encountered, the following characters, to a maximum of six, form the condition name, which is reversed and then pushed onto the condition stack. The process continues until the entire KS condition field has been decomposed. We catch unmatched parentheses ("(" or ")", not "<" nor ">", which are used only to delimit the condition parameter area) by setting a counter to zero, incrementing it each time a left parenthesis is encountered, and decrementing it for a right parenthesis. If when the condition field has been fully decomposed the counter is not again zero, we have an error.

Evaluating the conditions.: The next process is that of evaluating all the conditions, while retaining their order in the stack. We do this by popping each condition from the condition stack until the stack returns a "fail" status, indicating that is empty (there must be at least one condition), evaluating each condition as it is popped, and then enqueueing the values so that the order is retained.

When a condition is popped from the condition stack, the appropriate condition evaluation procedure is invoked, which then pops the number of condition operands fixed for that procedure from the condition operand stack. These operands must then be fixed: the process is described below.

Once the condition operands have been popped and fixed, the condition must be evaluated. This is a process that rather depends on how the condition is defined, but we will remark that when a condition fixes an operand and receives a vector, it will then use any scalar operand as a vector, by simply filling a vector of the given length with the scalar value, and return a vector result (which will contain as its n'th element the result of the condition operating on the operands in the n'th place of the operand vectors). The result returned by a condition be in the range 0 - 1 (totally false to totally true, respectively).

As the conditions return their results, the results are enqueued, ready for the *KS evaluator, which uses only them and* the contents of the operator stack to evaluate the KS.

Fixing condition operands.: To fix an operand, we associate with it a value and a type. As true variables are not used here, it is a question, not merely of finding the appropriate part of a variable space and retrieving a value, but of determining the type of the operand, and either associating with it the fixed value associated with that name in that type, or retrieving the region number, set of regions or number associated with it. We have four types at present: qualifiers, region numbers, numbers, and system primitives.

A qualifier is a member of the set VERY, QUITE, DK, QUITEN, VERYN, Q+, QN- [2], and is associated with a constant value or range of values, depending on what qualifier it is. This value is then compared to a value within the condition, for the condition to return a one (totally true) or a zero (totally false) value.

A region number is defined either as CENTRE, which recovers the region with which the system is dealing at the time it is encountered, or an integer with an "R" preceding the first digit, which will recover either the entire code, starting with the region occupying the place in the code referred to by the number, or a flag that will force the condition to evaluate as zero (totally false), if the number is greater than the length of the code. As the code starts with an arbitrary neighbour, it is meaningless to refer to the first region in the code, but as the system does not thereafter rotate the code, we can, using this scheme, refer for example to all pairs of neighbours separated by at least two other neighbours, which we would express as R1 and R4. This leads to the point that conditions often return vector, rather than scalar values, as they

---

[2] QUITEN => quite not

VERYN => very not

DK => don't know

Q+ => quite or more

QN- => quite not or less

are evaluated for a set of regions from the neighbour code. At this stage, the vectors will always have the same length, as we are do not yet allow more than one region's neighbour code to be referred to in a KS: the *scalers are then expanded to a vector* with this length, and the whole KS evaluated for conditions which are vectors. The KS evaluator will return either the vector value of the KS, or the maximum scalar value, depending on how it is invoked.

The number type consists of integers without prefixes, which are evaluated as the integer given; these operands allow the inspection of, for example, the number of neighbours a given region has.

It must be seen that a system primitive is in no way the same as an absolute primitive (for which we have used the word primitive), as an absolute primitive is defined by the rule base, and a system primitive may be either an absolute primitive, or a composite for which a match has been found. A system primitive operand refers to the set of matches that have been found for the system primitive named as the operand. As the proposed

algorithm for chaining forward to achieve the main system goal has not been implemented, and hence no condition requires system primitive operands, the details of their use *have not been fully tested. The intention* is, however, as follows:

System primitive operands will refer to sets of regions, and the conditions using these operands will have to be handled as such. Thus the condition will have to be evaluated for all available alternatives for the operands specified. It was intended to implement this for the prototype *simply by setting* up a structure which would hold the current

candidates for all system primitives referred to within a KS, loading that KS from the system structure list, and evaluating the KS. This process would be performed for each legal set of current candidates.

Evaluating the rule.: The rule evaluator uses a technique that is vested in our insistence on post-fix notation and a defined number of operands for each logical operator. We will discuss this in terms of scalars only, although the technique extends very simply to vectors, merely by performing operations on the vectors element-by-element, and storing and retrieving them as vectors. (Elements in vectors will not have to be stored in different parts of the temporary data structure used: that is, storage and retrieval need not be performed element-by-element.) We will first discuss the operations used.

When the truth value is defined as a member of a range of numbers, rather than as one of two, we must define functions on it to represent the logical operators. As we have accepted Dubois and Prade's system ( ), we must accept their functions. Thus, given two propositions, T and R, we define:

The truth value of (T R)OR as:

((T R)OR)TRUTHVALUE = MAX((T)TRUTHVALUE,(R)TRUTHVALUE)

The truth value of (T R)AND as:

((T R)AND)TRUTHVALUE = MIN((T)TRUTHVALUE,(R)TRUTHVALUE)

The truth value of (T)NOT as:

((T)NOT)TRUTHVALUE = 1-(T)TRUTHVALUE

And for any proposition S,

0 <= (S)TRUTHVALUE <= 1

using post-fix notation for typesetting simplicity. (For further information, and a justification of these functions, see the reference.)

The algorithm used to evaluate the KS is then quite simple, and relies on the stack order, and on careful use of a temporary data structure. The data structure is a two-dimensional array, which is two elements wide, and whose depth defines the limit to which unmatched left parentheses may be encountered in the process of evaluation. (That is, if we encounter 20 left parentheses during our evaluation before we see a right parenthesis, the structure must be 20 deep). The evaluation proceeds as follows:

- The temporary data structure (called temp) is initialised to all dummy values, and the bracket counter is set to zero. The first occupant of the operator stack is now popped.

- If the operator stack is empty, there can be only one condition, and the value of the condition at the head of the queue is the value of the rule.

- If a "(" is popped, the bracket counter increments, and the next operator is popped.

- If following a "(", a ")" is popped, and then an operator token (either "A","O" or "N"), we know that we may dequeue either one (for "N") or two (for "A" or "O") values, and combine them according to the operator popped. The resultant value is then stored.

- To store a value, we inspect the first location in temp at the current bracket depth; if this contains a non-dummy value, we inspect the second. If neither contains a dummy value, something has gone wrong, and the technique fails. If the first contains a dummy, we store our value in it; if it doesn't, and the second does, we store our value in the second.

- If instead of popping the sequence "(",")","<operator>", we pop only ")" "<operator>", implying that at least one operator has been evaluated, we must inspect temp for operands. If the first location at the current bracket depth plus one, contains a dummy value, we fail. If this location contains a real value, but the

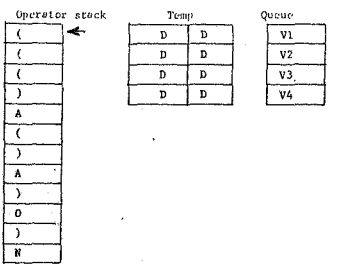
second location at this depth contains a dummy value, we use the value in the first location and a single dequeued value (if we need a second value) as the operands, and evaluate the operator and store the result in the fashion described above. If the second location also contains a real value, the two values in temp are used as operands for the operator (if the operator is a two-operand operator: if it is a one-operand operator, we must fail) and the results are stored as described.

- We continue this process until the operator stack is empty, or until the algorithm fails as a result of an anomalous temp or insufficient queued operands, both of which conditions are caused by KS syntax errors. We will then find the value of the KS in the first location in temp at bracket level zero.

Although the algorithm sounds complex, figure 8 will indicate how simple it really is. Its success is based on the structure of the KS's, which was carefully designed for an algorithm of this type. As long as the KS syntax is correct, the algorithm works.

