AISB

SUMMER CONFERENCE
JULY 1974

University of Sussex

CONTENTS

INFERRING THE POSITION OF BODIES FROM SPECIFIED SPATIAL RELATIONSHIPS

A.P. Ambler and R.J. Popplestone

Dept. of Machine Intelligence,
School of Artificial Intelligence,
University of Edinburgh,
Edinburgh EH8 9NW,
Scotland.

Abstract

A program has been developed which takes a specification of a set of bodies and of spatial relations that are to hold between them in some goal state, and produces expressions denoting the positions of the bodies in the goal state, together with residual equations linking the variables in these expressions.

1. Introduction

The work described in this paper is motivated by the desire to construct an easily instructable robot to work in the domain of automatic assembly. Earlier work at Edinburgh (1) produced a device capable of being easily "taught" to recognise solid bodies from a small number of viewpoints and in isolation, and to stack them in pre-determined places. It could also extract bodies from a heap and place them in isolation to recognise them. However the ensuing assembly of the objects from their standard positions was programmed by specifying the motions of the robot absolutely (e.g. moveto(9,7); graspto(0); ...). A much less painful way of instructing the robot would be to specify spatial relations that are to be established between parts being manipulated. Thus one might say "Place the cylindrical face of the rod against the sloping faces of the Vee block" (see Fig. 9). Even if such an English sentence had to be transcribed into a sort of predicate calculus form, the gain in instructability of the device would be great.

2. Positions

In order to describe the motions of a rigid body one needs the Affine group of rotations and translations of 3-space. We call a member of this group a position. If p_1 and p_2 are positions, then we write $p_1.p_2$ for the functional composition of p_1 and p_2. We will use the two positions valued functions twix and trans where twix(θ) is a rotation by θ about the X-axis and trans(x,y,z) is a translation by (x,y,z). We also use the constant position M which turns the X-axis back on itself. In the program, constant positions are represented by 4 by 4 matrices in the

usual way (for example see (2)).

3. Features and spatial relations

The spatial relations we consider here are *against* and *fits*. These hold not between bodies but between *features* of bodies. Features are either shafts (cylindrical), faces (planar), or holes (cylindrical). Examples of features are depicted in Figs. 1-3. Each body has a set of axes embedded in it. Each feature has axes associated with it. The feature axes are chosen according to certain conventions, namely that the X-axis of the axis set of a face is always pointing out from the face, with the Y and Z axes lying in the face, and the X-axis of a shaft or hole lies along the axis of symmetry of the feature, with the origin at the tip of the shaft or the mouth of the hole. The position of a feature in a body is defined to be that position which will transform the body axes into the feature axes.

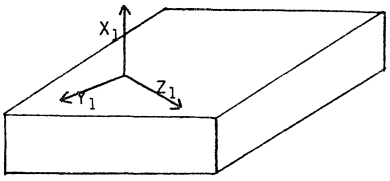


Figure 1. The axes of a face

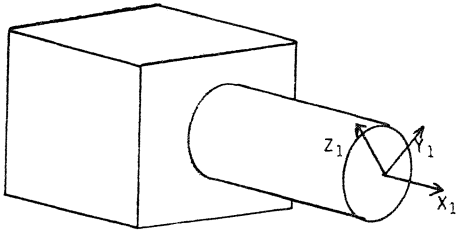


Figure 2. The axes of a shaft

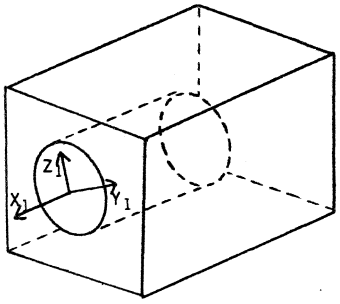


Figure 3. The axes of a hole

At present, the descriptions of the bodies are input manually, but work at Edinburgh is directed to being able to form suitable "body models" of objects whose surfaces are all plane or cylindrical by using a TV camera in conjunction with a projected light stripe. Such a body model contains a specification of all the features with their type (face, shaft, hole) and position.

In this paper we define the relation against to be such that:

a face is against another face when they are coplanar, and with their normals in opposition;

a face is against a shaft when the X-axis of the shaft lies in a plane parallel to the plane of the face, and removed from it by a distance equal to the radius of the shaft;

a shaft cannot be against a hole;

a shaft cannot be against a face;
etc.

We define the relation fits to be such that:

a shaft fits a hole when their X-axes lie along the same line, but in opposition;

a face cannot fit a face;
etc.

Note that this is a very incomplete description of against and fits.

We also need conditions involving the actual dimensions of the features, such as that a face is against another face only when their outlines overlap by a certain minimal amount, and that a shaft fits a hole only when its origin is between the origin of the hole and the point on the X-axis at minus the depth of the hole. We need to include facts about the space occupancy of objects - e.g. that two objects cannot occupy the same space at the same time. These latter conditions give rise to inequalities. It is envisaged that future work will concentrate on dealing with these inequalities. At present we only consider the equalities produced using our incomplete definitions of against and fits.

The program takes a list of body models, and a specification of the against and fits relations that are to hold, and returns a function G from bodies to expressions, where the expressions denote positions that the bodies must be in to satisfy the relations. These expressions will in general contain free variables, and the program also returns equations (possibly null) between these free variables, having attempted to eliminate these variables as far as possible (the equations are non-linear).

4. Relations between two features

The program for deriving the equalities is based upon the following considerations. Firstly it should be noted that if a feature of one body is spatially related to a feature of another then the number of degrees of freedom of the two bodies considered together is less than the 12 they would have if they were free to move. Suppose that a body B_1 has position p_1 and a body B_2 has position p_2 and face F_1 of B_1 is against face F_2 of B_2 (Fig. 4) then for some θ, y and z

$$p_2 = f_2^{-1}\ M\ \mathrm{twix}(\theta)\ \mathrm{trans}(o,y,z)\ f_1 p_1 \qquad (4.1)$$

where f_1 and f_2 are the positions of F_1 and F_2.

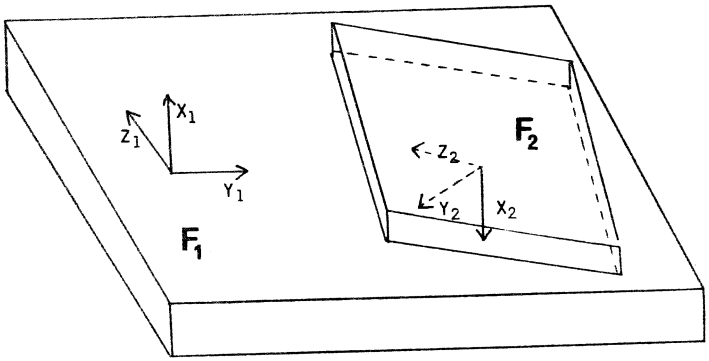


Figure 4. Face F_1 against face F_2

In equation (4.1) the variables y and z correspond to the ability of F_1 to be translated with respect to F_2 while still remaining in contact, and the variable θ corresponds to the ability of F_1 to rotate with respect to F_2 while preserving the contact. In the case where a shaft F_1 fits a hole F_2 or conversely (Fig. 5), we have a similar equation to (4.1) except that the relative translation is along the direction of the common axis of symmetry.

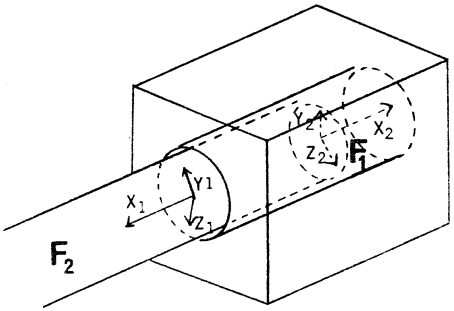


Figure 5. Shaft F_2 fits hole F_1

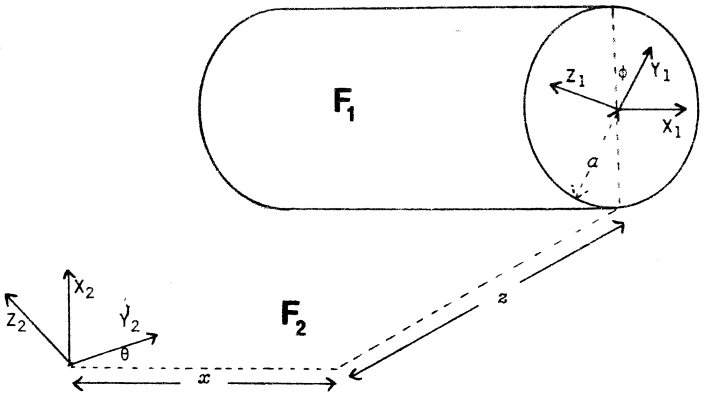


Figure 6. Shaft F_1 against face F_2

The remaining case that we deal with is when F_1 is a face and F_2 is a shaft, (Fig. 6). Here we get two equations, one for expressing p_2 in terms of p_1 and the other for expressing p_1 in terms of p_2. These are:

$$p_2=f_2^{-1}\ \text{twix}(\Theta)\ \text{XTOY}\ \text{trans}(x,-a,z)\ \text{twix}(\emptyset)\ f_1p_1 \qquad (4.2)$$

$$p_1=f_1^{-1}\ \text{twix}(\emptyset)\ \text{trans}(x,y,a)\ \text{XTOY}\ \text{twix}(\Theta)\ f_2p_2 \qquad (4.3)$$

where a is the radius of the shaft, x and z in the one case, and x and y in the other correspond to the translation of the shaft across the face, Θ corresponds to the rotation of the shaft about a normal to the face, and $\emptyset$ corresponds to the shaft rotating about its axis of symmetry and XTOY is a constant position which transforms the X-axis to the Y-axis.

5. Satisfying simultaneous relations

We have seen how to express the position of one body in terms of the position of another when the two bear a specified spatial relationship to each other. In general we are interested in making a number of relations hold simultaneously between a number of bodies (see section 8 for examples). The program derives expressions for the positions of these bodies by first selecting one to be the base. (A fixed one if possible, otherwise an arbitrarily chosen one.) A body which has a feature related to a feature of the base then has its position expressed symbolically in terms of the position of the base according to the equations of section 4. This new set of bodies with

positions defined is then used to provide expressions for the positions of the bodies related to them. Now it may happen that there are loops in the graph relating bodies (for instance the rod mentioned in section 1 is related in 2 ways to the Vee block, namely it is against both faces). In that case the program has two alternative expressions for the position of the body. It selects one of them to be the position, and forms an equation saying that the two are equal.

Thus at the end of this phase, assuming that the relation graph is connected all bodies will have an expression for their position, and a number of equations between positions will have been generated. Now, related to the fact that the Affine group is the semi-direct product of the group of translations and the group of relations, it is possible to consider the rotational component of the equations separately. (See (3) in which the treatment of these equations is considered in full) Briefly, however the system deals with the equations of the form

$$\text{twix}(\theta)=a \tag{5.1}$$

$$\text{twix}(\theta_1)\ a\ \text{twix}(\theta_2)=b \text{ and} \tag{5.2}$$

$$\text{twix}(\theta_1)\ a\ \text{twix}(\theta_2)\ b\ \text{twix}(\theta_3)=c \tag{5.3}$$

It is shown in (3) that, depending on the values of a, b and c, the above equations have solutions, giving constant values for θ_1, θ_2, θ_3. It should be noted that the rotation equations are first reduced by applying the transformations

$$\text{twix}(\theta_1)\ \text{twix}(\theta_2)=\text{twix}(\theta_1+\theta_2) \text{ and} \tag{5.4}$$

$$\text{twix}(\theta)\ M=M\ \text{twix}(-\theta) \tag{5.5}$$

The effect of this phase is to produce a number of linear equations on the θ's. These are used to eliminate as many of the θ's as possible.

Having deduced as much as possible by considering the rotations by themselves, and having substituted symbolically or numerically for the variables which have been eliminated, the program proceeds to attack the equations of the form

$$p_1=p_2 \tag{5.6}$$

by multiplying symbolically by the zero vector 0, to get

$$0p_1=0p_2 \tag{5.7}$$

It is shown in (3) that it is sufficient to solve this equation. The symbolic multiplication depends on the equalities

$$\text{trans}(a,b,c)\ ((x,y,z))=(x+a,y+b,z+c) \qquad (5.8)$$

and

$$\text{twix}(\theta)\ ((x,y,z))=(x,y\cos\theta - z\sin\theta,\ y\sin\theta + z\cos\theta) \qquad (5.9)$$

Using (5.8) and (5.9) an equation between two symbolic vectors is derived, and by equating the components, three real equations are obtained for each equation of the form (5.6).

6. The implementation

The expeditious implementation on a computer of the symbolic manipulation described in this paper obviously requires a language in which it is easy to implement a range of data-types and with "heap" rather than stack storage control. In fact we use POP-2 (4). Much of the algebraic manipulation is not specified in POP-2, however, but in terms of production which are input to an Algebra System written in POP-2. Equalities such as 5.4 are written as:

```
ALL THETA PHI; TWIX(THETA) MP TWIX(PHI) → TWIX(THETA + PHI)
```

meaning that anything that matches the expression to the left of the arrow is to be replaced by what is to the right. MP is an associative operator meaning matrix product. The Algebra System automatically performs certain simplifications such as working out constant subterms, the elimination of identity elements corresponding to operators, and the replacement of any subterm in which a zero of the operator occurs by the zero. The matching process in applying productions takes account of associativity. If an operator is both commutative and associative then the system automatically collects multiples of repeated subterms.

7. An example

Given a fixed block (the "world") of height 20, with position I and with a hole of depth 8 drilled into its top surface at (50,50,20) - i.e.

```
position of hole feature=XTOZ trans(50,50,20)
position of face at bottom of hole=XTOZ trans(0,0,12)
```

put a post into the hole so that the shaft (feature position of shaft=M) fits the hole, and the end face of the post is against the bottom of the hole, (Fig. 7).

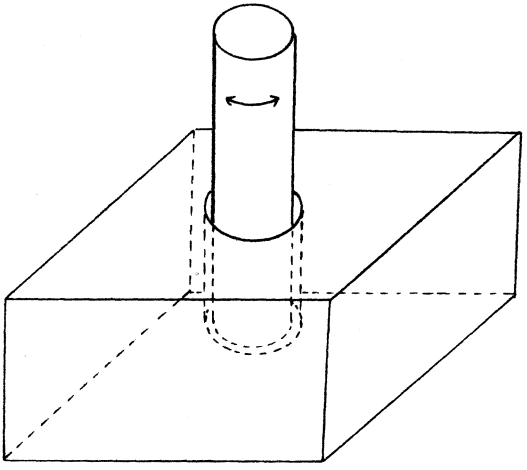


Figure 7. Post in hole, with 1 degree of freedom

Equating the position of the post derived through fits relation to the fixed world

$$(M^{-1}M\ \text{twix}(\Theta 1)\ \text{trans}(X1,0,0)\ \text{XTOZ}\ \text{trans}(50,50,20)I)$$

with the position derived through the against relation

$$(M^{-1}M\ \text{twix}(\Theta 2)\ \text{trans}(0,Y1,Z1)\ \text{XTOZ}\ \text{trans}(0,0,12)I)$$

produces the equation

$$\text{twix}(\Theta 1)\ \text{trans}(X1,0,0)\ \text{XTOZ}\ \text{trans}(50,50,20) = \text{twix}(\Theta 2)\ \text{trans}(0,Y1,Z1)\ \text{XTOZ}\ \text{trans}(0,0,12)$$

with G (post)=twix(Θ2) trans(0,Y1,Z1) XTOZ trans(0,0,12). Solving the rotation equations produces the real equation

$$\Theta 2 - \Theta 1 = 0$$

Now substituting $\Theta 2$ for $\Theta 1$ in the equation, and solving the translation equation gives

X1=-8, Y1=50, Z1=50

and G (post) becomes

TWIX(θ2) trans(0,50,50) XTOZ trans(0,0,12)

i.e. the post fitted into the hole has only one degree of freedom - rotation about its own axis.

8. Other problems

We have used the system to solve several other problems:

(1) Given a fixed world with a fixed wall on it, put a block down so that a particular side is against the worldtop and another particular side is against the wall.

(2) Given a fixed world with two fixed walls at right angles put one block down so that specified faces are against the worldtop and the wallside, and put a second block down so two specified faces are against the worldtop and the second wall and so that a particular pair of faces of the blocks are against each other, (Fig. 8). This produces a situation where one block has no degrees of freedom, and the other block is only free to slide along between the walls and the first block.

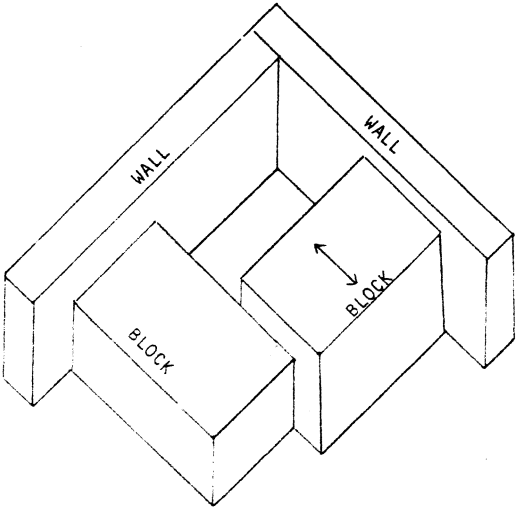


Figure 8. Two blocks against two walls and each other

(3) Given a fixed world with a fixed wall on it, put a cylindrical rod down so that it is lying on the worldtop with one end against the wallside.

(4) Given a fixed block with a "V" shaped groove cut into it, put a cylindrical rod down so that its cylindrical surface is against both sides of the groove, (Fig. 9).

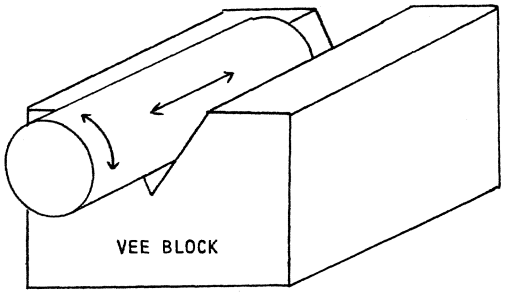


Figure 9. Rod resting against groove in Vee block

(5) Given a fixed world with two holes drilled into it, fit two posts into the holes, with their ends against the bottoms, and fit a crossbar into two holes drilled into the posts, so that its ends are against the bottoms of the holes, (Fig. 10). In this case the posts have no degrees of freedom, and the crossbar can only rotate about its own axis. During the course of solving this problem, five equations are set up, and the rotation of the posts in their holes can only be determined by considering both the fits relations of the crossbar.

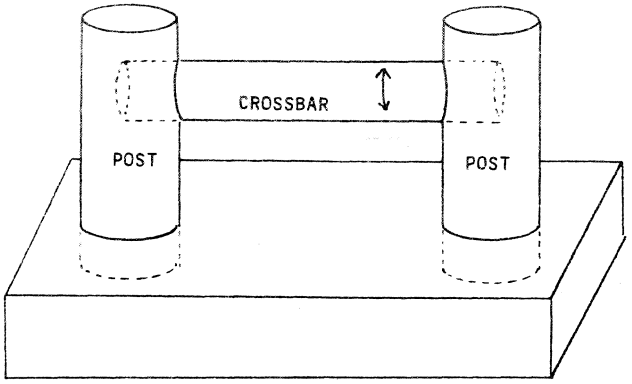


Figure 10. Crossbar fitted into two posts in holes

(6) Given three blocks, with holes drilled in each end, and pins fitting into the holes to join the blocks into a triangle, determine the position of two of the blocks, given that one is fixed, and they are of relative lengths 3, 4 and 5.

9. The relation to previous work

Most work with robot manipulators requires the solution of equations of one sort or another, but usually such equations are stereotyped, that is to say it is required to get the manipulator to grip one block or to put a block in a known place. For instance see Paul (5) and Ejiri et al (6), Feldman (7).

Nevins et al (8) have dealt with the automatic production of the dynamic equations for an arbitrary manipulator whose connection graph is linear.

Fikes (9), Moore[1] and Foster (10) have considered the solution of equations as part of a general problem solving system.

Acknowledgements

We thank the Science Research Council and the Dalle Molle Foundation for support.

Footnote

[1] Moore, J S. A personal communication.

References

1. Ambler, A.P., Barrow, H.G., Brown, C.M., Burstall, R.M. and Popplestone, R.J. (1973) A versatile computer-controlled assembly system. Proceedings of Third International Joint Conference on Artificial Intelligence, Stanford, California, pp. 298-307.

2. Roberts, L.G. (1965) Machine perception of 3-dimensional solids. In Optical and Electro-optical Information Processing, (eds. J.T. Tippet, et al), M.I.T. Press, Cambridge, Mass.

3. Ambler, A.P. and Popplestone, R.J. (1974) Turning spatial relations into equations. Research memorandum MIP-R-107, Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh.

4. Burstall, R.M., Collins, J.S. and Popplestone, R.J. (1971) Programming in POP-2. Edinburgh: Edinburgh University Press. (A revision of POP-2 Papers, Edinburgh: Edinburgh University Press, (1968), with much new material.

5. Paul, R. (1971) Trajectory control of a computer arm. Proceedings of Second International Joint Conference on Artificial Intelligence, London, pp. 385-390.

6. Ejiri, M., Uno, T., Yoda, H., Goto, T. and Takeyasu, K. (1971) An intelligent robot with cognition and decision-making ability. Proceedings of Second International Joint Conference on Artificial Intelligence, London, pp. 350-358.

7. Feldman, J., Pingle, K., Binford, T., Falk, G., Kay, A., Paul, R., Sproull, R. and Tenenbaum, J. (1971) The use of vision and manipulation to solve the "Instant Insanity" puzzle. Proceedings of Second International Joint Conference on Artificial Intelligence, London, pp. 359-364.

8. Nevins, J.L., Whitney, D.E. and Simunovic, S.N. (1973) System architecture for assembly machines. A Report in Advanced Automation R764, Charles Stark Draper Laboratory Inc., Cambridge, Mass.

9. Fikes, R.E. (1970) REF ARF: A system for solving problems stated as procedures. Artificial Intelligence 1, No. 1, 27-120.

10. Foster, J.M. (1970) The philosophy behind Abset. S.R.C. Computer Research Group, Dept. of Engineering, Marischal College, Aberdeen.

AI and Sensori-Motor Intelligence

John Burge*
Department of Psychology, University of Durham

In this talk I shall discuss the earliest period of human cognitive development, the period of sensorimotor intelligence (SMI), in terms provided by AI. One aim in doing so is to show that the application of AI concepts and techniques in the study of this period will prove useful for the understanding of infancy (and incidentally to AI itself). A few issues central to AI will be taken and used as a basis for discussion of aspects of sensorimotor intelligence. SMI can be a good test-bed for AI ideas - at least as informative as that provided by, say, chess. The world of an infant is in no sense a toy world, yet it is small. It is more than an arbitrary subset of our world but can be computed manageably. Moreover, most of the phenomena anyone would ascribe to the action of "intelligence" may be found in the first two years of human life. Another aim of this paper is to give some feeling for the significance of this earliest phase of human development for our understanding of man.

The presentation will begin with a brief outline of the course of sensorimotor development as described by Piaget. Attention will then be focussed on the last stage, that of "rapid internal coordination", which will be compared with devices with a solution to the frame problem in AI. An investigation of the behaviour of infants will be proposed as an approach to the solution of the frame problem, and a number of issues involved in such an attempt will be discussed. The paper ends with some comments aout the relation between AI and Psychology.

---

*Currently at the Department of Computer Science,Carnegie -Mellon University

2 The Psychology of SMI

Without a doubt the most famous student of children's cognitive development has been Piaget. He was interested in creating an experimental epistemology by finding out how children interpret the world and seeing how their interpretation develops with time and experience. He summed up his interests in the origin of cognition with the name "genetic epistemology". He began his research by observing his own children's spontaneous actions and their reactions to situations which he set up himself, as this example shows:

> Obs 16. ... Laurent, at 0;7(5) [i.e. at 7 months and 5 days] loses a cigarette box which he has just grasped and swung to and fro. Unintentionally he drops it outside the visual field. He then immediately brings his hand before his eyes and looks at it for a long time with an expression of surprise, disappointment, something like an impression of its disappearance. But far from considering the loss irremediable, he begins once again to swing his hand, although it is empty; after this he looks at it once more! For anyone who has seen this act and the child's expression it is impossible not to interpret such behavior as an attempt to make the object come back. Such an observation ... places in full light the true nature of the object peculiar to this stage: a mere extension of the action. ... he grasps and swings the cigarette box ...; when he loses it right after having taken it he searches on the coverlet with his hand. However, when he drops it again under any other circumstance, he does not try to find it again. I then again offer him the same box above his eye level; he makes it fall by touching it but does not search for it! (Piaget '54a)

Clearly the conceptual world of the baby is somewhat different from our conceptual world and Piaget, by the use of simple but judicious experimentation, has shown that it is organised on an immediately practical basis throughout the first two years of contact with the physical world. He found that as development proceeded there was an increasing capacity for the representation of absent states of affairs, facility with which was held to mark the advent of the next period. He called the first period, from 0 to 2 years, the period of Sensori-Motor Intelligence. The brief survey of it which follows will show what vast progress the child makes in his construction of reality within this first period of intellectual growth.

Piaget ('54a) describes six stages in the development of SMI. Stage I is characterized by instinctive reflexes, stage II by habitual (acquired) reflexes, stage III by secondary circular reactions, stage IV by means-end behaviour, stage V by tertiary circular reactions and the final stage, stage VI by "rapid internal coordination" of problem solving processes.

As an example of a reflex schema, Piaget cites sucking. At this stage objects are known only through their capacity to enter into reflex activity. In stage II the reflexes are modified so as to become extended in scope - to, for example, systematic thumbsucking. In Piaget's terms, this "involves the formation of a schema of a higher order (a genuine habit), which then integrates the lower schema [i.e. the reflex] with itself."

Stage III, starting at about 3 or 4 months, is marked by the appearance of the secondary circular reaction. An example of this is that of the infant shaking some rattles on the pram cover by means of a string attached to it. Initially, the child grasps the string and inadvertently shakes the rattles. On hearing the result he repeats the process. In a typical circular reaction this repetition will recur for some time. In the primary circular reaction of stage II (e.g. thumbsucking) the body itself is affected repetitiously; in the secondary circular reaction, external objects are affected, usually via prehension.

The fourth stage, starting around 8 to 10 months, involves the concatenation of schemata which produces means-end behaviour. An example is the removal of a screen to retrieve an object placed behind it. Because the schemata may be concatenated in an arbitrary order, whereas the habitual coordinations of the second stage are fixed firmly together in discrete uncommunicating schemata, Piaget refers to an increase in the 'mobility' of the schemata.

The characterization of these mobile means is the preoccupation of stage V. The tertiary circular reactions consist,as do all circular reactions, of repetitions of new phenomena, but this time with "variations and active experimentation" - for example, dropping a toy from various heights and studying the trajectory.

The final stage, stage VI (around 18 months) involves the internal solution of problems. Piaget cites the example of a stick, with which his child had previously had no contact, affording insight into its practical potentialities for reaching things without actual trial and error. Another example will be discussed in the next section.

Figure 1 may help to conceptualize the stages and the relations between them. In it are distinguished the stages involving a new type of information processing behaviour - the "modes" - from the stages involving the mere acquisition of data in in association with these modes of organization - "explicit data gathering". Figure 2 attempts to relate these modes to a progressively bifurcating development of descriptive terms which are justified by the modes. These modes may be related to Kant's "a priori" categories - "a priori", that is, so far as Kant's introspections on his own, adult, state were concerned.

Piaget's rather homely methods and theoretical analysis may be attacked in a variety of ways. To consider them would quickly generate a complex argument for which there is no space here, but it would be inappropriate to take all that Piaget has written without criticism. His writings can thus be put to only a relatively weak use here - merely to provide a framework within which to appreciate the character of sensorimotor development. Another aspect of Piagetian theory for which there is insufficient space is its structural aspects. These are less pronounced for the sensorimotor period than for the later periods. The structures he uses for these are regular mathematical structures (Beth and Piaget '66). Had he had a knowledge of the use of the irregular structure manipulations of AI, he might have been able to characterize the structural aspects of infant behaviour more completely. This line of enquiry will not be pursued explicitly here. It will, however, be implicit in the following discussion of insightful behaviour and its origin.

3 The Frame Problem and Stage VI

The frame problem, simply stated, is that of keeping track of what is going on in the world while attempting to change some aspect of it. The difficulty is that an action may have side-effects not immediately representable in the

data-base describing the state of the world. One is tempted at first sight to believe that the problem should be quite readily soluble, but this illustration from Raphael ('71) should dispel such naivety. Suppose that initially a situation is described by four facts:

(f1) A robot is at position A.
(f2) A box called B1 is at position B.
(f3) A box called B2 is on top of B1.
(f4) A, B, C and D are all positions in the same room.

Suppose moreover that two kinds of actions are possible:

(a1) The robot goes from x to y, where x and y may be any of A, B, C and D.
(a2) The robot pushes B1 from x to y.

Consider two tasks:

(t1) The robot should be at C.
(t2) B1 should be at C.

t1 can be accomplished by the action of type a1, 'go from A to C'. After performing the action, the system should know that facts f2 to f4 are true, but that f1 must be replaced by

(f1') The robot is at position C.

t2 requires the use of a2, and both and f1 and f2 must be changed. The problem is to work out which facts have changed as a result of the action. Raphael says that although one can think of ways of doing this, they all seem to break down in complicated cases. He gives two examples:

(p1) 'Determine which facts change by matching the task specification against the model.'

This would fail for t1 if the robot got to C by pushing B1 there (which is not unreasonable if the box were between the robot and C and pushing it there were easier than going round), thus changing f2.

(p2) 'Specify which facts are changed by each action operator.'

This procedure is also not sufficient, since derived information such as

(f5) B2 is at position B.

will be made false by t2.

According to Raphael the solution to the frame problem is unknown. (Its solution may, indeed, require abandoning the first-order predicate calculus of, for instance, McCarthy and Hayes ('69), which anyway has rather definite limitations according to Anderson and Hayes ('72). Be that as it may Jacqueline Piaget demonstrated that she had solved it before her second year was over:

Observation 181 repeated.- ... Jacqueline at 1;8(9) arrives at a closed door - with a blade of grass in each hand. She stretches out her right hand toward the knob but sees that she cannot open it without letting go of the grass. She puts the grass on the floor, opens the door, picks up the grass again and enters. But when she wants to leave the room things become complicated. She puts the grass on the floor and grasps the doorknob. But then she perceives that in pulling the door toward her she will simultaneously chase away the grass which she placed between the door and the threshold. She therefore picks it up to put it outside the door's zone of movement. (Piaget '54b)

Perhaps this shows only a partial solution, as Jacqueline did make a mistake as she was about to leave the room. The mistake was short-lived and easily corrected. What is interesting about it is that the consequences of it could be foreseen and that the correction was made without trial and error. She certainly gained very little information about the nature of the problem of getting herself and her blades of grass out of the room. Her 'mistake' - and if she had not made it, it could have been argued that the blade of grass had been placed outside the range of the door by chance - allows one to see what was missing from a complete insightful action. A number of such observation of her actions in similar situations might well provide a list of all the necessary components. It is only towards the end of the sensorimotor period that the internal factors that define the next stage begin to obscure the origination of performed behaviour, so that the components which are invisible at stage VI will have been on display in the preceding mode, stage IV. It is precisely this invisibility which makes it plain that the child has solved the frame problem. In order to find out how he has done it one could do worse than follow the advice of Chairman Mao, quoted by Anderson and Hayes:

You can't solve a problem? Well, get down and investigate the present facts and its past history! ... Only a blockhead cudgels his brains on his own, or

together with a group, to 'find a solution' or 'evolve an idea' without making any investigation. It must be stressed that this cannot possibly lead to any effective solution or any good idea.

4 Factors in Tracing the Child's Solution

In following this advice and investigating the history of the problem (or, rather, the history of the infant's solution to it) we find that external feedback governs the younger infant's attempts, rather than some internal feedback producing 'insightful' actions. Moreover, exactly as quoted in the examples above, one can see exactly what processes are occurring so that theories can reflect the data rather closely.

One also needs well-understood theoretical constructs in order to fabricate a viable theory. Fortunately those applicable to the mode preceding that of rapid internal coordination (see Figure 1) have been elaborated in AI following the work of Newell and Simon ('63) on GPS. Some aspects of this branch of AI have even been amenable to systematic analysis, for instance the traversing of graphs (Michie '70). The lack of not only these formalised concepts but also the computational power to deduce rigorously the results of theories based on them may be important reasons why Tolman's ('32) attempt to apply "means-end analysis" to learning by rats did not catch on in the Thirties.

Another issue which bears on this is that of representation, both in general and particular terms. Generally, the infant constructs his reality in a manner quite different from that of the adult, as has been illustrated above. The classifications of Figure 2 correspond to gross structures for representation, to Kant's categories (Korner '55), perhaps. For the finer details let us consider putting some meat on a suggestion of Meltzer's ('70) that a sensible way to acquire generalizations about the world is to generalize, i.e.

$$\text{from } P(a) \text{ infer } (x)P(x)$$

All structures are generalizations, although they may not be due to just such a process of generalization. Let us, however, look at Meltzer's process. He gave two examples. The first was the inference of almost all the axioms of group

theory from ten statements about two groups. The second was from a more complex domain. That was of a child throwing a stone and seeing it sink in a pool. What is it that allows a child to make - as Meltzer claims he does - the "(deductively invalid) inference that all stones if dropped on water will sink" and not any of a host of more or less general inferences, some of which are incorrect? Clearly the child must make careful use of his current representation of the world in order to learn anything new about it and at the same time remain in a state of approximate adaptation.

One could do experiments with children to explore this issue of how current structures limit generalization in early learning. Here is a rather entertaining example from linguistic development quoted by McNeill ('66):

Child: Nobody don't like me.
Mother: No, say "nobody likes me".
Child: Nobody don't like me.

.
.
.

(eight repetitions of this dialogue)

.
.
.

Mother: No, now listen carefully; say "nobody likes me".
Child: Oh! Nobody don't likes me.