We have described in detail how the system evaluates a KS; we will now discuss how the system factors a goal.

The KS : ((((C1 C2)A (C3 C4)A)O)N



Step 1: pop (Br = 2

Step 2: pop (Br = 3

Step 3: pop (Br = 4

Step 4: pop) Br = 3

Step 5: pop A We have now seen ()A. Thus, we dequeue two operands and form V1.V2, which goes into the first place in temp at level 3.

We now have:

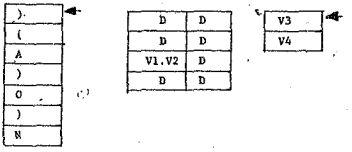


FIGURE 8 : To Illustrate Evaluating a KS. Part 1

Step 6: pop (Br = 4

Step 7: pop) Br = 3

Step 8: pop A We have seen ()A, and are as before entitled to dequeue two operands, and form V3.V4, which goes into the second location at level 3 in temp.

We now have:

Operator stack

)
0
)
A

Temp

D	D
D	D
V1.V2	V3.V4
D	D

(the queue is empty) Br = 3

Step 9: pop) Br = 2

Step 10: pop 0 We have popped)0, and must inspect temp at a level one deeper than the bracket depth to find out what should be done about operands. We find both locations occupied, and hence must use their occupants. We form V1.V2+V3.V4, and store it in the first location in temp at the bracket depth.

We now have :

)
N

D	D
V1.V2+V3.V4	D
D	D

Step 11 : pop) Br = 1

Step 12 : pop N And again, we must inspect temp for operands. There is one, and we require only one, so we form $\overline{V1.V2+V3.V4}$, and place it in the first location at the bracket depth.

Step 13 : The operator stack is empty, so we inspect temp, and fin our result where we expected it. Br is again 1, so we may terminate "happy".

FIGURE 8 : To Illustrate Evaluating a KS. Part 2

How the prototype factors a goal.

The prototype uses some techniques popular in the current Artificial Intelligence literature: this block, for example, uses a technique known as backward chaining, which really deserves description only because the language in which the prototype was implemented does not allow recursion, and some ingenuity was required to get the prototype to chain backward. We will briefly describe the algorithm that the system uses.

When we require the system to factor a goal, we are requiring it to record all KS's necessary to describe the goal completely in terms only of primitives and conditions on primitives. Thus, any reference to an object must be able to be resolved into a set of relationships defined on primitives; the relationships and the descriptions of the primitives must have been stored. Only KS's describing the structures of composites are stored in anything like their original form: *KS's describing primitives are decomposed*, and the descriptions are stored in the primitive list, and later in the system primitive list.

Achieving this is relatively simple. We do it as follows:

- The system composite list is initialised with the main system goal as its only member.
- When fgoal is invoked, the action field of the KS that has been passed to it, is inspected. If the name that appears there, does not appear in the system composite list, then the KS is ignored, and fgoal terminates.

- If the name does, however, appear in the system composite list, the parameter of the name is examined; if the KS describes a primitive, it is inspected and the description is stored in the primitive list. If it describes a composite, the rule is decomposed, and the members of the condition operand stack are entered in the system composite list, if they do not already appear there. (Objects may appear in composite descriptions only as the operands of relationships, which explains why the condition operand stack is searched.) The condition operand stack is searched, but its members are not popped; this is so that at this stage, all three stacks generated by the KS may be entered into the system structure list as a record under the name of the object. Whether the object described is a primitive or a composite, its name is now removed from the system composite list.

- This process continues until the system composite list is empty, when the goal has been fully factored.

Although no checks have been implemented in the prototype, it is easy enough to see that if the entire knowledge base for the block fgoal has been consulted, and the system composite list is unchanged, the goal cannot be factored. This would be caused either by a spelling error, or by an insufficient knowledge base.

This algorithm hardly requires further comment; we will not discuss it further, and will move to a discussion of how the system generates and tests merges.

How the system generates and tests merges.

The "generate and test" technique is also well known in the Artificial Intelligence literature, and we have used a very much simplified version by limiting the scope of both merge generation and merge testing. Merges are generated only up to the primitive level; after that, the merges result from forward chaining through the structural description.

The "merge generate" duties lie solely with the knowledge base: KS's inform the gtmer block which regions should be tested with a view to merging. The process implemented in the present system is extremely simple, and goes as follows:

- The block evaluates KS's and records those candidates recommended for merging by KS's which evaluate to greater than 0,75 true (this figure is purely arbitrary) and continues to do so until it has evaluated all KS's relevant to it for all regions in the picture. This is unwieldy, granted, but the system is a prototype.

- When the evaluator is satisfied, all merge candidates are recovered, and the values that would result for texture, average grey level and centre of gravity for the candidate region, are found. (These parameters are not claimed to be the best for testing a merge.)

  At this stage, we test only that the new average grey level and texture are the same as those of the merge candidates, but it

is intended that the primitive list be consulted to allow more sophisticated, goal-directed merging. If the results are satisfactory, the candidates remain in the merge list.

- The merge list is then passed to the neighbour-code merging section, and the picture representation is updated accordingly.

Although this segment of the prototype is still extremely primitive, it is a segment that will allow great expansion as it is here that the use of active knowledge[3] could be seen to be productive. The intention would be to add to the generate stage's list of candidate regions those regions that structure indicates may well be merged to form a missing primitive. The test stage would be instructed to be lenient in its testing of these candidates, and should also pass out information indicating how "good" a merge is performed under these conditions.

The suggestions described above would lend great power to the system, but are far too sophisticated to have been implemented in the present prototype. We will discuss next how the prototype finds a goal in a picture.

---

[3] Active knowledge was mentioned in the introduction as a concept introduced as far as I can see, by Freuder in his 1976 thesis. The idea is that if one has found part of an object, one has a good idea of where to look for the rest of it.

Finding the goal.

The first point to note here is that this section of the prototype has not yet been implemented* ; thus, the algorithm described has neither been tested, nor deeply examined, and potential implementation difficulties have not yet become apparent.

The process that would have been used is another one familiar in the Artificial Intelligence literature, that of forward chaining. This would have been directed toward finding the main system goal, given the location of the primitives. For the process to start, matches must already have been found for all members of the system primitive list. We intend the process to follow these steps:

- The structural primitive list has been initialised with the names of the absolute primitives; it now contains (because these primitives have been matched), against each name a list of the region numbers which match that primitive. The goal finder must now find in the system structure list a description of an object,

* This was as a result of problems with time, and the fact that the technique of running blocks as segments to fit them into the 32K space allowed, was failing: the segments themselves were getting too big. Any implementation of the algorithm described would have ended up in the gtmar segment, which simply had no space left, and a major re-arrangement of the prototype's structure would have been necessary.

which is described only in terms of relationships between members of the structural primitive list. If such a description cannot be found, the goal cannot be found, and the system fails.

- A data structure is set up, which contains an ordering of the system primitives required for the object to be found, recalling that LEG1 and LEG2, for example, refer to distinct legs, and must therefore be matched to distinct regions.

- The rule is evaluated for this ordering: if it is successful, the name of the object is demoted to the structural primitive list, if it is not there already, and the matching set is entered against it. If it is unsuccessful, nothing is done.

- A new ordering of the primitives is now tried, and the process continues either until the system fails as a result of being unable to find a KS which is in terms only of members of the system primitive list, or until the structure list is empty, in which case the system has found the goal.

It can be seen that this is a simple technique, and that it is in a fairly primitive form, but that it will allow of expansion, and could well become powerful.

The first necessity for the technique to become truly powerful would be the performing of some form of sensitivity analysis on the KS, to determine which primitives were absolutely necessary for the KS to evaluate as true. At this stage, not being able to match all absolute primitives is fatal; with a more sophisticated system, the concept of active know-

ledge may be employed to allow more lenient classification of regions that appear in the right place for primitives, but have not been so classified. Thus, for a powerful system, we need some form of two-way information flow between gtmer and this segment, allowing each block to reconsider its decisions in the light of new evidence uncovered by the other block. This freer flow of information will require a great deal of work to implement, but should generate a system of enormous power and considerable potential.

CONCLUSION.

In the above discussions, we have shown the design of a prototype system intended to show the feasibility and desirability of building the system described in the chapter on the new paradigm. Many of these design decisions were taken during the building of the system, although the major structure of the design had been sketched out beforehand. As it is, the design has produced a system which works to the extent of stimulating curiosity, and in my view, one which has raised more questions than it has answered. Some of the more interesting ones are noted below:

- How can inexact knowledge be handled in a system of this kind ?
- How can active knowledge be deployed really effectively ?
- We have described forward chaining to the goal. Can this be made to work well with an information interchange between this block, and the generate and test merge block ?

- Just what level of explicit knowledge is necessary in this system? *Do we need to define different "knowledge languages"* with different capacities for different blocks, to any greater extent than we have done already ?

- Will a large set of merging KS's manage all classes of pictures, or must sets of merge KS's which are potentially contradictory, be associated with different classes of pictures ?

- Whatever the answer to the previous question, how can these KS's be discovered ?

These are among the major issues raised by the design, and provide ground for extensive research.

THE IMPLEMENTATION OF THE PROTOTYPE.

As we have briefly mentioned the segmentation of the code, the language of implementation and the system on which the system was implemented, we will not discuss these issues again. We will discuss the way in which the prototype was implemented in terms of the data structures, the error handling, the techniques used to code the system, and the code generated.

THE DATA STRUCTURES.

As we have stated above, the system data structures were coded as procedures, with the data in the original implementation being local to the procedure; operators and data were then passed in and out of the structures as parameters for these procedures. We have said above that two categories of data are recognised: system data, which are required either by more than one KAB, or by one KAB on more than one occasion, and simple data, which exist local to a block.

System data had to be placed in a common block declared in the loader, as well as in the block, as the HP-1000 when it loaded a segment, overwrote the previous segment's local data space, which was not restored when the previous segment was reloaded; only data common with the loader were safe. This situation is not ideal from a software engineering point-of-view, and led to an amount of debugging that would otherwise have

been unnecessary, caused by errors such as misspelt common block names, and inappropriate access to structures. The compiler used stored common blocks contiguously, with no run-time bound checking on accesses to subscripted variables. This created an appalling overhead in debugging: about 75% of the errors found, would have been found and rejected by a language that checked bounds on subscripted variables at run time.

Simple data did not in general present such problems. These data (for example the stacks generated by a rule when it was being evaluated) were more tractable, largely as no modifications had to be made to procedures using simple data, when the code had to be segmented.

Data structures were coded to return, depending on the data structure, parameters other than the data requested. In appendix I, where the subroutines that comprise the system are individually described, we have documented the meaning of returned parameters.

ERROR HANDLING.

The error handling provided in the implementation is simple, but it has proved effective at this level of implementation. Each routine is required to return a parameter which indicates whether it has been successful or whether it has failed; in general, if a routine fails, the system must fail, although there are exceptions.

When a routine detects a condition as a result of which it will fail, it will write a message to the output device describing the failure and its source (the number of the current KS). It will then set its status parameter to zero, and will terminate, by going to the statement numbered "70" (70 was chosen for no particular reason). Hence, in any routine, "goto 70" means that that routine should fail and terminate. A routine that calls a routine that then fails, will itself fail, unless the error is one of the small class of non-fatal errors that will be described. Thus, the failure of any one routine causes a chain of failures, until the loader itself fails. In most cases, errors will be reported along the chain, and one error may generate a chain of error messages.

Non-fatal errors are:

- An attempt to pop from an empty condition stack, if at least one condition has already been popped (this means that we have evaluated all our conditions, and must move to the next phase of evaluating the KS).

- An attempt to access information about the n'th region of a region's neighbour code, when that region has fewer than n neighbours (this will return a status value that forces the condition value to zero).

These error-handling techniques have so far proved adequate to the needs of debugging the present simple knowledge base.

CODING TECHNIQUES USED FOR BUILDING THE PROTOTYPE SYSTEM.

The system was built on sound software engineering principles. Data structures were "hidden" from higher-level procedures, by using the approach described. The code itself was structured, although in some procedures the structure is messy, largely as a result of the practice of putting "fixes" in, without performing any revisions to regain structural elegance. All routines are required to indicate whether they have failed or succeeded and, although one may argue that it is possible for a routine to be unable, by virtue of its failing, to report that it has failed, the technique works well enough in practice.

The language used was standard FORTRAN 77 with two exceptions. We have used:

- An option provided by Hewlett-Packard that allows one to force the compiler to demand explicit typing on all variables, which to a very large extent simplifies debugging.

- A routine specific to the HP-1000 was used which recovers a parameter list which is given when the program is run. This meant that the output device could be specified at runtime, by passing t the program, in the invocation, the number of the device to which the output should be directed.

The DO WHILE statement, although standard to FORTRAN 77, was not used; the structure shown below was used instead, largely as I did not realise until too late, that FORTRAN 77 provided this statement.

```
aaa   CONTINUE
      IF (inverse condition) GOTO bbb
        (start of block)
        (.............)
        (end of block)
      GOTO aaa
bbb   CONTINUE
```

Logical variables were not used, as a result of a compiler bug that made their behaviour unpredictable. Instead, all variables which were intended to be of the logical type were declared as INTEGER*2 (as opposed to INTEGER; this associates with the class of variable a unique type, making changes to their declaration simple) and then assigned either zero (false) or one (true) as necessary.

The coding techniques used were fairly standard; the code that was generated is more interesting.

THE CODE.

The prototype system was coded as a large set of subroutines, which are individually described in appendix I. This technique allowed segmentation to be implemented easily when it was required, and led to a system from which simpler subsystems could be cloned.

Using these techniques, some 4300 lines were written, to implement the prototype described above, with the exception of the section of gtmer that chains forward through the system structure list to find the main system goal. This code is presented, in its segmented form, in appendix III, with compiler reports and the load map. It can be seen when certain subroutines are inspected ("strlist", the system structure list, for example), that there are sections of code that are not used by the system as it is at present. These are for the use of the section that was not implemented.

The algorithms used are described in "The Design of the Prototype System" and the neighbour-code merging techniques in "Neighbour Codes". The code implements these algorithms correctly, as far as I can see. The prototype has been tested such that the present version has produced no errors when required to deal with KS's of different types in different blocks. Testing does not allow one to certify a system error-free, but all errors encountered have been dealt with, and I have seen no errors from the present version in a number of tests.

CONCLUSION.

The implementation of the prototype has been described. We have discussed the issues and techniques involved, rather than the details of the code. The system, it is claimed, works, and results will be given and discussed in the following chapter.

RESULTS.

In this chapter, results obtained from work in this paradigm will be described. We will attempt to show that they encourage the undertaking of further work in this paradigm. The results are of two types: those obtained within the paradigm, but not as a result of the use of the system described above, and the results obtained by prototyping this system. The former are more directly encouraging.

RESULTS OBTAINED WITHIN THE PARADIGM.

We have discussed above the work on the particle-size distribution analyser. It is not immediately obvious that this is within the scope of the paradigm offered, but it can be seen that the picture is being handled in terms of regions, and that the process is knowledge-guided, although the knowledge used is not explicit. This work has generated some promising results.

The photographs provided as figure 11 show the quality of picture taken on Deelkraal gold mine, while data were being collected for this project. These pictures are hardly comprehensible, but show gold ore on a conveyor belt. The ore is being conveyed to a mill, and has not been crushed. The problem posed by the large quantity of adsorbed fine particles can be seen to be considerable.