The hapless child did not make the generalization his mother wanted him to. Indeed, it is a little difficult to see how she could have signified that she wanted him to delete the "do" with the consequent double negative, even if she could have formulated the problem in the first place. (And as it happens, the "tense" was correct anyway, given the presence of a "do" in the sentence.) It is not easy to envisage a generalization procedure which the child could sensibly employ to improve his grammar in the direction intended by his mother.

Investigation of the details of the child's structural representation must bear these considerations in mind.

5 Structural Learning

Now that sophisticated software has made it possible to consider implementing theories of sensorimotor development in detail as process models, the question of how the structures develop with experience may be examined afresh. Hopefully it will become practicable to identify exactly the potential cracks in each structural level which allow the next level to unfold. A start has been made in this direction by Newsted ('73). He has begun to implement Cunningham's ('72) interpretatiopn of the Piagetian first period in terms of Hebb's ('49) hypothetical neural learning processes.

It may be found that mere differentiation and recombination of reflexes (as proposed by Piaget) cannot provide for early intellectual development and that something additional will have to be added. That would lead to a position similar to that of Chomsky ('65). He proposed that normal learning methods would be inadequate for first language acquisition, and that extra principles would be needed to latch on to "universal" (i.e. general) properties of all adult languages. Supposing that we did use the same basic learning mechanism as animals, these supernumary principles would help characterize what it is to be a man. This is an important goal of psychology which seems to have been forgotten long ago, and it may be that AI is in a position to help achieve it. This will be especially so after AI-based work on learning by infants, for Chomsky had no clear idea how children ordinarily learn about the world. The discovery of this will be impossible without the wholesale importation into developmental psychology of AI techniques and results. Another drawback for the Chomskian view of universals of adult language as an explanatory aid in first language acquisition is that they provide nothing more than constraints on the solution, rather than specify the acquisition process. An AI approach would work close to each structure as it grew and so could provide a better account of the relationship between innate heuristics and generalizations in the fully-developed structure than Chomsky's conjecture that the former somehow lock on to the latter. It could also distinguish any specifically linguistic prior knowledge from that which is general to sensorimotor activity.

6 Conclusion

This talk has attempted to show that AI and early developmental psychology may be of value to each other. It has been suggested that the infant's gradual approach to the insightful solution of problems may be a valid technique to employ in solving the frame problem. However the problem itself would not have become apparent had it not been for the detailed analysis of action forced upon AI by the neutrality of software ignorant of the ways of the world.

The position advanced does not fall wholly into one of Newell's ('70) eight possible relationships between AI and psychology. They are

(1) No relationship
(2) Metaphor/attention-focussing
(3) Forces operationality
(4) Provides language
(5) Provides base (ideal) models
(6) Sufficiency analysis
(7) Theoretical psychology
(8) Self sufficient

In none of these is there provision for transferring ideas or results from psychology to AI, and that this is possible and indeed desirable is one of the contentions of this paper. Work on chess problems provides another example of AI making use of psychology. Simon and Chase ('73) consider observations on the perceptual abilities of grandmasters as a means of isolating the important factors to develop in the evolution of better programs - rather as has been suggested here for the frame problem. Good ('69) proposed a collection or principles of play from good books on chess. These principles were the results of their propounders' introspective analyses of their own methods of working. In present-day chess technology it is difficult to tease apart the psychological from the purely AI components.

Clowes' ('73) timely attempt to proselytize AI to a rather powerful subset of psychology takes this one stage further. His argument was based partly on the historical priority of psychology. What, he asked, was the point of continually rediscovering the wheel?

In contrast with both Clowes and Newell, this paper has argued neither that "psychology proposes but AI disposes" or its reverse, but that the two disciplines may best be developed together.

References

Anderson and Hayes '72 "An Arraignment of Theorem-Proving or the Logician's Folly" Edinburgh DCL Memo No. 54

Beth and Piaget '66 "Mathematical Epistemology and Psychology" Reidel

Chomsky '65 "Aspects of the Theory of Syntax" MIT Press

Clowes '73 AISB Summer School, Oxford

Cunningham '72 "Intelligence: Its Organisation and Development" Academic Press

Good '69 "A Five-Year Plan for Automatic Chess" p89 Machine Intelligence 2 EUP

Hebb '49 "The Organisation of Behavior" Wiley

Korner '55 "Kant" Penguin

McCarthy and Hayes '69 "Some Philosophical Problems from the Standpoint of Artificial Intelligence" p463 Machine Intelligence 4 EUP

McNeill '66 "Developmental Psycholinguistics" p15 of "The Genesis of Language" eds Smith, F and Miller, G.A. MIT Press

Meltzer '70 "Generation of Hypotheses and Theories" Nature vol 225, p972

Michie '70 "Heuristic Search" Computer Journal vol 14, p96

Newell '70 "Remarks on the Relationship between Artificial Intelligence and Cognitive Psychology" p363 of "Theoretical Approaches to Non-Numerical Problem-Solving" eds Banerji and Mesarovic Springer-Verlag

Newsted '73 "Simulation as a Way to Understand a Theory: A Preliminary Model of Cunningham's Hebb-Piaget Theory of Intelligence" School of Business Administration, University of Wisconsin, Milwaukee

Piaget '54a "The Construction of Reality in the Child" Basic Books

Piaget '54b "The Origins of Intelligence in the Child" RKP

Raphael '71 "The Frame Problem in Problem-Solving Systems" p159 of "Artificial Intelligence and Heuristic Programming" eds Findler and Meltzer EUP

Simon and Chase '73 "Skill in Chess" American Scientist vol 61, p394

Tolman '32 "Purposive Behavior in Animals and Men" Appleton-Century

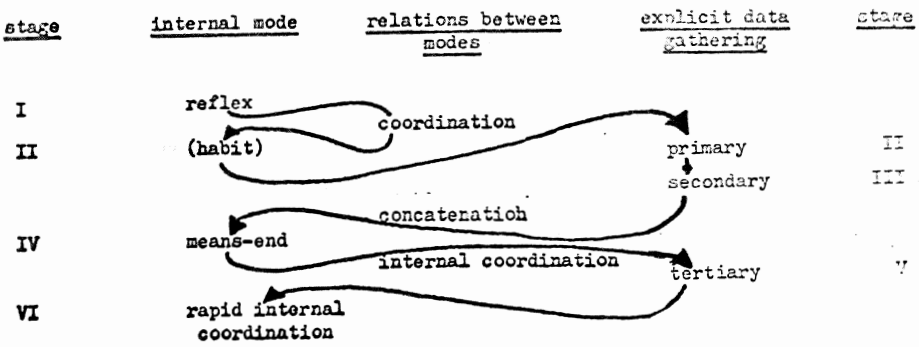
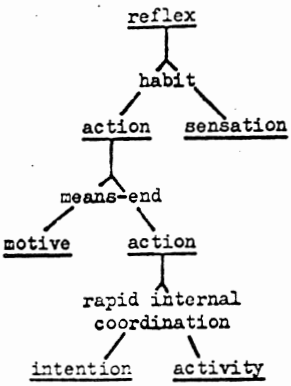


Figure 1



categories underlined

modes of action not underlined

Figure 2

REPRESENTING NEGATION IN A PLANNER SYSTEM

D.J.M. Davies

Theoretical Psychology Unit,
School of Artificial Intelligence,
Edinburgh University.

Abstract

A program is described which carries on a dialogue with the operator, accepting English statements and questions, noting the statements and answering the questions. A method is described for representing negative information. The program goes beyond previous question-answering systems in that new information can be given in English even where this entails selectively removing older information. Universal and existential 'quantifiers' and negation may be used in both statements and questions. The treatment of the quantifiers is outlined. The program uses POPLER 1.5, a PLANNER-like system.

Key-words

Natural Language, PLANNER, Negation, Quantifiers, Question-answering, Procedures.

Introduction

This paper concerns a Natural-language question-answering program which will accept new information in English as well as answering questions. The program uses POPLER 1.5 (Davies 1973), a PLANNER-like system, rather than predicate logic; methods are presented for

(i) representing negative information such as in (1) and (2);

John doesn't own that house. (1)

No-one ate any apples. (2)

and

(ii) removing or 'forgetting' old items of information which conflict with new statements.

These methods depend on the use of "self-erasing procedures". The relationships between negation and universal and existential quantifiers will be outlined.

The program can be given new information by typing in a suitable English indicative sentence, and the information will then be used in answering subsequent questions to which it is relevant. A particular problem arises when such a statement contradicts information stored previously: the out-of-date information must be removed. This problem is exceedingly difficult to solve in a system based on the storing of predicate calculus formulas. Inconsistencies must be avoided; however

it is generally hard to know which formulas to remove when a new formula is added, particularly when quantifiers are involved. So far, no system based on predicate calculus has demonstrated a solution to this, and that is an important reason for using a PLANNER-like system.

In a PLANNER-like system, propositional information can be represented by two different methods. First, a list-constant 'assertion' can be stored in the 'data-base'; this method is suitable for 'atomic facts' which do not contain any quantifiers or variables, e.g. [IN COW1 FIELD2]. Secondly, information can be represented by procedures - programs called through pattern matching. A procedure has a pattern as well as its body of program. In appropriate circumstances, procedures are called and their bodies run; the procedures called are those whose patterns 'match' a given 'target pattern' item. Before any procedure is used, it has to be 'asserted', telling the calling mechanism that it is available for use. It is possible subsequently to 'erase' the procedure withdrawing it from use again.

Two main types of procedure are used in the program: asserting and infer procedures. Asserting procedures are called with the function draw, which may be read as "draw conclusions from ...". Draw is applied to an item (which represents a proposition) and calls all asserting procedures whose patterns match that item. One call of draw may cause several procedures to be called.

Infer procedures are called with the system function infer, which also takes an argument item which represents a proposition; infer will try to infer the proposition's truth. Infer calls an infer procedure whose pattern matches the item; if several such procedures are available then only one is called. If, however, a back-tracking 'failure' backs up, then infer calls another procedure instead if there is one.

The operator maintains a dialogue with the program by typing English statements and questions on a teletype, and the program responds to each one in turn. The program is of interest as a (very incomplete) model of a 'hearer' of English, not of a 'speaker', and the program's responses are stereotyped. A typical dialogue is shown in Figure 2; the marginal notes will be explained later. The operator may use negation and 'quantifier' words in his questions and statements; this covers the words: each, every, any, all, some, a, an, not, there is, no-one, something, etc. Collective uses ("I paid £500 for all those cows"), cardinals ("Five sheep were stolen") and "many" and "few" are not handled.

The domain of discourse is very limited in subject matter. A

number of people own various animals and keep them in various fields. Certain facts are known to the program beforehand, but its 'beliefs' will change in accordance with what it is told. The program demonstrates some 'understanding' of negation and the quantifiers, but it is not a detailed model for the concepts of ownership and place.

The program is based on the principles put forward by Davies and Isard (1971) for a model of a hearer. The response to any utterance takes place in two stages as shown in Figure 1. The utterance is first 'compiled' into a piece of program which represents the (cognitive) meaning of the utterance. That is, if the hearer then runs this piece of program he will respond appropriately. For instance, a statement 'compiles' into a program to store the information (and erase out-of-date information). A question 'compiles' into a program which, if run, will compute and print a suitable reply. This 'compilation' of the utterance may be regarded as 'understanding' it. The program then goes on to run the compiled utterance, thus producing a response. There are no interesting peripherals available to the program, so there is no provision for responding to imperatives (e.g. "Shut the door"). In what follows, we shall not examine the 'compilation' process in detail but will look at the programs which various types of utterance compile into.

Simple Assertions

The simple assertion (3) compiles into the program (4).

```
Cow1 belongs to Brown.                                    (3)
(ACKNLDGE (DRAW [BELONG COW1 BROWN]))                     (4)
```

This program looks like a mixture of LISP and POP-2. Parentheses mean a function call in LISP format, while square brackets mean a list as in POP-2. Every statement compiles into a program of the form (ACKNLDGE (DRAW list)) where list represents the proposition.

When (4) is run, draw is applied to the given list and (as described earlier) calls asserting procedures whose patterns match it. Draw returns some result, and the function acknldge is applied to it. Actually, all the work is done by the procedures, and acknldge merely prints "OK". In (4) the argument of draw is a 'simple fact' and there are two standard procedures to be run. The procedure ASSTINFO is called for any 'simple fact' and inserts it into the data-base. The procedure BELONG1 is specific to assertions about BELONG; it checks (in this case) whether COW1 belonged to someone other than Brown, and if so removes the out-of-date information. It may be that draw will also call one or more other asserting procedures

which have been created by previous statements.

Other statements are handled in a generally similar way; draw is applied to a list which represents the information, and calls procedures. There are different standard procedures for all the various types of statements (negations, existentials, etc).

Simple Yes-No Questions

A Yes-No question compiles into a program closely related to that for the corresponding statement. For example, (5) 'compiles' into (6):

```
Does cow1 belong to Brown?                          (5)
(ANSWER (YESNO [BELONG COW1 BROWN]))                (6)
```

The program (6) differs from (4) only in the two functions called. Yesno is a function which takes a list representing a proposition, and looks to see whether (on the basis of the stored information) it is true, false, or undecidable. It returns true, false or undef as its result, and the function answer prints a suitable reply.

Yesno first tries to establish the truth of the proposition by using the function deduce. Deduce takes the list and looks in the data-base to see whether it is a 'simple fact' which has already been recorded there. If that fails, deduce calls infer to see whether there is an infer procedure which will establish the truth of the proposition. If deduce succeeds then yesno returns true.

If deduce fails, then yesno has to discover whether the proposition is refutable. The method adopted for doing this in most question-answering systems is to try to 'prove' (i.e. deduce) its negation. In this system, however, we capitalise on the fact that new information can be added and that inconsistent old information is thereupon removed. That is, yesno applies draw to its argument, and watches to see whether this entails erasing any piece of information. If anything has to be erased then the given proposition is inconsistent with the available information, so yesno returns false. If nothing was erased then yesno returns undef since the question is undecidable. This application of draw by yesno is done in 'Sceptical Mode', which means that any attempt to erase something will immediately be spotted (causing execution of draw to be terminated), and that in any case all side-effects of the draw will be undone afterwards.

For example, suppose that (3) has already been typed in, and we now ask (5). Deduce will find [BELONG COW1 BROWN] in the data-base, so

yesno returns true and the reply is "Yes". On the other hand, suppose that we ask (7) which 'compiles' into (8):

```
Does Williams own cow1?                                    (7)
(ANSWER (YESNO [BELONG COW1 WILLIAMS]))                    (8)
```

Yesno tries deduce first, but that will fail; the given list is not in the data-base, and no procedure will be able to infer it. Therefore draw is applied to the list in Sceptical Mode. Draw operates as usual for a simple fact, calling ASSTINFO and BELONG1 in this case. BELONG1 will now find that [BELONG COW1 BROWN] is recorded in the data-base, and therefore erases it. Since this is done in Sceptical Mode, the erasure is 'trapped', the effects of the draw are undone again, and yesno exits with result false.

Let us now consider the treatment of a simple denial such as (9), which compiles into (10):

```
Cow3 does not belong to Brown.                             (9)
(ACKNLDGE (DRAW [NOT [BELONG COW3 BROWN]]))                (10)
```

When (10) is run, the standard procedure DENYFACT is called, which specialises in denials of simple facts. This procedure first erases the list [BELONG COW3 BROWN] from the data-base if necessary. However, this is not sufficient, because yesno would now merely return undef if applied to that list. DENYFACT therefore also creates (and asserts) a new asserting procedure (11) which represents the specific denial.

```
(PLAMBDA ASSERTING [] [BELONG COW3 BROWN]  (ERASEA (FRAMEDATA 1)))   (11)
```

This procedure has pattern [BELONG COW3 BROWN] so it will be called if draw is applied to that list. When it is called, its only effect is to erase itself from the index of procedures in use. (The expression (FRAMEDATA 1) will evaluate to the procedure itself at run-time.)

If we now ask the question (12), which compiles into (13),

```
Does Brown own cow3?                                       (12)
(ANSWER (YESNO [BELONG COW3 BROWN]))                       (13)
```

yesno will try deduce first as usual. As in the previous case, deduce will fail, because Brown does not own cow3. Once again, yesno then applies draw to its argument, in Sceptical Mode. As we have just remarked, draw will now call procedure (11), which erases itself. This erasure is 'trapped' in sceptical mode, so yesno finally returns false again, which is correct.

On the other hand, if we subsequently state "Brown owns cow3", then draw is applied to the same list, but not in Sceptical Mode this time.

Procedure (11) is called, and erases itself; it will have no further effect. It can be seen that procedure (11) represents the meaning of the denial (9) in a very direct manner.

Quantifiers

Figure 3 summarises the treatment of the various quantifiers by draw and deduce. Each of the various actions is implemented by means of a "specialist" procedure; the whole system is recursive (with the exception of IFANY statements), permitting complicated propositions to be handled.

It can be seen from Figure 3 that there are three quantifiers, THEREIS, FORALL, and IFANY, not two as in predicate logic. THEREIS is the existential quantifier, while FORALL and IFANY are distinct universal quantifiers; this distinction arises because propositions may change in truth value as the dialogue progresses. Each type of quantifier binds a variable, of a specific type, over a 'matrix'. FORALL claims that the matrix is true for all individuals of a given type, at the time of the utterances (all utterances are in the present tense). IFANY goes further, claiming that the matrix will remain true of all such individuals for all future time (until a later statement explicitly rejects this again). For example "Every one of my friends takes drugs" applies to my friends now (and presupposes that I have at least one); it is a FORALL statement. In contrast, "Anybody caught trespassing will be prosecuted" is an IFANY statement, applying on future occasions (but not presupposing that anybody will get caught).

A FORALL statement can be 'run' by checking off all the relevant individuals in turn, asserting the matrix of each. This usually reduces to a series of simple facts or simple denials. An IFANY statement, however, also requires the creation of one or more procedures, depending on the structure of the matrix. The procedures represent the meaning of the statement as a rule of inference, permitting it to be used in changed situations and about newly introduced individuals. The procedures depend on the structure of the matrix. Each separate structure requires different treatment, and consequently only a limited variety can be handled.

In general, the words every and each translate into FORALL, while all translates into IFANY. (Collective uses are not yet handled.) Any also translates into IFANY in some contexts, but becomes THEREIS

in questions and negative contexts, and in the antecedent of another universal quantifier. 'Definite' constructions such as "any of the ..." become FORALL. This scheme follows Vendler (1967) and Johnson-Laird (1970).

Consider, for example, the statement (14) which compiles into (15).

Any cow in field1 belongs to Brown. (14)

```
(ACKNLDGE (DRAW "[IFANY [(COW) V1]
                        [IMPLIES [IN £*V1 FIELD] [BELONG £*V1 BROWN]]]))
```
(15)

When (15) is run, draw calls the appropriate specialist procedure which creates (and asserts) two new procedures. These new procedures will read thus (slightly simplified):

proc1=

```
(PLAMBDA INFER [[(COW) V1]] [BELONG £*V1 BROWN]
        (DEDUCE [IN £*V1 FIELD1])
        (DRAW [BELONG £*V1 BROWN]))
```
(16)

and

```
(PLAMBDA ASSERTING [[(COW) V1]]   [NOT [BELONG £*V1 BROWN]]
        (COND [(HASRICC (DEDUCE [IN £*V1 FIELD1]))
                (ERASEA proc1 )
                (ERASEA (FRAMEDATA 1))]
              [ELSE (DRAW [NOT [IN £*V1 FIELD1]]) ]))
```
(17)

The first procedure (16) permits one to infer that a cow belongs to Brown if it can be deduced to be in field1. Whenever this inference is made, the conclusion is also put into the data-base to avoid having to repeat the computation. The other procedure is triggered by a denial that Brown owns a certain cow, and normally draws the conclusion that the cow concerned is not in field1. The complication is required because it must be possible to erase these two procedures again. This is done by stating 'in one breath' that some cow is in field1 but is not owned by Brown. When (17) is triggered by a denial that Brown owns a certain cow, it checks whether that cow is known to be in field1 by virtue of the current utterance (hasricc does this). If so, then the two procedures are erased since the asserted conjunction is inconsistent with the IFANY statement.

It can be seen from Figure 3 that the negation of a universal is an existential, and vice versa. This is in accordance with predicate logic. The negation of (14) will be converted to the existential (18), by virtue of the rule for negating IMPLIES.

```
[THEREIS [(COW) V1] [AND [IN £*V1 FIELD1] [NOT BELONG £*V1 BROWN]]]] (18)
```

When a THEREIS list is supplied to draw, the specialist procedure concerned first tries to deduce it, which is done by looking for any instance. If no individual already satisfies the matrix, then an 'arbitrary' individual of the given type is hypothesised and the matrix asserted of it. This will, for example, serve to erase the procedures (16) and (17), and to permit THEREIS to be subsequently deduced. However this technique is unsatisfactory; perhaps the system should ask the operator which individual is involved. A similar problem arises with statements involving cardinals, e.g. "Brown owns five animals".

In a statement, the negation of THEREIS becomes IFANY (rather than FORALL). This means that a procedure will be created. For example, "Brown doesn't own any animals" gives rise to the procedure (19), which is self-erasing rather like (11).

```
(PLAMBDA ASSERTING [[(ANIMAL) V2]]  [BELONG £*V2 BROWN]
         (ERASEA (FRAMEDATA 1)))                                    (19)
```

On the other hand, in deduce the negation of THEREIS becomes FORALL. To answer "Who does not own anything?" the system looks for people with no known possessions, rather than for people who are explicitly known not to have possessions.

Relationship to Previous Work

The main previous attempt to translate English into a PLANNER-like language is Winograd's system (1972). Winograd's system provided only limited scope for the operator to tell the program things; on the whole, the BLOCKS program knows most things already. It was possible to tell that program who "owned" various blocks, and subsequently to ask questions. However, his treatment of denials was not entirely satisfactory. When something was denied, an infer procedure (in our terminology) was created which is triggered by the attempt to infer the proposition denied; the procedure makes it its business to ensure that the false proposition cannot be inferred. The trouble with this is that no distinction is drawn between the answers "No" and "Dunno" to a yes-no question.

The treatment of "quantifiers" described here is based primarily on Vendler's (1967) account, but has obvious affinities with predicate logic.

Acknowledgments

I am glad to acknowledge the encouragement of and helpful discussions with my colleagues, especially Professor H.C. Longuet-Higgins and Stephen Isard. This work was supported in part by the Science Research Council.

References

Davies D.J.M. (1773) POPLER 1.5 Reference Manual TPU Report 1, Edinburgh University.

Davies, D.J.M. & Isard, S.D. (1971) Utterances as Programs, in Machine Intelligence 7 (eds Meltzer & Michie) Edinburgh University Press.

Johnson-Laird P.N. (1970) The interpretation of quantified sentences, in Advances in Psycholinguistics (eds Flores D'Arcais & Levelt) North-Holland, London.

Vendler Z. (1967) Each and Every, Any and All, in Linguistics in Philosophy Cornell University Oress, Ithica, N.Y.

Winograd T (1972) Understanding Natural Language. Edinburgh University Press.

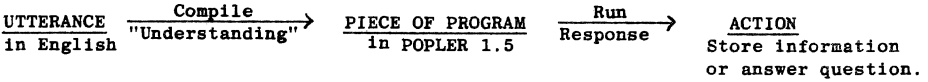


Figure 1. Response to an Utterance

: Sheep4 belongs to Brown.	
OK	
: Does Williams own sheep4?	
I BELIEVE [BELONG SHEEP4 BROWN]	*Sceptical; item being erased.*
NO	
: Willaims owns all animals in field2.	*IFANY inference.*
OK	
: Sheep4 is in field2.	
OK	
: Who owns sheep4?	
THE ANSWER IS: WILLIAMS	*The inference has been made.*
: Is every cow in field2 the property of Williams?	*FORALL question.*
[NO (COW) WHICH [IN £*V3 FIELD2]]	*No referents.*
YES	
: Is each sheep in field2 owned by Williams?	
YES	
: Does any sheep in field2 belong to Brown?	*THEREIS question.*
I BELIEVE PROCEDURE (PLAMBDA INFER ...)	*Sceptical; procedure erased.*
NO	
: There is a cow in field2 which is owned by Brown.	
OK	*Procedures erased now.*
: Do all the sheep in field2 belong to Williams?	*FORALL question.*
YES	
: Do all sheep in field2 belong to Williams?	*IFANY question.*
DUNNO	
:	

Figure 2. A Sample Dialogue

	To Draw (DRAW xx)	To Deduce (DEDUCE xx)	To Deny (DRAW [NOT xx])	To Deduce Denial (DEDUCE [NOT xx])
Simple fact	Store in data-base.	Look in data-base.	Remove from data-base; Make self-erasing procedure.	Look in data-base to check that it is not present.
[THEREIS var m]	Do nothing if deduce-able; otherwise construct 'arbitrary' instance.	Look for an instance.	(DRAW [IFANY var [NOT m]])	(DEDUCE [FORALL var [NOT m]])
[FORALL var m]	Draw for every individual.	Deduce for every individual.	(DRAW [THEREIS var [NOT m]])	(DEDUCE [THEREIS var [NOT m]])
[IFANY var m]	Draw for every individual; then construct procedures.	Deduce for an 'arbitrary' instance.	(DRAW [THEREIS var [NOT m]])	(DEDUCE [THEREIS var [NOT m]])
[AND a b]	Draw both.	Deduce both.	(DRAW [IMPLIES a [NOT b]])	(DEDUCE [IMPLIES a [NOT b]])
[IMPLIES a b]	If a is deduce-able then draw b.	Draw a, then deduce b.	(DRAW [AND a [NOT b]])	(DEDUCE [AND a [NOT b]])
Used in:	Statements & yesno	Yesno and WH-questions.	Statements.	WH-questions.

This table shows what is done in each case by the standard procedures. Other actions may be performed in particular cases where procedures have been constructed by previous statements.

Figure 3. Table of Quantifiers and Connectives

UNDERSTANDING SIMPLE PICTURE PROGRAMS

Ira P. Goldstein
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

A collection of powerful ideas--description, plans, linearity, insertions, global knowledge and imperative semantics--are explored which are fundamental to debugging skill. To make these concepts precise, a computer monitor called MYCROFT is described that can debug elementary programs for drawing pictures. The programs are those written for LOGO turtles.

Keywords: debugging, program writing, planning, linearity

1. Introduction

This paper reports on progress in the development of a monitor for debugging elementary programs. Such research is important both for its practical applications as well as for its investigation of concepts which are fundamental to programming skill. A computer monitor called MYCROFT has been designed that can repair simple programs for drawing pictures [Goldstein 1974]. The reasons to develop such monitors are:

1. to provide a more precise understanding of the fundamentals of programming;

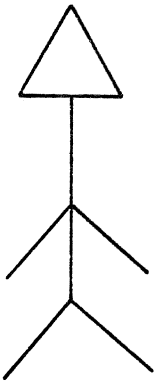
2. to facilitate the development of machines capable of debugging and expanding upon the programs given them by humans; and

3. to produce insight into the problem solving process so that it can be described more constructively to students.

MYCROFT is intended to supply occasional advice to a student to aid in the debugging of programs that go awry. (Just as the system's namesake, Mycroft Holmes, occasionally supplied advice to his younger brother Sherlock on particularly difficult cases.) In this interaction, the user supplies statements that describe aspects of the intended picture and plan, and the system fills in details of this commentary, diagnoses bugs and suggests corrections. In this paper, however, I shall not emphasize this interactive role. Instead, my primary purpose will be to describe MYCROFT as a model of the debugging process. This is reasonable since MYCROFT's utility as an advisor stems directly from its understanding of debugging skill.

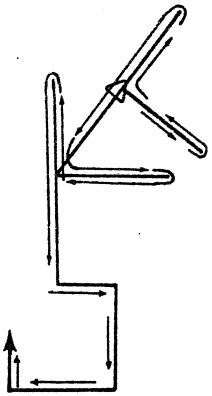
MYCROFT is able to correct the programs responsible for the bugged pictures shown in figures 2, 3, 4 and 5 so that the intended pictures are achieved. In this paper, the debugging of figure 2, a typical

example, will be thoroughly explained. Figures 3, 4 and 5 are corrected in analogous ways: see [Goldstein 1974] for details.



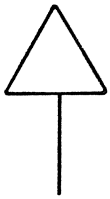
Intended MAN

FIGURE 1

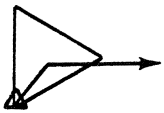


Picture drawn by NAPOLEON

FIGURE 2

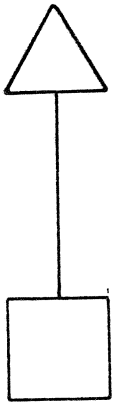


INTENDED TREE

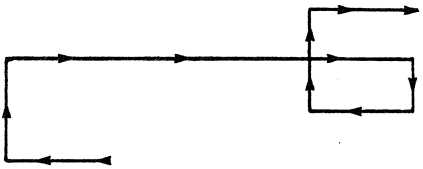


Picture drawn by
bugged TREE program

FIGURE 3

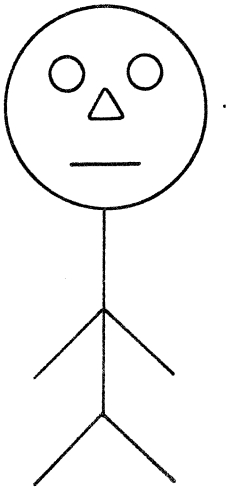


Intended WISHINGWELL

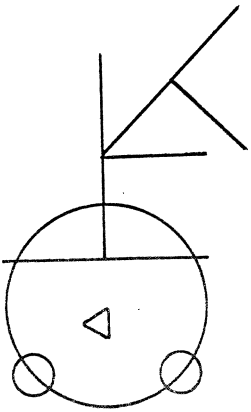


Picture drawn by bugged WISHINGWELL program

FIGURE 4



Intended FACEMAN



Picture drawn by bugged FACEMAN program

FIGURE 5

These pictures are drawn by program manipulation of a graphics device called the turtle which has a pen that can leave a track along the turtle's path. Turtles play an important role in the LOGO environment where children learn problem solving and mathematics by programming display turtles, physical turtles with various sensors, and music boxes [Papert 1972]. Turtle programs have proven to be an excellent starting point for teaching programming to children of all ages, and therefore provide a reasonable initial problem domain for building a program understanding system.

The context of MYCROFT's activity is the interaction of three kinds of description: graphical (i.e. the picture actually drawn), procedural (the turtle program used to generate the picture) and predicative (the collection of statements used to describe the desired scene). For MYCROFT, debugging is making the procedural description produce a graphical result that satisfies the set of predicates describing intent. Thus, debugging here is a process that mediates between different representations of the same object.

2. Flowchart of the System

The organization of the monitor system is illustrated in figure 6. Input to MYCROFT consists of the user's programs and a model of the intended outcome. For the graphics world, the model is a conjunction of geometric predicates describing important properties of the intended picture. MYCROFT then analyzes the program, building both a Cartesian annotation of the picture that is actually drawn and a plan explaining the relationship between the program and model. (Any or all of the plan can be supplied directly by the user, thereby simplifying MYCROFT's task.)

The next step is for the system to interpret the program's performance in terms of the model and produce a description of the discrepancies. These discrepancies are expressed as a list of the violated model statements. The task is then for the debugger to repair each violation. The final output is an edited turtle program (with copious commentary) which satisfies the model. (Occasionally, the plan that MYCROFT hypothesizes requires implausible repairs--for example, major deletions of user code--resulting in the debugger asking the plan-finder for a new plan.)

The remainder of this paper introduces MYCROFT by describing the debugging of NAPOLEON (figure 2) and discussing some important ideas about the nature of plans. For a discussion of the other modules shown in the flowchart, see [Goldstein 1974].

3. Picture Models

To judge the success of a program, MYCROFT requires as input from the user a description of intent. A declarative language has been designed to define picture models. These models specify important properties of the desired final outcome without indicating the details of the drawing process. The primitives of the model language are geometric predicates for such properties as connectivity, relative

FLOWCHART OF MYCROFT

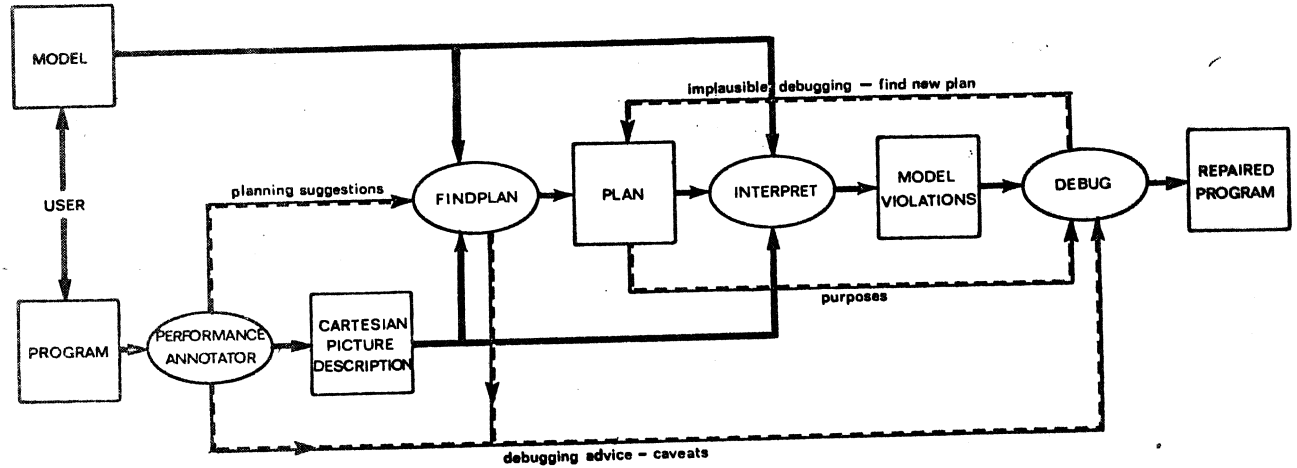


FIGURE 6

position, length and location. The following models are typical of those that the user might provide to describe figure 1.

```
MODEL MAN
M1 PARTS HEAD BODY ARMS LEGS
M2 EQUITRI HEAD
M3 LINE BODY
M4 V ARMS, V LEGS
M5 CONNECTED HEAD BODY, CONNECTED BODY ARMS, CONNECTED BODY LEGS
M6 BELOW LEGS ARMS, BELOW ARMS HEAD
END

MODEL V
M1 PARTS L1 L2
M2 LINE L1, LINE L2
M3 CONNECTED L1 L2 (VIA ENDPOINTS)
END

MODEL EQUITRI
M1 PARTS (SIDE 3) (ROTATION 3)
M2 FOR-EACH SIDE (= (LENGTH SIDE) 100)
M3 FOR-EACH ROTATION (= (DEGREES ROTATION) 120)
M4 RING CONNECTED SIDE
END
```

The MAN and V models are underdetermined: they do not describe, for example, the actual size of the pictures. The user has latitude in his description of intent because MYCROFT is designed only to debug programs that are almost correct. Therefore, not only the model, but also the picture drawn by the program and the definition of the procedure provide clues to the purpose of the program.

4. The NAPOLEON Example

MYCROFT is designed to repair a simple class of procedures called Fixed-Instruction Programs. These are procedures in which the primitives are restricted to constant inputs. Sub-procedures are allowed; however, no conditionals, variables, recursions or iterations are permitted. Given below are the three programs which drew figure 2--NAPOLEON, VEE, and TRICORN. The "<-" commentary is called the plan and was generated by MYCROFT to link the picture models--MAN, V and EQUITRI--to the programs.

```
TO NAPOLEON              <- (accomplish man)
10 VEE                   <- (accomplish legs)
20 FORWARD 100           <- (accomplish (piece 1 body))
30 VEE                   <- (insert arms body)
40 FORWARD 100           <- (accomplish (piece 2 body))
50 LEFT 90               <- (setup heading (for head))
60 TRICORN               <- (accomplish head)
END
```

```
TO VEE                  <- (accomplish v)
10 RIGHT 45             <- (setup heading for 11)
20 BACK 100             <- (accomplish 11)
30 FORWARD 100          <- (retrace 11)
40 LEFT 90              <- (setup heading for 12)
50 BACK 100             <- (accomplish 12)
60 FORWARD 100          <- (retrace 12)
END

TO TRICORN              <- (accomplish equitri)
10 FORWARD 50           <- (accomplish (piece 1 (side 1)))
20 RIGHT 90             <- (accomplish (rotation 1))
30 FORWARD 100          <- (accomplish (side 2))
40 RIGHT 90             <- (accomplish (rotation 2))
50 FORWARD 100          <- (accomplish (side 3))
60 RIGHT 90             <- (accomplish (rotation 3))
70 FORWARD 50           <- (accomplish (piece 2 (side 1)))
END
```

The turtle command FORWARD moves the turtle in the direction that it is currently pointed: RIGHT rotates the turtle clockwise around its axis. A complete description of LOGO can be found in [Abelson 1974], but is not needed here.

A Cartesian representation of the picture is generated by an annotator that describes the performance of turtle programs. The plan is used to bind sub-pictures to model parts. This allows MYCROFT to interpret programs with respect to their models and produce lists of violated model statements. MYCROFT produces the following list of discrepancies for NAPOLEON:

```
(NOT (LINE BODY))          ;The body is not a line.
(NOT (BELOW LEGS ARMS))    ;The legs are not below the arms.
(NOT (BELOW ARMS HEAD))    ;The arms are not below the head.
(NOT (EQUITRI TRICORN))    ;The head is not an equilateral
triangle.
```

MYCROFT is able to correct these bugs and achieve the intended picture using both planning and debugging knowledge.

5. Plans

This section introduces a vocabulary for talking about the structure of a procedure which is useful for understanding both the design and debugging of programs. A main-step is defined as the code required to achieve a particular sub-goal (sub-picture). A preparatory-step consists of code needed to setup, cleanup or interface between main-steps. Thus, from this point of view, a program is understood as a sequence of main-steps and preparatory-steps. A similar point of view is found in [Sussman 1973]. The plan consists of the purposes linking main- and preparatory-steps to the model: in the turtle world, the purpose of main-steps is to accomplish (draw) parts of the model; and the purpose of preparatory-steps is to properly setup or cleanup the turtle state between main-steps or, perhaps, to retrace over some previous vector.

A modular main-step is a sequence of contiguous code intended to accomplish a particular goal. This is as opposed to an interrupted main-step whose code is scattered in pieces throughout the program. In NAPOLEON, the main-steps for the legs, arms and head are modular; however, the code for the body is interrupted by the insertion of the code for arms. The utility of making this distinction is that modular main-steps can often be debugged in private (i.e. by being run independently of the remainder of the procedure) while interrupted main-steps commonly fail because of unforseen interactions with the interleaved code associated with other steps of the plan.

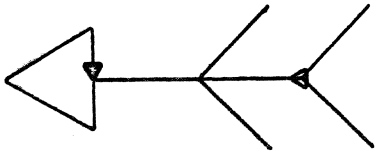
Linearity is an important design strategy for creating programs. It has two stages. The first is to break the task into independent sub-goals and design solutions (main-steps) for each. The second is then to combine these main-steps into a single procedure by concatenating them into some sequence, adding (where necessary) preparatory-steps to provide proper interfacing. The virtue of this approach is that it divides the problem into manageable sub-problems. A disadvantage is that occasionally there may be constraints on the design of some main-step which are not recognized when that step is designed independently of the remainder of the problem. Another disadvantage is that linear design can fail to recognize opportunities for sub-routinizing a segment of code useful for accomplishing more than one main-step. A linear plan will be defined as a plan consisting only of modular main-steps and preparatory steps: a non-linear plan may include interrupted main-steps.

6. Linear Debugging

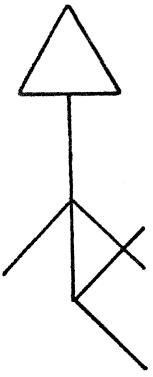
Linearity is a powerful concept for debugging as well as for designing programs. MYCROFT pursues the following linear approach to correcting turtle programs: the debugger's first goal is to fix each main-step independently so that the code satisfies all intended properties of the model part being accomplished. Following this, the main-steps are treated as inviolate and relations between model parts are fixed by debugging preparatory-steps. This is not the only debugging technique available to the system, but it is a valuable one because it embodies important heuristics (1) concerning the order in which violations should be repaired and (2) for selecting the repair-point (location in the program) at which the edit for each violation should be attempted.

Following this linear approach, MYCROFT repairs the crooked body and the open head of NAPOLEON before correcting the BELOW relations. Repairing these parts is done on the basis of knowledge described in the next two sections. Let us assume for the remainder of this section that these property repairs have been made--NAPOLEON appears as in figure 7--and concentrate on the debugging of the violated relations.

Treating main-steps as inviolate and fixing relations by modifying setup steps limits the repair of (BELOW LEGS ARMS) to three possible repair-points: (1) before the legs as statement 5, (2) before the first piece of the body as statement 15 and (3) before accomplishing the arms as statement 25. MYCROFT understands enough about causality to know that there is no point in considering edits following the



NAPOLEON with parts corrected
FIGURE 7



NAPOLEON with statement 15
as RIGHT 135
FIGURE 8

execution of statement 30 to affect the arms or legs. The exact changes to be made are determined by imperative semantics for the model primitives. This is procedural knowledge that generates, for a given predicate and location in the program, some possible edits that would make true the violated predicate. MYCROFT generally considers alternative strategies for correcting a given violation: it prefers those edits which produce the most beneficial side effects, make minimal changes to the user's code or most closely satisfy the abstract form of the plan.

For BELOW, the imperative semantics direct DEBUG to place the legs below the arms by adding rotations at the setup steps. More drastic modifications to the user's code are possible such as the addition of position setups which alter the topology of the picture; however, MYCROFT tries to be gentle to the turtle program (using the heuristic that the user's code is probably almost correct) and considers larger changes to the program only if the simpler edits do not succeed. The first setup location considered is the one immediately prior to accomplishing the arms. Inserting a rotation as statement 25, however, does not correct the violation and is therefore rejected. The next possible edit point is as statement 15. Here, the addition of RIGHT 135 makes the legs PARTLY-BELOW the arms and produces figure 8. This edit is possible but is not preferred both because the legs and arms now overlap and because the legs are not COMPLETELY-BELOW the arms. MYCROFT is cautious, being primarily a repairman rather than a designer, and is reluctant to introduce new connections not described in the model. Also, given a choice, MYCROFT prefers the most constrained meaning of the model predicate. If the user had intended figure 8, then one would expect the model description to include additional declarations such as (CONNECTED LEGS ARMS) and

(PARTLY-BELOW LEGS ARMS).

Adding RIGHT 90 as statement 5 achieves (COMPLETELY-BELOW LEGS ARMS) and the NAPOLEON program now produces the intended picture (figure 1). This correction has beneficial side effects in also establishing the proper relationship between the head and arms, confirming for MYCROFT that the edit is reasonable, since a particular underlying cause is often responsible for many bugs. Thus the result of (DEBUG (BELOW LEGS ARMS)) is:

```
5 RIGHT 90 <- (setup heading such-that (below legs arms)
                                       (below arms head))
              (assume (= (entry heading) 270))
```

The assume comment records the entry state with respect to which the edit was made. If the program is run at a future time in a new environment, then debugging is simplified. The cause of a BELOW violation will now immediately be seen to be an incorrect assumption, and the corresponding repair is obvious -- insert code to satisfy the entry requirements described by the assumption. This illustrates the existence of levels of commentary between the model and the program, each layer being more specific, but also more closely tied to the particular code and runtime environment of the program.

Linear debugging greatly restricts the possibilities that must be considered to repair a violation. It is often successful and constitutes a powerful first attack on the problem of finding the proper edit; however, it is not infallible. Non-linear bugs due to unexpected interactions between main-steps would not be caught by this technique.

Figure 9 illustrates a non-linear bug. (INSIDE MOUTH HEAD) is

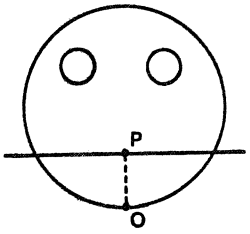


FIGURE 9

violated but it cannot be repaired by adjusting the interface between these two parts (indicated in figure 9 by the dotted line OP) since the mouth is longer than the diameter of the head. The imperative semantics for fixing INSIDE recognize this. Consequently, MYCROFT resorts to the non-linear technique of modifying main-steps to repair a relation between parts. The imperative semantics suggest changing the size of one of the parts because this transformation does not

affect the shape of the part and consequently will probably not introduce new violations in properties describing the part. Advice is required from the user to know whether shrinking the mouth is to be preferred to expanding the head. Two more non-linear debugging techniques are discussed in the next two sections: one is based upon knowing the abstract form of plans, and the other uses domain-dependent theorems about global effects.

7. Insertions

In programming, an interrupt is a break in normal processing for the purpose of servicing a surprise. Interrupts represent an important type of plan: they are a necessary problem solving strategy when a process must deal with unpredictable events. Typical situations where interrupts prove useful include servicing a dynamic display, and arbitrating the conflicting demands of a time sharing system. In the real world, biological creatures must use an interrupt style of processing to deal with dangers of their environment such as predators.

A very simple type of interrupt is one in which the program associated with the interrupt is performed for its side effects and is state-transparent, i.e. the machine is restored to its pre-interrupt state before ordinary processing is resumed. As a result, the main process never notices the interruption. In the turtle world, an analogous type of organization is that of an inserted main-step (insertion). It naturally arises when the turtle, while accomplishing one part of a model (the interrupted main-step), assumes an appropriate entry state for another part (the insertion). An obvious planning strategy is to insert a sub-procedure at such a point in the execution of the interrupted-step. Often, the insertion will be state-transparent: for turtles, this is achieved by restoring the heading, position and pen state. The insertion of the arms into the body by statement 30 of NAPOLEON is an example of a position- and pen- but not heading- transparent insertion.

Insertions do not share all of the properties of interrupts. For example, the insertion always occurs at a fixed point in the program rather than at some arbitrary and unpredictable point in time. Nor does the insertion alter the state of the main process as happens in an error handler. However, if one focusses on the planning process by which the user's code was written, then the insertion as an intervention in accomplishing a main-step does have the flavor of an interrupt.

The FINDPLAN module aids the debugger in a second way beyond just the generation of the plan. This is through the creation of caveat comments to warn the debugger of suspicious code that fails to satisfy expectations based on the abstract form of the plan. In particular, if FINDPLAN observes an insertion that is not transparent, then the following caveat is generated:

```
30 VEE  <- (caveat findplan (not (rotation-transparent insert))).
```

The non-transparent insertion may have been intentional, e.g. the preparation for the next piece of the interrupted main-step may have

been placed within the insertion. The user's program may have prepared for the next main-step within the insertion. Hence, FINDPLAN does not immediately attempt to correct the anomalous code. Only if subsequent debugging of some model violation confirms the caveat is the code corrected. There will often be many possible corrections for a particular model violation. The caveat is used to increase the plausibility of those edits that eliminate FINDPLAN's complaint. In this way, the abstract form of the plan helps to guide the debugging.

For NAPOLEON, analysis of (NOT (LINE BODY)) leads MYCROFT to consider (1) adding a rotation as statement 35 to align the second piece of the body with the first or (2) placing this rotation into VEE as the final statement. Ordinarily, linear debugging would prevent the latter as it does not respect the inviolability of main-steps. However, it is chosen here because of the corroborating complaint of FINDPLAN. The underlying cause of the bug is a main-step error (non-transparent insertion) rather than a preparatory-step failure. Thus, (DEBUG (LINE BODY)) produces:

70 RIGHT 45 <- (setup heading such-that (transparent vee))

8. Geometric Knowledge

Linearity, preparation and interrupts are general problem-solving strategies for organizing goals into programs. However, it is important to remember that domain-dependent knowledge must be available to a debugging system. The system must know the semantics of the primitives if it is to describe their effects.

The debugger must also have access to domain-dependent information to repair main-steps in which the sub-parts must satisfy certain global relationships. For example, TRICORN has the bug that the triangle is not closed. Each main-step independently achieves a side but the sides do not have the proper global relationship. Debugging is simplified by the explicit statement in the model that:

(FOR-EACH ROTATION (= (DEGREES ROTATION) 120)).

But suppose the model imposed no constraints on the rotations. Then the design of the rotations would have to be deduced from such geometric knowledge as the fact that N equal vectors form a regular polygon if each rotation equals 360/N degrees.

The pieces of an interrupted-step such as the first side of TRICORN are not always separated by a state-transparent insert. (This would be a local interruption.) Instead, it is possible that more global knowledge is needed to understand the properties of the intervening code which justifies the expectation that the pieces will properly fit together. In TRICORN, the second piece (drawn by statement 70) must be collinear with the first (drawn by statement 10). The global property of the code which justifies this is that equal sides and 120 degree rotations results in closure. Thus, debugging violations of globally interrupted-steps requires domain-dependent knowledge.

9. Conclusions

The design of MYCROFT required an investigation of fundamental problem solving issues including description, simplification, linearity, planning, debugging and annotation. MYCROFT, however, is only a first step in understanding these ideas. Further investigation of more complex programs, and of the semantics of different problem domains is necessary. It is also essential to analyze additional planning concepts such as ordering, repetition and recursion as well as the corresponding debugging techniques. Ultimately, such research will surely clarify the learning process in both men and machines by providing an understanding of how they correct their own procedures.

10. Bibliography

[Abelson 1973]
Abelson, H., Goodman, N. and Rudolph, 1.
LOGO manual
LOGO Memo 7 LOGO Project, MIT AI Laboratory (August 1973)

[Floyd 1967]
Floyd, R. W.
"Assigning Meaning to Programs"
Proc. Symp App. Math AMS vol. XIX (1967)

[Fahlman 1973]
Fahlman, Scott
A Planning System For Robot Construction Tasks
AI-TR-283 MIT AI Laboratory (May 1973)

[Goldstein 1974]
Goldstein, I.P.
Understanding Simple Picture Programs
AI-TR-294 MIT AI Laboratory (March 1974)

[Hewitt 1971]
Hewitt, C.
"Procedural Embedding of Knowledge in PLANNER"
Proc. IJCAI 2 (September 1971)

[McDermott 1972]
McCermott, D.V. and G.J. Sussman
The CONNIVER Reference Manual
AI Memo 259 MIT AI Laboratory (July 1973)

[Papert 1972]
Papert, Seymour A.
"Teaching Children Thinking"
Programmed Learning and Educational Technology, Vol.9, (Sept. 1972)

[Sussman 1973]
Sussman, G.J.
A Computational Model of Skill Acquisition
AI-TR-297 MIT AI Laboratory (September 1970)

Automatic Induction of LISP Functions

Steven Hardy

Essex University, December 1973

Abstract

A program that infers and codes the LISP function "naturally" intended by a single input-output pair (sample computation) is described. The program uses a knowledge of LISP programming and an extended LISP system to develop and test hypotheses about the function. The program is written in POPCORN, a POP2 implementation of many of the ideas embedded in CONNIVER.

Keywords

Automatic Programming, Induction, LISP, CONNIVER.

Automatic Induction of LISP Functions

Steven Hardy*

The task of Automatic Programming is to make it easier to use computers. Initial developments were languages, such as FORTRAN, which make it possible to specify a numerical algorithm without all the details of its implementation. Within the language, though, it is necessary to specify, precisely, the algorithm.

Recent work has centred on the extent to which the specification of the algorithm itself can be made unnecessary. Thus we have two major problems - firstly, how to describe the problem solved by a program and secondly, given such a description, how to generate, automatically, a program solution. Feldman discusses this, at length, in his paper "Automatic Programming" [4], as does Baltzer in his review of the topic [1].

A popular approach has been to specify the problem with an input/output predicate, usually relying on a resolution based theorem prover to construct a proof that implicitly contains the necessary program [16]. This approach has a number of drawbacks. Existing theorem provers are not very powerful and this limits the size of problem that can be tackled. Further, it seems as difficult, and presumably as error prone, to describe a program - especially one that will involve iteration - in predicate calculus as to actually code it. Also, as has been pointed out by several recent writers [7, 8 and 18], any intelligent proof system needs to employ knowledge not only in the form of axioms defining the problem domain, but also in the form of "control" statements embodying one's understanding of how such proofs might well be achieved. This has led to the development of the ideas embedded in languages such as PLANNER [8] and CONNIVER [12 and 15].

Another approach relies on debugging an existing program to achieve the wanted effect - which might be a LOGO drawing [5] or some action in the BLOCKS world used in Winograd's program [14 and 17].

---

*The work reported here was carried out under the support of the Science Research Council.

I should like to thank the members of the Computing Centre for their advice and guidance. Special thanks go to Mike Brady whose constant help has been invaluable.

Alternatively we could base an automatic programming system on a capacity for inductive generalization. Despite the fact that there are infinitely many functional extensions of the input-output pair ("iopair"):

```
(A B C D) =>= ((A) (B) (C) (D))                    (1)
```

there is only one function that would be regarded by LISP programmers as the "obviously" intended one, viz:

```
(A B --- Z) =>= ((A)(B) --- (Z))
```

I have written a program, called GAP (Generalizing Automatic Programmer) which attempts to model the LISP programmer to this extent. When presented with iopair (1) GAP produces:

```
(LAMBDA  (X)
  (COND  ((ATOM X) NIL)
         (T (CONS (LIST (CAR X)) (SELF (CDR X))))))
```

If presented with, say,

```
(A B C D) && (E F G H) =>= (A E B F C G D H)
```

it produces:

```
(LAMBDA (X1 X2)
   (COND ((OR (ATOM X1)(ATOM X2)) NIL)
         (T (CONS (CAR X1)
                  (CONS (CAR X2)
                        (SELF (CDR X1) (CDR X2))))))).
```

GAP is written in POPCORN [6], an extension of POP2 [2] that provides many of the features of CONNIVER [12 and 15]. The program contains a number of heuristic routines which embody knowledge about various possible formats for LISP functions. GAP looks for features ('cues') of the input, output and the relationship between them. These are used to activate the appropriate heuristic routine. This makes a hypothesis about the basic format of the function, which GAP attempts to verify, and complete, using an extended LISP system. Whilst doing this, GAP can discover new cues which can either affect the current hypothesis or be used to generate new hypotheses.

The routines to notice cues and take the necessary action are stored in the POPCORN data base, indexed by situations to which they are applicable. This makes it possible to add new routines without having to alter the rest of the program. It can be seen that the

detailed flow of control will be very dependent on which cues are noticed and in what order they are noted.

The complete program occupies less than 30k words on a PDP-10. It takes three or four seconds of CPU time for each of the above examples.

In section two I show GAP at work on a few simple examples; in section three there is a fuller discussion of the LISP system. In conclusion I point out some shortcomings of the program and describe the direction of my current work.

Section Two - GAP at work

When given the iopair (A B C D) =>= ((A) (B) (C) (D)) the cue that GAP notices is that the length of the output is proportional to the length of an input - in fact equal to the only input's. This is often the case with simple CDR-loop functions - which are those recursively written LISP programs whose recursion line has the form f(cdr(ℓ)). Such programs embody an essentially iterative process [11], and are one of the most commonly occurring types of LISP function.

Because of this GAP hypothesises that the function is recursive, with the basic body:

(CONS <form x> (SELF (CDR X)))

GAP divides functions into various types; a simple composition of CARs and CDRs is of type PARTOF, for example. <form x> denotes an expression that is an application of a function of type FORM to x. Type FORM functions are the whole range of GAP and are those functions where the output is formed directly out of the input, without special reference to any particular atoms.

To validate its hypothesis GAP tries to make the above body evaluate to ((A) (B) (C) (D)) when x is (A B C D). It realises that <form x> must evaluate to (A) and sets itself the subsidiary problem (A B C D) =>= (A), which it solves by a call on the POPCORN data base and hence a possible recursive call of GAP. It uses the function it gets to decide that <form x> can be replaced by (LIST (CAR X)).

Having done this GAP works out that the body would 'explain' the output if (SELF NIL) evaluated to NIL. GAP also knows that most recursive functions stop before they would have caused an error - in other words the recursion line of a function needs some 'minimum' value of the inputs which should be checked for in some appropriate test. In this case x must be a pair, as it has a CAR and a CDR, and so an appropriate test is (ATOM X). GAP extends its hypothesis to:

```
(COND ((ATOM X)<partof x>)
      (T (CONS (LIST (CAR X)) (SELF (CDR X)))))
```

GAP now uses the fact that this must evaluate to NIL when X is NIL to replace <partof x> by NIL. Notice GAP has a double check on the point at which the function stops recursing.

If we had given GAP the iopair

```
(A B C D) =>= ((A) (B) (C) (D) (B) (C) (D) (C) (D) (D))
```

then the cue noted would have been that the length of the output is proportional to N * (N+1)/2, where N is the length of some input. This can happen if a CDR-loop function has another CDR-loop function as a subroutine. So GAP splits off the first four elements of the output and finds an expression to produce them before proceeding to 'solve' the iopair in a similar way to the previous example.

GAP has a little trick when given functions with more than one input. It looks at the output, element by element, in terms of which input it came from. Thus, if given the iopair:

```
(A B C D) && "Q" =>= (A B Q B C Q C D Q)
```

GAP looks at the list (X1 X1 X2 X1 X1 X2 X1 X1 X2). It is trivial to recognize the repeated - X1 X1 X2 - in this list and so GAP decides to investigate the body:

```
(APPEND  (LIST <form x1><form x1><form x2>)
         (SELF <partof x1><partof x2>))
```

The trick generates plausible hypotheses because many functions, of multiple inputs, produce their output by interleaving the inputs in some way.

The LISP system, using the matcher, realises that (LIST <form x1> <form x1><form x2>) must evaluate to (A B Q) and can therefore be replaced by (LIST (CAR X1)(CAR (CDR X1)) X2).

whilst processing the recursive call of SELF, all the LISP system knows of X1 and X2 is that they are parts of, respectively, (A B C D) and Q. It soon finds that (CAR X1) is B though and so it knows X1 is (B.UNKNOWN). This can only be the CDR of (A B C D) and so <partof x1> is replaced by (CDR X1). Similarly <partof x2> is replaced by X2.

The function is now nearing completion. An appropriate test - in this case (OR (ATOM X1) (ATOM (CDR X1))) - is put in and after final polishing up GAP produces:

```
(LAMBDA  (X1 X2)
  (COND ((OR (ATOM X1) (ATOM (CDR X1))) NIL)
        (T (CONS (CAR X1)
                 (CONS (CAR (CDR X1))
                       (CONS X2 (SELF (CDR X1) X2)))))))).
```

The method just described (looking for repeated patterns in an 'origin list') is a homomorphic mapping of the problem, to create a new problem with a smaller search space, which can be solved to provide a plan for the solution of the main problem. This is a common method of solving problems and several researchers have used it, notably [3], [10] and [13].

This method can be extended if it is unsuccessful, by regarding multiple occurrences of the same origin as a single occurrence. The modified origin list for the iopair

(A B C D) && "Q" = = (A B C D Q B C D Q C D Q D Q)

is (X1 X2 X1 X2 X1 X2 X1 X2) and the repeated - X1 X2 - suggests the recursion line:-

```
(APPEND  (APPEND <form X1><form X2>)
         (SELF <partof X1><partof X2>))
```

which can be expanded to a complete function in the way already described.

The cues described above all assume that atoms from the front of the inputs come at the front of the output. Therefore, when we give GAP the iopair

(A B C D) && "Q" =>= (D Q C Q B Q A Q)

It splits the output into the segment (D Q) and (C Q B Q A Q) - which leads to an incorrect recursion line hypothesis. When the

method discovers it has made a wrong hypothesis it tries to find out if it should have split from the back rather than the front of the output. It does this by replacing the atoms in the two initial segments (in this case (D Q) and (C Q B Q A Q)) by numbers representing which element of an input they came from - and so has (4 Ø) and (3 Ø 2 Ø 1 Ø). The 'average atom' in the first of these is 2 (= (4+Ø)/2) - but this is greater than the average for the second segment 1 (= (3+Ø+2+Ø+1+Ø)/6). If the second segment is due to a recursive call of the function it ought to have a higher average. As this is not so, GAP tries splitting from the back and tries the hypothesis

```
(APPEND (SELF <partof x1><partof x2>)
        (LIST <form x1><form x2>)).
```

Some cues used by GAP recognise immediately that the output is being built up from the back. Suppose a function recurs on the CDR of some input, and otherwise only references the CAR of that input. If this is so, it might be possible to split the output into three segments - the inner one, due to the recursive call of the function, containing no atoms from the CAR of the relevant input and the outer segments containing none from the CDR. This method splits the output of the iopair

```
(A B C D) =>= (A B C D D C B A)
```

into (A), (B C D D C B) and (A), and hence suggests the recursion line:-

```
(APPEND (LIST (CAR X1))
        (SELF (CDR X1))
        (LIST (CAR X1))).
```

The principle of guessing which atoms will be in the three segments of the output is extended by another cue. This counts the times the atoms from a particular input occur in the output. For example,

```
(A B C D E F) =>= (A B B C D D E F F)
```

has an atom count list (1 2 1 2 1 2) - meaning A occurred once, B occurred twice and so on.

The repeated -1 2- in this list suggests splitting the output into three segments - the outer ones containing one A and two B's, the inner one containing one C, two D's, one E and two F's. This splits the output into (A B B), (C D D E F F) and ( ). The length of -1 2- in two - and this suggests a function recursing on the CDDR of its input.

Thus the method suggests the recursion line

```
(APPEND  (LIST  (CAR  X1)  (CAR  (CDR  X1)))
         (SELF  (CDR  (CDR  X1))))
```

POPCORN allows GAP to work in a backtracking mode if it gets a problem it can't solve in any other way. GAP considers a hypothesis that could be represented by a body something like:

(APPEND <form inputs> (SELF <partof inputs>)).

<form inputs> is allowed to evaluate to successively larger segments from the front of the output until the whole expression can be made consistent with the output. We know that <form input> is unlikely to evaluate to, say, NIL - but it is not possible to tell the LISP system facts like this. So the way GAP actually investigates this alternative is a little messier than described. A comprehensive actor system, like that described by Hewitt [9] would probably make this easier.

Section Three - The Program's Knowledge of LISP

GAP has a powerful LISP system to analyse expressions. One part performs simple optimisations, for example:

((LAMBDA (W X)(CAR W)) Y Z) is replaced by (CAR Y)

(APPEND (LIST X) Y) is replaced by (CONS X Y).

This simplifies the task of keeping expressions in a reasonably efficient, natural format.

A second part is a conventional LISP evaluator - except that it has a capability for partial evaluation of expressions whose values are not completely defined. If all we know of Y is that it is an atom, and we know nothing of X then

(CONS X Y) evaluates to (UNKNOWN.SOMEATOM)
(COND ((ATOM X) 1) (T 2)) evaluates to UNKNOWN
(COND ((ATOM Y) 1) (T 2)) evaluates to 1.

A final, more complex, part uses the result of evaluating an expression to deduce things about the expression itself and about what the value of things on the alist must be. It takes as argument a, possibly incomplete, alist, an expression and what one wants the

expression to evaluate to. It returns a list of alists that are consistent with the inputs. If given, for example, a null alist, the expression (APPEND X Y) and the result (A B) it returns:-

```
(((X.NIL)  (Y.(A B)))
 ((X.(A))  (Y.(B)))
 ((X.(A B))(Y.NIL)))
```

If told that X is a pair, by giving it the initial alist ((X.(UNKNOWN.UNKNOWN))) then it does not, of course, return the possibility with X equal to NIL.

If there are no consistent alists, for example (LIST X Y) to evaluate to (A B C), then it returns NIL.

It also completes expressions. If given a null alist, the expression (APPEND X <partof x>) and the result (A B C D B C D) then one possibility it returns is:

((X.(ABCD)) (<partof x>. (CDR X)))

It calls on the POPCORN data base - and hence the whole GAP program - for the solution of any functions it needs.

The control structure of POPCORN is such that it need not generate its alternatives all at once - it does this by returning a tag that allows the computation of alternatives to continue if necessary. However, as the routine works by backtracking it is best avoided when expected to try a lot of alternatives - processing a CONS is far simpler than processing an APPEND.

Conclusion

At present GAP assumes that atoms in the input of an iopair given to it are universally quantified over all S-expressions. Thus it takes no note of the identity of particular atoms, nor of the fact that they are atoms. Thus the iopair

(A A B (X Y)) =>= ((A) (A) (B) ((X Y)))

describes the same function as

(A B C D) =>= ((A) (B) (C) (D))

This means that because it doesn't understand the concepts involved, GAP could not possibly build functions like:-

```
"C" && ((A.W) (B.X) (C.Y) (D.Z)) =>= "Y"          (Assoc)
((AB) ((C) D) ((E))) =>= (A B C D E)              (Flatten)
(A B C D) && (C D E F) =>= (A B C D E F)          (Union)
```

At present I am studying, and trying to implement, ways of solving some of these kind of problems. GAP will need to decide what type of function a particular iopair describes. The range of types GAP can cover is, at present, so small that GAP need only decide whether to hypothesize that a function is recursive - described as type RECUR - or a simple composition of CARs CONSs and APPENDs - described as type BUILD.

Some progress can be made by enriching the information content of an iopair by using ellipsis. In this way, one can more precisely describe a function by an iopair since the ellipsis mechanism is, in fact, an abbreviation for an infinity of iopairs. Thus the automatic programming problem remains, but the inductive generalisation is less difficult. We can use the mechanism to disambiguate an iopair. The function described by:

```
(A B C D) =>= ((A B) (C D))
```

might include either of the following iopairs:

```
(A B C D E F) =>= ((A B) (C D) (E F))
(A B C D E F) =>= ((A B C) (D E F))
```

This ambiguity is removed by the description:

```
(A B --- Y Z) =>= ((A B) --- (Y Z))
```

Ellipsis can be given a useful meaning that requires no 'intelligence' to unpick. Suppose we say that ellipses in the output of an iopair come from a recursive call of the function applied to the ellipses in the input. Using this definition we can see that the following pairs of iopairs describe the same function - but the iopair with ellipsis is unambiguous:-

```
(A ---) && (E ---) =>= (A E ---)
```

is the same as

```
(A B C D) && (E F G H) =>= (A E B F C G D H),
```

```
(A ---) && "Q" =>= (A Q ---)
```

is the same as

```
(A B C D) && "Q" =>= (A Q B Q C Q D Q),

(A ---) =>= (A --- A)
```

is the same as

```
(A B C D) =>= (A B C D D C B A),

(A B ---) =>= (A B B ---)
```

is the same as

```
(A B C D E F) =>= (A B B C D D E F F),

(A ---) =>= (--- A)
```

is the same as

```
(A B C D) =>= (D C B A).
```

Functions can be described to GAP in this language, and the relevant code of GAP is quite small and very fast. This isn't very surprising as we have reduced ellipsis to an unambiguous syntactic device.

As I try to understand what is needed to build an automatic programming system, several facts become increasingly clear. A program will need a mixed description - a single iopair is woefully inadequate - and the program should be interactive - in part to complete its own internal description of a problem. This is, of course, to be expected - the same is true of people.

For these reasons I feel GAP will be difficult to extend unless it can include the person for whom the function is being written in its discussion of a problem. This means that GAP's internal description of a problem must be understandable by people. Much of my effort has been, and will continue to be, devoted to this end. If this is so, it is possible to give GAP hints without a detailed knowledge of its workings.

References

[1] Baltzer R., A Global View of Automatic Programming. IJCAI-3, pp 494-499.

[2] Burstall R.M., Collins J.S. and Popplestone R.J., Programming in POP2. Edinburgh University Press 1971.

[3] Duda R.O. and Hart P., Experiments in the Recognition of Hand-Printed Text. Proc. FJCC 1968 pp 1139-1151.

[4] Feldman J.A. Automatic Programming. AIM-160, CS-255, Stanford University.

[5] Goldstein I. Forthcoming Ph.D. Thesis. Massachusetts Institute of Technology.

[6] Hardy S., The POPCORN Reference Manual. SCM-1 Essex University 1973.

[7] Hayes P.J., Computation and Deduction. Proceedings of the Conference on the Mathematical Basis of Computation. Czechoslovakia 1973.

[8] Hewitt C., PLANNER - a language for proving theorems in robots. IJCAI-1, pp 295-300.