It can also be seen that the prototype of the particle-size distribution analyser (PSDA) handles the problem fairly well, as it has segmented the picture, despite its quality, into regions and has merged those regions *sensibly*. The PSDA has been seen to handle many such pictures; the techniques it uses are closely linked to the ideas described above, with the exception that they are in general more deeply embedded in the structure of the system. Thus, the PSDA finds the particles in the picture by:

1. Segmenting the picture into regions.

2. Merging these regions according to the rules devised.

3. Going over the merged picture to undo those merges which have led to regions of a suspect shape. (e.g. hourglass-shaped regions are not encouraged)

This technique is redolent of the ideas that motivate the paradigm presented, of a knowledge-guided process, and of active knowledge. These ideas have been basic to the work done both by this author and by Berger, who has obtained these recent results with the PSDA. The point here is *that these ideas can lead to useful working systems.*

The PSDA was developed using regions as the feature first extracted, after it became apparent that they would be the simplest feature to work with. (The process described above would be far more cumbersome if edges were to be extracted.) Thus, this work can be claimed to be within the paradigm proposed on this count as well, and its present apparent success can be used to strengthen the justification for the paradigm.

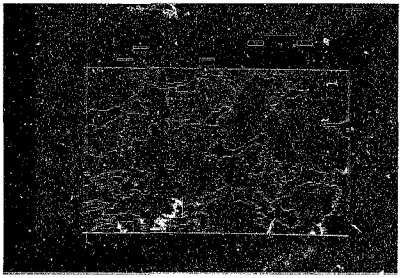




FIG 9 : The above photographs show the results of the PSDA work. The top photograph is of the picture taken on the mine: the bottom one shows the segmentation arrived at by the PSDA.

RESULTS OBTAINED BY PROTOTYPING THE SYSTEM.

I cannot claim yet that the prototype is analysing scenes; it factors scenes, and generates, tests, confirms and implements merges between regions, but the latter activity is not yet guided toward the primitives required, or toward finding an object in the picture. We will show here output from the system as it expands a goal, and merges regions, although the merging is not goal-directed.

Refer to figure 10, which shows the knowledge base on which the system operates in this example. It consists of 12 KS's, the 12'th of which is not used, as the KAB finish has not been implemented, with the result that the system will cycle endlessly through the rule base, when its tasks have been performed, as opposed to terminating cleanly. The KS's will be interpreted below.

There are only two KS's with "start" in their SBC field, and their meaning has been given above, as the KS's used in the start block do not vary. The KS's with fgoals in their SBC field [5] form the description of an elephant, although an arbitrary one. The elephant has been described as having four legs, two eyes, two ears, and a trunk, with no spatial re-

---

[5] Notice a potential source of confusion here. The tags in the SBC fields for fgoal and gtmer are fgoals and gtmerg, for entirely historical reasons, and changing them would have involved unnecessary complexities.

lationship specified between these primitives. The description may be trivial, and inaccurate, but it is sufficient to illustrate the process.

We have described a leg as consisting of a thigh on top of a knee on top of a shin, and "eye", "ear", "trunk", "thigh", "shin" and "knee" are all primitives, whose description although open to argument, is sufficiently comprehensible in the KS language to need no futher discussion. There is only one rule with gtmerg in its SBC field, which expresses the intention that regions with neighbours with the same grey-level, should be merged to them.

At this stage of our demonstration, we must compose a picture and render it into the representation for the system to use, as we are not yet using the software ascribed to Berger in the text above, for computing neighbour codes, because it is easier to compose a picture to the system's liking than to take one. The picture we have used is shown with the representation it generates in figure 11. The output of the system for this picture, given this knowledge base, is shown in figure 12. We have annotated the output as being more useful in that form, than if it were described.

What is worth noting is:

- The system has factored the goal successfully, and has a full description of it in terms both of primitives and of structure.

- The system has generated, tested, confirmed and implemented merges between regions, and the results of these merges are correctly reflected in the representation.

Thus the prototype is doing what we have claimed it can do. What is not perhaps clear is why this is important.

It is important because it is a clear indication that a prototype of this nature can be successfully built, and can be expected to work with simple pictures. It is most unfortunate that the problems that prevented the implementation of the block that would have merged forward to the main system goal, occurred when they did, as the success of this block would have made the prototype a more convincing argument for further work toward building a more sophisticated version of the system described; as it is, the prototype is open to the argument so often successful against work in the artificial-intelligence field, "Sure, it looks good now, but what happens when it encounters the real problems?".

This is unfortunate, as the argument is strong, and as we have said above, there do exist problems far harder than the ones solved here, many of which will not respond well to being sidestepped. I must answer this criticism frankly, by saying that I am not certain that these problems may be solved in that the system in the form described, will analyse real scenes, I must express more faith in the paradigm described (The evidence in its favour is stronger.), which not only has reasonable tenets, but which can also boast a working system (the PSDA) designed within it.

The prototype can best be described as part of an unfinished long-term experiment: it has worked to the extent that it has been implemented, and this indicates that further work on it would be fruitful. If in the full form, it can be made to work on simple pictures, it would represent a strong argument for a larger research effort into the problems raised here, with a view to building an even more powerful system. I am unable

to say here that such a system would work; what I can say is that the work performed to date indicates no reason why it shouldn't, and offers hope for its success.

CONCLUSION.

We have shown results from work on the particle-size distribution analyser, and from the prototyping of the system discussed. The output of the system for a given knowledge base was shown, and the prototype system could be seen to factor a goal, and to generate, test, confirm and implement merges. These results do confirm that further work in this paradigm is justified, but do not yet *fully validate* the *system design* presented above. There is, however, hope for the design, as although it has not been validated, the prototype suggests that it may be. Strong arguments may still be raised against the design and the prototype, but further work along these lines seems to be justified.

```
*****************************************
*                                       *
*  COMPLETE FILE LIST                   *
*                                       *
*****************************************

001*START    *TRUE:DUM*ASK*/FGOAL *



002*FGOALS    *(((((((HAS<LEG1> HAS<LEG2>)A HAS<LEG3>)A HAS<LEG4>)A HAS
<EYE1>)A HAS<EYE2>)A HAS<EAR1>)A HAS<EAR2>)A HAS<TRUNK1>)A*ELEFNT<COMP>*



003*FGOALS    *(OK<KNEE1,THIGH1> OK<SHIN1,KNEE1>)A*LEG<COMP>*



004*FGOALS    *(SHAPE<ROUND> SIZE<VSMALL>)A*EYE<PRIM>*



005*START    *TRUE:DUM*CHCTXT<FGOALS>*



006*OTHERG    *SIMG<CENTRE,R1>*MERGE<CENTRE,R1>*



007*FGOALS    *((((SHAPE<ROUND> TEXTUR<LOW>)A REFLEC<UK>)A DIMENSION<MEDIUM>
)A SIZE<LARGE>)A*EAR<PRIM>*
```

FIG.10.1: The Knowledge base used for the example.

008*FGOALS *((SHAPE<L*T> REFLECTION TEXTOR<MEDIUM>)& CONCAVITY
& SIZE<MEDIUM>)& *PRIM-FRIM*

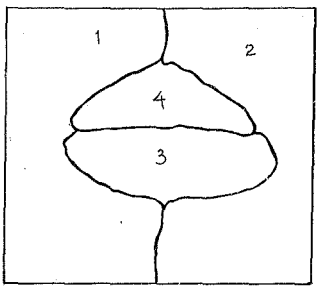
009*FGOALS *(SHAPE<L*T> SIZE<MEDIUM>)&*THIGH<PRIM>*

010*FGOALS *(SHAPE<ROUND> SIZE<SMALL>)&*KNEE<PRIM>*

011*FGOALS *(SHAPE<L*T> SIZE<MEDIUM>)&*SHIN<PRIM>*

012*FINISH *TRUE<DUM>*CHCTXT<FINISH>*

FIG.NO.2 : The Knowledge base used for the example.