[9] Hewitt C., Bishop P. and Steiger R., A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI-3, pp 235-245.

[10] Kelly M.D., Edge Detection in Pictures by Computer using Planning. Machine Intelligence 6. Edinburgh University Press.

[11] McCarthy J., Towards a Mathematical Science of Computation. Proc. IFIP 1963.

[12] Mabermott D.V. and Sussman G.J. The CONNIVER Reference Manual. AIM-259, MIT.

[13] Shirai Y. A Context Sensitive Line Finder for Recognition of Polyhedra. Artificial Intelligence Vol. 4 (1973)

[14] Sussman G.J. Teaching of Procedures - a Progress Report. AIM-270, MIT.

[15] Sussman G.J. and McDermott D.V. Why Conniving is better than Planning. AIM-255A, MIT.

[16] Waldinger R.J. Constructing Programs Automatically using Theorem Proving. Ph.D. Thesis, Carnegie-Mellon University.

[17] Winograd T. Understanding Natural Language. Edinburgh University Press.

[18] Winston P.H. The MIT Robot. Machine Intelligence 7, Edinburgh University Press.

Some Problems and Non-Problems in Representation Theory

Patrick J. Hayes

0. Introduction

The purpose of this paper is to give a brief survey of some general issues and problems in representing knowledge in AI programs. This general area I will call representation theory, following John McCarthy. Its boundaries are, like those of all interesting subjects, not crisply defined. It merges in one direction with programming language design, in another with philosophical logic, in another with epistemology, in another with robotics. Nevertheless, it is an increasingly important aspect of AI work. Since my main concern here is to draw attention to problems which seem to me to be difficult, and issues which seem to be important, this paper should be read as an appeal for help rather than a statement of achievements (and comments, criticisms and suggestions are welcome).

Inevitably, to believe that some issues are important, and some problems difficult, is to believe that others aren't. At the end of the paper I draw attention to some specific points of disagreement with other authors. It may be helpful, however, to point out immediately that my goals here are not philosophical, but technical. Some commentators on an earlier draft seemed to take it as an essay in philosophical analysis in the modern Oxford style. My aim rather is to substitute, for informal and apparently endless philosophical discussion, the precision of mathematics. (This aim is not achieved in this paper, I hasten to add, but is I hope brought nearer.) To emphasise this, I will, when introducing a technical word intended (ultimately) to have a precise meaning, underline it.

1. Semantics

There are many ways known of systematically representing knowledge in a sufficiently precise notation that it can be used in, or by, a computer program. I will refer generally to such a systematic representational method as a scheme. It is not a very good word, but one cannot say 'language' as that begs an important question (see section 2). Examples of schemes include logical calculi, some programming languages, the systematic use of data structures to depict a world (e.g. as in the early Shakey's use of an array as a room-map), musical notation, map making conventions, circuit diagrams, 'JCM Schemas', 'Conceptual Dependency' notation, 'Semantic Templates' (all in [21]), etc. A configuration is a particular expression in a scheme: an assertion, a program, a data structure, a score, a map, etc. Thus one might, formally, define a scheme to be a set of configurations.

All of these examples are formal in the sense that the question, whether a particular arrangement of marks is a well-formed configuration, always has a definite answer: there is a definite notion of well-formedness. Many ways which humans have of conveying meaning will not be allowed as schemes, for they fail this criterion: drawings, photographs, poems, conversational English, musical performances, TV pictures, etc. In brief, I wish to draw a distinction between (formal) schemes, in which knowledge can be stored and used by a program, and on the other hand, (informal)

scenes or perceptual situations requiring the deployment of knowledge for their successful interpretation.

I am aware of several philosophical problems in analysing this distinction further. As a rough-and-ready guide, schemes can be recognised by the fact that one can construct ill-formed 'configurations'. There is no such thing as an ill-formed photograph. Natural language is a borderline case, as are accurate line drawings of polyhedra.

Schemes are usually intended as vehicles for conveying meanings about some 'world' or environment. In order to be clear about this important topic, a scheme must have an associated semantic theory. A semantic theory is an account of the way or ways in which particular configurations of the scheme correspond to (i.e. have as their meanings), particular arrangements in the external world, i.e. the subject matter about which the scheme is intended to represent knowledge. Some of the schemes referred to above have very precise semantic theories, others have none (and seem to rejoice in this lack: see section 7 below), others (music, maps, circuit diagrams) have informal semantic theories which can be made precise by the approach outlined in section 2 below.

It is not at all fashionable in AI at present to give semantics for new representational schemes, and this is, I believe, a regrettable source of confusion and misunderstanding. Now, one cannot prove such an opinion, of course. One can point to other fields where syntactic confusion and proliferation of ad-hoc formalisms has been or is being replaced by the development of semantic insights: notably, philosophical logic and the design of programming languages. One can point to the way in which, in AI itself, elementary semantic ideas have been re-invented by various authors over the years (especially the Frege/Tarski notion of individuals and relations between them, which crops up with remarkable regularity [, ,]). And one can point to several important questions which simply cannot be answered without a semantic theory. Of these, the most urgent concern the equivalence or otherwise of different formalisms. Is there a difference in meaning between a conjunction of atomic predicate-calculus assertions and the corresponding semantic network? Is there anything which can be expressed in the notation of Merlin [16] which cannot be expressed in a logical notation? The answer to both these questions is yes, in fact: but without a semantic theory the questions cannot even be precisely formulated. Finally, discussion in the AI literature, on, for example, the different roles of deductive, inductive and analogical reasoning and the relative merits or demerits (either technical or philosophical) of various formalisms, is often ill-informed or at best vague due to a lack of a clear model theory for the systems under discussion.

Nothing so far has been an argument for any particular sort of semantic theory: for example, some kinds of 'intensional', 'operational', 'meaning-intentional' or 'procedural' semantics, may eventually enable the meanings of configurations in a scheme to be rigorously defined. However, as a matter of fact, the only mathematically precise account which I have seen of how a scheme can talk of entities outside of the computer, is the Tarskian model theory for first-order logic (but see section 2 below). I believe there are important reasons for going beyond this semantics, but many of the arguments in the AI literature against the use of predicate logic as a scheme are based on misunderstandings of one kind or another, especially the assumption that the use of predicate calculus necessarily involves the use of a general-purpose theorem-proving program. (See section 7 for more discussion.) To defend first-order logic is unfashionable: however, I do want to emphasise that it is the semantics of predicate logic

which I wish to preserve. I have no brief for the usual syntax: networks, for example, can be used as a syntactic device for expressing predicate calculus facts. Some other authors advocate rather the use of predicate calculus syntax either without semantics [19], or with an alien semantics imported from computational theory [6]. This is throwing out the baby and keeping the bathwater.

To insist on a semantic theory is not, of course, to insist that the expressions comprising a program's beliefs are accurate, i.e. that what they express about the world is in fact the case. (This common misunderstanding may be caused by the phrase "truth-recursion", which leads people to think that metamathematics guarantees infallibility.) Without a semantics, one cannot even say precisely what is being claimed about the world: that is the point.

It is important to emphasise that to regard a formalism 'simply' as a programming language: that is, a way of getting the machine to do what one wants, is to adopt a rather different point of view towards it. (Unless, that is, the semantics of the scheme are concerned with machines and what they do.) For example, many people argue that PLANNER is to be regarded 'simply' as a programming language which provides useful facilities for the sorts of programming one finds oneself involved in when writing AI programs. Much of the force of the criticism in [23] for example, is from this position. While this is a perfectly respectable point of view, it is different from the one which regards PLANNER as a scheme which refers to external worlds of, say, blocks. It is even different from the idea that PLANNER is a scheme which refers to problem-solving processes or the like. For the 'programming language' view encourages the user (for example), if he needs a new semantically primitive notion, like negation, to encode it - that is, to implement it - in PLANNER in some way. In terms of schemes this is a change of scheme, since the semantics have been enriched.

To put it extremely: the only difference, in this view, between (say) CONNIVER and (say) FORTRAN, is user convenience: for one could implement the one in the other. (I have heard precisely this view forcibly maintained by professional systems programmers). Hewitt characterises the essence of PLANNER in terms of schemas [13]. While this syntactic approach works up to a point, the relationships between programming languages are, I feel, greatly clarified by giving them natural semantics. The trivial universality which FORTRAN possesses can then be eliminated by the requirement that in embedding one language in another there is a corresponding embedding of the meanings of programs. "Implemented in", as a relation between languages, then ceases to be an embedding since the meaning of (say) THCONSE does not correspond to the meaning of the rather large piece of (say) FORTRAN which would be in the implementation (actually, several pieces scattered about the program but related by context.) The former has to do, presumably, with goals and facts and such things: the latter, probably, with arithmetic relationships between numbers which represent list structures in some way.

In saying all this, one must admit that there is much force in the position that it is too early in AI to settle on particular schemes with fixed semantics. According to this view, AI programs should be implemented using all possible programming skill and ingenuity and we should leave to the future the (perhaps rather arid) task of tidying-up. Much very good AI work has been done from this standpoint, and will probably continue to be done. I do not wish to give the impression of arguing against pragmatic expediency in writing advanced programs. But I do feel that it is not too early to investigate schemes with organised semantics, both on general grounds of scholarliness and because I believe that such schemes are

ultimately easier to use in programming.

2. Linguistic and direct representations

Several authors have drawn attention to a distinction between representations consisting of a description in some language and representations which are in some sense more direct models or pictures of the things represented. I first met this distinction in [1], and it has been more recently emphasised by Sloman [22]. It seems to be clearly important but I have met with surprising difficulty in trying to make the distinction precise.

One problem is to suitably define what is meant by a descriptive language, for we must not beg the question by being too restrictive in our definitions of language. Thus Sloman's emphasis on what he calls analogical representations is really a plea for the consideration of a wider class of languages than those in which the only semantic primitive is the application of a function to arguments (Sloman's term is 'Fregean' languages, like predicate calculus and PLANNER. Some authors seem to have interpreted Sloman as arguing against the use of descriptive representations [3], but this is a misunderstanding.)

Another problem is that a representation which appears to be a direct model at one level of analysis, may, upon enquiring further, be itself represented in a descriptive fashion, so that it becomes impossible to describe the overall representation as purely either one or the other. For example, a room may be directly represented by a 2-dimensional array of values which denote the occupants of various positions in the room: but this array may itself be implemented by the programming system as a list of triplets <i,j,a[i,j]>, i.e. by a sort of description. It seems essential, therefore, to use a notion of level of representation in attempting to make the distinction precise.

Third, any representation must also be a direct representation of something. For, the pattern of marks which is a configuration of the scheme, can convey meaning only by virtue of the fact that its parts are physically arranged in some definite way. This physical arrangement has to be a direct representation of (at least) the way in which meanings of some configurations are compounded into meanings of larger configurations.

Fourthly, the notion of direct representation seems to depend upon some similarity between the medium in which the representation is embedded, and the thing represented. Thus a map of a room is a direct representation of the spatial relationships (in the horizontal place) in the room, by virtue of the similarity between the 2-dimensional plane of the paper and the 2-dimensional plane of the floor of the room. The paper is a direct homomorph of the room: they are the same sort of structure (2-D Euclidean space), admitting the same sorts of operations (sliding, rotation, measurement), but the map is a simplification of the reality, in the sense that certain properties present in reality (colour, exact shapes, etc.) and certain relations (the third dimension, comparisons of value) are missing in the map. Another example is an ordered vector of items in a core store: here the medium is the address structure of the store, which is similar to the integers in respect of its ordering relationships, but not (for example) in respect of its cardinality (stores are finite).

Putting all this together, one arrives at the following general position. There are things called media in which one can construct certain

configurations of marks or symbols: that is, arrangements of marks in which relations exhibited directly in the medium hold between the marks. A language is defined (syntactically) by a set of 'primitive' symbols and a set of grammatical rules which define new configurations in terms of old ones. One gets the usual ideas of parsing. (It could be mathematically interesting to see how much of formal language theory can be generalised to this setting from the conventional 'string' case of 1-dimensional media. One can certainly define context-free, and context-sensitive grammars, but I am not so sure about finite-state, for example.) A model for such a language is provided by a set of entities acting as meanings of the primitive symbols; and, for each grammatical rule, a semantic rule which defines the meaning of the configuration in terms of the meanings of its parts. (One needs variables and variable-binding expressions also, so this account needs elaboration and qualification, but space does not permit a full discussion.) This, so far, is the usual Tarskian idea of a truth-recursion, generalised to this more general notion of language. But now, we also insist that each medium-defined relation used in constructing configurations corresponds to a similar relation in the meanings, and that the representation is a structural homomorph of the reality with respect to these relations. That is, the meanings of configurations must exist in a space which is similar to the representing medium, and the syntactic relations which are displayed directly by the symbol-configurations of the language, must mirror semantic relations of the corresponding kind. The directness of a direct representation lies in the nature of the relationship between the configurations and the reality they represent (it is a relation of homomorphism rather than denotation). A scheme is not direct because of any syntactic features (such as being 2-dimensional) of its schemes, or because of any special qualities (such as being continuous) of the worlds it describes.

It is possible to give formal grammars for simple maps, to emphasise how this account fits the facts, along the lines of Rosenfeld's isotonic grammars [18]. To emphasise again: map-making conventions are, in this view, a language, of which the maps are expressions. The relationship of these expressions to reality is that the primitive symbols denote features of a terrain in a way defined by the map key, and the positional relationships between symbols directly display corresponding relationships between the denoted features.

In electrical circuit diagrams, lines joining symbols denoting components directly denote, in their topological structure this time, the electrical connectivities in the actual circuit. Another example is provided by the simple narrative convention. In "He got up. He got dressed. He went out. He walked to the shop ... ", we understand a time-sequence which is directly denoted by the ordering of the (timeless) separate propositions. This convention is also used in programming languages and cartoon strips, with the same sort of semantics. A final example is provided by networks. A network is a configuration which is a relational structure. Web grammars are the appropriate parsing device. The most obvious way of giving this a semantics is by declaring that a model is any relational structure into which the network can be homomorphically embedded. According to this semantics, a network has the same meaning as the conjunction of predicate calculus atoms corresponding to the arcs of the network. (It is a straightforward exercise in system programming to convert a list of such atomic assertions into a network, represented in the core-store medium by using 'addresses' as the direct analog of 'is connected to', for efficient retrieval.) As we will see, however, one can give a rather different semantics to networks, which makes them more expressive in an important way.

A more complete and rigorous account of this will be published elsewhere. The major problem is to find a general precise characterisation of what is meant by "medium" and "similar". I am currently working on an algebraic account (in which a medium is a category), but it is not yet altogether satisfactory. (Suggestions are welcome.)

The importance of all this, apart from the intrinsic interest of the subject, seems to me to lie in three points. (1) It shows that direct representations are not incompatible with linguistic representations, and can be given a precise model theory along Tarskian lines (which supports Sloman's view in [22]). (2) It suggests ways in which efficient deductive systems may be generalised from work in computational logic. (3) The notion of 'medium' captures the idea of levels of representation mentioned earlier. For a medium may not be physically directly present, but may itself be represented by configurations in some quite other medium, as in the array example. Or again, consider a simulation language like SIMULA. This provides a medium consisting of processes and events and certain relations between them. This medium, taken in its own terms, gives a direct representation of time which is often extremely useful. But if one goes deeper, time is represented in a quite indirect way involving numerical descriptions and long chains of inference. This 'looking-deeper' means not treating SIMULA as a medium to be used to represent, but rather as a reality which is itself represented in some medium (say, FORTRAN or assembly language). The choice of primitive relationships defines both the medium and the level at which analysis will cease.

This shows, incidentally, that Sloman's arguments for the utility of analogical representations, based on the idea that they are somehow more efficient in use than Fregean representations, are fallacious. For an analogical representation may be embedded in a medium which is itself represented in a Fregean way in some other medium. Any discussion of efficiency must take into account the computational properties of the medium.

3. Exhaustiveness and plasticity

An important fact about schemes with Tarskian semantics is that a configuration in such a scheme is, in general, a partial description of the environment. It constrains the form of a satisfying world, but does not (in general) uniquely determine one. And even if it does uniquely determine a world (is categorical, in the technical term), this fact can only be determined by metamathematical analysis: there is no sense in which one can say in the scheme itself, "this is a complete description".

Now this means that one has the opportunity of adding information ad lib, further specifying the world. (Hence the idea of conjunction arises very naturally). The process of adding information can be arrested only by the whole configuration becoming inconsistent, i.e. making an assertion about the world which is so strong that no such world exists. Different schemes will have different particular notions of consistency, but this general outline follows from the abstract properties of the satisfaction relationship between configurations and worlds. This ability to accept new pieces of information and to gradually accumulate knowledge piecemeal is one of the most valuable aspects of Tarskian schemes. Thus, the idea of a 'knowledge base' of separate pieces of information, to which new pieces can be added freely without a need,in particular, to pay attention to control flow or other organisational matters, is very familiar and important.

This possibility of adding information is one aspect of a scheme's plasticity, i.e. the ease with which changes can be made to configurations in the scheme. Plasticity is essential for nontrivial learning, and for any system working on limited information in an uncertain world.

However, there are times when one does want to be able to make a claim of exhaustiveness in a representation. For example, we might want to represent that all the relations of a certain kind, between the entities represented in the configuration, are also represented in the configuration; or, that all the facts about some entity, which are in some sense relevant to some problem or task, are present in the configuration.

One important example of the need for this sort of assumption is the well-known frame problem. Consider a traditional description of the monkey-bananas problem, in natural English. How do you know there isn't a rope from the box, over two pulleys, and down to the bananas (so that as you move the box, the bananas ascend out of reach)?* Well, we assume that the simple description has given us all the relevant information to do with causal chains in the situation: we assume it is an exhaustive account of the machinery of the room. Much of the difficulty of the frame problem lies in the impossibility of expressing this assumption in the predicate calculus. (Using the causal-connection theory developed in [9], we could say there was no causal connection between the box and the bananas; but that is not strictly true: the monkey can throw one at the other, for example. In any case it is unsatisfactory as a general solution.)

(Parenthetically, I would like to take this opportunity of suggesting that we should stop talking about the frame problem. There are, it is now clear, several independent difficulties bound up in the normal formulation. One was just noted; another is the lack of a good representation of the way in which causal chains follow trajectories determined by mechanisms in the environment; another is the heuristic problem of organising inferences involving causality. The presence of state-variables in the language is not part of the problem, as some authors seem to have believed.)

Another, rather different, example of a claim of exhaustiveness is provided by the sort of analogy reasoning epitomised by Evan's well-known program, and formalised in the Merlin system [16]. This is normally regarded as essentially non-deductive reasoning, but it can be regarded as deductive reasoning from some rather strong hypotheses. Thus, suppose we decide that a certain collection of properties of an individual, taken together, constitutes an exhaustive description of it, from a certain 'point of view'. For example, we might say that a man was a mammal with a nose and feet. What could this mean? Well, it might mean that certain facts about men can be established by the use of these properties only: that is, an essentially proof-theoretic assertion. Now, with this meaning, if we replace the properties in the description with others (of the same 'type', in some sense: e.g. with corresponding sort structures in a multi-sorted logic), then corresponding facts can be established relative to the alternative properties. Thus, in the example of [16], if a pig is a mammal with a snout and trotters, then we can regard a pig as a man with a snout for a nose and trotters for feet. The existence of the 'analogy' follows from the (presumed) sufficiency of the list of properties. It follows deductively from the claims expressed in the putatively exhaustive descriptions of men and pigs.

---

*This example due to Alan Newell

This account of analogy (which is related to Kling's ideas) suggests natural explanation of (for example) the breakdown of an analogy (the claim of exhaustiveness fails: e.g. some property of men needs other hypotheses than those of noses and feet), and naturally relates 'analogical' and 'deductive' reasoning.

Now, there is a way in which a direct representation can be considered to be exhaustive, by a slight alteration to the semantic rules. We may insist that the medium-defined relations of a configuration completely mirror the corresponding relations in the reality: that is, that a medium-defined relation holds between subconfigurations if and only if the corresponding relation holds in the world between the entities denoted by the subconfigurations. Let us call such a representation, strongly direct.

For example, a map is strongly direct in this sense: all the 2-dimensional spatial relationships which hold between towns, rivers, etc. also hold in the map between the symbols denoting them. (They are also, often, exhaustive in a stronger sense; that all the entities (towns, rivers) present in the reality are denoted by symbols in the map. Thus we say, of a map with a river missing, that it is wrong, not just incomplete. It misleads us because we assume that if a river isn't marked, it isn't there.)

An example of a direct representation which isn't strongly direct is provided by networks: a relation may well not be displayed in the graph. However, we can also use networks as a strongly direct representation, if we consider the medium to be the algebra of relational structures with a given signature. Thus we would insist that either all or none of the instances of a certain relation are displayed in the network. A family tree is a strongly direct representation in this sense, relative to the relationships 'child of' and 'married'. With this semantics, (which can be specified algebraically) a network is no longer equivalent in meaning to the simple conjunction of the atomic facts represented in it. (If we call this conjunction C, it is equivalent to C with the added rule: if C $\not\vdash \emptyset$ then $\neg\emptyset$, for any atom $\emptyset$ in the appropriate vocabulary.) Winston's use of networks to describe concepts [26] seems to be closer to this latter semantics than to the former one, for example.

In unpublished work at Stanford, Arthur Thomas is developing a different approach to combining exhaustiveness with a Tarskian semantics, based on Hintikka's 'model sets'.

Strongly direct representations are less plastic than direct/Tarskian representations, in that information cannot be accumulated piecemeal in them. To add information to a strongly direct representation is to alter the information expressed by it. Alterations, as opposed to mere additions, raise problems of their own.

The trouble with alterations is that the information being altered may have been used earlier as a premis in a deduction of some kind. Thus, other pieces of information which obtain their support in some sense, from the altered information, are now endangered, and should probably be re-examined. This seems to require the system to keep an explicit record of how it formed its beliefs: a history of its own thinking. And this seems prohibitively expensive (of either space or time: one could recompute rather than store), due to exponential factors in the amount of information required.

Under some circumstances, it may be possible to re-evaluate a belief on criteria independent from its original derivation, as for example in adjusting the fit of lines to a gray-level picture (this observation due to Aaron Sloman), but in general I do not think one can avoid the problem.

This dilemma seems insoluble. There must be a clever series of compromises which steer us between its horns, but I don't know of any work in this direction.

More far-reaching alterations to a representation which one can envisage include changes to the basic ontology, to the sorts of entity to which it refers. The introduction of substances into a scheme oriented towards describing individuals is such a change, for example (see section 6). Minsky and Papert [15] give another rather simpler example: the change from a two-place relation of support between objects to a support relation between an object and a collection of objects, needed to describe e.g. an archway or a table. As they remark, this alteration seems to require a complete rebuilding of all knowledge about support, for the actual logical grammar of the assertions has changed. However, in this and similar cases one can see the general outlines of how it might be done. The fundamental step is to introduce the new notion of support as a new primitive idea (this is the really 'creative' act), and then define the old notion in terms of the new one, i.e. regard the old concept henceforth as an abbreviation for its definition in terms of the new one. In the example, support (a,b) would be defined as support (a,{b}). This preserves the old theory of support as a special case of a new, more general, theory (which is yet to be defined). There is, however, a strong constraint on the new theory, viz. that it'explains' the old theory. Thus, statements of the new theory which translate statements of the old theory must be derivable (in the new theory).

This corresponds to the idea that the alteration is somehow a refinement of, or an improvement upon, the former representation. A similar change, but in which the new concept completely replaced the older concept, which was rejected as wrong or unusable, could not be handled this way.

This whole issue of plasticity in representation is important not only for learning, but also for everyday program development reasons, and for debugging. For we must be able to modify and improve the representations of knowledge in the programs we write, and this is often far from easy.

4. Evidential Reasoning

There is a continual need, especially in perception, to represent information concerned with one belief being evidence for another. It seems clear that one needs to make reasonings concerning such matters explicit so that they can be properly related to other reasonings, and can be adjusted in the light of experience (see section 3). The problem is how to adequately express the notion of one knowledge-fragment (or collection of fragments) being 'good evidence' for another.

There seem to be several notions of good evidence, but all can be put into a common framework: A is good evidence for B (under assumption Th, say) if the conjunction (A & not B) is somehow unlikly or implausible (or: if this follows from Th). Thus, for example, if A entails B then A is very good evidence for B, for then (A & not B) is impossible. In Guzman's work [8] back-to-back 'T's are good evidence for occlusion of

one body by another, since the former without the latter is an unlikely coincidence. In a world where lines of bricks were common, back-to back 'T's would be weaker evidence since the conjunction of such an observation with the hypothesis of a single occluded body would be less implausible: the possibility of a line of bricks being occluded would be an alternative explanation of the evidence.

This sort of observation suggests an account of 'plausible' as follows: (A & not B) is implausible if B entails A (occluded body entails back-to-back 'T's) and no other B of the suitable sort (e.g. no other hypothesis about physical arrangements of bodies) entails A. If there are several such explanations of A then A is evidence that one of them holds, but it doesn't distinguish which one. This decision has to be made on some other basis, for example by the use of Baye theorem in a probabilistic scheme, or by choosing the simplest hypothesis or the one most compatible with other entrenched beliefs.

An important problem is how to discover the collection of possible or likely explanations. (This point was emphasised to me by Aaron Sloman). How many ways can back-to-back 'T's arise? I can think of three, and am pretty convinced there aren't any more; but I have no idea where that conviction comes from, or how I would prove it. The 'theory' of lighting and perspective which is welded into Waltz's program has this nice exhaustive character, expressed in effect as a collection of explicit disjunctions. This works up to a point, but how could a program derive these lists from a description of, for example, the lighting conditions and geometry of the scene?

Involving the background theory of lighting, etc., in this way is not just of academic interest. A vision system which could make hypotheses about the lighting conditions, the sorts of reflectivity in the scene, etc. would find it necessary to be explicit about the role of such assumptions in interpreting pictorial phenomena. Thus we might have: if there is strong unidirectional lighting then shadows have sharp edges and are dark; so if this is the corner of a shadow then it will have a dark interior: $\overline{Th} \supset (B \supset A)$; from which we may use corners with sharp edges and dark interiors as evidence for shadows. Reasonings like this will be essential in any system with the ability to percieve a range of scenes. (Similar remarks apply to other perceptual situations, e.g. understanding speech, handwriting, children's stories.)

5. Control

A system which makes inferences to generate new facts must control its inference-making capabilities in some way. This control itself requires the storing and using, by the system, of information about the deductive process. That is: the system must represent and use knowledge about its own deductive behaviour.

In conventional programming languages this information is sometimes represented implicitly in, for example, the ordering of statements in the body of a program (which is a strongly direct representation of the time-order of control flow, provided jumps are forbidden) and sometimes explicitly in, for example, the correspondence in names which relates procedure calls to their corresponding procedure bodies. In PLANNER-like languages, the latter representation breaks down since 'procedures' are called not by name but by pattern matching, and is replaced by the more flexible device of advice lists. The ordering information is still represented implicitly, however.

Now, this metadeductive information needs to be made explicit and separated from the factual information represented in the scheme, for reasons of semantic clarity, plasticity and deductive power. For example, the residue of PLANNER upon separating out control information is a logic which resembles intuitionist predicate calculus . Results like this are important: they give us an inkling of how a semantic theory might be put together. (Unfortunately, intuitionist logic itself has a rather murky semantics.) The control information which can be represented in PLANNER is rather limited, as the CONNIVER authors emphasise [23]. Their solution, to give the user access to the implementation primitives of PLANNER, is however, something of a retrograde step (what are CONNIVER's semantics?), although pragmatically useful and important in the short term. A better solution is to give the user access to a meaningful set of primitive control abilities in an explicit representational scheme concerned with deductive control. This is the basic idea of the GOLUX project now underway at Essex [11].

The problem is to find a good set of control primitives. What is control? One answer to this is to pick on a fixed mechanism (the interpreter) associated with the language, and to relate control to this mechanism in, more or less, the way an order code relates to an actual computer. But this tends to be inflexible and arbitrary. The GOLUX answer is that control is a description of the behaviour of the interpreter. The exact nature of the interpreter is not defined, only that it constructs proofs according to some predefined structural rules. The descriptions in control assertions constrain its behaviour more or less tightly. It is, I believe, important that control information be represented in a scheme compatible with the scheme used for 'factual' information, so that control can be involved in inferences, added to, and changed.

Control primitives in GOLUX include predicates on, and relations between, partly constructed proofs in the search space; descriptions of collections of assertions; and primitives which describe temporal relations between events such as the achievement of a goal (e.g. the construction of a proof). The major source of difficulty is the tension between the expressive power of these primitives and their implementability: it is important that they be sufficiently simple that their truth can be rapidly tested against the actual state.

GOLUX is based on recent ideas in computational logic [10,12]. Other authors have also recently emphasised that computational logic provides a powerful theoretical framework for problem-solving and computational processes [14,28,19], although we are not in complete agreement as to which is the best framework.

A common area of difficulty both here and in evidential reasoning is to get a good notion of a 'theory': an organised body of knowledge about some subject-area.

6. Substances, Parts and Assemblies

Every representational scheme known to me is based ultimately, like predicate calculus, on the idea of separate individual entities and relations between them.

But our introspective world-picture also has quite different 'stuff', viz. substances: water, clay, snow, steel, wood. Linguistically, these are meanings of mass terms. Substances are fundamentally very different from individuals, and I know of no scheme which seems capable of satisfactorily handling them. I became aware of this problem from reading

Davidson [5].

We often speak as though substances were individuals having properties and relations one to another and to more conventional individuals: steel is dense, blood is thicker than water, his head is made of wood. The relation "made of" seems particularly important. But appearances are deceptive.

Does 'water is wet' mean the same as 'all samples of water are wet'? I think it does: we certainly want to be able to infer from 'water is wet', that 'this sample of water is wet'. This suggests at first sight that we should treat pieces of stuff as individuals, which seems fairly acceptable. But these individuals are also rather strange, especially for fluids. If you put together two pieces of water you get one piece, not two: we have to speak of quantity (of stuff) before we can use arithmetic. (It is significant that, as Piaget has shown, children properly understand the concept of quantity only at quite a late stage of development.) Moreover, we should distinguish properties which a piece of stuff has by virtue of its being a piece (size, shape), from those which it has by virtue of its being made of stuff (density, hardness, rigidity): for the former, but not the latter, can be easily altered by physical manipulations. It really seems that we cannot get away from substances no matter how hard we try.

Let me emphasise that this problem is not a by-product of a nominlist philosophical position. I have no objections to platonic, abstract, non-physical individuals. That's not the difficulty. The difficulty is 'individuals' which appear and disappear, or merge one with another, at the slightest provocation: for they play havoc with the model theory.

This seems to me to be one of the most difficult problems in representation theory at present. The only way I can imagine handling substances is by regarding each substance as a (special sort of) individual, to which such properties as hardness, density, etc. are attributed. These individuals can be regarded as platonic ideals, or alternatively as the physical totality of all samples of the substance: you can take your nominalism or leave it. We have the naïve axiom

$$Stuff(x) \;\&\; madeof(y,x) \;\&\; z(x). \supset z(y)$$

(e.g. : a lump of hard stuff is hard).

which transmits properties from substances to pieces of them. (Care is needed: steel ships float, for example; a fact which often amazes young children.) Notice this axiom is first-order (in a sugared syntax). Quantity is now a function from (pieces)X(stuff) to some scale of measurement, so we can express conservation of quantity through some physical alteration Q by:

$$quantity(piece,stuff) = quantity(Q(piece),stuff).$$

And so on. This works up to a point, but seems to me to be essentially unsatisfactory.

There is a close analogy between being made of a substance, and being made up of a number of parts. And a corresponding analogy between quantity (of stuff) and number (of parts). Sand and piles of small pebbles are intermediate cases: and we often treat an assembly of individuals as a fluid, e.g. as in "traffic flow". The major difference seems to be that different scales of measurement are used in common-sense reasoning (but not in physics, where quantity is number of atoms), as the "paradox of the

heap" shows. This runs as follows: a heap with one stone in it is small. If you add just one stone to a small heap, it's still a small heap. Hence by mathematical induction all heaps are small. The 'paradox' comes by switching from the informal quantity scale of 'small-large' to the precise number scale. Induction is not valid in the former, which (for example) exhibits hysteresis.

Things are often made up of parts joined or related in some way. Obvious examples are physical objects made of pieces glued or assembled together: cups, cars, steam engines, animals. But there are others: processes made up of subprocesses; time-intervals made up of times. The idea of organised collections of entities being regarded themselves as entities permeates our thinking.

Now this fact strikes at the root of an 'individual-based' ontology in the same sort of way that substances do. The only way of handling collections is to count both the collection and its parts as individuals, related by some sort of made of or has-as-part relation. But then these assembled individuals behave in odd ways: they sometimes merge (two heaps make one heap) like pieces of stuff: sometimes they can be disassembled, cease to exist for a time and then perhaps be reassembled: is it the same individual? (Our intuition says: yes, in most cases).

Modal logicians now have very elegant semantic theories which can accommodate such odd behaviour in individuals. But these allow any pattern of vanishing, reappearing and changing properties. The point is to find a way of representing the fact that composite individuals have this special way of vanishing (being taken apart), and to distinguish, for example, those composites which cannot be reassembled (animals, cups) from those that can (cars, steam engines): and to do all this in a framework which assumes that things, by and large, don't just vanish and reappear spontaneously. Composites are thus a different sort of individual, in a very deep sense.

A related issue is how to state criteria upon which we reify a collection into a composite individual. Physical compactness is sometimes sufficient (a heap), but not always necessary (the wiring system of a house), for example. Of course, one does not expect a single general answer, but I do not know of any reasonable answers at all, even for special cases.

I have already remarked on the similarities between being made of (stuff) and being made up of (parts). Is this anything more than a facile analogy? Is there some common framework in which the fundamental ontological notion, rather than existence, is space-occupancy? It might be useful to strive for a representation which allowed the simultaneous expression in different schemes of both 'existence' and 'space-occupancy'. (The schemes would, I believe, have to be essentially different.) Indeed, in a crude way one can see how it might be done directly by "arrays of facts": the array subscripts give one access via spatial relationships to the local presence of objects, which also partake of relationships (represented by a network, say) between themselves and other, non-space-filling, individuals (such as colours). Decomposability is indicated in the array also by 'break lines' which separate the space into regions: different sorts of connection could be fairly easily handled (glued, detachable ...). But this is very crude and has several crucial drawbacks (notably plasticity: imagine moving an object through the space, preserving its shape.)

7. Some non-issues

7.1 Irrelevant classifications

Much heat is generated by disputes based on classifications which do not correspond with the facts, or which at least have outlived their usefulness. Tow such are the "generality vs. expertise" debate and the more recent "procedures vs. assertions" debate. Both of these arise from a revulsion against a particular early naive idea about how to organise intelligent programs, which one could (perhaps unfairly) call the general problem-solver fallacy. (Seymour Papert calls it, the blinding white light theory.)

This was the early insistence that problem-solving methods had to be wrapped up in black boxes called problem-solvers, whose (only) input was a problem and whose (only) output a solution. Problem-solvers were supposed to be as powerful and as general as possible. One had not to "cheat" by "giving" the problem-solver the solution in any sense, e.g. by reprogramming it or cleverly coding the problem in some way (this is made explicit in [7]). Unfortunately, of course, this collection of rules means that there is no way of getting subject-matter-dependent knowledge into the black box; for it cannot be there a priori (violates generality), and it cannot be put into the problem (cheating), and there aren't any other inputs. This is a caricature, but not much of a caricature. Much work in automatic theorem-proving was done with the implicit idea that the theorem-provers were to be regarded as problem-solvers in this sense (c.f. the widely felt 'need' for adequate criteria of relative efficiency of theorem-provers: "my problem-solver is more powerful than yours". (See [2, 10] for a fuller discussion).

The MIT school have now succeeded admirably in destroying this idea, but unfortunately have gotten it confused with some others. Surely we need both generality and expertise: the fallacy is not the emphasis on generality, but the insistence upon the black box and the "no cheating" rules. The general mechanisms of means-end analysis, heuristic search and computational logic should not be rejected, but rather incorporated into more flexible systems, rather than wrapped up in closed 'problem-solving subroutines' or 'methods' or whatever. Thus, to reject conventional uniform theorem-proving systems because they work with assertional rather than 'procedural' languages, is to miss the point. (Whether a language is considered to be a programming language or not, is largely a matter of taste, in any case. LISP can be regarded as (an incomplete) higher-order predicate calculus, or as a first-order applied predicate calculus: predicate calculus can be regarded as a programming language, although by itself not a very good one.) The force of the MIT criticism of computational logic is directed against the 'problem-solver' view and its consequences, especially the lack of any accessible and manipulable (programmable) control stucture in conventional theorem-proving systems. The GOLUX system referred to earlier is an attempt to fill this lack directly with an especially devised control language.

A more recent attack on conventional theorem-proving [7] is that it is too concerned with "machine oriented" logic, and not enough with "human oriented" logic. I confess to being quite unable to understand what this could possibly mean.

7.2 Semantics

Some authors, usually concerned with comprehension of natural language,

use 'semantic' as a vague term roughly synonymous with 'to do with meanings', where this means the same as 'not to do with grammars'. This follows a long and honourable tradition in linguistics (c.f. the use of such terms as "semantic markers" and the idea that linguistic deep structure is semantics).

I wish to emphasise however that this is not the same usage as that adopted here and in formal logic. And it is, I believe, very misleading. It militates against an understanding of the fundamental point that the meanings of linguistic expressions are ultimately to be found in extra-linguistic entities: chairs, people, emotions, fluids.....

As a recent example, Wilks' "semantic units" [24] are syntactic objects in a scheme; nowhere does he tackle the difficult and vital problem of describing exactly what sorts of extra-linguistic entities his "semantic units" refer to. It is easy to say: we must have substances and things and ... ; but what are these? There does seem to be the beginnings of some sort of sketchy semantic theory behind Wilks' formulae (actions have agents which are animate, etc.), but it is not articulated: and if it were, all the problems I have discussed would promptly appear. Similar remarks apply to Schank's work [20], and others.

I am not arguing that natural language should be given an extensional semantics. I distinguish sharply between a natural language, which is an informal and probably not even completely defined means of communication in the real world (is "Eh?" a sentence? Eh?), and a formal deductive scheme for representing knowledge. (It has been suggested to me that the distinction may be related to Sassure's distinction between Langue and Parole, but I have not investigated this.) I suspect that those who deny the usefulness of extensional semantics would also deny the validity of this distinction. That is probably a perfectly respectable philosophical position: but I submit that it is bad engineering.

7.3 Fuzziness and Wooliness

Several authors have recently suggested that more exotic logics, especially 'fuzzy logic', are necessary in order to capture the essentially imprecise nature of human deduction. While agreeing that we have to look beyond first-order logic, I find the usual arguments advanced for the use of fuzzy logic most unconvincing.

Introspection does not suggest to me that intuitive reasonings are essentially imprecise; still less that they are precise in terms of a real-valued truth-value in the unit interval (which is what fuzzy logic would have us accept). Even ignoring introspection, fuzzy logic does not seem very useful, for where do all those numbers come from? (This is McCarthy's point.)

The typical example brought forward to illustrate the need for fuzzy logic concerns the everyday use of such words as 'large', 'small', 'old', 'expensive'. Now it seems to me that, when I say a heap is small, I mean just that. If asked, "Is what you say true?", I will correctly answer "yes", and become impatient with the protagonist. These are precise words but they refer to vague measuring scales. As remarked earlier, for example, the scale 'small-large' exhibits a different topology from the integers or from real intervals: it is more like a tolerance space [27] and it may have hysteresis (an intermediate heap will be considered small if it began as small and grew, and considered large if it began as large and shrank), and it may have gaps in it. The point however is, that we should keep the vagueness of the scale localised into it, rather than

letting it infect the whole inferential system. This enables different 'fuzzy' measuring scales to coexist, which is important. We should investigate what sorts of measurement scales are useful for various purposes.

The most drastic alteration to the actual logic which seems to be needed to handle words like this is to move from a 2-valued to a 3-valued logic, and it is not absolutely clear that even this small step is really necessary.

The view expressed here is different from the one I held some years ago. I have become more respectful, since then, of the unexplored possibilities of predicate logic.

Acknowledgements

Many people have helped me with conversations, suggestions and criticisms. I would like especially to thank John Laski (section 1); Aaron Sloman (sections 2 and 4); Harry Barrow (section 3); Jim Doran (section 4); Bruce Anderson, Carl Hewitt, Johns Rulifson (section 5); Seymour Papert, Gerald Sussman, Bruce Anderson (section 7.1). More generally, I owe much to many conversations with Richard Bornat, Mike Brady, Jim Doran and Bob Kowalski. Alan Bundy, Aaron Sloman and Yorick Wilks made many useful criticisms on an earlier draft.

References

(1) S. Amarel. More on Representations of the Monkey Problem. Internal Report, Carnegie-Mellon University (1966)

(2) D.B. Anderson & P.J. Hayes. The Logician's Folly. DCL Memo 54, Edinburgh University (1972)

(3) R. Balzer. A global View of Automatic Programming. 3rd IJCAI proc. Stanford (1973) (see "Problem Acquisition", paragraphs 3&4)

(4) E. Charniak. Jack & Janet in Search of a Theory of Knowledge. 3rd IJCAI proc., Stanford University (1973)

(5) D. Davidson. Truth and Meaning. Synthese 17 (1967)

(6) M. van Emden & R. Kowalski. The Semantics of Predicate Logic as a Programming Language.

(7) G. Ernst & A. Newell. Some Issues of Representation in a General Problem-Solver. Proc. Spring Joint Comp. Conf. (1967)

(8) A. Guzman. Computer Recognition of Three-Dimensional Objects in a Visual Scene. Report MAC-TR-59, MIT (1968)

(9) P.J. Hayes. A Logic of Actions. Machine Intelligence 6, Edinburgh University Press (1971)

(10) P.J. Hayes. Semantic Trees. Ph.D. thesis, Edinburgh University, (1973)

(11) P.J. Hayes. Computation & Deduction. Proc. MFCS Symposium, Czech. Academy of Sciences, (1973)

(12) P.J. Hayes. Simple and Structural Redundancy in Nondeterministic Computation. Research memorandum, Essex University (1974)

(13) C. Hewitt. PLANNER. MIT AI Memo 258 (1972)

(14) D. Loveland. A Hole in Goal Trees. Proc. 3rd IJCAI, Stanford (1973)

(15) M. Minsky & S. Papert. Progress Report. AI Memo 252, MIT. (1972)

(16) J. Moore & A. Newell. How can Merlin understand? Internal memo, Carnegie-Mellon University (1973)

(17) A. Nevins. A Human Oriented Logic for Automatic Theorem Proving. MIT AI Lab. Memo 268 (1972)

(18) A. Rosenfeld. Isotonic Grammars. Machine Intelligence 6, Edinburgh University Press (1971)

(19) E. Sandewall. Representing Natural Language Information in Predicate Calculus. Machine Intelligence 6, Edinburgh (1971)

(20) R. Schank. The Fourteen Primitive Actions and their Inferences. Stanford AIM-183, Stanford University (1973)

(21) Schank & Colby (eds). Computer Models of thought and language. Freeman (1974)

(22) A. Sloman. Interactions between Philosophy and Artificial Intelligence. Artificial Intelligence 2, (1971)

(23) G. Sussman & D. McDermott. Why Conniving is Better than Planning. MIT AI memo 255A, 1972

(24) Y. Wilks. Understanding Without Proofs. Proc. 3rd IJCAI, Stanford (1973)

(25) T. Winograd. Understanding Natural Language. Edinburgh University Press (1971)

(26) P. Winston. Learning Structural Descriptions from Examples. Ph.D. Thesis, Report MAC-TR-76, MIT (1970)

(27) C. Zeeman. Homology of Tolerance Spaces. Warwick University, 1967

(28) R. Kowalski. Predicate Calculus as a Programming Language. DCL Memo 70, Edinburgh University (1973)

(29) E. Sandewall. The conversion of Predicate-Calculus Axioms, Viewed as Non-Deterministic Programs, to Corresponding Deterministic Programs. Proc. 3rd IJCAI, Stanford (1973)

PROGRAMS THAT WRITE PROGRAMS AND KNOW WHAT THEY ARE DOING.

John Knapman, Bionics Research Laboratory, School of Artificial Intelligence, University of Edinburgh.

Abstract

The concept of run-time structure, expounded by Stansfield[1], is explored in the light of its use in a computer program currently being developed that is to acquire a natural language. Special facilities have been provided for programs to modify and extend themselves by interacting with a record of their behaviour and experience.

Descriptive Terms

Run-time structure, control structure, list programming, procedural representation, language acquisition.

* * * * * * * * *

1. Before the programming system is described, a brief outline will be given of the application for which it is being used. The program is to acquire a natural language through the medium of a teletypewriter. It begins with no vocabulary and, on encountering an unfamiliar word, synthesises its meaning by examining the situation in which it is found and constructing a sub-program. For instance, the word asterisk is taught by making the program print an asterisk (by enclosing the instruction in square parentheses) and then supplying the word.

```
: [PRINT('*')]

*

: ASTERISK
```

In this dialogue, lines entered by the human tutor are preceded by a colon; the others are printed by the program.

As a result of the above sequence, a sub-program is written (the text appears in section 4 below) and, in future, it will be executed whenever the word "asterisk" is read, whether alone or as part of a sentence. The effect of running the sub-program depends on the context. The word "print" causes "asterisk" to be interpreted in the imperative sense. It is taught thus.

```
: PRINT AN ASTERISK [(PRINT('*')]

*
```

The teaching of "and", "a" and "you" will not be described here. Instead, numbers will be introduced since they involve an interesting new principle.

```
: PRINT AN ASTERISK AND AN ASTERISK

**

: TWO ASTERISKS
```

In the first line, the sub-program for print will itself carry out the interpretation of "an asterisk". The main program will then set in /

/in motion the phrase "and an asterisk". In the last line, "two" is a new word and, after "asterisks" (taken as synonymous with "asterisk") has been run (it compares the instructions that caused two asterisks to be printed this time with the original single instruction), "two" becomes associated with the asterisk left over.

At this stage the command to print two asterisks would be obeyed correctly but the following incorrect result would also occur.

```
: PRINT TWO DOTS

.*
```

A second example is necessary to derive the proper meaning.

```
: PRINT A DOT AND A DOT

..

: TWO DOTS
```

A comparison takes place within "two" and out of the conflict between the given and the expected, its meaning is induced. It takes on the function of duplicating the object that follows it. Higher numbers are similar.

More details will be given below after an explanation of the programming system. The working of the following example is also presented there. The word "did" is introduced to the program in a question.

```
: PRINT AN ASTERISK

*

: DID YOU PRINT AN ASTERISK [PRINT('Y');PRINT('E');PRINT('S')]

YES
```

"Did" now means: "Examine the record of the immediate past to see whether the action performed there matches the meaning of the rest of the sentence following 'did'". The meaning of "did" can be extended to reply "no" when the two do not match.

```
: PRINT A COMMA

,

: DID YOU PRINT A DOT [PRINT('N');PRINT('O')]

NO
```

As presently implemented, the program can also learn to answer questions like: "What did you print?" and "How many dots did you print?". It can associate the numerals with strings of appropriate length (e.g. 3 with '£££'), and be taught to perform addition on them. It is hoped to extend these numerical capabilities and to explore other concepts, such as that of time and of language as an activity in its own right.

The /

/The application program will serve to illustrate some of the uses to which the programming system can be put. The system will be described by first showing how it fits into the historical development of programming. Then its unique attributes will be presented.

* * * * * * * * *

2. "Programs that know what they are doing" is a quotation from Stansfield(1) revealing the philosophy behind his programming system PROCESS 1. To have "programs that write programs" is not a new idea. LISP was invented by McCarthy et al (2) with the express purpose (as stated in the last paragraph of the introduction to the manual) of permitting programs to act on programs. Programs in LISP are stored with each instruction containing an explicit pointer to the next (i.e. in a list) so that changes are convenient to make. An equally important fact about LISP is its functioning as an interpreter; programs are not converted to another form for execution. This makes it possible for a program in temporarily interrupted execution to be modified by another program, or even to modify itself without interruption. Such facilities are presented below in section 3.

A convenience provided by most programming languages is the procedure - also known as the sub-program, sub-routine or function. In FORTRAN and similar systems the flow of control through a number of functions follows a hierarchical discipline. Dissatisfaction with the rigidity of this kind of control structure has led to the development of back-tracking (3) and generalized jumps (4). A formulation of generalized jumps independent of a goal formalism is found in (5). The intuitive advantages of procedures (functions, etc.) are retained while some of the flexibility of machine-level programming is restored.

As well as providing such flexibility, Stansfield (1) had the idea of making available to the program a representation of the control structure, known as the run-time structure. Now a procedure may inspect a record of which procedures were previously invoked and are possibly awaiting a result. The procedure might send a result directly to the one expecting it, bypassing the intervening hierarchy, or indeed take any action at all, after obtaining this information. By sending a result, of course, control is also transferred to the procedure receiving that result. However, the run-time structure representing the execution of the lower level procedure is still available and can be stored for a restart at a possible time in the future. The execution of a procedure is a process (hence the name PROCESS 1) and the run-time structure that has been furnished at the time of transfer of control represents a suspended process. There can be any number of suspended processes saved by a program. In addition, there is always one procedure actually being executed. The execution of this procedure is known as the current process and it too possesses a run-time structure available for inspection and modification at any time.

The format of a run-time structure is that it contains one record for each invocation of a procedure. This means that if a procedure is called several times, as in recursion, there will be one record (an "activation record") for each call. If, however, a process that suspends itself is later restarted, the same activation record will be employed again. The record points to the next instruction to be executed in the text of the procedure. These records are chained together on the run-time structure starting with the most recent activation in the process that the structure represents and linking back to the earliest.

A system of levels is introduced and is used here to make these chains of records more manageable. Stansfield describes a method of level numbering which I call "absolute". I have implemented PROCESS 1.5 (6) incorporating absolute and relative levels and the discussion below refers to the latter, since they are the ones used in this application. A procedure can be started by being either called or run. If it is called, its activation record appears in the run-time structure on the same level as its predecessor; if it is run, a new level is made.

As an example consider the analysis of the sentence: "Print two dots and a comma". Two or three procedures are involved in the interpretation of each word, including at least one that is unique to the word in question. By the use of run and call, each word's processing can be confined to a separate level, making possible the kind of communication between words that will be outlined when the examples of "did" and "two" are explained.

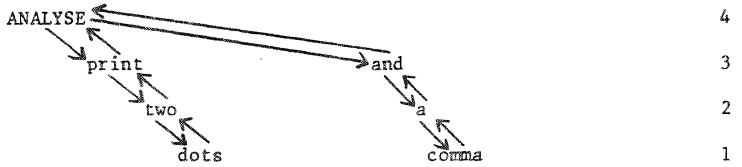


Fig. 1 Flow of control in interpreting a sentence.

The levels make it natural to draw the diagram of fig. 1 in which they are numbered from the point of view of the procedures involved in "dots" or in "comma". Each downward arrow represents a run to a new level. Within each level the processing for the word involves calling appropriate procedures. If the syntax of the word demands the interpretation of the following word or phrase before processing can be completed, the procedure for the word will itself contain the necessary run instruction to bring that about. The verb "print" requires an object and so it runs the interpretation of the next phrase. The number "two" also requires an object on which to perform its duplication function. "Dots", on the other hand, has no such requirement and returns control back to "two". It may pass control back to any of the levels by means of the operation "rise" (a generalization of "return" in many programming languages) followed by a number: RRISE 1 will return to the dictionary procedure (known as FIND) that called "dots" on the same level, RRISE 2 will return to "two", RRISE 3 to "print" and RRISE 4 to ANALYSE. The numbers are relative to the current process: after RRISE 2 to "two" another RRISE 2 goes to "print". In fig. 1 each upward arrow represents RRISE 2. The total structure that you see never exists at any instant, although it could be reconstructed automatically. The first branch is built and destroyed, followed by the second, as the words are being interpreted.

An interesting property of fig. 1 is that not only does it represent the flow of control through the process of interpretation but it corresponds to the syntactic structure of the sentence. It can be viewed as a parsing tree. According to Halliday (7), grammar imposes a second dimension on the linear succession of elements that is the substance of language and it does this by a process of segmentation into units. Syntax exhibits the shape of the process of comprehension; it is one aspect of that single process.

Winograd (8) was able to demonstrate the enormous power and flexibility gained by his program having a procedure representing the meaning of each word. His strong arguments for the close interaction of syntax and semantics carry great conviction and he used the best programming tools available at that time, i.e. back-tracking, MICRO-PLANNER (9) and PROGRAMMAR (10) to assist that endeavour. But his program still possesses a separable syntactic component.

With the concept of run-time structure, it has been possible in the examples considered to incorporate both the syntactic and semantic aspects of processing for a word into one procedure and the learning process is greatly simplified by the consequent uniformity as compared with a program which has a distinct syntactic component. Learning is further facilitated because in a run-time structure the procedure that performs the synthesis for a new word has available a representation of the whole activity in progress, including linguistic activity. The method has intuitive appeal because, for one thing, the program will answer a grammatically ill-formed question such as: "What did you printed" which most English speakers would understand. ("Printed a dot" would not be interpreted as a command, however).

* * * * * * * * *

3. Another substantial advance made by high level programming languages is the convention of naming areas of storage, which are then known as variables, rather than numbering them. In a procedure, the programmer declares the names of variables to be used in that procedure. When it is executed, space for the variables declared is provided in the activation record and whenever a value is assigned to a variable the value is put into that space. This kind of variable is local. If a name is mentioned without being declared then it refers to the variable of another procedure and is termed non-local. The rule for determining to which activation record a non-local variable refers in an ambiguous case is the binding convention. Dynamic binding has been found the most suitable when procedures are manipulated as objects. To quote Burstall et al (11): "It allows functions (i.e. procedures) to be produced as the results of other functions which is quite impractical with the ALGOL 60 way of handling non-locals. This adds greatly to the power of the language". This point of view has been borne out in the application program. The procedures for "what" and "how many" include parts adapted from the synthesising routines. After inclusion their non-local variables automatically refer to the new environment without the need for any textual modification. In this respect PROCESS 1.5 differs completely from PROCESS 1. Dynamic binding is used in LISP and POP-2 (11). A fuller treatment of this matter (the frame problem) and its extension to processes as manipulable objects is given in (6).

A record on the run-time structure in PROCESS 1.5 contains all the information relevant to performing a process. There is provision for programs to access and modify, by name, the value of any variable in any process.

The primitive is VALUE. As an example, the following will initiate a process by running the function FUN which leads to a rise back from a subordinate process. The run-time structure for that process is supplied by the system in a global variable CONTINUE. The value of X and A will be assigned to Y and B in the subordinate process which will then be resumed.

```
RRUNS FUN;  VALUE(CONTINUE, [X A])->VALUE(CONTINUE,[Y B]);
RRUNS CONTINUE;
```

Thus these objects constitute a highly structured presentation of the entire state of the machine, since the records also provide access to the text of the procedures whose execution they govern. It only remains to describe the form of their text and to present the facilities for them to generate and manipulate one another. These are new and experimental facilities and do not appear in the earlier documentation for the system.

The textual format of a procedure is a sequence of instructions separated by semicolons. For example:

```
RRUNS  COMPARAND  FINDASSOCS;
CALL  CHECKASSIGN:  ASSIGN  DIFFUN1;
```

As implemented, PROCESS 1.5 will, in addition, accept statements in an extended POP-2 notation and convert them to the above form. In fact, the programs for this application are written in a hybrid language. In the basic form (illustrated above) each instruction begins with an operation (the operations are CALL, RRUNS, RRISE, ASSIGN, NOOP, GOTO, RUNS and RISE) followed by variable names or actual data. For every operation except GOTO and ASSIGN, items in the instruction are placed on a push-down stack. An assignment removes items from the stack and places them into the variables specified. In the case of RISE and RRISE, the last entry on the stack is taken to be the level to which return is made. For CALL, RRUNS and RUNS, the last entry refers to the procedure to be invoked. RUNS and RISE perform as in PROCESS 1; RRUNS carries out the "run" function outlined in section 2. The performance of RRISE is also set out there. CALL initiates a procedure on the current level.

If we prefix C- or NC- on to any of these they become conditional on the outcome of the preceding instruction. For instance, altering CALL CHECKASSIGN to CCALL CHECKASSIGN would cause the procedure to be called only if the result from FINDASSOCS was TRUE. Similarly, NCCALL would mean call only if the result was FALSE. This was devised so that choices would automatically leave a clear indication on the run-time structure of the fact that they arose and of which branch was taken. It was also designed for convenience in making changes to the text consequent upon the outcome of the conditional. Such amendments are essential in this application because a procedure will normally be generated from one situation and the ability to make later modifications in the light of new but related experience is of paramount importance.

As a matter of fact, in the present program all the data are procedures except in the first stage of input from the teletypewriter and that is soon converted to procedural form. As a result, the only source of conditionals is the comparing of two procedures. (Comparing two run-time structures reduces to a series of comparisons between the procedures to which they refer directly or indirectly).

The program for comparing two procedures is called DIFFERENCE. It will examine the first procedure to see if it contains the second and it will return the result TRUE or FALSE, accordingly. In the TRUE case, it will also yield the difference between them in the form of two more procedures which will be the extra parts of the first procedure preceding and following the common portion. There is a special case of DIFFERENCE known as PEMPTY which simply ascertains whether a /

/a procedure contains any text at all. Such a check is usually necessary at the end of a series of DIFFERENCE operations during which a procedure has been broken down.

Building procedures up also demands special facilities, as procedures are somewhat cumbersome objects to manipulate (mostly because of the use of variables). As with any programming system, they can be typed in by the user but the object here is to allow programs to write them as well. To start a new procedure, a skeleton is first created by a call to INITPROC, which places the skeleton in a standard global variable and makes it available to be assigned to a variable. Additions can then be made to it using a program called ADD. An example appears below. Text is supplied by programs in the internal PROCESS 1.5 form or in the extended POP-2 form or a mixture of the two. (It is written in square parentheses [ ] denoting a POP-2 list). Two procedures can also be joined to produce a third.

Another use for the program ADD is to insert text into the middle or at the beginning of a procedure. There is a method for positioning a pointer to a particular instruction or part of an instruction before adding or, alternatively, deleting text. The pointers are set up by means of the function EDIT. After writing EDIT(ACTION); the procedure ACTION may be modified by means of various search and delete commands as well as by use of ADD. There is special provision for doing this in conjunction with the run-time structure when a procedure has been executed. If we write EDPOSITION(PFINDRET(1)); then the procedure that invoked the current one will be prepared for modification with the pointers positioned at the place in the text where the call was made. For instance, if a DIFFERENCE test yields an unexpected result and causes another procedure to be called, the instructions related to the test (and the values of variables) can be inspected and changed. In fact, this is the normal way in this application by which procedures representing the meanings of words are extended after their initial synthesis. Carroll (12) remarks that the development of a word in child language is far more complex and interesting a process than its initial acquisition. This is my justification for seeking a method of procedural modification that interacts with the record of a process in the run-time structure.

* * * * * * * * *

4. Two illustrations will be given of these methods at work. The first involves learning the word "two" and the second is the "no" reply to questions beginning: "Did you print". The teaching situations were outlined above and the first to be considered is "two asterisks".

The text of the procedure (slightly simplified) for the word "asterisks" is given below. You will recall that the procedure to be invoked appears at the end of a CALL instruction.

```
CALL MAKEASTERISK;

CALL REFERENCE  RESULT  DIFFERENCE;  NCCALL  XRESOLVE;

ASSIGN  DIFF1  DIFF2;

CALL DIFF1  PEMPTY;  NCCALL  EXRESOLVE;

CALL  DIFF2  PEMPTY;  NCCALL  EXRESOLVE;
```

This procedure was synthesised in an earlier situation, as was MAKEASTERISK, which consists of the following:

```
CALL INITPROC;  ASSIGN RESULT;

CALL [CALL '*'PRINT] ADD;
```

The processing for this word is to generate a procedure for printing an asterisk and to compare that procedure (stored in RESULT) with the REFERENCE which, by default, has previously been set to the preceding action in the dialogue (i.e. the printing of two asterisks). After the comparison, checks are made to ensure there is nothing left over. Each test is followed by a call to a resolving routine conditional upon the failure of that test.

In the "two asterisks" situation, where "two" is an unknown word, processing commences with a procedure known as SYNTHESISE with the objective of ascertaining the meaning of the new word. It causes the word "asterisks" to be interpreted and the code presented above is executed, resulting in a call to EXRESOLVE because DIFF1 has been left with the second asterisk. The run-time structure is illustrated in fig. 2, where FIND is the procedure that locates the meaning of a word. EXRESOLVE is able to extract the contents of DIFF1 from the "asterisks" procedure at level 1 and, detecting the presence of SYNTHESISE at level 2, passes it the information.

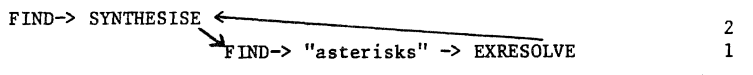


Fig. 2 Run-time structure for "two asterisks". The downward arrow indicates RRUNS, the upward one RRISE 2, and the horizontal arrows CALL.

After the first example, SYNTHESISE generates a procedure as the meaning of "two". It is like the one for "asterisks" shown above, except that MAKEASTERISK is replaced by a procedure that runs the imperative interpretation of the following word before adding its own asterisk printing instruction on to RESULT. The imperative is forced by setting the point of reference to a null value. (Local variables named REFERENCE are used to eliminate interference of contexts).

When in the second situation two dots are encountered, the assumption that "two" just means an asterisk in particular circumstances is violated and the program must seek a different explanation. In terms of code, that means that the DIFFERENCE comparison in "two" failed to match a procedure for printing two dots with one for an asterisk and a dot. Consequently XRESOLVE was called with the task of finding a match for the alien procedure and it has a number of sources from which to do this. These include that part of the sentence preceding the word in question, the word or phrase immediately following, further segments of a long sentence and actions indicated by the tutor. In the present case, the imperative interpretation of the following word does yield a match (in the sense of a successful DIFFERENCE operation) and recursion within XRESOLVE disposes of the remainder from this comparison. Notice that XRESOLVE is also involved in the process of imperative interpretation: if REFERENCE is null it cannot perform its matching operations and issues RRISE 2, leaving the action in the global variable RESULT.

XRESOLVE now synthesises a new procedure, again like "asterisks", wherein the equivalent of MAKEASTERISK is a run of the imperative interpretation of the following word and duplication of the result. XRESOLVE concludes by inserting a call to the new procedure in place of the call to XRESOLVE in the original version of "two". (The run-time structure identifies the location of the instruction to be modified). "Two" will now behave correctly, using the original portion when examining asterisks and the new portion in other contexts, including the imperative.

The second illustration involves learning and extending the word "did", which has a somewhat more elaborate procedure than "asterisks" but still contains DIFFERENCE tests, the main one being a comparison between the point of reference (the past action) and the interpretation of the rest of the sentence. In the "no" situation the result of the test is negative and a call to XRESOLVE takes place.

In fact, none of the remedies described yields a match. The intended meaning is, after all, that the program should answer "no" when this particular comparison fails. Thus XRESOLVE is once more required to replace the call to itself. An instruction is substituted in "did" that will generate a procedure to print "no" when the comparison fails between the point of reference and the result of the following clause.

That concludes the examples, which have been intended to illustrate how the run-time structure presents a record of the behaviour of a program in a way that is usefully related to the actual procedures that give rise to that behaviour. This usefulness has two aspects. One is that a procedure has access to the context in which it is acting. The other is that the right kind of information is available for a program to expand and develop itself.

Acknowledgements.

I should like to thank Dr. Howe and my colleagues at the Bionics Research Laboratory for constructive criticism. Financial support from IBM United Kingdom Ltd. and from the Science Research Council is gratefully acknowledged.

References

1. Stansfield, J. L. (1972) "PROCESS 1: a Generalisation of Recursive Programming Languages", Bionics Research Reports No. 8, School of Artificial Intelligence, University of Edinburgh.

2. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I. (1962) "LISP 1.5 Programmer's Manual", M.I.T. Press.

3. Hewitt, C. (1972) "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", AI TR-258 (Ph.D. Thesis) Artificial Intelligence Laboratory, M.I.T.

4. Sussman, G. J. and McDermott, D. (1972) "CONNIVER Reference Manual", AI Memo No. 259, Artificial Intelligence Laboratory, M.I.T.

5. Bobrow, D. G. and Wegbreit (1972) "A Model and Stack Implementation of Multiple Environments", BBN Report No. 2334, Bolt, Beranek and Newman, Inc., Cambridge, Mass.

6. Knapman, J. M. (1973) "PROCESS 1.5: Description and User's Guide", Bionics Research Reports No. 11, School of Artificial Intelligence, University of Edinburgh.

7. Halliday, M. A. K. (1961) "Categories of the Theory of Grammar", WORD, Vol. 17, No. 3.

8. Winograd, T. (1972) "Understanding Natural Language", Edinburgh University Press.

9. Sussman, G. J., Winograd, T. and Charniak, E. (1970) "Micro-Planner Reference Manual", AI Memo 203, Project MAC, M.I.T.

10. Winograd, T. (1969) "PROGRAMMAR: A Language for Writing Grammars", AI Memo 181, Project MAC, M.I.T.

11. Burstall, R. M., Collins, J. S. and Popplestone, R. J. (1971) "Programming in POP-2", Edinburgh University Press. See especially page 45.

12. Carroll, J. B. (1960) "Language Development" in Bar-Adon, A. and Leopold, W. F. (eds.) "Child Language: A Book of Readings", Prentice-Hall, Englewood Cliffs, N. J. (1971) (page 205).

DEFINING SOME PRIMITIVES FOR A COMPUTATIONAL MODEL OF VISUAL MOTION PERCEPTION

C Lamontagne
Bionics Research Laboratory, School of Artificial Intelligence,
University of Edinburgh.

Abstract. Primitive computational concepts, expressed in terms of neural nets, are created as a basis for a model of visual motion perception. These primitives are explicitly derived within the context of a complete visual system.

1. Introduction

Our current goal is to produce a working model of visual motion perception, and to use this specific concern as an alley into visual perception as a whole. The main idea is to approach the problem on very broad grounds, keeping our motion perception "sub-system" open in the context of a complete visual system.

We want our model to be a computational one, that is we want it to be expressed in terms of explicit computations which are detailed enough to be simulated on some computer or built into hardware. In building the model we want to use to the fullest possible extent the observations provided by Physiology and Psychology as basis for induction, and the general principles of computation offered by Computational Sciences as basis for deduction. We want our model's achievements to be at a human level of sophistication and its computational strategies to be as efficient as possible. It is important to realise that we do not claim that our model will necessarily be a model of human visual perception, not any more than we claim that it will necessarily be a model of a computationally optimum visual system, but we do claim that the model will be perfectly suitable, at any stage of its construction, as a hypothetical statement about how the human system could work or about how the optimum system could work. In fact the model has already shown its power as a source of hypotheses for Experimental Psychology by providing, at a very early stage of development, quite a strong theoretical framework for the prediction of a whole family of new visual phenomena which constitute a complete experimental paradigm for a part of the human visual system (1).

Now to represent computations, that is to talk about our model, we needed to choose a language. As primary language, i.e. the language which is used to express the model itself, we chose the language of neural nets; the reasons behind this choice are that first we consider neural nets as very suitable tools for "visual thinking" (i.e. they are easy to manipulate in one's head), and that secondly (and most importantly) we consider them as very suitable tools for talking about parallel processing as well as for talking about serial processing. As secondary language, i.e. the language of simulation for the neural nets, we chose POP-2 (for purely accidental reasons), but only a very small part of the model as it stands /

(1) See Lamontagne, C., 1973, "A new experimental paradigm for the investigation of the secondary system of human visual motion perception", Perception, 2, 167-180.

/stands at present has been simulated; most of it is still only expressed in terms of the primary language, that is neural nets. This brings us to talk briefly about the stage which our model has reached, and to outline which part of it will be described in this paper.