Region	Code	Ave grey level
1	2 1 -1 2 4 3	0,0
2	1 3 4 1 -1	0,0
3	1 4 2	63,0
4	1 2 3	63,0

FIG. 11 The picture used in the example with its representation.

```
 11
FNUM
 1
[illegible]
```

The system starts and recovers the main system goal.

```
AGET(GOAL)
SYSTEM PEDURES
GOAL

CORRECT SPELLING,<=6 CHARS

ELEFNT
RNUM
 2
RNUM
 3
RNUM
 4
RNUM
 5
005*START   *IR\              \rGOALS)*
```

```
CHCTXT(FGOALS)
START
FGOALS
RNUM
 6
RNUM
 7
EAR(PRIM)
RNUM
 8
TRUNK(PRIM)
RNUM
 9
THIGH(PRIM)
RNUM
 10
KNEE(PRIM)
RNUM
 11
SHIN(PRIM)
RNUM
 1
RNUM
 2
ELEFNT(COMP)
```

FIG.12.1: The output of the system for the given knowledge base and the given picture.

```
FNUM
 3
LEG(COMP)
FNUM
 4
EYE(PRIM)
RNUM
 5
RNUM
 6
RNUM
 7
EAR(PRIM)
RNUM
 8
TRUNK(PRIM)
RNUM
 9
THIGH(PRIM)
RNUM
 10
KNEE(PRIM)
RNUM
 11
SHIN(PRIM)
RNUM
 1
RNUM
 2
RNUM
 3
RNUM
 4
RNUM
 5
RNUM
 6
MERGE(CENTRE,R1)
  1.000      1MERGE    2    1    2
  1.000      1MERGE    2    1    2
STATE
 1
RNUM
 7
RNUM
 8
RNUM
 9
RNUM
 10
RNUM
 11
RNUM
 1
RNUM
 2
RNUM
 3
RNUM
 4
```

the goal has been expanded.

a merge is recommended.

FIG12.2: The output of the system for the given knowledge base and the given picture.

```
RNUM
 5
RNUM
 6
MERGE-CENTRE,R1>
  1.000       2MERGE     2    2    1
  1.000       2MERGE     2    2    1
STATE
 1
RNUM
 7
RNUM
 8
RNUM
 9
RNUM
 10
RNUM
 11
RNUM
 1
RNUM
 2
RNUM
 3
RNUM
 4
RNUM
 5
RNUM
 6
MERGE<CENTRE,R1>
  1.000       3MERGE     2    3    4
STATE
 1
RNUM
 7
RNUM
 8
RNUM
 9
RNUM
 10
RNUM
 11
RNUM
 1
RNUM
 2
RNUM
 3
RNUM
 4
RNUM
 5
RNUM
 6
MERGE<CENTRE,R1>
  1.000       4MERGE     2    4    3
STATE
```

FIG92.5: The output of the system for the given knowledge base and the given picture.

```
 1
RNUM
 7
RNUM
 6
RNUM
 9
RNUM
 10
RNUM
 11
RNUM
 1
RNUM
 2
RNUM
 3
RNUM
 4
RNUM
 5
RNUM
 6
STATE
 2
RNUM
 7
RNUM
 8
RNUM
 9
RNUM
 10
RNUM
 11
RNUM
 1
RNUM
 2
RNUM
 3
RNUM
 4
RNUM
 5
RNUM
 6
STATE
 3
RNUM
 7
RNUM
 8
RNUM
 9
RNUM
 10
RNUM
 11
RNUM
```

FIG12.4: The output of the system for the given knowledge base and the given picture.

```
 1
RNUM
 2
RNUM
 3
RNUM
 4
RNUM
 5
RNUM
 6
STATE
 4
PRIMLIST FOLLOWS
EYE   ROUND
EAR   ROUND MEDIUMLOW   OK
TRUNK L+T   OK   MEDIUMLOW
THIGH L+T
KNEE  ROUND
SHIN  L+T

          STRUCTURE FOLLOWS

ATORSTACK  1  ANDSTACK  1  CANDSTACK
IN THIS ORDER
ELEFNT
LEG
ATORSTACK
A)A)A)A)A)A)A)A)(((((((((
ANDSTACK
HAS   HAS   HAS   HAS   HAS   HAS   HAS   HAS   HAS   HAS
CANDSTACK
TRUNK1EAR2  EAR1  EYE2  EYE1  LEG4  LEG3  LEG2  LEG1
ATORSTACK
A)(
ANDSTACK
ON    ON
CANDSTACK
KNEE1 SHIN1 THIGH1KNEE1
NCTAB FOLLOWS
1    -1 -1 2 4 5 0 0 0 0 0 0 0 0 0 0 1
2    1 3 4 1 -1 0 0 0 0 0 0 0 0 0 0 2
3    1 4 2 0 0 0 0 0 0 0 0 0 0 0 0 1
4    1 2 3 0 0 0 0 0 0 0 0 0 0 0 0 3
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
NCXREF FOLLOWS
1 1 3 3 5 6 7 8 9 10
RNUM
 7
RNUM
 8
RNUM
 9
RNUM
 10
```

the primites

the lengths of the codes: thus, the code for 1 is -1

4 merged to 3

2 merged to 1.

FIG12.5: The output of the system for the given knowledge base and the given p[illegible]ce.

CONCLUSION

A new paradigm to allow general scene analysis has been presented. A system has been designed within this paradigm, and prototyped in a limited form, to show that the paradigm does indeed simplify this problem. The design and implementation of the prototype have been discussed, and the output of the prototype, operating on a given knowledge base, has been presented. Support from other work for the paradigm has been presented.

We have said above that this work by its nature could not yet demonstrate the validity of the paradigm, or of the system designed within it, but must be limited to showing that there have been no gross errors in their construction, and that they are feasible and valid goals to work toward. I believe that this has been done, by the presentation both of the success of the PSDA work, which has been shown to have been done within this paradigm, and by the results of the work on the prototype, which have been encouraging, as we have shown that the representation used is tractable, that a prototype can be built that may handle pictures using this representation, and that this prototype may factor the goal that it has been presented with, into a form of this representation.

The unfortunate aspect of this work is its necessary superficiality: vision is a hard problem, but it is trying to be in the position I must adopt, where one can say of this work only that it presents a line of approach by which the problem may possibly be solved, and shows that this approach has no glaring errors, and that systems can be built using it, that work under certain circumstances.

REFERENCES.

Armco Autometrics, Manual for a Particle Size Distribution Analyser, 1984.

Barr, A. and Feigenbaum, E.A. "The Handbook of Artificial Intelligence", William Kaufman inc.,1981.

Berger, G.F., "Particle Size Distribution Analysis", unpublished B.Sc.(Elec. Eng.) dissertation submitted to University of the Witwatersrand, Johannesburg, 1984.

Berger, G.F., (private conversations), 1984(a), and 1985.

Brachman, R.J., et al, "What are Expert Systems?", in Hayes-Roth,F. et al (eds), "Building Expert Systems", Addison-Wesley, 1983.

Buchanan, R.J. and Shortliffe, E.H., "Rule-based Expert Systems", Addison-Wesley, 1984.

Charniak, E.C. "A Common Representation for Problem-solving and Language-Comprehension Information", Artificial Intelligence vol.16 pp225-255, 1981.

Davis, R., "Reasoning about Control", Artificial Intelligence vol.15 pp179-222, 1980.

Davis, R., "Content Reference: Reasoning about Rules", Artificial Intelligence vol. 15 pp223-240,1980 (a).

Dubois, D. and Prade, H., "Fuzzy Sets and Systems: Theory and Applications", Academic, 1980.

Forsyth, D.A., "Particle Size Distribution Analysis", unpublished B.Sc. dissertation submitted to University of the Wiwatersrand, Johannesburg, 1984.