After two and a half years of work along the lines sketched in the above paragraphs we have got to the stage where the model can detect ten different types of motion, and track objects involved in translatory motion relative to its retina. All ten types of motion are essentially interpreted as two-dimensional ones but we are in the process of developing a learning scheme to raise the level of interpretation to three dimensions. The two restrictions imposed on the system from the very beginning of the research are still holding; the model has a single eye (i.e. we are developing a monocular system), and the eye's retina is homogeneous (i.e. there is no duality in its receptors' structure). A most important point is that as it stands now the model is highly homogeneous, being almost entirely built out of similar atomic processing structures, or primitives, combined and re-combined in all sorts of ways in order to reach the desired computational specificity; furthermore the planned extension of the model into the three-dimensionality involves using these same primitives as building blocks. The present paper will be exclusively devoted to the detailed description of these primitives, and this will be done in the explicit context of a complete visual system.

2. Preliminaries.

2.1 Input device: structurally detected features

We should start by choosing as our input device a single homogeneous two-dimensional array of receptors sensitive to different light intensities, where a signal fired by any receptor would qualitatively represent a specific position (by construction, or structure, of the retina) and would quantitatively represent a specific intensity. This input device would allow us to work directly on "real world" visual stimuli.

We will however adopt a slightly modified version of input device which is in every point similar to the one we just described but for the fact that it will not detect <u>different</u> intensities, being restricted to "all-or-none" responses to light intensities; moreover we will restrict the valid input stimuli to the class of bright line drawings on dark backgrounds. This modification is far from being as drastic as it might seem; we see it as equivalent to bringing the different intensities detected by the original input device down to 1's and 0's according to whether or not they reach a certain difference threshold when compared with their immediate neighbours. This computation would in fact bring out the contrasting elements in the picture, and this is exactly what we are doing by restricting the input stimuli to line drawings and bringing the intensity discrimination to an "all-or-none" mode. Since the problem of going from our simplified version of the input device to the originally desired one is well defined and since our simplified version is easier to handle in the process of building the visual system, we decided that we could wait until more important questions have been tackled before lifting the "line drawings" constraint on the input device.

So /

/So let us go ahead with the simplified version of the input device, calling "primitive array" or "retina" its array of receptors, and calling "primitive objects" the informational entities created by the specific retinal positions detected at this level. It should be stressed here that since our input device reacts in an all-or-none manner to light intensities falling on its different receptors (or positions) we are left with the very general piece of information "there is light", which is of little help when we are figuring out the physical constraints around us, and the more helpful although rather primitive information about where this light falls on our retina. The important point to notice here is that however primitive this basic information might seem, it is potentially very rich in the sense that the detected feature "retinal position" has a repertoire of values (i.e. "retinal position" is a multi-valued feature), each receptor on the retina representing one "value" of this feature. By combining these values in different ways we can get at new features with their own sets of values which can themselves be combined into still higher level features with their own sets of values, and so on. We therefore consider the main task of any visual system as being one of deriving features by grouping and re-grouping values of other features under some criterion or other, and consequently we consider the task of defining a visual system as being one of finding the adequate criteria under which the grouping should occur, i.e. under which new features should be derived.

2.2 What is meant by "motion": main types of derivable features

We can see two main types of strategy for deriving new features: analysing the values of a feature as they stand in one given moment, or as they stand in successive moments. On this basis we will make the distinction between two types of derivable features: frozen features which are derived from different values of some feature detected in one processing moment, and running features which are derived from different values of some feature detected in successive processing moments.

For instance let us consider the case where a straight line covering nine retinal receptors is used as stimulus and is actually projected on to the retina at moment 1. Then at this moment 1 we have nine receptors firing together, specifying nine different retinal positions. Since these retinal positions are directly provided (or detected) by the input device within but a single moment we say that retinal position is a frozen feature. Now we go on to say that any higher level feature derived from some or all of those retinal positions alone (i.e. the retinal positions worked out in a single moment) will itself be a frozen feature. For instance finding out that the "occupied" positions on the retina are adjacent, in a straight line, in a given orientation, and that there are nine of them creates as many new frozen features.

Now if on the other hand we concentrate on analysing values of features detected through successive moments we can derive a rather different type of feature. For instance let us consider the case where we have the same straight line as before (with orientation X, and size 9) projected on to the retina at moment 1 but where, at moment 0 (i.e. the moment just before moment 1), we had the line in a different orientation and with a different size (let us say orientation Y and size 6). We can then say a few more things about our line at moment 1, for instance we can say that (orientation) X and (size) 9 are values of features which the /

/the line possesses at moment 1 after not possessing them the moment before, and similarly we can say that (orientation) Y and (size) 6 are values of features which the line does not possess at moment 1 after possessing them the moment before. Furthermore we can go on deriving more features by relating the actual values which have undergone "death" or "birth" from moment 0 to moment 1, deriving new multi-valued features which can themselves be analysed through time. The main point here is that all these new features are essentially derived by comparing values of features as they "flow" through successive moments, and this is why we group all these features under the general label "running feature". We expect it to be clear by now that computing running features is what motion perception is all about, and that it is our criterion for defining the boundaries of motion detection as a specific ability within the context of a complete visual system.

3. Primitive and quasi-primitive running features

In the case of frozen features it is easy to grasp the fact that retinal position is a primitive feature in the sense that it is just about the most basic piece of information detected by the system, and that it serves as a basis for deriving all other frozen features detected by the system. The question which we are asking now is: can we find a primitive running feature which constitutes the basis for deriving any other running feature?

The most primitive running feature which we found seems to fit quite well the concept of a primitive feature, although it must be appreciated that a running feature, since it necessarily rests on the temporal analysis of the values of some other feature, cannot be considered as being "completely" primitive. Our primitive running feature however rests on a frozen feature which is even more primitive than retinal position, although any detected value of retinal position necessarily specifies it, and this frozen feature is the existence state of some value of some feature. This rather trivial feature (i.e. existence state) has two possible values: 1 or 0 (existing or not existing). Since the value of this feature is directly available at any moment from the signal that represents any value of any other feature, we did not bother to talk about its detection as a separate frozen feature; but now it turns out that considering existence states is necessary in order to compute the primitive running feature which we are looking for. This desired primitive running feature will in fact characterize the type of change of existence state for any value of any feature from moment to moment, and we will call it the "transistence state" ("transistence" meaning "existence through time") of the value considered.

In order to understand what all this means in concrete terms we must first realize that any "motion" involves some change in the values of some feature - e.g. a translation involves changes in values of the feature "position", a rotation involves changes in values of the feature "orientation", an expansion (or a contraction) involves changes in values of the feature "size", and an acceleration (or a deceleration) involves changes in the values of the feature "speed". It follows that in order to analyse any motion the lowest level essential task is to keep track of what happens to each possible value of the feature concerned so that at every moment we are aware of which values come to existence and which values lose it. This is where we need transistence states.

As we said before, the existence state of any value of some feature at any moment is either 1 or 0, and is directly available from the signal that represents the value itself. The transistence state of a value is then worked out by pairing the existence state of this value at any moment with its existence state the moment before; we therefore have four possible transistence states: 0-0 or the "still absent" state, 0-1 or the "on" state, 1-0 or the "off" state, and 1-1 or the "still present" state.

In the case of a single-valued feature (e.g. straightness of a line, concavity or convexity, connectedness, "squareness", etc...etc...), the single value's transistence state has a "global" significance which in fact makes motion impossible within the feature itself (i.e. "squareness" cannot possibly move); but in the case of a multi-valued feature (e.g. position) the transistence state of each value only has a "local" significance. This local character of transistence states in the context of multi-valued features opens up a door for further running (and frozen) feature computing. What we mean by "local character" of transistence states is that they only refer to particular values, and that they do not convey any information about what is happening "globally" (through time) in the "pool" of values which belong to the feature concerned. Such global events can only be grasped by grouping the different values in the "pool" under transistence states as criteria. This is in fact the way to get at motion itself, by comparing "off" values with "on" values ("off" and "on" being the criteria for comparing such and such values) and deriving from this comparison what we will call the two quasi-primitive running features: direction and speed (or type of change in value and rate of change in value). It is indeed the case that when some motion occurs different values of the feature involved succeed each other, that is one value goes "off" and another one goes "on" and direction and speed can only be derived by comparing the values which behave in this way.

For instance if we consider the feature "orientation" with values ranging from 1 to 180, a motion within this feature (i.e. a rotation) could be something like this: at moment 1 orientation 45 turns "off" and orientation 46 turns "on", at moment 2 orientation 46 turns "off" and orientation 47 turns "on", at moment 3 orientation 47 turns "off" and orientation 48 turns "on", etc...etc.. Computing motion in such a case consists in identifying which value goes "off" and which value goes "on" and in deriving from them the fact that nothing has "globally" disappeared or appeared but that "something" has moved clockwise at a rate of one unit of resolution per unit of time.

Now to combine the actual values going "off" and "on" in order to get at direction and speed one needs to consider quite closely the actual feature involved, because there is no reason to believe that the same number and the same type of possible directions will have to be dealt with whatever feature happens to be considered, no more than we have reason to believe that the requirements for working out the rate of change will be uniform for all. We will not discuss the details of this just now, but will restrict the present discussion to acknowledging the completely general principles of velocity detection and then concentrate on finding precise computational tools to suit them.

The general principles of velocity detection (or of quasi-primitive running features computation) are on the one hand the analysis of the type /

/type of difference (or "qualitative" difference) between "off" and "on" values of some given feature, and on the other hand the analysis of the amount of difference (or "quantitative" difference) between the same ("off" and "on") values. The former type of analysis yields direction and the latter type yields speed.

4. Computational concepts to deal with the primitive and the quasi-primitive running features

What we want to do in this section is describe effective decision procedures which will act as precise computational concepts to represent the deriving of the primitive and the quasi-primitive running features. We want these computational concepts to be simple enough to allow us to use them with complete control over their significance as we proceed from the embrionic state which our visual system is in at the moment up to the most sophisticated level which we wish the system to reach; and we want these concepts to be precise enough to allow us to build (explicitly in hardware or implicitly through simulation) actual systems which will carry out the type of computation which the concepts are meant to cover.

4.1 Computing the primitive running feature (or transistence state): CDU's

From what we have said in section 3, detecting the transistence state of some value of some feature involves a procedure which takes as input the existence state (either 1 or 0) of the value at some given moment, and, by pairing this existence state with the one detected the moment before, produces as output one of the four possible transistence states. An effective procedure which does just this is expressed by the network shown in Fig. 1.

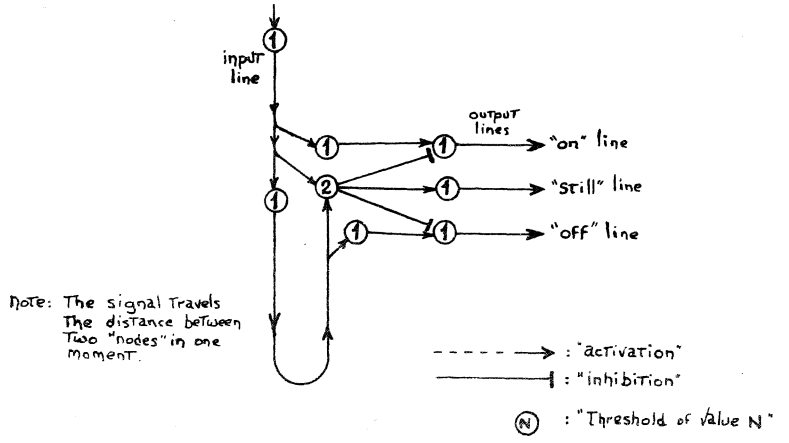


Figure 1

Fig. 2 shows a precise situation where the network actually computes transistence states: at moment 1 the value which this particular network happens to be set on is absent, i.e. its existence state is 0 (and there is no signal already running in the network); at moment 2 the value is present (it turns "on"); at moment 3 the value is present again (it remains "still"), and at moment 4 the value is absent (it turns "off").

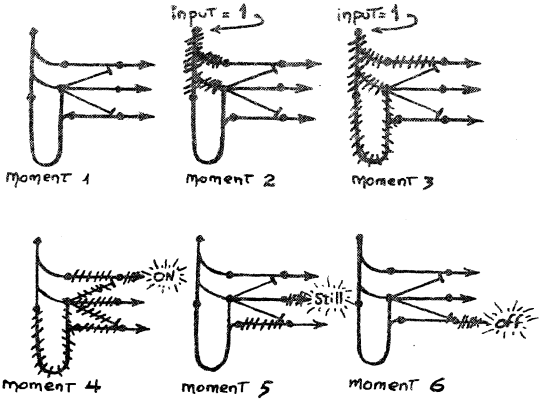


Figure 2

We can see from this example that each possible matching (of the input at one moment with the input at the moment before) is represented by a specific outcome in the network: 0-1 (the "on" state) is specified by a signal in the upper output line, 1-1 (the "still present" state) by a signal in the middle output line, 1-0 (the "off" state) by a signal in the lower output line, and 0-0 (the "still absent" state) by "none of these signals". Obviously one and only one of these possibilities is activated at any moment. The computation is achieved by using a delay loop to keep in the network the input received the moment before (memory requirement), and by a combination of activating and inhibiting signals controlled by thresholds at particular junction points to carry out the matching process and generate the specific output. This procedure is precise enough to be actualized in electronic hardware or simulated on a digital computer (using for instance straightforward Boolean functions) and is simple enough to be grasped in a single "glimpse" whenever needed. We will hereafter refer to it as the Change Detection Unit (CDU).

Before turning to the precise characterization of computing direction and speed, the quasi-primitive running features, we feel that we should make the following remarks concerning the CDU.

First we want to stress the fact that since the CDU is designed to compute the transistence state of particular values of a given feature, if we want to keep an eye on every possible value of the feature then we have the choice between considering one single CDU as a "sub-routine" which is called to compute the transistence state of each value as the system exhaustively /

/exhaustively goes from one to the next, or considering an individual CDU for each possible value, thereby making parallel processing possible. The simplicity of the CDU allowed us to choose the much more satisfying parallel setup, which means that if the feature considered has N possible different values there will exist N different CDU's, each specifically linked to one particular value.

Secondly we want to emphasise the general purpose character of the CDU. We tried to convey this characteristic by saying that the CDU could compute the transistence state of any detected value of any detected feature, including of course transistence states themselves as detected values of a detected feature. Right now it might be obvious that the nature of the feature whose values are analysed through time interferes in no way with the analysis as such; but later on, when we start talking mostly about particular cases, it might happen that the general purpose character of the CDU drowns in the specificity of the context, and this could create undesirable misunderstandings.

And thirdly we want to make it clear that we do not propose the CDU as an anatomical unit that will be found in actual nervous systems. The network which we are proposing is exclusively intended to be a conceptual tool to tackle the problem of motion perception. In other words any resemblance with any existing natural anatomical network is a pure coincidence.

4.2 Computing the quasi-primitive running features (or direction and speed): VDU's

We saw in section 3 that the detection of speed and direction of motion is achieved by comparing the actual values which go "off" and "on" from moment to moment. What we want to discuss in this section is a precise scheme to carry out explicitly this comparison process.

Since some transistence states (namely "off" and "on" states) are needed as criteria for choosing the relevant values for comparison, we clearly want to use the output from the CDU's as a starting point. Knowing which values ought to be compared we then want to carry out a comparison which will yield the type of difference (or direction of motion) and the amount of difference (or speed of motion) between the values. What we therefore propose is a network where "off" signals originating from our value-specific CDU's (remember that we decided to link a CDU to every value of each detected feature) will "travel" along lines projecting in all possible directions through every feature's pool of value-specific CDU's in search for "on" signals which will in fact be made to "cross" the"travelling off" lines at points which are specific to the respective values which they characterize; the "off" signals will keep track of the distance travelled by adding 1 to their quantitative content every time they meet an intersection with "on" lines where there is no "on" signal. So when an "off" signal meets an "on" signal at one of these intersections a velocity signal is triggered, the distance travelled by the "off" signal along the particular line specifying the speed of motion (i.e. the amount of difference between the "off" value and the "on" value), and the actual line which led the "off" signal to the "on" signal specifying the direction of motion (i.e. the type of difference between the "off" value and the "on" value).

Such /

/Such a scheme obviously requires a careful arrangement of the CDU's within every feature's pool of values, CDU's having to be set in a highly ordered way in order to allow the "off" signals to "spread out" in an adequate way. Let us then see in more concrete terms how all this is achieved. Fig. 3 shows what the network would look like for a single direction and from a single CDU's point of view (i.e. only one CDU's "off" signal can travel along the line in search for an "on").

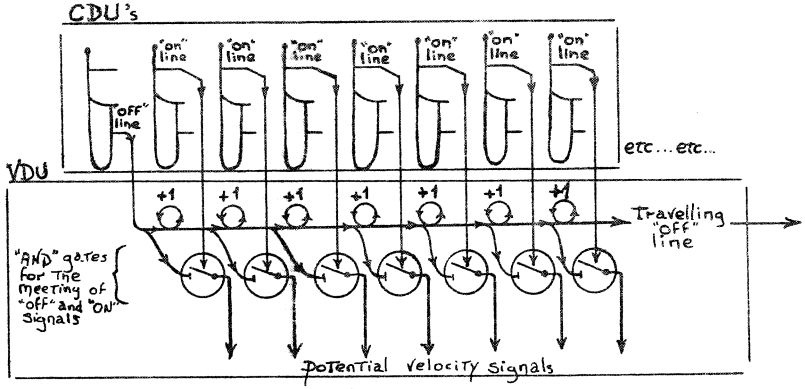


Figure 3

Fig. 4 shows an example of how this Velocity Detection Unit (VDU) would work for the case of an "off" signal computed by the first CDU and an "on" signal computed by the fifth CDU.

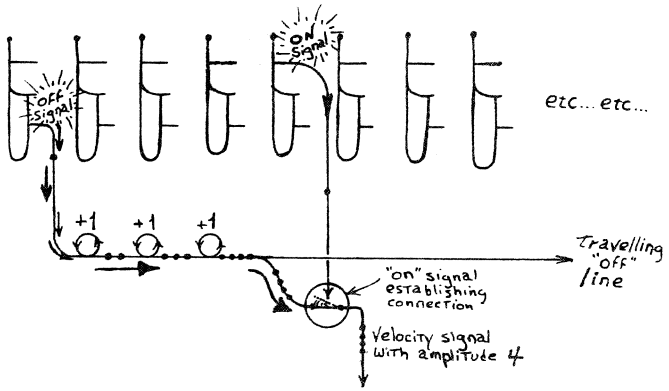


Figure 4

By choosing a particular feature, let us say "orientation", and following it through the network let us now try to clarify how this network leads the system to compute speed and direction efficiently and fast. In the case of orientation let us remind ourselves that we have a CDU to represent each particular orientation (i.e. each value of the feature orientation) which can be detected. All these CDU's are ordered in a single line, as in figures 3 and 4; the order according to which they are set should make the distance between any two of them correspond to the difference in amount between the values which they specify. In this way we make it possible to derive speed by computing the actual distance between the "off" value and the "on" value over a unit of time. Now as far as the direction of motion is concerned we have to realise first that in the case of orientation there are only two possible directions: clockwise and anti-clockwise (this is in fact the reason why we decided to put the CDU's along a single line); these two directions can be accounted for by having two "travelling off" lines linked to our line of CDU's, one going from left to right and the other one going from right to left. In fact what we would really need is a ring of CDU's linked to two circular "travelling off" lines, but there is no need to go into that for the moment since the general idea can be grasped quite adequately from considering a "straight line" network. Fig. 5 shows how the system would work for two CDU's only, an "off" signal being detected at 5^{o} and an "on" signal at 10^{o}.

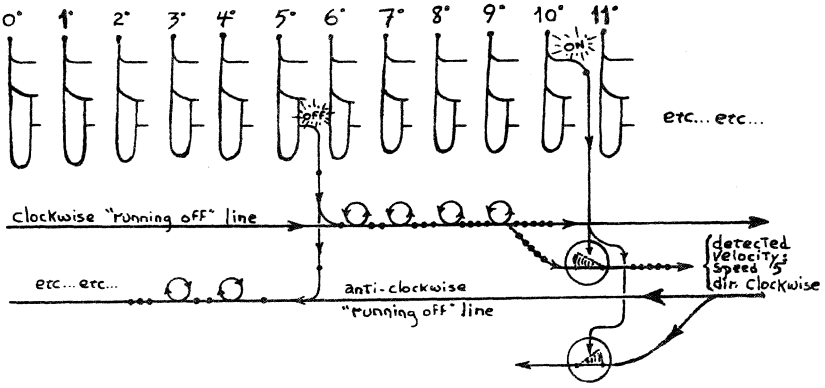


Figure 5

We will call VDU (Velocity Detention Unit) of a given feature the "travelling off" lines setup for this feature. We hope that it is now clear that VDU's can vary quite a lot from one multi-valued feature to another. For instance, since a much greater number of directions of motion have to be accounted for, the VDU required by the feature "position on the retina" will be very different from the one we just saw for orientation. Thinking about other possible multi-valued features like size or even speed and direction themselves should bring enough evidence to convince anyone of the need for different types of VDU's. However we/

/we want to stress that the general computing principles of a VDU as discussed in the above paragraphs remain completely general whatever feature they happen to be applied to. Another important point about VDU's is that their relative complexity does not allow them to process many velocities at the same time; a much too complex control system would be required to make this possible. In fact each VDU will be allowed to work on one velocity only at a time, but we will have many VDU's working in parallel, each one computing velocity for its particular multi-valued feature.

What comes out of all this is that every multi-valued feature for which we want to compute motion will have to be given its own set of CDU's (the number of CDU's in the set depending on the number of different values the particular feature allows) and its own VDU (the number and the lay-out of"travelling off" lines depending on the particular feature). In order to underline the unity inherent to this "pairing" of a set of CDU's with a particular VDU whenever we decide to compute motion for some multi-valued feature, we found a single label to cover it: Motion Detection Unit (MDU). An MDU therefore is this two-storey network (CDU's over VDU) which we stick under the set of values of each multi-valued feature for which we want to compute motion.

5. Conclusion: a glimpse at the rest of the story

Reaching the level of MDU's was the final step in defining primitives for motion perception. However we are very well aware of the fact that the power of any system rests as much on the way primitives are used as it rests on the primitives themselves. This is why we want to conclude this paper by at least hinting at how the simple primitives which we just described will be used to create a powerful visual system.

We said in section 2.1 that for us the task of defining a visual system is one of finding the adequate criteria for grouping different values of different features into new values of new features. Since these criteria are always themselves abstracted from values of features we can say that values can be used either as criterion for grouping or as element for grouping. In the context of a whole visual system this means for instance that frozen features can be derived by using some value(s) of a running feature as grouping criterion ("frozen" meaning only that the actual values grouped together are all detected within the same processing moment). For example, a set of positions could be analysed as a line using as criterion for choosing these particular positions the fact that they are all moving in an identical way. This gives an idea of how running features can get entangled in matters other than straightforward motion detection, and of how intertwined frozen and running features can become. But there is more to it; even when we stick to our standard MDU's we can get quite a lot done by applying them to the right multi-valued features (by the way we hope it is clear that MDU's can be, and will be, applied to running multi-valued features as well as to frozen ones). To realise this let us consider sticking an MDU to the set of values specified by our primitive frozen feature "retinal position"; this MDU would compute translation of "dots" relative to the retina, but would do it for one "dot" only at a time. If we want to see a line (i.e. a set of "dots") rotating we could then provide our system with as many MDU's as there are dots in the line, and relate their respective outputs in a way which is specific to rotations. But much more simply /

/simply we could go up one level by computing a single frozen feature, namely orientation, which could then be linked to a single MDU (the general purpose character of CDU's and VDU's making this perfectly legitimate) which would then compute rotation without problem. If one now tries to generalise this type of strategy to much more complex types of motions (going right up to three-dimensional motion) one can get a feel for what can be achieved by putting such simple structures as MDU's in the right places.

Finding the adequate features for running feature computing, and discussing their relevance as criteria for deriving other types of required features constitute our two main preoccupations for the last eighteen months or so, but we won't go any deeper into this for the moment, the scope of the present paper having already been outranged sufficiently.

AUTOMATIC GENERATION OF PROGRAMS CONTAINING CONDITIONAL STATEMENTS

by

David C. Luckham
Artificial Intelligence Laboratory
Stanford University

and

Jack R. Buchanan
Department of Computer Science
Carnegie-Mellon University

March 1974

ABSTRACT

An experimental system for automatically generating certain simple kinds of programs is described. The programs constructed are expressed in a subset of ALGOL containing assignments, function calls, conditional statements, while loops, and non-recursive procedure calls. The system has been used to generate programs for symbolic manipulation, robot control, every day planning, and computing arithmetical functions. This system has previously been described in [Buchanan and Luckham 1974]. The present report focuses on the generation of conditional statements and describes applications to mechanical assembly and symbolic manipulation problems.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contracts [DAHC15-73-C-0435] and [F44620-73-C-0074].

1. INTRODUCTION

A potentially useful area of application for automatic program generation is the class of problem domains in which the solutions usually have the form of programs or plans containing alternative paths for processing various cases but very little looping structure or recursion. Let us say that the main complexity in the planning is the contingency or conditional branch, although some loops may occur. Such problem domains include scheduling (travel itineries, office procedures) medical diagnosis, and machinery repair procedures. Certainly, the problem of automatically generating simple contingency plans correctly is important, and it is not an entirely straightforward business.

In this paper we describe some methods for constructing contingency programs that have been implemented in our system [Buchanan and Luckham 1974]. We give examples within some possible areas of application, of the sort of conditional branching procedures that are generated by the system. In addition, example 1 dealing with the generation of assembly and repair procedures for very simple machinery, seems to present a potentially practical context requiring much further research into such questions as differentiating the functions of various kinds of knowledge, and developing languages for describing those functions. The present system provides an experimental tool for such research.

The system requires as input a programming environment (called a FRAME) consisting mainly of primitive procedures and rules of composition (i.e., programming methods). A programming environment is defined using a declarative language, the FRAME language. The rules of inference, axioms and other logical facts expressed in this language are translated into a backtrack problem reduction system augmented by special search procedures. This system, which does most of the searching, recursively applies to a given goal the primitives and rules of the programming environment to generate subgoals whose solution will imply a solution to the goal. The basis of the Frame language is a free variable first order logic in which statements may have one of three truth values (TRUE, FALSE, or UNDETERMINED).

When the system has generated a program satisfying the given input-output conditions, the user may incrementally extend the program by asking for another program which takes the output conditions of the first program as its input conditions. He can choose to have the solution program optimized according to some simple criteria, or generalized and placed on a library of non-primitive procedures. If the program contains conditional branches calling other procedures, he can choose to have those secondary procedures constructed. Figure 1 shows the main components of the system and how they interact. The system is implemented in LISP using primitives and backtracking facilities of Micro-Planner [Hewitt 1971, Sussman and Winograd 1972].

The forms of the definitions of the elements of the programming enviornment, i.e. primitive procedures and rules of composition, correspond to axioms and rules of inference in a logic of programs currently used to define the semantics of the programming language PASCAL [Hoare 1969, Hoare and Wirth 1972; see also Igarashi, London, Luckham 1973]. The contents of these definitions vary with the actual environment. Problems to be solved are stated as pairs of conditions, the initial input

condition and the goal output condition which may be regarded as the input-output assertions of formulas in the logic of programs. The construction of a solution program may therefore be viewed as a search for a proof in the logic of programs that the generated program satisfies the given input-output assertions. Under certain sufficient conditions this approach enables us to prove that the system will construct correct programs.

In the remainder of this section the logical basis and the formalism used to describe the programming environment will be summarized.

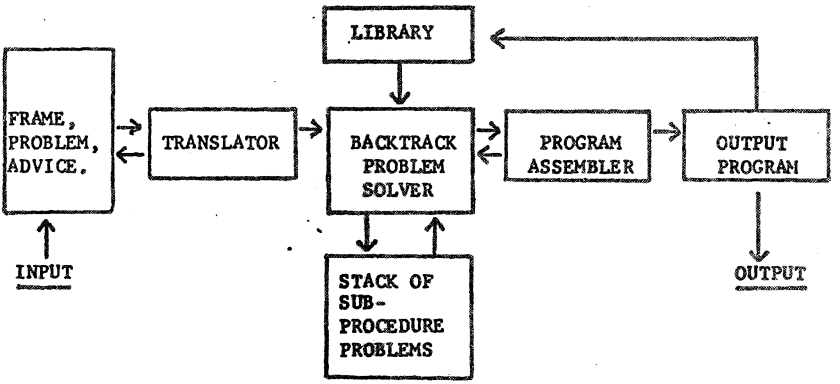


Figure 1. Main System Components

1.1 LOGIC OF PROGRAMS

We review briefly the elements of an inference system for proving properties of programs [Hoare 1969]. Further details may be found in [Igarashi, London, Luckham 1973].

NOTATION: x,y,z,u,v,w...variables,
f,g,h.... functions,
s,t... functional terms,

G,P,Q,R,S,... Boolean expressions (essentially formulas of first order logic with standard functions and predicates for equality, numbers, lists and other data types),

A,B,C,... programs and program parts in our Algol-like plan language,

p,q,.. procedure names,

$\alpha,\beta,\lambda,\ldots$ substitutions of terms for variables, also denoted by $(<x \leftarrow t>)$.

P(t) denotes the result of replacing x by t everywhere in P(x).

$\alpha\beta$ denotes the COMPOSITION of α and β; $E\alpha\beta = (E\alpha)\beta$ for all expressions E.

STATEMENTS of the logic are of three kinds.

(i) Boolean expressions, (henceforth often called ASSERTIONS)

(ii) statements of the form P{A}Q where P,Q are Boolean expressions and A is a program or program part.

P{A}Q means "if P is true of the input state and A halts (or halts normally in the case that A contains a GO TO to a label not in A) then Q is true of the output state".

(iii) Procedure declarations, p PROC K where p is a procedure name and K is a program (the body of p).

A RULE OF INFERENCE is a transformation rule from the conjunction of a set of statements (premisses, say $H_1,\ldots,H_n$) to a statement (conclusion, say K) of kind (ii).

Such rules are denoted by

$$\frac{H_1,\ldots,H_n}{K}$$

The concept of PROOF in the logic of programs is defined in the usual way as a sequence of statements that are either axioms or obtained from previous members of the sequence by a rule. A proof sequence is a proof of its end statement.

NOTATION: We use H ||- K to denote that K can be proved by assuming H. H |- K denotes the same thing for first order logic. Problems for the program generator to solve are denoted by P{?}Q. QuF⊃R denotes that R is a first order consequence of Q and the the axioms of F.

The logical rules used in the system are:

R1. Rule of Consequence: $\dfrac{P\supset Q, Q\{A\}R \quad P\{A\}Q, Q\supset R}{P\{A\}R \qquad P\{A\}R}$

R2. Rule of Concatenation: $\dfrac{P\{A\}Q, Q\{B\}R}{P\{A;B\}R}$

R3. Rule of Invariance: if P{A}Q and IuF⊃P then I{A}I'
where $I'=Q\wedge\{R:R\in I\wedge\neg(QuF\supset\neg R)\}$

R4. Change of Variables: $\dfrac{P(x)\{A(x)\}Q(x)}{P(y)\{A(y)\}Q(Y)}$

R5. Conditional: $\dfrac{P\wedge Q\{A\}R,\ P\wedge\neg Q\{B\}R}{P\{\text{IF Q THEN A ELSE B}\}R}$

R6. Undetermined Values: if I'{?}G cannot be solved and ¬(I'uF⊃¬G) then G is UNDETERMINED in I'.

R7. Primitive Procedures: The rule defining p is an axiom of the form P{p}Q.

R8. Iterative Rules: An iterative rule definition containing the Boolean expressions P(basis),Q(loop invariant),R(iteration step goal),L(control test) and G(rule goal) is a rule of inference of form:

$$\frac{P\ |\!|\!-\ Q, Q\wedge L\{?\}R,\ R\{??\}Q\vee\neg L}{P\{\text{WHILE L DO ?;??}\}G}$$

where the free variables of R occur in Q and ?? is restricted to be a sequence of assignment statements.

R9. Definitions: A definition of G in terms of P is a logical equivalence |-P≡G.

R10. Axioms: A axiom P is a logical axiom |-P.

In the definition of a Frame F provided by the user, instances of rules R7-R10 may be given whereas rules R1-R6 are built into the program construction system. A problem is represented as the formula I{?}G, where I is an input assertion or initial state and G is an output assertion or goal and the objective is to generate a solution program for ? that transforms I into G using the rules of F.

The above summary does not include system rules for conditional assignments used in constructing loops, nor the strongest form of the rule of invariance [Buchanan and Luckham 1974].

1.2 FRAME LANGUAGE

The Frame language consists of the following elements:

1.2.1 ASSERTIONS: Boolean expressions are used as conditions in rules, axioms and problem representations.

1.2.2 INPUT CONDITIONS: In specifying a problem I{?}G, the input condition (initial state) I is given by a conjunction of literals.

1.2.3 AXIOMS: Axioms are stated in either of the forms P⊃Q or P, where P and Q are assertions. They hold in all states and are used to complete a given state description by deduction of other elements of a state from those given.

1.2.4 RULES: There are three types of rules: primitive procedures, definitions, and iterative rules.

(a) A primitive procedure is specified by a name, an argument list, and its pre and post - conditions, i.e.

$$P\ \{f(x_1,\ldots,x_k)\}Q$$ where P and Q are assertions in

which $x_1,\ldots,x_k$ are free, and f is the procedure name.

The variables are formal parameters of the procedure. They may be "bound" by substitution of actual parameters when the procedure is applied to a state.

When a primitive procedure is defined it may be declared to be an ASSUMPTION. If it is used in a successful program construction, then the user is informed and is given the opportunity to carry out a structured program development of this non-primitive operation. This is described in [Buchanan and Luckham 1974].

(b) A definitional rule is of the form R≡S where R and S are assertions. The relation, S, is given as the post-condition of the rule. The meaning of a definition is that whenever it is desired that S be true it is equivalent to establish the truth of R. A definition is often used to shorten assertions in rules by defining a single relation as equivalent to an often used condition.

(c) Iterative rules specify conditions that if satisfied justify the assembly of a "while" loop to achieve the associated goal. They are instances of the iterative rule R8, and are defined by giving:

(i) A name, e.g. TLOOP, (without parameters).
(ii) A basis condition P.
(iii) A loop invariant condition Q that specifies relations that must be true in the state prior to each iteration.
(iv) An iteration step condition R that specifies the goals to be achieved during an execution of the loop body.
(v) An iterative goal G, the condition considered achievable by the iterative process.
(vi) A loop control test and an output assertion may be specifed.

1.2.5 SPECIAL AXIOMS: After the rules and initial state have been defined the system requests the following information for each predicate symbol P that has been mentioned.

a) "Is P a function of the state?" The intent of this classification is to separate those relations whose truth value may be affected by a state transformation, i.e., a FLUENT relation, from those whose truth value is constant over all achievable worlds, i.e., NON-FLUENT relations such as "ROBOT(X)", "INTEGER(Y)".

b) "Is knowledge represented using P partial?" A partial relation may have truth values TRUE, FALSE, or UNDETERMINED. Partial relations may be used to represent incomplete knowledge of the world which may cause conditional statements to be generated as explained in Section 2. A relation may be declared UNCERTAIN which implies an absence of knowledge about it so that it is assigned a truth value of undetermined a priori. If P is not PARTIAL it is TOTAL and can only have truth values of either true or false. Thus rule R6 applies to partial predicates only.

c) "Does P have a uniqueness property in certain argument positions?" A "yes" answer indicates that P cannot be true for two sequences of argument values that differ only at one of those positions that are unique. The unique positions are given using the notation, (X1,*,X3,*,...,Xn), for example, to designate the second and fourth argument positions. For each unique argument position in relation P(a1,...,an), an axiom is "built-in" from which a contradiction may be established with P(b1,...,bn) that differs in a unique position and matches elsewhere. For example the statement, "an object can only be in one place at one time",is expressed by, AT(X1,*). If we add, "and only one object can be at any place", then we use AT(*,*).

Conditional statements are generated when the problem solver encounters states in which it cannot determine the truth value of its current subgoal. This can happen either in situations where the rule of undetermined values applies or when the outcome of a primitive procedure is uncertain. In the next sections the system methods for constructing conditionals will be described, examples given and program correctness considered.

2. CONDITIONAL STATEMENTS

As previously mentioned, relations involving partial predicates may have truth values of TRUE, FALSE, or UNDETERMINED, whereas all other relations must be either TRUE or FALSE. During program generation, knowledge may be incomplete, for example properties of the value currently held by a program variable, e.g. C(X,Y)∧ZEROP(Y), may not be known, or it probably would not be known whether or not a traffic light would be green when a robot vehicle approached an intersection while following a generated procedure. However it is assumed that when the program is executed there will be a recursive test yielding TRUE or FALSE for all conditions.

2.1 UNDETERMINED VALUES. During the generation of a program, uncertainty may arise when a pre-condition for the application of a rule is UNDETERMINED with respect to the current state. The implementation of the rule R6 is described by the following definitions.

DEFINITION. A literal l is UNDETERMINED in a state S if the following conditions hold:
(i) pred(l) is partial,
and (ii) l cannot be achieved from S,
and (iii) ¬l cannot be proved in S.

Condition (ii) means that l is not true in S nor can S be transformed into a state in which l is true. If condition (ii) is true and ¬l is true in S then l must retain a truth value of FALSE and the precondition subgoal l must fail. Failure to find ¬l in S establishes a truth value of UNDETERMINED for l with respect to S. This definition applies to fluent and nonfluent literals but since the truth value of a "nonfluent" cannot be changed by a state transformation, for them, it is sufficient to use only the logical axioms in deciding condition (ii).

For the more general case in which the uncertainty may be a disjunction of literals we have the definition,

DEFINITION A disjunction of literals $\{l_i\}_{i=1}^{n}$ is UNDETERMINED in a state S if at least one literal is UNDETERMINED and no literal can be achieved from S.

2.2 CONDITIONAL STATEMENTS: When a pre-condition P is UNDETERMINED in a state S, a conditional branch is inserted in the solution program. If P is a single literal l, then program generation may continue either along the path in which l is assumed to be TRUE and in which future goals are attempted with respect to state S ∧ {l}, or along the path in which ¬l is assumed to be TRUE using state S ∧{¬l}. The system convention has been to generate a call to a yet ungenerated procedure for the latter case. The tasks of generating such contingency programs are placed in a stack for later attention. The structure and use of this stack is described in Section 2.5. Program generation continues, by convention, along the path using state S ∧{l}. This path is referred to as the "trunk" program of the tree of contingency programs generated while attempting to achieve the main goal.

The path selection at present is rather ad hoc since no assignments of probability are

made at the points of uncertainty and no path is considered more likely to be successful in general.

If an undetermined disjunctive precondition $\{l_i\}_{i=1}^{n}$ occurs in which literals $\{l_i\}_{i=1}^{m} m \leq n$, are UNDETERMINED in S, then a nested conditional statement of the following form will be generated:

```
if ¬l1    then
  if ¬l2    then
    .
     .
      if ¬lm    then pm
      else pm-1
     .
    .
   else p1
else p0
```

where each p_i is a call to a program to achieve a selected goal G from state $S_i = S \wedge \{l_i : i=j+1 \ \& \ i<m\} \wedge \{\neg l_i : 1 \leq i \leq j\}$ and p_0 is the trunk program segment which satisfies $S \wedge l_1 \{p_0\} G$ and forms the else-statement in the main-clause of the conditional. Each member of the set of triples $\{(p_i, S_i, G): 1 \leq j \leq m\}$ is placed in the stack of contingencies and program generation continues for p_0 The assumed literal, l_1, is removed from the state following the generation of the ELSE clause in the trunk program. This is the point in the trunk program where the contingency programs rejoin and the assumption l is not valid for all computation paths leading there.

2.3 SELECTION OF CONTINGENCY GOAL: The goal G to be achieved by the contingency programs is selected from the set of goals in the subgoal tree that are global to the undetermined precondition. Let us refer to the set of goals which are below G in the subgoal tree, as the SCOPE of G.

The particular G chosen and its associated scope affect the length of p_0 , duplication among contingency programs, degree of difficulty in generating contingency programs and validity of their use. If the structure of the trunk program is to remain fixed during contingency program generation then the choice of G cannot be deferred. The block structure of our program language imposes the restriction that for any conditionals in p_0 , a contingency goal G' must not have a greater scope than G. There is also the problem that if G is not fully instantiated then inconsistent instantiations may occur in different contingency programs which must validly rejoin the main program following the ELSE clause. The present system selects the least global fully instantiated goal thereby satisfying the block nesting constraint and minimizing the scope while avoiding the problem of handling deferred instantiation. This selection process is always effective in the present system since the top level goal is fully instantiated (i.e. all of its variables occur in the initial state).

2.4 REJOIN CONDITIONS When a contingency program is generated its output state must satisfy certain conditions, hereafter called the rejoin condition, for return of control to the trunk program to be correct. Consider the case of an undetermined goal L in state S and a contingency goal G in Figure 8 . Let A and B be program segments that satisfy $S \wedge L\{A\}G$ and $S \wedge \neg L\{B\}G$ and let C be the rest of the trunk program.

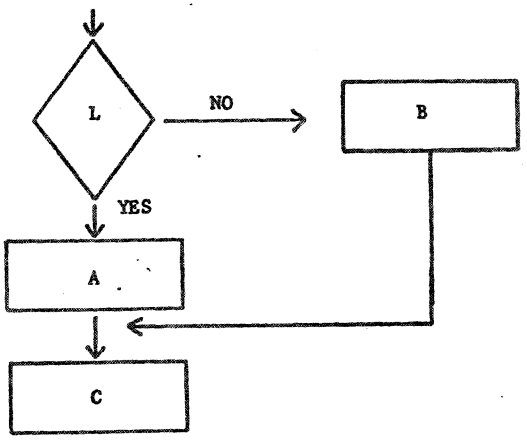


Figure 2

Let R be the total output state of B, i.e. $S \wedge \neg L \{ B \} R$,where $R \supset G$. Let Q be a sufficient input assertion computed for C. Then the REJOIN CONDITIONS for p(B) is, $R \supset Q$. A contingency program is said to have SIDE EFFECTS when its execution results in state changes in addition to the achievement of G. The difficulty in satisfying a rejoin condition occurs when B has had side effects resulting in an output state from which Q cannot be proven. The implication for program correctness of satisfying rejoin conditions is obvious.

2.5 SUBPROBLEM STACK OF CONTINGENCIES The task of generating a contingency procedure is specified by the quadruple:

(<procname> <state> <goal> <rejoincond>)

where,

<procname> is the name of the yet ungenerated procedure that must satisfy <state>{<procname>}<goal> $\wedge$ <rejoincond>.

At the point in the planning when the uncertainty is encountered, the first three elements of the quadruple are placed in a stack as explained previously. The rejoin conditions are not known at this time since it involves the input assertion for the trunk procedure segment following the point where control returns from the contingency plan to the trunk plan. After this segment is generated, the rejoin condition is computed and stored as the fourth element of the quadruple.

When planning has been completed for some trunk procedure, if the contingency stack is not empty then contingency planning may be done by removing a quadruple from the list and posing this as a program generation task. The state of the system is initialized to the specified contingency state and the subgoaling system is given <goal> as its main goal. If it is successful in achieving a state in which the main goal is true then a test is made to see if the rejoin condition is true in that state. If it is then the procedure declaration is adjoined to its trunk program. If the condition is false then the system allows the user two alternatives, i.e.

(i) Mark the call to the program as an error exit in the trunk program, or (ii) "Fit" the program to the trunk program by posing currently untrue rejoin conditions as goals and constructing a new program segment that achieves them and appending it to the end of the contingency program.

This process of generating a trunk procedure which may create new contingency tasks then generating contingency procedures as directed by the user may continue until all contingencies have been processed and the stack is exhausted.

2.6 COMPUTATION OF INPUT/OUTPUT ASSERTIONS. In Section 1 primitive procedures were viewed as Frame rules of the form P{p}Q, where P and Q are the pre and postconditions for p. The conditions P and Q may also be viewed as the minimal input and output assertions for p , that must be satisfied by the actual parameters of p.

For any generated program segment A, the input assertion I_a is computed as the conjunction of all literals, l, from a state that were used in achieving subgoals encountered during the generation of A and did not occur in that state as a result of a postcondition of a procedure whose generation in A preceded the addition of l to I_a . The output assertion O_a is the conjunction of literals added to a state during the generation of A that are true in the final state.

The computation of input/output assertions for programs consisting of compositions of primitive procedures is straightforward as described above, however the uncertainty as to which path computation will follow in a program containing conditional statements complicates these assertions. The input/output assertions in this case must be computed incrementally as each contingency program is generated.

In the conditional statement shown in Figure 2, suppose we know the minimal input and output assertions for A and B, say P{A}Q and R{B}S. then the input and output assertions for the conditional statement are

$$(L \wedge P) \vee (\neg L \wedge R)\{\text{if } L \text{ then } A \text{ else } B\}Q \vee S.$$

To reduce computation, We use the simpler sufficient conditions, $P \wedge R$, for input assertions.

The conditional statement may be correctly executed in any state in which $P \wedge R$ is true. There doesn't appear to be a simplifying approximation for output assertions unless on the assumption of no side effects in the contingency program B, i.e. $Q \supset S$, we take Q as the output assertion.

It can be shown by induction that if the computation of input/output assertions is correct for atomic program constructs, i.e. primitive procedures and while statements then using the the composition rule, the computation of input/output assertions for generated programs is correct.

2.7 UNCERTAIN PRIMITIVE PROCEDURES A primitive procedure q defined by P{q}Q has an uncertain outcome if Q is a disjunction. In the present system, disjunctive post-conditions use the exclusive OR connective, "⊕". This allows us to define frame procedures that have an intended result but may be unreliable. It is assumed that exactly one of the possible outcomes will be true in the output state and that none of them are true in the input state. At the point where an uncertain operator is applied, the problem solver has no knowledge of what the outcome will be and a conditional statement must be generated. Let Q be the disjunction of literals $\{l_i\}_{i=1}^{n}$. The first outcome l_1 is considered to be the normal result of executing q. Following the inclusion of q in the program in state S, a conditional statement of the following form is generated.

if $\neg l_1 \wedge l_2 \wedge \neg l_3 \wedge \ldots \wedge \neg l_n$ then p_2

else if $\neg l_1 \wedge \neg l_2 \wedge l_3 \wedge \neg l_4 \wedge \ldots \wedge \neg l_n$ then p_3

.

.

.

else if $\neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_{n-1} \wedge l_n$ then p_n

else p_{n-1}

where each p_j, $2 \leq j \leq n$, is a call to a program to achieve l_1 from state $S_j = S \cup \{l_j\} \cup \{\neg l_i : i \neq j \ \& \ 1 \leq i \leq n\}$.

The contingency states will correspond to the n ways of assigning exactly one literal true and the remaining literals false.

2.8 CORRECTNESS Conditional statements will be correctly generated if the system methods are an accurate implementation of the conditional rule, R5, presented in Section 1. Referring to Figure 2 in section 2.4, if we let S be the output state of C then by construction and by verifying the rejoin conditions we have,

(1) $I \wedge L\{A\}G \wedge Q$,
(2) $I \wedge \neg L\{B\}G \wedge R$,
(3) $Q\{C\}S$,
(4) $\vdash R \supset Q$, (rejoin condition verification)

and the correctness proof may then be completed as follows,

(5) $I \wedge \neg L\{B\}G \wedge Q$, (2,4,Consequence Rule)
(6) $I\{\text{if } L \text{ then } A \text{ else } B\}G \wedge Q$, (1,5,Conditional Rule)
(7) $I\{\text{if } L \text{ then } A \text{ else } B;C\}S$, (3,6,Composition Rule).

It should be noted, however, that if conditional statements occur in B then R may only

be an approximation of the true output state resulting from executing B as discussed in Section 2.6. Similarly Q may be only an approximation of the true input assertion for the remainder of the program. In these cases an incorrect program may result.

3. EXAMPLES

3.1 Assembly and Repair of a Model T Ford Water Pump.

The problem is to make a water pump given the various parts placed at locations on a pallet. This task is actually accomplished by a mechanical hand controlled by programs written in a specially developed Hand Language [R. Bolles and R. Paul, 1973]. There are three major parts, a casing (or pump base), a gasket, and a top assembly, and these must be fastened together by screws. The pallet may contain more than the minimum quantity of parts. The frame consists of simple idealized Hand Language operations and definitions of concepts dealing with the assembly world of the mechanical hand, such as ALIGN, ASSEMBLY, POSITION, and FASTEN. There is a specific order in which most of the building operations must take place; in particular, the problem of lining up holes in the pump casing with holes in the gasket and top requires the use of auxiliary tools called PINS. Pins must be placed in holes in the casing, and other parts slipped over the pins, and then some free holes must be fastened (to prevent slipping and misalignment) before the pins are finally removed . This assembly order can be represented graphically, but is in fact encoded by the way definitions are built up from other definitions in the frame. The reader will see the sequence in PROC1 below.

The frame also contains a simple scheme of definitions dealing with diagnosing faults and repairing them. At the top level, the concept DIAGNS is defined simply as an OR of possible faults. If a new fault is discovered it can be added (by extending the disjunction for DIAGNS). Each fault is defined as an OR of pairs of the form CAUSEn∧FIXn, where CAUSEn is the nth possible cause of the fault and FIXn is the definition of what must be done to fix that cause. As more causes or repair procedures are discovered they may be added. So the diagnostic definitions are easily extended to encompass new situations. A repair procedure for CAUSEn is the positive branch on the test "is CAUSEn true" of the complete program to achieve the goal "diagnose the fault". It will be generated as the nth contingency plan, PROCN; the user may choose to have it generated before any other sub-procedures, if for example he believes CAUSEn to be the problem. The generation of repair procedures involves repeatedly dismantling the pump and rebuilding it, and is a good test of the updating algorithms of the system (implementation of R3).

Definitions of concepts such as ASSEMBLY, ALIGNMENT, FAULT, CAUSE, REMEDY in this example are, to say the least, unsatisfactory. Intuitively, these are "general" concepts, where we might, with a little good will, interpret "general" to mean that more accurate definitions of these concepts ought to be part of FRAMES for assembling a wide variety of different machinery. In other words, we should be able to put our words into other worlds! The definitions given here are clearly not general enough. This is not a fault of the system but of our lack of analysis of the concepts. The definitions here are directly functional in the sense that the bodies of the definitions state exactly what to do with the parts of the water pump instead of how to reason or deduce what to do. The example is in the nature of a feasibility study.

RELATIONS USED IN THE FRAME DEFINITION:

RELATION	INTERPRETATION	FLUENT	PARTIAL	UNIQUENESS
AT(X,Y)	"X is at Y"	TRUE	FALSE	AT(X,*)
=(X,Y)	"X is equal to Y"	FALSE	FALSE	FALSE
ISPIN(X)	"X is a pin"	FALSE	FALSE	FALSE
ISHOLE(X)	"X is a hole"	FALSE	FALSE	FALSE
IN(X,Y)	"X is in Y"	TRUE	FALSE	IN(X,*)
ISFDR(X)	"X is the feeder"	FALSE	FALSE	FALSE
ISGASK(X)	"X is a gasket"	TRUE	FALSE	FALSE
ISCASE(X)	"X is a casing"	FALSE	FALSE	FALSE
ALIGN(X,Y)	"X is aligned with Y"	TRUE	FALSE	FALSE
ISCREW(X)	"X is a screw"	FALSE	FALSE	FALSE
FASTND(X,Y,Z)	"X,Y,Z are rigidly fastened"	TRUE	FALSE	FALSE
ISTOP(X)	"X is a top unit"	FALSE	FALSE	FALSE
EMPTY(X)	"X is empty"	TRUE	FALSE	FALSE
POSITN(X,Y,Z)	"X,Y,Z are correctly positioned"	TRUE	FALSE	FALSE
PINNED(X,Y,Z)	"X,Y,Z are pinned together"	TRUE	FALSE	FALSE
UNPNNED(X,Y,Z)	"X,Y,Z are unpinned"	TRUE	FALSE	FALSE
MAKE(X)	"X is to be made"	TRUE	FALSE	FALSE
ISLOC(X)	"X is a location on the pallet"	FALSE	FALSE	FALSE
ISPUMP(X)	"X is a pump"	FALSE	FALSE	FALSE
ASSMBL(X,Y,Z)	"X,Y,Z are assembled"	TRUE	FALSE	FALSE
UNDUN(X)	"X is not rigidly fastened"	TRUE	FALSE	FALSE
DSMNTL(X)	"X is disassembled"	TRUE	FALSE	FALSE
LOOSE(X)	"X is loose"	TRUE	TRUE	FALSE
FAULT1(X)	"X is a fault of type 1"	TRUE	FALSE	FALSE
FIX1(X)	"X is a remedy for fault1"	TRUE	FALSE	FALSE
FAULT2(X)	"X is a fault of type 2"	TRUE	FALSE	FALSE
FIX2(X)	"X is a remedy for fault2"	TRUE	FALSE	FALSE
BROKEN(X)	"X is a broken part"	TRUE	TRUE	FALSE
RJCT(X)	"X is rejected"	TRUE	FALSE	FALSE
ISNEW(X)	"X is a new part"	FALSE	FALSE	FALSE
ISLEAK(X)	"X is a leak"	TRUE	FALSE	FALSE
DILEAK(X)	"X is a diagnostic for leaks"	TRUE	FALSE	FALSE
DINFLW(X)	"X is a diagnostic for bad pressure"	TRUE	FALSE	FALSE
DIAGNS(X)	"X is a fault diagnosis"	TRUE	FALSE	FALSE

****

PRIMITIVE PROCEDURE	PRE-CONDITIONS	POST-CONDITIONS
move(X,Y) "move X to Y"	ISCASE(X) ∧ AT(X,Z)	AT(X,Y)
pin(X,Y) "put pin X in hole Y"	ISPIN(X) ∧ ISHOLE(Y) ∧ EMPTY(Y) ∧ IN(X,PNPLCE)	IN(X,Y) ∧ ¬EMPTY(Y)
putgsk(X,Y) "put gasket X on casing Y"	ISGASK(X) ∧ ISCASE(Y) ∧ AT(X,Z)	ALIGN(X,Y) ∧ ¬AT(X,Z)
putop(X,Y) "align top assembly"	ISTOP(X) ∧ ISGASK(Y) ∧ ISCASE(Z) ∧ ALIGN(Y,Z) ∧ AT(X,V)	ALIGN(X,Z) ∧ ¬AT(X,V)
screwd(X,Y) "put screw X into hole Y"	ISFDR(W) ∧ ISCREW(X) ∧ ISHOLE(Y) ∧ EMPTY(Y) ∧ AT(X,W)	IN(X,Y) ∧ ¬EMPTY(Y) ∧ ¬AT(X,W)
unpin(X,Y) "remove pin X from hole Y"	ISPIN(X) ∧ ISPPLC(V) ∧ REQUEST(IN(X,Y))	EMPTY(Y) ∧ IN(X,V)
unscrew(X,Y) "remove screw X from hole Y"	ISCREW(X) ∧ ISHOLE(Y) ∧ REQUEST(IN(X,Y)) ∧ ISFDR(V)	EMPTY(Y) ∧ AT(X,V) ∧ ¬IN(X,Y)
reject(X) "reject gasket X"	ISGASKET(X)	RJCT(X) ∧ ¬ISGASK(X)
remove(X,Y,Z) "disassemble top of gasket"	(ISTOP(X) ∨ ISGASK(X)) ∧ REQUEST(ALIGN(X,Y)) ∧ ISLOC(Z)	AT(X,Z) ∧ ¬ALIGN(X,Y)

DEFINITIONS:

BODY OF DEFINITION	RELATION DEFINED
PINNED(T1,G1,C1)∧ALIGN(G1,C1)∧ALIGN(T1,C1)	POSITN(T1,G1,C1)
ISHOLE(H1)∧ISHOLE(H2)∧¬=(H1,H2)∧ISPIN(X1)∧ ¬=(X1,X2)∧ISPIN(X2)∧IN(X2,H2)	PINNED(T1,G1,C1)
ISHOLE(H1)∧ISHOLE(H2)∧ISHOLE(H3)∧¬=(H1,H2,)∧ ¬=(H2,H3)∧¬=(H3,H1)∧ ISPIN(P1)∧ISPIN(P2)∧REQUEST(IN(P1,H2))∧ REQUEST(IN(P2,H3))∧ ISCREW(S1)∧IN(S1,H1)∧EMPTY(H2)∧EMPTY(H3)	UNPNND(T2,G2,C2)

ISHOLE(H1)∧ISHOLE(H2)∧ISHOLE(H3)∧¬=(H1,H2)∧ ¬=(H2,H3)∧¬=(H3,H1)∧ ISCREW(P1)∧ISCREW(P2)∧¬=(P1,P2)∧ISCREW(P3)∧ ¬=(P1,P3)∧¬=(P2,P3)∧IN(P1,H1)∧ IN(P2,H2)∧IN(P3,H3)	FASTND(T1,G1,C1)
ISCASE(C3)∧ISLOC(LOC)∧AT(C3,LOC)∧ISGASK(G3)∧ ISTOP(T3)∧ASSMBL(T3,G3,C3)	MAKE(PUMP)
ISTOP(T2)∧ISGASK(G2)∧ISCASE(C2)∧ POSITN(T2,G2,C2)∧UNPNND(T2,G2,C2)∧ FASTND(T2,G2,C2)	ASSMBL(T2,G2,C2)
ISHOLE(H1)∧ISHOLE(H2)∧ISHOLE(H3)∧¬=(H1,H2)∧ ¬=(H2,H3)∧¬=(H3,H1)∧EMPTY(H1)∧EMPTY(H2)∧ EMPTY(H3)	UNDUN(P)
ISPUMP(PUMP)∧ISGASK(G1)∧ISTOP(T1)∧ISCASE(C1)∧ REQUEST(ALIGN(G1,C1))∧REQUEST(ALIGN(T1,C1))∧ UNDUN(PUMP)∧AT(C1,P3)	DSMNTL(PUMP)
DILEAK(Z1)∨DINFLW(W1)	DIAGNS(X1)
(FAULT1(Y1)∧FIX1(Y1))∨(FAULT2(Y1)∧FIX2(Y1))	DILEAK(Y1)
ISPUMP(P1)∧LOOSE(P1)∧ISLEAK(Y1)	FAULT1(Y1)
ISTOP(T1)∧ISGASK(G1)∧ISCASE(C1)∧ISPUMP(P1)∧ UNDUN(P1)∧FASTND(T1,G1,C1))	FIX1(Y1)
REQUEST(ALIGN(G1,C1))∧BROKEN(G1)∧ISLEAK(Y1)	FAULT2(Y1)
ISPUMP(P1)∧REQUEST(ALIGN(G1,C1))∧DSMNTL(P1)∧ ISGASK(G2)∧¬=(G1,G2)∧RJCT(G1)∧MAKE(P1)	FIX2(Y1)

****

INITIAL STATE:

ISGASK(GSKT1)∧AT(GSKT1,P1)∧ISLEAK(LEAK)∧ISNEW(NEWGSK)∧ISLOC(P1)∧
ISLOC(P2)∧ISCASE(CASE)∧ISGASK(GSKT2)∧ISTOP(TOP)∧ISLOC(LOC)∧
ISHOLE(HOLE1)∧ISHOLE(HOLE2)∧ISHOLE(HOLE3)∧AT(CASE,P1)∧AT(GSKT2,P2)∧
ISPPLC(PNPLCE)∧AT(TOP,P3)∧ISPIN(PIN1)∧ISPIN(PIN2)∧IN(PIN1,PNPLCE)∧
IN(PIN2,PNPLCE)∧ISCREW(SCREW1)∧ISCREW(SCREW2)∧ISCREW(SCREW3)∧
AT(SCREW1,FEEDER)∧AT(SCREW2,FEEDER)∧AT(SCREW3,FEEDER)∧EMPTY(HOLE1)∧
EMPTY(HOLE2)∧EMPTY(HOLE3)∧ ISFDR(FEEDER)∧ISPUMP(PUMP)

THE_GOAL_ (MAKE PUMP)_IS_ATTAINABLE_BY_THE_FOLLOWING_PROGRAM:

```
PROC1 (PUMP)
   BEGIN
  MOVE(CASE LOC);
  PIN(PIN2 HOLE3);
  PIN(PIN1 HOLE2);
  PUTGSK(GSKT2 CASE);
  PUTTOP(TOP GSKT2);
  SCREWD(SCREW3 HOLE1);
  UNPIN(PIN2 HOLE3);
  UNPIN(PIN1 HOLE2);
  SCREWD(SCREW2 HOLE3);
  SCREWD(SCREW1 HOLE2);
  END
```

REMARKS:

The order in which some operations are done is crucial; the order structure is encoded by layered definitions. Thus it doesn't matter which pins go in which holes, but it is crucial that one hole is fastened before the pins are removed.

56 ___RULES_ENTERED
21 ___RULES_SUCCESSFUL

THE_GOAL_ (DIAGNS LEAK)_IS_ATTAINABLE_BY_THE_FOLLOWING_PROGRAM:

```
PROC2 (LEAK)
COMMENT
PROC3 ATTEMPTS_TO_ACHIEVE_(DIAGNS LEAK);
  BEGIN
  MOVE(CASE LOC);
  PIN(PIN2 HOLE3);
  PIN(PIN1 HOLE2);
  PUTGSK(GSKT2 CASE);
  PUTTOP(TOP GSKT2);
  SCREWD(SCREW3 HOLE1);
  UNPIN(PIN2 HOLE3);
  UNPIN(PIN1 HOLE2);
  SCREWD(SCREW2 HOLE3);
  SCREWD(SCREW1 HOLE2);
  IF ¬LOOSE(PUMP) THEN
    PROC3(LEAK)
  ELSE
    BEGIN
    UNSCRW(SCREW2 HOLE3);
    UNSCRW(SCREW1 HOLE2);
    UNSCRW(SCREW3 HOLE1);
    SCREWD(SCREW3 HOLE3);
    SCREWD(SCREW2 HOLE2);
    SCREWD(SCREW1 HOLE1);
    END
  END
```

REMARKS:

PROC2 is actually added onto PROC1 since the problem of diagnosing a leak is posed from a state in which the pump has been made. PROC2 repairs the pump in the case that the gasket is loose.

```
14 ___RULES_ENTERED
14 ___RULES_SUCCESSFUL
```

THE_GOAL_ (DIAGNS LEAK)_IS_ATTAINABLE_BY_THE_FOLLOWING_PROGRAM:

```
PROC3 (LEAK)
COMMENT
PROC4 ATTEMPTS_TO_ACHIEVE_(DIAGNS LEAK);
  BEGIN
  IF ¬BROKEN(GSKT2) THEN
    PROC4(LEAK)
  ELSE
    BEGIN
    UNSCRW(SCREW3 HOLE1);
    UNSCRW(SCREW1 HOLE2);
    UNSCRW(SCREW2 HOLE3);
    REMOVE(TOP CASE);
    REMOVE(GSKT2 CASE);
    REJECT(GSKT2);
    MOVE(CASE P1);
    PIN(PIN1 HOLE1);
    PIN(PIN2 HOLE2);
    PUTGSK(GSKT1 CASE);
    PUTTOP(TOP GSKT1);
    SCREWD(SCREW1 HOLE3);
    UNPIN(PIN1 HOLE1);
    UNPIN(PIN2 HOLE2);
    SCREWD(SCREW2 HOLE1);
    SCREWD(SCREW3 HOLE2);
    END
  END
```

REMARKS:

PROC3 repairs the leaky pump in the event that the gasket is broken. It is the first contingency plan in this example.

```
73 ___RULES_ENTERED
35 ___RULES_SUCCESSFUL
```

3.2 A Simple Translator from Infix to Polish Notation

This example illustrates the generation of conditional branches within loops in a program to convert strings of symbols in infix form into strings in polish form, i.e. "(X+Y*Z)" converts to "XYZ*+". This is a common symbol manipulation task in a compiler. The example shows how the system can be used to program in a structured "top down" manner.

A fully parenthesized, syntactically correct infix expression of a specified length is given as input and on output a result stack S contains the Polish string. A working stack R is used during the translation. We may consider the basic data structures (stacks)i.e. variables, constructors,(e.g. push) and selectors (e.g. pop)),and the primitive operators as given. Then,in this case,the user proceeded in the following steps.

(1)First the actions of the top level of the program were described by declarative statements (i.e. the definitions of RECOGNIZED and PROCESSYM in terms of basic concepts such as "X is a left parenthesis", and intermediate concepts such as "pop operators from stack X and push them onto stack Y".

(2) Then at the second level, Rules - in this case iterative rules - were given for writing loops that implement the intermediate concepts. In doing this,the user specified the major characteristics of a loop and left the system with the details of deciding whether to write such a loop,and if so, with the choice of local variables,the actual operations in the loop body and their order,(in so far as that was not specified) and with looking after the updating of the local variables. Thus in order to write the top level loop, TSLOOP, to achieve TSL(T,U,V), the user must have "thought out" an invariant relation between the elements manipulated by the loop body and what the goals of the loop body were (in this case one of the goals is a top level concept, RECOGNIZED(X,Y,Z)). The system, if it uses this rule in constructing the output, will construct a loop body including update assignments, and assemble it into a WHILE statement. Similary, in this example the user has supplied iterative rules for POPOPS and POPHOPS.

The output program consists of a main program, i.e. PROC1, containing a compound conditional statement which splits up the cases for processing as a function of the input symbol. Each allowable input symbol must be either of type variable, operator, left parenthesis, or right parenthesis. The main program processes the case in which the input symbol is an operator and generates calls to contingency programs, PROC3, PROC4, & PROC5, to be generated for the other three alternatives. The procedure calls PROC2, PROC6, & PROC7 result in error exits.

The various parts of the Frame definition will be given below followed by the generated programs.