Freuder, E.C., "A Computer System for Visual Recognition Using Active Knowledge", A.I.-TR-345, M.I.T., 1976.

Hayes-Roth, F. et al, "An Overview of Expert Systems" in Hayes-Roth, F. et al,(eds) "Building Expert Systems" , Addison-Wesley, 1983.

Lenat, D.B., "The Nature of Heuristics", Artificial Intelligence 19 pp189-249, 1982

Marr, D. , "Representing and Computing Visual Information" in Winston, P.H. and Brown, R.H. (eds) "Artificial Intelligence: an M.I.T. Perspective", M.I.T. press, 1979.

Mandelbrot, B., "The Fractal Geometry of Nature", William Freeman, 1982.

Nazif, A.H., and Levine, M.D., "Low-Level Image Segmentation: an Expert System", I.E.E.E. trans. Pattern Analysis and Machine Intelligence, vol. 6 no. 5 ,pp555-576, September, 1984.

Niemann, H. et al, "A Knowledge Based System for Analysis of Gated Blood Pool Studies", I.E.E.E. trans. Pattern Analysis and Machine Intelligence, vol. 7 no. 3, pp246-259, May, 1985.

Sommer G., (private conversation), 1984.

SPIDER Manual, MITSUI corp., 1985.

READINGS.

Many good introductions to image processing and scene analysis exist: amongst the better are books by Rosenfeld and Kak, Pratt, Gonzales and Wintz, Ballard and Brown, and Duda and Hart. A good introduction to Artificial Intelligence is "Artificial Intelligence", by P.H. Winston, and "The Handbook of Artificial Intelligence" cited above.

APPENDICES

APPENDIX A. THE SUBROUTINES INVOLVED

The subroutines will be presented one by one, with a list of the routines called by the subroutine, and any necessary comments on the subroutine. The variables have been extensively and carefully declared in each routine; we will not do this here.

Program RINT

Calls these subroutines.

- LGBUF
- RMPAR
- CONMAT
- CONHEAD
- BTS

Comments. RINT is the main loader program: it collects the image representation from disk, and then schedules the program segments, as described in the text.

Program INITS

Calls these subroutines

COMPLIST

- PRMLIST
- STRLIST
- SEGRT

Comments. INITS is a segment, which is why it must be defined as a program, that initialises the system data structures that require it.

Program START

Calls these subroutines

- CLFIND
- CSSETUP
- AL

- ASETUP
- APSETUP
- AHEAD
- SEGRT

Comments. START is the segment that is referred to above as the start block. START must evaluate KS's, and hence contains most of the KS evaluation and action code, which is present in all KS evaluating routines. This section of start does nothing but call the appropriate routines.

Program FGOAL

Calls these subroutines

- LISTPGOALS
- COMPLIST
- SEGRT

Comments. FGOAL is the main program for factoring the main system goal. FGOAL calls the necessary routines, but does little itself but perform

an elementary check on whether the main system goal is in the knowledge base, which works like this: if the present KS number is less than the number of the previous KS seen by FGOAL, and the main system goal is still in the system composite list, then we have cycled through the knowledge base without seeing the main system goal, and it is not there.

Program GTMER

Calls these subroutines

- GTMER
- PRMLIST
- STRLIST
- SEGRT

Comments. GTMER is the segment that generates and tests merges for regions. This program again simply calls the appropriate routines: the subroutines do the work.

Subroutine AGET

Calls these subroutines

- COMPLIST

Comments. AGET implements the action GET, which recovers the main system goal from the operator, and puts it into the system composite list.

Subroutine AHEAD

Calls these subroutines

- MRECORD
- CHCTXT
- AGET

Comments. AHEAD calls the appropriate routine to implement the action described by the action name in the KS action field.

Subroutine ANDSTACK

Calls these subroutines (NONE CALLED)

Comments. ANDSTACK implements the operand stack (contains condition names, not condition operand names).

Subroutine APSETUP

Calls these subroutines (NONE CALLED)

Comments. APSETUP recovers the parameter list that is associated with the action in the action field of the current KS, and returns it as an array of parameter names.

Subroutine ASETUP

Calls these subroutines

- BTS

Comments. ASETUP recovers the action field from the KS, once it has been stripped of the number, SBC and condition fields.

Subroutine ATORSTACK

Calls these subroutines (NONE CALLED)

Comments. ATORSTACK implements the operator stack described in the main text.

Subroutine BACKE

Calls these subroutines

- CANDSTACK
- CVARFIX

Comments. Same as "ROCKE" (q.v.), but for membership of the background class.

Subroutine BTS

Calls these subroutines (NONE CALLED)

Comments. Bts makes a loud noise at the operator terminal if the system fails.

Subroutine CANDSTACK

Calls these subroutines (NONE CALLED)

Comments. CANDSTACK implements the condition operand stack.

Subroutine CHCTXT

Calls these subroutines (NONE CALLED)

Comments. CHCTXT changes the SBC to the value passed in, implementing the CHCTXT action.

Subroutine CLEAN

Calls these subroutines (NONE CALLED)

Comments. CLEAN cleans up a neighbour code as described in the text above, by removing multiple instances and end-around instances.

Subroutine CLFIND

Calls these subroutines

- BTS

Comments. CLFIND finds the condition list in the KS, and returns it. BTS is called if an error is encountered.

Subroutine CMERGE

Calls these subroutines

- INSTNO

- SEARCH1

Comments. CMERGE is used to remove sets of internal regions from the codes that are about to be merged. (See text for details.)

Subroutine COMPLIST

Calls these subroutines (NONE CALLED)

Comments. COMPLIST is the system composite list, mentioned in the text above. Names may be added to this list only if they do not match any name in the list. The list may be searched, and returns a parameter that indicates whether it contains a match to the pattern submitted. The length of the list is returned as a parameter, as when it is empty, the system goal has been factored.

Subroutine CONHEAD

Calls these subroutines

- SEGLD

Comments. CONHEAD loads the appropriate block, and passes it the KS.

Subroutine CONMAT

Calls these subroutines (NONE CALLED)

Comments. CONMAT tests the SBC field of the current KS against the current SBC.

Subroutine CSSETUP

Calls these subroutines

- ATORSTACK
- ANDSTACK
- CANDSTACK
- REVERSE
- BTS

Comments. CSSETUP sets up the condition stacks in the fashion described above.

Subroutine CVARFCL

Calls these subroutines

- NUMEVAL
- RNUMEVAL
- QUALEVAL

Comments. CVARFCL, as its name implies, is a clone of CVARFIX, which allows any region reference to appear together with CENTRE in a condition parameter list. CVARFIX will not produce usable results with this type of parameter list, as it was intended to be used with lists consisting of a region reference and any other type of parameter (for ROCK, for example). A maximum of two region references may appear in any parameter list; if two are used, one must be CENTRE.

Subroutine CVARFIX

Calls these subroutines

- NUMEVAL
- RNUMEVAL
- QUALEVAL

Comments. CVARFIX fixes the condition variables as described above. The variables are passed in as strings, and CVARFIX returns the values in a set of arrays, one for each type. Thus if we wished to fix the parameter string CENTRE, Q+, 100, the array passed (uvar) in would contain as its first element the string "CENTRE", and so on. The fixed parameters would be returned as follows: The first parameter will be in regvar(1) (a region type), the second in qualvar(2) (a qualifier), and the third in numvar(3) (a number). Thus typing is performed. The routine cannot handle the system primitive type yet. (See the comment on CVARFCL, before using this routine.)

Subroutine EDGE

Calls these subroutines

- CANDSTACK
- CVARFIX

Comments. EDGE implements the edge condition, which has one operand, a region reference, and returns the value 1. if the region referred to is an edge, and a 0. otherwise. As with all conditions that use a region reference operand, EDGE uses vector operands and returns vector results.

Subroutine GTMERCON

Calls these subroutines

- CLFIND
- CSSETUP
- VECREVAL
- ASETUP

- APSETUP
- AHEAD
- MEROK
- NCFUL
- NEWSTATS

Comments. GTMERCON generates, tests, confirms and implements merges between regions. The generate phase occurs first: all KS's with gtmerg in their SBC field are evaluated for all regions in the picture. Merge candidates are recorded on disk.: The test and confirm phase occurs next. All recommended merges are passed to MEROK, which accepts or rejects them; if they are accepted, they are automatically confirmed by the confirm phase, which occurs next.

The confirm phase then calls NCFUL, and NEWSTATS to update the representation of the picture.

Note that GTMERCON is invoked rather strangely, as it starts in the first phase, and is invoked once for every KS for every region, leaving temporary results each time. It then changes phase, but can still only execute when gtmer is loaded by rint; thus, in the last two phases, it is only invoked when a gtmer KS is found, executes until it decides to change phase, and terminates leaving the number of the next phase as a system variable. This may sound complex, but it was in fact the easiest way I could see of getting the routine to work as I wanted it to.

Subroutine INSTNO

Calls these subroutines (NONE CALLED)

Comments. INSTNO counts the number of instances of a given region in another region's neighbour code.

Subroutine LGBUF

Calls these subroutines (NONE CALLED)

Comments. This is a Hewlett-Packard routine that enlarges the disk buffer area.

Subroutine LISTPGOALS

Calls these subroutines

- CLFIND
- CSSETUP

- ASETUP
- NAMFIND
- COMPLIST
- STRLIST
- PRMLEXP

Comments. LISTPGCALS takes a KS, sets up the appropriate stacks (with CSSETUP), retrieves the name of the object being described, and checks if the name is in the system composite list. If it is, the description is passed to the system structure list, and then to PRMLEXP. If it isn't, the routine ignores the KS, and terminates.

Subroutine MATCHP

Calls these subroutines (NONE CALLED)

Comments. MATCHP matches a given stack against the system primitive list (which contains both absolute and system primitives) to ascertain whether the KS whose condition operand stack it is, may be used next for forward chaining to the goal. If all the names in the stack, are in the primitive list, the KS may be used.

Subroutine MEROK

Calls these subroutines (NONE CALLED)

Comments. MEROK tests merges, and returns a flag to accept or reject them.

Subroutine MRECORD

Calls these subroutines

- CVARFCL

Comments. MRECORD records details on disk of a merge that has been recommended by a KS. The details are recorded on disk simply because they were in the earliest system built, and this has not been removed from the code.

Subroutine NAMELIST

Calls these subroutines (NONE CALLED)

Comments. NAMELIST holds the pointers for strlist, and the names of the structure records. A "next pointer" facility has been provided in this list.

Subroutine NAMFIND

Calls these subroutines (NONE CALLED)

Comments. NAMFIND is a routine cloned from APSETUP, that retrieves from a KS that describes an object, the name of the object described, and whether it is a primitive or a composite, both of which pieces of information are in the action field of the KS.

Subroutine NCFUL

Calls these subroutines

- CLEAN
- NCMERGE

Comments. NCFUL is the calling routine for merging routines using neighbour codes, which checks in the neighbour cross reference table

whether the regions have been merged, and cleans up the codes before they are merged.

Subroutine NCMERG

Calls these subroutines

- INSTNQ
- ROTATE
- CMERGE

Comments. NCMERG uses CMERGE to remove internal regions, and links the neighbour codes when this has been done.

Subroutine NEWSTATS

Calls these subroutines (NONE CALLED)

Comments. NEWSTATS calculates the average grey level, grey level variance, centre of gravity, and number of pixels of a composite region formed by a merge.

Subroutine NNUME

Calls these subroutines

- CANDSTACK
- CVARFIX

Comments. NNUME is the procedure corresponding to the condition NNUM, which has two condition operands, a region reference and a number. If the region referred to has the given number of neighbours, NNUME assigns the vector the value 1. in the corresponding element, which otherwise is assigned 0. NNUM at present is used with only CENTRE as a legal region reference, for simplicity.

Subroutine NUMEVAL

Calls these subroutines (NONE CALLED)

Comments. NUMEVAL returns the numerical value of a string of characters representing a number, for example, the string "100" is evaluated to the number 100.

Subroutine PRMLEXP

Calls these subroutines

- COMPLIST
- PRMLIST
- ANDSTACK
- CANDSTACK

Comments. PRMLEXP expands the primitive list by factoring the main system goal. PRMLEXP is passed a KS's condition field, the name of the object described by the KS, and whether it is a primitive or a composite. If it is a primitive, its description is placed in the primitive list, the operators being ignored, and the condition operands being placed in the field forced by the condition. Fields exist at present for CONCAV, SHAPE, TEXTUR, REFLEC and SIZE. These data are then passed to the primitive list.

If the object is a composite, all the condition operands in its description are passed to COMPLIST, which adds them to the system composite list, if they are not there already (We emphasise that an object may appear in another object's description as a condition operand.).

The name of the object described by the KS is then removed from the system composite list.

Subroutine PRMLIST

Calls these subroutines (NONE CALLED)

Comments. PRMLIST implements the system primitive list, which contains a record of the absolute primitives encountered when the goal was factored. This structure may also be searched for matches to a name, and will return a list of the region numbers that match the primitive.

Subroutine PUTIT

Calls these subroutines

Comments. PUTIT stores the temporary results encountered during rule evaluation. The algorithm is described in the text.

Subroutine QUALEVAL

Calls these subroutines (NONE CALLED)

Comments. QUALEVAL returns a numerical value for a qualifier. The present defined qualifiers are: QUITE, VERY, QUITEN, VERYN, DK, Q+, QN-. Q+ and QN- specify a range rather than a value, and as such are returned as negative numbers, and the condition subroutines receive them in this form. The numbers used are not important, but are chosen here to be in the range 0 - 1 for convenience, with negative numbers being used to specify a range.

Subroutine REVAL

Calls these subroutines

- VECREVAL

Comments. REVAL is a higher level form of VECREVAL, and calls it. A call to VECREVAL will return the vector value of a KS. A call to REVAL will return the maximum element value of the vector value of a KS.

Subroutine REVERSE

Calls these subroutines (NONE CALLED)

Comments. REVERSE reverses names as necessary (see text on decomposing KS's).

Subroutine RMPAR

Calls these subroutines (NONE CALLED)

Comments. This is a Hewlett-Packard routine that recovers parameters from the invocation of the program. It has been used to allow the output of the main program to be directed to different devices: so, for example, "RU, RINT,25" means run the program with output directed to device 25.

Subroutine RNUMEVAL

Calls these subroutines

- NUMEVAL

- CLEAN

Comments. RNUMEVAL evaluates and returns the region reference vector generated by the region reference operand.

Subroutine ROCKE

Calls these subroutines

- CANDSTACK
- CVARFIX

Comments. ROCKE evaluates the "rockness" of a given region, and corresponds to the condition "ROCK", which has two operands, a region reference and a qualifi[illegible] in that order. ROCKE returns a value of either 0. or 1., depending on whether the region indicated belongs to the rock class to the degree indicated by the qualifier, or not. The value is returned as a vector, each member of which indicates the "rockness" of a different candidate for the given region identifier (see text). The length of the vector is returned as well, being -1 for an identifier with only one possible match (like "CENTRE").

Subroutine ROTATE

Calls these subroutines

- ROUND

Comments. ROTATE uses ROUND to rotate the neighbour code until the required region (normally the region's merge partner) is in the first position in the code.

Subroutine ROUND

Calls these subroutines (NONE CALLED)

Comments. ROUND rotates the neighbour code one place.

Subroutine RQUEUE

Calls these subroutines (NONE CALLED)

Comments. RQUEUE implements the queue used to keep conditions in order when they been evaluated. (See text for details.)

Subroutine SEARCHI

Calls these subroutines (NONE CALLED)

Comments. SEARCHI finds the position of the next occurrence, after a given position, of an indicated region in a neighbour code.

Subroutine SEGLD

Calls these subroutines (NONE CALLED)

Comments. SEGLD loads a segment and transfers control to it: it is another Hewlett-Packard routine.

Subroutine SEGRT

Calls these subroutines (NONE CALLED)

Comments. SEGRT is a Hewlett-Packard routine that returns control from the segment to the loader.

Subroutine SIMGE

Calls these subroutines

- CANDSTACK
- CVARFCL

Comments. SIMGE implements the condition SIMG, which is required to have as its operands two region references. At present one of these references must be to CENTRE, as we have not made provision for comparing all members of a set in all possible ways with all members of another set. A 1. is returned as the condition value if the grey levels are similar, a 0. if they are not.

Subroutine SIMTE

Calls these subroutines

- CANDSTACK
- CVARFCL

Comments. *SIMTE is similar to SIMGE, but compares textures (grey-level variance) rather than grey levels.*

Subroutine STRLIST

Calls these subroutines

- NAMELIST
- MATCHI

Comments. STRLIST implements the system structure list. Records may be entered if their name does not already appear in the list, deleted, or demoted to primitives, when a match list must be specified. The list also allows a search to be made for the next legal structural description (in terms primitives only). The records are kept in a simple array, the

list structure being imposed by NAMELIST, which stores the pointers to the list.

Subroutine TRUEE

Calls these subroutines

- CANDSTACK

Comments. TRUEE implements the condition TRUE. It always pops one operand from the condition operand stack, and always returns the value 1. : thus, TRUE must always have one operand, conventionally called DUM, but not inspected.

Subroutine VECREVAL

Calls these subroutines

- RQUEUE
- ANDSTACK
- ROCKE

- BACKE
- NNUME
- EDGE
- TRUEE
- SIMGE
- SIMTE
- ATORSTACK
- PUTIT

Comments. VECREVAL evaluates a KS, returning a vector value for the entire KS. The technique used is exhaustively described in the text. Note that this subroutine may not always appear with its full complement of condition evaluation calls (calls to: ROCKE, BACKE, TRUEE, EDGE, NNUME, SIMGE, or SIMTE) but is documented here as needing them all, for simplicity and generality.

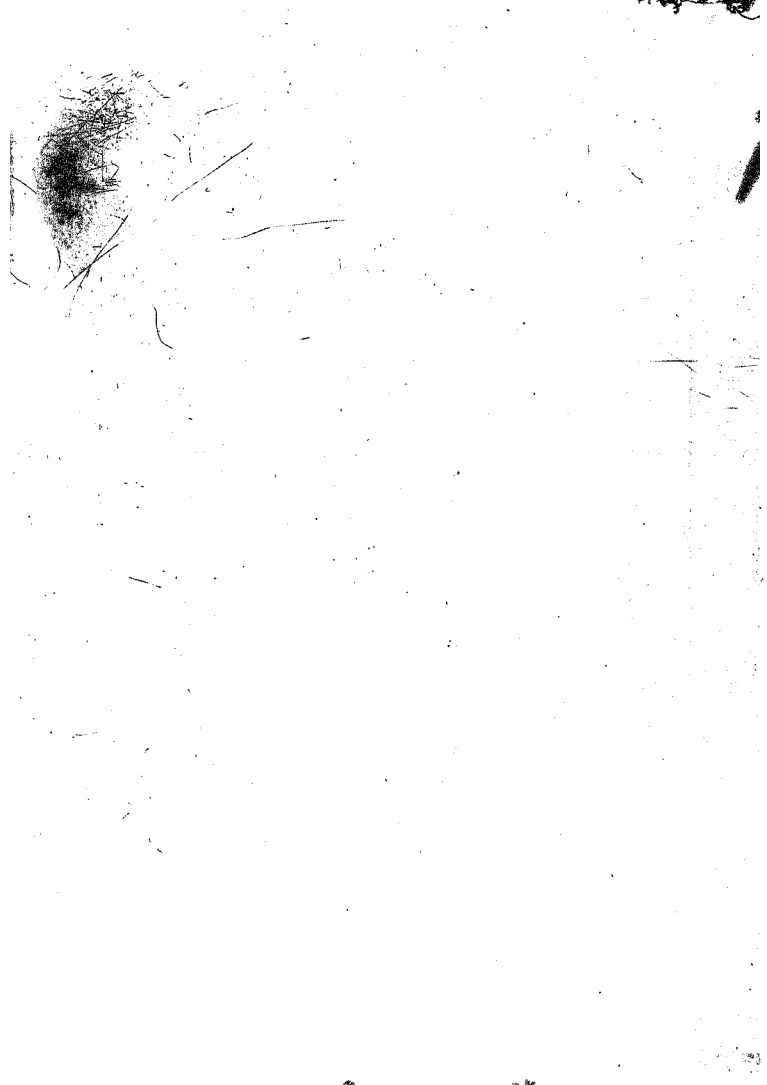
APPENDIX B. THE BACKUS NAUR FORM SYNTAX OF THE KS'S.

```
BNF    T=00004 IS ON CR  00043 USTHE 00012 BLKS R=0000
                                    11:17 AM  FRI., 10  OCT., 1980

0001
0002
0003        Backus Naur Form Expression Of Rules .
0004        ---------------------------------------
0005        The rules require the use of the characters ( and );to evoid
0006        confusion,however,we use here [ and ].
0007
0008   (rule)::=(digit)(digit)(digit)*(clist)*(alist)
0009   (clist)::=[(operand) (operand)]A[[(operand)] (operand)]D[[(operand)]N
0010   (operand)::=(condition)|(clist)
0011   (condition)::= BACK((regionno),(qualifier))|
0012                  ROCK((regionno),(qualifier))|
0013                  TRUE(DUMD)|
0014                  SINCE((regionno),(regionno))|
0015                  SINCE((regionno),(regionno))|
0016                  ONE((objname),(objname))|
0017                  NEAR((regionno),(objname))|
0018                  NEAR((regionno),(integer))
0019                  TOUCH((sides))|
0020                  SHAPE((shdes))|
0021                  SIZE((sides))|
0022                  REFLEC((rfdes))|
0023                  CONVEX((cdes))
0024   (ides)::=HIGH[MEDIUM]LOW
0025   (shdes)::=L[TISOSHAPEDNAME
0026   (rfdes)::=HIGH[MEDIUM]LOW
0027   (cdes)::=(qualifier)
0028   (sides)::=BIG[LITTLE[MED]]M
0029   (regionno)::=R(integer)|CENTRE
0030   (integer)::=(digit)|(digit)(integer)
0031   (digit):::=0|1|2|3|4|5|6|7|8|9
0032   (qualifier)::=QUITE[QUITE]VERY[VERY]HIGH-LO*
0033   (alist)::=MERGE((regionno),(merlist))|
0034            CHCTXT(<newcontext>)|
0035            AGET(GOAL)|
0036            <objname>|(objdescriptor>)
0037   (objdescriptor>)::=PRIM|CONP
0038   (objname)::=(charlist)
0039   (charlist)::=(char)|(char)(charlist)
0040   (char)::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
0041   (merlist)::=(regnum)|(regnum)(merlist)
0042
0043   Note:merlist and charlist are limited to six member and seven members
0044   respectively,for reasons of practicality.
0045   For the language to work,an objname should have a definition to be used.
0046
```

APPENDIX C. A COMPLETE LISTING OF THE CODE.

This listing is bound in a separate volume for simplicity, and may be obtained from Professor M.G. Rodd, Department of Electrical Engineering, University of the Witwatersrand, 1 Jan Smuts Ave., Johannesburg, South Africa.



Author Forsyth David Alexander
Name of thesis A paradigm for general scene analysis. 1985

PUBLISHER:
University of the Witwatersrand, Johannesburg
©2013

LEGAL NOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed, or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page) for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.