RELATIONS USED IN THE FRAME DEFINITION:

RELATION	INTERPRETATION	FLUENT	PARTIAL	UNIQUENESS
C(X,Y)	"Contents of X is Y"	TRUE	FALSE	C(X,*)
INTEGER(X)	"X is an integer"	TRUE	FALSE	FALSE
VAR(X)	"X is a variable"	FALSE	TRUE	FALSE
LP(X)	"X is a left paren"	FALSE	TRUE	FALSE
RP(X)	"X is a right paren"	FALSE	TRUE	FALSE
OP(X)	"X is an operator"	FALSE	TRUE	FALSE
ISVAR(X)	"X is a program variable"	FALSE	FALSE	FALSE
NEXTSYM(X)	"A value for X is input"	TRUE	FALSE	FALSE
RECOGNIZED(X,Y,Z)	"Symbol X is recognized wrt stacks Y & Z"	TRUE	FALSE	FALSE
PROCESSYM(X)	"Symbol X is processed"	TRUE	FALSE	FALSE
>(X,Y)	"X is greater than Y"	FALSE	FALSE	FALSE
<(X,Y)	"X is less than Y"	FALSE	FALSE	FALSE
POLISH(X)	"X contains a Polish sequence"	TRUE	FALSE	FALSE
POLTSL(X,Y,Z)	"Translate an infix string x symbols long to Polish using stacks Y and Z"	TRUE	FALSE	FALSE
=(X,Y)	"X is equal to Y"	FALSE	FALSE	FALSE
PUSHED(X,Y)	"X is pushed onto Y"	TRUE	FALSE	FALSE
POPPED(X)	"X is popped"	TRUE	FALSE	FALSE
TOPPED(X,Y,Z)	"The top symbol of	TRUE	FALSE	TOPPED(X,Y,*)

	stack Y of size Z is assigned to X"			
POPOPS(X,Y)	"Pop operators from X and push onto Y"	TRUE	FALSE	FALSE
POPHOPS(X,Y,Z)	"Pop operators from Y that have greater priority than X and push onto Z"	TRUE	FALSE	FALSE
STACKSIZE(X,Y)	"Size of stack X is Y"	TRUE	FALSE	STACKSIZE(X,*)
STACK(X)	"X is a stack"	FALSE	FALSE	FALSE
EMPTY(X)	"Stack X is empty"	FALSE	TRUE	FALSE

ITERATIVE RULES: ----------

NAME: TSLOOP
BASIS: NEWVAR(X,Y) ∧ C(X,0)
INVARIANT: C(X,W) ∧ INTEGER(W) ∧ STACK(V) ∧ STACK(U) ∧ ISVAR(Y)
ITERATION STEP: C(X,(ADD1 W)) ∧ NEXTSYM(Y) ∧ RECOGNIZED(Y,U,V)
CONTROL TEST: >(X,T)
OUTPUT ASSERTION: POLISH(V)
GOAL: POLTSL(T,U,V)

NAME: RLOOP
BASIS: NEWVAR(X) ∧ STACKSIZE(U,Z) ∧ TOPPED(X,U,Z)
INVARIANT: C(X,Y) ∧ =(Y,(TOP U)) ∧ STACK(U) ∧ STACK(V) ∧ STACKSIZE(U,W)
ITERATION STEP: PUSHED(X,V) ∧ POPPED(U) ∧ TOPPED(X,U,W)
CONTROL TEST: ¬OP(X)
OUTPUT ASSERTION: POPOPS(U,V)
GOAL: POPOPS(U,V)

NAME: OLOOP
BASIS: NEWVAR(X) ∧ STACKSIZE(U,T) ∧ TOPPED(X,U,T)
INVARIANT: C(X,Y) ∧ =(Y,(TOP U)) ∧ STACK(U) ∧ STACK(V) ∧ STACKSIZE(U,W)
ITERATION STEP: PUSHED(X,V) ∧ POPPED(U) ∧ TOPPED(X,U,W)
CONTROL TEST: ¬OP(X) ∨ <((PRIORITY X)(PRIORITY Z))
OUTPUT ASSERTION: POPHOPS(Z,U,V)
GOAL: POPHOPS(Z,U,V)

PRIMITIVE PROCEDURE		PRE-CONDITIONS
push(X,Y) "Push symbol X onto stack Y"	ISVAR(X) ∧ STACK(Y) ∧ STACKSIZE(Y,Z)	PUSHED(X,Y) ∧ STACKSIZE(X,(SUB1 Y))
pop(X) "Pop stack X"	STACK(X) ∧ STACKSIZE(X,Y)	POPPED(X) ∧ STACKSIZE(X,(SUB1 Y))
getnext(X) "Get next symbol"	ISVAR(X)	NEXTSYM(X)
←(X,Y) "Assign Y to X"	ISVAR(X)	C(X,Y)
top(X,Y) "Put top of stack Y in X"	ISVAR(X) ∧ STACK(Y) ∧ STACKSIZE(Y,Z)	TOPPED(X,Y,Z) ∧ C(X,(TOP Y))

DEFINITIONS:

BODY OF DEFINITION	RELATION DEFINED
(VAR(X) ∨ LP(X) ∨ RP(X) ∨ OP(X)) ∧ PROCESSYM(X,Y,Z)	RECOGNIZED(X,Y,Z)
VAR(X) ∧ PUSHED(X,Z)	PROCESSYM(X,Y,Z)
LP(X) ∧ PUSHED(X,Y)	PROCESSYM(X,Y,Z)
RP(X) ∧ POPOPS(Y,Z) ∧ POPPED(Y)	PROCESSYM(X,Y,Z)
OP(X) ∧ POPHOPS(X,Y,Z) ∧ PUSHED(X,Y)	PROCESSYM(X,Y,Z)
=(X,0) ∨ INTEGER((SUB1 X))	INTEGER(X)

INITIAL STATE:

STACK(S) ∧ STACK(R) ∧ STACKSIZE(S,I) ∧ STACKSIZE(R,J)

ALGEBRAIC SIMPLIFICATION: (SUB1(ADD1 X)) → X

```
PROC1 (N R S)
ISVAR(X1);ISVAR(X2);ISVAR(X3);STACK(S);STACK(R);
COMMENT
INPUT:CONDITIONS:
STACKSIZE(R J)∧STACKSIZE(S I)
OUTPUT:CONDITIONS:
POLISH(S);
COMMENT
PROC6 ATTEMPTS:TO:ACHIEVE: (POPPED R)
PROC5 ATTEMPTS:TO:ACHIEVE: (PROCESS X2 R S)
PROC4 ATTEMPTS:TO:ACHIEVE: (PROCESS X2 R S)
PROC3 ATTEMPTS:TO:ACHIEVE: (PROCESS X2 R S)
PROC2 ATTEMPTS:TO:ACHIEVE: (PROCESS X2 R S);
  BEGIN
  X1 ← 0;
  WHILE ¬>(X1 N) DO
    BEGIN
    Z1 ← (X1+1);
    GETNEXT(X2);
    IF ¬OP(X2) THEN
      IF ¬RP(X2) THEN
        IF ¬VAR(X2) THEN
          IF ¬LP(X2) THEN
            PROC2(X2 R S)
          ELSE PROC3(X2 R S)
        ELSE PROC4(X2 R S)
      ELSE PROC5(X2 R S)
    ELSE
      BEGIN
      TOP(X3 R);
      WHILE OP(X3) ∧ ¬<((PRIORITY X3)(PRIORITYX2)) DO
        BEGIN
        PUSH(X2 S);
        IF EMPTY(R) THEN
          PROC6(R)
        ELSE
          BEGIN
          POP(R);
          END
        TOP(X3 S)
        END
      PUSH(X2 R);
      END
    X1 ← Z1
    END
  END


PROC3 (X2 R S)
ISVAR(X2);STACK(R);
```

```
COMMENT
INPUT:CONDITIONS:
STACKSIZE(R I)
OUTPUT:CONDITIONS:
STACKSIZE(R (ADD1 I))∧PUSHED(X2 R);
  BEGIN
  PUSH(X2 R);
  END


PROC4 (X2 R S)
ISVAR(X2);STACK(S);
COMMENT
INPUT:CONDITIONS:
STACKSIZE(S I)
OUTPUT:CONDITIONS:
STACKSIZE(S (ADD1 I))∧PUSHED(X2 S);
  BEGIN
  PUSH(X2 S);
  END

PROC5 (X2 R S)
ISVAR(X4);STACK(S);STACK(R);
COMMENT
INPUT:CONDITIONS:
STACKSIZE(R J)∧STACKSIZE(S I)
OUTPUT:CONDITIONS:
POPOPS(R S);
COMMENT
PROC7 ATTEMPTS:TO:ACHIEVE: (POPPED R) ;
  BEGIN
  TOP(X4 R);
  WHILE OP(X4) DO
    BEGIN
    PUSH(X4 S);
    IF EMPTY(R) THEN
      PROC7(R)
    ELSE
      BEGIN
      POP(R);
      END
    TOP(X4 R)
    END
    IF EMPTY(R) THEN
      PROC8(R)
    ELSE
      BEGIN
      POP(R);
      END
  END
```

REFERENCES

Bolles, R.,Paul, P., "The Use of Sensory Feedback in a Programmable Assembly System", Stanford AIM-220, October 1973

Buchanan, J.R., Luckham, D.C.,"On Automating The Construction of Programs," Stanford AI Project Memo, Stanford University, 1974.

Buchanan, J.R.,"A Study in Automatic Programming," Ph.D. Thesis, Stanford University, 1974.

Hewitt, C. 1971. "Description and Theoretical Analysis of Planner" Ph.D. Thesis, M.I.T., 1971.

Hoare, C.A.R. 1969. An axiomatic basis for computer programming, Comm. ACM, 12, 10, October 1969, 576-580, 583.

Hoare, C.A.R.; and Wirth, N. 1972. An axiomatic definition of the programming language Pascal, Berichte der Fachgruppe Computer-Wissenschaften 6, E.T.H., Zurich, November 1972.

Igarashi, S.; London, R.L.; Luckham, D.C. 1973. "Automatic Program Verification I: A Logical Basis and Implementation", Stanford AIM 200, May 1973.

Sussman, J.; Winograd, T. 1972. "Micro Planner Reference Manual", M.I.T. Project MAC Report 1972.

USING MODELS TO SEE

Alan K. Mackworth
University of Sussex*

Abstract

Scene analysis programs offer the hope of providing a more adequate account of human competence in interpreting line drawings as polyhedra than do the current psychological theories. This thesis has several aspects. The aspect concentrated on here is that those programs have explored a variety of methods of incorporating a priori knowledge of objects through the use of models. After outlining the range of models used and sketching some psychological theories, the various proposals are contrasted. This discussion leads to two new proposals for exploiting model information that involve elaborations of an existing program, POLY.

1. Introduction.

In one of its many roles, artificial intelligence is cast as the vanguard of an army of psychologists who seek a new paradigm for cognitive and perceptual processes. Despite several clarion calls to this effect (Minsky and Papert, 1972; Clowes, 1972; Sutherland, 1973) AI may well be a vanguard without an army. This paper attempts to show that a small part of the scouted territory is ripe for capture.

The interpretation of line drawings as polyhedral scenes has been the focus of most attempts to build AI vision systems. As it is a natural human task, several psychologists have also studied it. In sketching and contrasting various resultant theories, we will concentrate on how they represent the a priori knowledge of the objects that exist in the world. Of necessity, other essential themes such as non-model knowledge of the world (for example, support and the picture-formation process itself) or the use of picture cues to access the models are slighted.

Sections 2 and 3 of the paper sketch the use of models in several AI and human vision proposals. Section 4 briefly contrasts them using a few examples. Some of the weaknesses exposed lead to two proposals in section 5.

2. Models in Machine Vision.

Roberts (1965) used the three simple models of Fig.1. These can be expanded along each of their coordinate axes. Compound objects are created

---

*Now at Department of Computer Science,
University of British Columbia,
Vancouver 8, B.C, Canada.

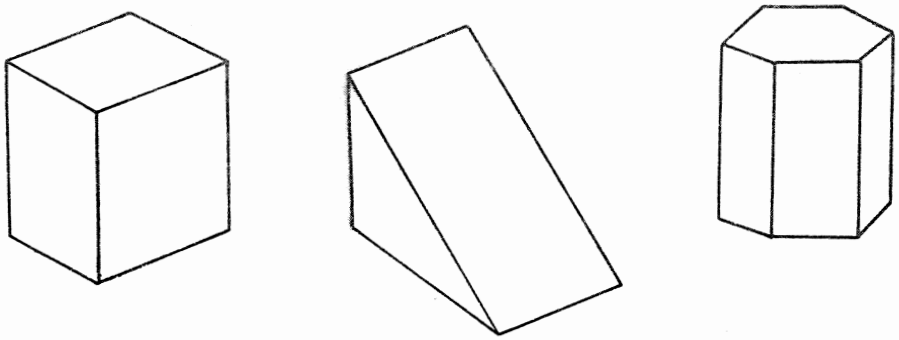


Figure 1. Roberts' simple object models

by abutting simple ones. Falk's (1972) recent state-of-the-art scene analysis system expected its visual world to be composed of instances of the nine polyhedral prototypes of specified dimensions shown in Fig.2.

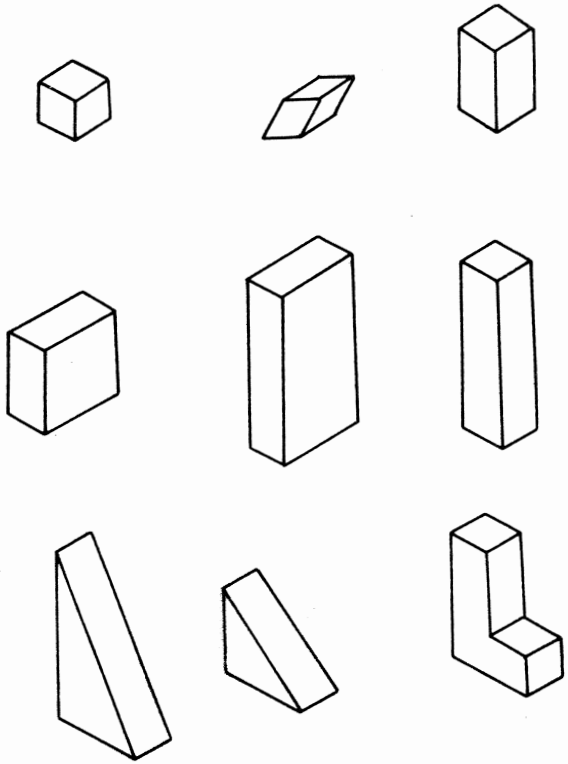


Figure 2. Falk's object prototypes

The size-specificity of the prototypes was exploited by the object recognition phase of the program in its use of the actual heights of the blocks and the lengths of their base edges.

At the other end of the size and shape specificity spectra for models are the edge-labelling procedures. These originated in Guzman's SEE (1968) which produces surface groupings corresponding to objects. Huffman (1971) and Clowes (1971) developed a procedure that relies on four prototype corners: the trihedral corners in which the object occupies 1, 3, 5 or 7 'octants'; the corners have no further shape-specificity. The corner models are accessed by the shape of the picture junctions. For each of four picture junction classes (L, FORK, ARROW, T), there is a list of possible corner/viewpoint configurations. These lists are used to label the edges depicted as convex or concave. The convex category is subdivided into three according to the viewpoint: either both surfaces depicted at the edge belong to it or the surface on the right, which does, is partially occluding the one on the left which doesn't or vice versa.

It has been shown (Mackworth, 1974) that SEE implicitly uses a single prototype corner: the one in which the object occupies only one 'octant'; whereas, Waltz (1972) has expanded the range of corner prototypes far beyond those of Huffman-Clowes.

The model information embedded in POLY (Mackworth, 1973) is minimal, confined as it is to a requirement that surfaces be planar and edges be occluding or connect (non-occluding); however, there is a marked preference for connect edges. With this apparatus somewhat augmented, POLY interacts with a representation called the gradient configuration (originally suggested by Huffman (1971))to produce a labelled interpretation. (The gradient of an edge is vector in a 2D gradient space whose direction is that of the corresponding picture line. Its length is the tangent of the angle between the edge and the picture plane. The gradient of a surface is in the direction of steepest descent in the surface away from the picture plane; the magnitude of the gradient is the tangent of the dihedral angle between the surface and the picture plane.) The final gradient configuration needs only the origin and scale of the gradient space defined before it represents the absolute orientations of the object surfaces. (POLY assumes orthographic projection; see (Mackworth, 1974) for the perspective case.)

3. Some Psychological Theories.

Attempts to provide psychological theories of the interpretation of line drawings have not usually provided an algorithm by which interpretation may

proceed though, presumably, the usual monocular depth cues are thought to be relevant. Rather, such theories seem to assume the existence of such an algorithm and concentrate on the tension set up between the 3D (scene) and 2D (picture) organisations. Kopfermann (1930) held that the impression of tridimensionality varies with the degree to which the scene organization is simpler than that of the picture. In extending that theory Hochberg and Brooks (1960) provided a quantified measure of simplicity as the sum of the number of lines, the number of angles and the number of angles differing in magnitude. Attneave and Frost (1969) presented a similar theory in which the competition between the scene and the picture is resolved by figural simplicity criteria.

Finally Hochberg (1968) almost anticipated the Huffman-Clowes algorithm as he demonstrated, with an ingenious experiment, that junctions act as 'local depth cues'.

4. Some Examples.

The discussion of this section uses, as examples, the two pictures in Fig. 3, which the reader should look at without reading further. The usual

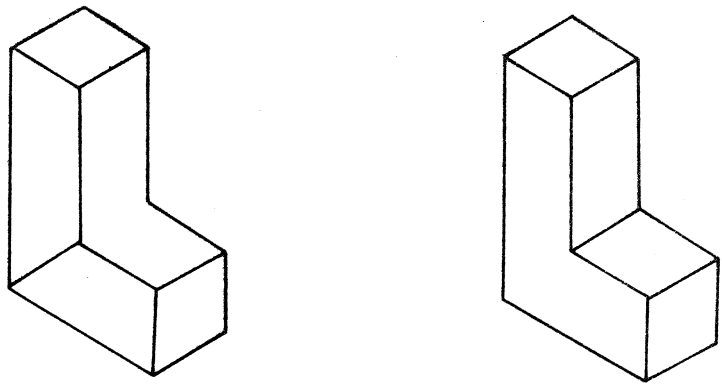


Figure 3. Two examples

impression given by Fig. 3(a) is that it remains obstinately flat on the page (provided one can suppress the tendency to see an edge that is not depicted and to ignore one that is thereby transforming the picture into Fig. 3(b)). Fig. 3(b) has a solid, three-dimensional appearance. This major difference which holds even though both are equally faithful depictions of polyhedra is a phenomenal fact requiring explanation.

The Huffman-Clowes algorithms do not offer that explanation. The pictures are successfully labelled with equal ease. The corners are all trihedral and both interpretations require three hidden surfaces to complete the object.

Perhaps the scene interpretation of Fig. 3(b) derives from its familiarity as Falk's program suggests: the L-beam is one of its nine prototype objects. Look then at Fig. 5(a) which is surely unfamiliar to the reader (although it is derived from pictures used by Shepard and Metzler (1971)); is that object any less solid than the one depicted in Fig. 3(b)?

Both Fig. 3(a) and Fig. 3(b) are interpreted by Roberts' program as compound objects made from cuboids (two and four, respectively). However, that program cannot, contrary to expectation, similarly interpret the two objects in Fig. 4.

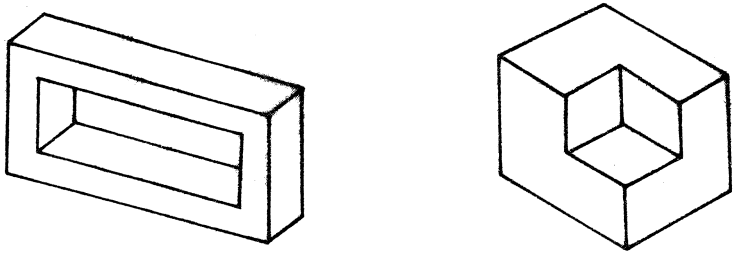


Figure 4. Two concave rectangular objects

The traditional depth cue theory of picture interpretation has nothing to say. In four of the pictures (Figs. 3(a), 3(b), 4(b), 5(a)) there are no traditional depth cues at all! Yet, in Fig. 5(a) for example, corner 1 is clearly nearer the observer than corner 2.

The Hochberg-Brooks criterion doesn't contradict the phenomenon. (By that criterion, the pictures have equal complexity while the scene in Fig. 3(b) is just marginally simpler than that in Fig. 3(a)). And yet, that doesn't take us very far. What mechanisms produced those scene interpretations in the first place? Hochberg (1968) has provided an excellent rebuttal of his earlier theory.

On the other hand, Hochberg's claim (1968) that the junction configurations act as 'local depth cues' is not very powerful. The only depth evidence given by edge labelling is provided by the occluding edges; that is the traditional depth cue of partial occlusion. Even if the edges of Fig. 5(a) are appropriately labelled convex, concave or occluding there is still no evidence that corner 1 is closer to the observer than corner 2; that is, an ordinary polyhedron with that appearance and those edge labels can be constructed for which that is not true.

5. Two Proposals.

Surely the solidity of Fig. 3(b) and Fig. 5(a) as contrasted with the 'flat' appearance of Fig. 3(a) can be explained as follows: in the former cases, the polyhedron interpretation can be seen as made up of surfaces of very familiar shape (in this case, rectangular) whereas in the latter that is not possible. This explanation suggests extensions to POLY that use the shapes of surfaces as models. Here two such extensions are presented. Rectangularity is often a major feature of the worlds we build for ourselves. The first proposal shows how a straightforward extension to POLY can exploit that feature. The second proposal is more of a substantial upheaval than an extension in that it suggests integrating the use of prototypes into the interpretation process.

5.1 Rectangularity.

Consider the rectangular object of Fig. 5(a). For this object POLY produces the gradient configuration that appears as a triangle in Fig. 5(b) There are only three possible values for the gradient so several surfaces

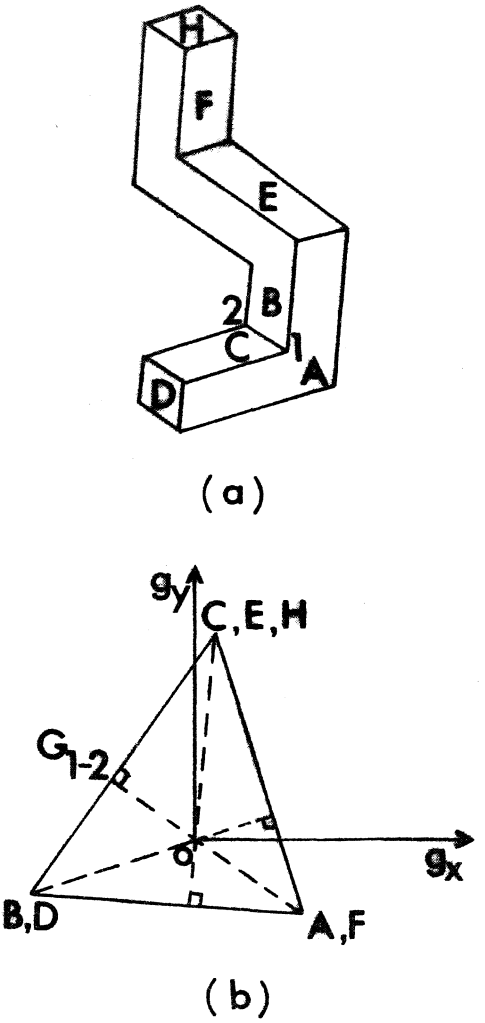


Figure 5. A rectangular object and its gradient space configuration

are superimposed at each position. This obscures the fact that the configuration is intricately connected: each pair of surfaces meeting in a connect edge is joined by a line perpendicular to the picture line showing that edge. Neither the position of the origin nor the size of the triangle is yet specified but note that E and A are ordered in the gradient

configuration just as they are across their common edge in the picture so that edge is convex whereas the relative positions of B and C are reversed in the picture and gradient spaces so that edge 1-2 is concave; however, as the actual values of the gradients are not determined, we still cannot say that corner 1 is closer than 2.

At any corner such as corner 1 in Fig. 5(a) there are three edges (which may not all be visible). Each pair of edges defines a surface at that corner. Each edge is normal to the surface defined by the other two. Since the direction of the gradient vector is the direction in the picture in which the normal appears to point, the direction of the gradient of each surface at the corner is given by the edge that does not belong to it. Thus gradient A must be in the direction of picture line 2-1. Since the vector difference between gradients B and C is required to be perpendicular to picture line 2-1, the origin must be on a perpendicular dropped from gradient A to the opposite side of the gradient triangle. Hence the origin must be at 0 shown in Fig. 5(b). The scale is immediately determined by the requirement that the product of the magnitudes of the gradient of A and the gradient of edge 1-2, G_{1-2}, must be unity. Now that the orientations of all the surfaces and edges are defined it is an obvious consequence that corner 2 is further from the picture plane than corner 1; that is shown by the fact that G_{1-2} points up to the left (not down to the right).

5.2 Using prototype surfaces.

The idea of using specific prototypes is attractive but as suggested in Section 4 complete polyhedral prototypes are, in a sense, too monolithic. In this section we show how the use of prototype surfaces can be integrated directly into the POLY interpretation process.

Consider Falk's list of nine prototype objects. They have in all fifty-four separate faces; yet those faces have only fourteen distinct polygonal shapes. The size-specificity of these shapes will be dropped for the sake of this argument although it could be retained. Dropping size-specificity (so that a 1 x 2 rectangle represents itself and the 2 x 4 rectangle etc.) leaves a total of twelve distinct surface shapes.

First, a geometrical fact must be stated (Mackworth, 1974). Suppose one is given the true shape of a surface in the form of a polygon (where the dimensions may be uniformly scaled up or down by a factor, k), the projected shape of that surface and three or more pairs of non-collinear points on the true and projected shapes that correspond. From this information it is easy to

compute whether the true shape could produce the projected shape and, if it does, the value of k and the gradient of the surface.

For each picture region, by considering the topologically identical surfaces, a set of possible surfaces each with a corresponding k and gradient could be computed. If that set is empty then the region depicts a partially occluded surface.

This is now a labelling situation comparable to the corner labelling algorithms of Huffman, Clowes and Waltz. In those algorithms each junction has associated with it a set of possible corners; the aim of the interpretation is to discover a unique corner corresponding to each junction. Here, besides labelling each edge, the aim is to assign a unique surface to each region. Agreement between the interpretations of adjacent regions is necessary if the edge is taken to be connect. The agreement takes two distinct forms. First, the POLY coherence rules must be satisfied and second, model-based coherence rules must be used. Such model-based rules would, at the lowest level, be of the form: Are there two such surfaces meeting at an edge in the set of prototypes? If so, do those surfaces meet at this dihedral angle? Do they agree on the scale factor? Higher levels would also be required: Are there three such surfaces meeting at a corner?

Procedurally, this approach need not be implemented in a depth or breadth-first fashion. It is amenable to the two-stage Waltz search procedure which would first weed out the lists of possible surfaces (just as Waltz weeded the lists of possible corners) based on consideration of the mutual interpretation of each pair of adjacent regions and only then try to build complete coherent interpretations.

6. Conclusion.

World knowledge of the type incorporated as models in scene analysis programs is an essential feature of any psychological theory that attempts to explain human competence in interpreting line drawings as polyhedra. Furthermore, in those programs that knowledge is used in a procedural fashion; they demonstrate, at the very least, how a scene interpretation can be achieved.

The discussion of Section 4 has pointed out some of the ways in which the available range of models is deficient for purposes of psychological explanation. The two proposals of Section 5 are designed to provide mechanisms that reflect particular human competence in this task domain.

Acknowledgements

The author is grateful to Max Clowes, Roddie Cowie, Frank O'Gorman and Aaron Sloman for discussion and criticism.

The research reported is part of a project supported by the Science Research Council.

Bibliography

Attneave, F. and Frost, R. (1969) The determination of perceived tridimensional orientation by minimum criteria. Perception & Psychophysics 6, 391-396.

Clowes, M.B. (1971) On seeing things. Artificial Intelligence 2,1, 79-112.

Clowes, M.B. (1972) Artificial intelligence as psychology. AISB Bulletin, November, 1972.

Falk, G. (1972) Interpretation of imperfect data as a three dimensional scene. Artificial Intelligence 3, 2, 101-144.

Guzman, A. (1968) Decomposition of a visual scene into three-dimensional bodies. AFIPS Proc. Fall Joint Comp. Conf. Vol. 33, pp.291-304.

Hochberg, J. and Brooks, V. (1960). The psychophysics of form: reversible-perspective drawings of spatial objects. Amer. J. Psychol. 73, 337-354.

Hochberg, J. (1968) In the mind's eye. Contemporary Theory and Research in Visual Perception Haber, R.N. (Ed.), Holt, Rinehart and Winston, N.Y., pp. 309-331.

Huffman, D.A. (1971) Impossible objects as nonsense sentences. Machine Intelligence 6 Meltzer, B. and Michie, D. (Eds.), Edinburgh University Press, Edinburgh, pp.295-323.

Kopfermann, H. (1930) Psychologische Untersuchungen uber die Wirkung zweidimensionaler Darstellungen korperlicher Gebilde. Psychol. Forsch., 13, 293-364.

Mackworth, A.K. (1973) Interpreting pictures of polyhedral scenes. Artificial Intelligence, 4, 2, 121-137.

Mackworth, A.K. (1974) On the interpretation of drawings as three-dimensional scenes. D.Phil. thesis, Laboratory of Experimental Psychology, University of Sussex.

Minsky, M. and Papert, S. (1972) Progress Report Memo. No. 252. Artificial Intelligence Lab., Mass. Inst. of Technol., Cambridge, Mass.

Roberts, L.G. (1965) Machine perception of three-dimensional objects. Optical and Electro-Optical Information Processing Tippett, et al (Eds.), MIT Press, Cambridge, Mass., pp. 159-197.

Bibliography

Shepard, R.N. and Metzler, J. (1971) Mental rotation of three-dimensional objects. Science 171, 701-703.

Sutherland, N.S. (1973) Some comments on the Lighthill report and on Artificial Intelligence. Artificial Intelligence: a paper symposium Science Research Council, London, pp. 22-31.

Waltz, D.L. (1972) Generating semantic descriptions from drawings of scenes with shadows. AI TR-271(Thesis), MIT, Cambridge, Mass.

A theory of evaluative comments in chess.

by

Donald Michie

Abstract

Classical game theory partitions the set of legal chess positions into only three evaluative categories: won, drawn and lost. Yet chess players employ a wide variety of evaluative terms, distinguishing (for example) a "drawn" from a "balanced" position, a "decisive" from a "slight" advantage, and a "blunder" from a "mistake".

As an extension of the classical theory, a model of fallible play is developed. Using this, two quantities can in principle be associated with each position, its "game-theoretic value" and its "expected utility". A function of these two variables can be found which yields interpretations of many evaluative terms used by chess commentators.

A theory of evaluative comments in chess

Introduction

The game tree of chess contains about 10^{46} positions (Good, 1968) a substantial proportion of which are terminal. The rules of the game assign a value to every terminal position, +1, 0 or -1 according as the position is won, drawn or lost for White. These values can be backed up the game tree using the minimax rule, so that in principle every position can be given a value, including the initial position. This last is known as "the value of the game", and is widely conjectured to be 0 for chess. If this conjecture is correct, and if both sides play faultlessly, i.e. only execute value-preserving moves (it follows from the "back-up" method of assigning values that there is at least one such move available from every non-terminal position), then the game must end in a draw. A fragment of a hypothetical game tree is depicted in Figure 1. In Figure 2 the method of attaching game-theoretic values to positions is illustrated.

An evaluation function could in principle map board positions into a larger set of values, making it possible to express a distinction between positions which are "marginally" won and positions which are "overwhelmingly" or "obviously" won, or between drawn positions in which White, or Black, "has the edge" and drawn positions which are "equally balanced", and so forth. Two circumstances suggest that a useful purpose might be served by multi-valued functions.

(i) Chess Masters and commentators have developed a rich descriptive language for the expression of such distinctions.

(ii) Computer chess programs employ real-valued functions for evaluating terminal positions, not of the game tree which is too large, but of the lookahead tree. Values backed up according to the minimax rule are used to select the next move. It would be nice to have a theory/

theory which allowed us to assign some definite interpretation to such values.

There is thus a *prima facie* need for a stronger theory of position-evaluation. This paper discusses chess, but the treatment is general and covers all two-person zero-sum games of perfect information without chance moves.

Requirements of a theory

A good theory should explicate a wide variety of commentators' concepts. The following is a representative list. Where a conventional symbol is available it precedes the verbal comment.

(1) A dead draw (nothing that either player can do can avert a draw).
(2) A complicated position.
(3) =, a balanced position.
(4) $\pm$, White has a slight advantage.
(5) $\pm$, White has a clear advantage.
(6) +−, White has a decisive advantage.
(7) A certain win for White.
(8) A difficult position for White.
(9) A losing move.
(10) An inaccurate move: White weakens his position.
(11) White strengthens his position.
(12) ?, a mistake.
(13) ??, a blunder.
(14) !, a strong move.
(15) !!, a very strong or brilliant move.
(16) !?, a brilliant but unsound move.
(17) Best move.
(18) (!), best move in difficult circumstances.
(19) A safe move.
(20) White should press home his advantage.
(21) Black should play for time.

Main/

Main features of the theory

The game-theoretic model pre-supposes perfect play, whereas in the real-life game of chess (whether human or computer) both sides are susceptible to error. Our theory is based on this distinction, and presents the following main features:

(1) We follow I.J. Good (1968) and interpret the values of terminal positions as utilities as though the game were played for a unit stake. Values for pre-terminal positions are then calculated as expected utilities. In order to avoid confusion we shall refer to these throughout as "expected utilities", never as "values", reserving the latter term for game-theoretic values.

(2) A model of imperfect but skilled play is developed. Chess skill appears in this model as an adjustable parameter running from 0 (random play) to ∞ (perfect play).

(3) In the new model the classical game-theoretic treatment appears as a special case.

The calculation of expected utilities

Consider a state, s_0, from which transitions to successor states $s_1, s_2, s_3, \ldots s_n$ can occur with respective probabilities $p_1, p_2, p_3, \ldots p_n$. Let us suppose that these successor states have associated utilities $u_1, u_2, u_3, \ldots u_n$. Then the expected utility associated with s_0 is $\sum_{i=1}^{n} p_i u_i$. It follows trivially that if we interpret as utilities the values attached by the rules of chess to the terminal positions then the values assigned to the non-terminal positions by minimaxing can be interpreted as expected utilities. In this special case the p's associated with those arcs of the game tree which carry a change of game-theoretic value are all 0. Consequently the evaluation of $\sum_{i=1}^{n} p_i u_i$ at each node reduces to obtaining the min or the max of the successor-values according/

according as White or Black has the move. The above specification is ambiguous in the case when two or more of the moves applicable to a given board position are value-preserving. We can either select one of these at random and assign a probability of unity to it and zero probabilities to the rest, or we can divide the unit probability equally among them. In the case of error-free play calculation of expected utilities according to either procedure leads to the same result. As the basis of a model of actual play we shall adopt the second alternative, which is illustrated in Figure 2.

We now relax the game-theoretic condition that at each choice-point on the tree there is probability 1 that a value-preserving move ("sound" or "correct" move) is chosen, and we introduce the possibility of error. In constructing a model of error, we express the relative probabilities of making alternative moves from a given position as a monotonic increasing function (decreasing function for Black, since all utilities are expressed from White's standpoint) of the expected utilities of the corresponding successor positions. Thus the move leading to the highest expected utility will be chosen with highest probability (but not with probability 1 as in the game-theoretic, error-free, model), the move leading to the next highest expected utility with next highest probability, and so on. We thus envisage an idealised player whose statistical behaviour reflects the rank-ordering of the expected utilities of chess positions. Using such a model it is again possible to label all the nodes of the tree, working upwards from the terminal nodes, but by a procedure which differs from the minimax method.

The notion of discernibility

In order to carry out some illustrative computations based on this idea, we now choose an actual monotonic function. No significance is claimed for the particular choice, since the points which we seek to establish are qualitative rather than quantitative. Certain ideas must, however, be reflected in any such function. A central one is that of discernibility. We conceive the player as standing upon a given node of the game-tree and/

and looking towards its successors. These are labelled with their expected utilities, but the labels are not fully discernible to him. Discernibility is directly related to the strength of the player (the labels are fully discernible to an infinitely strong player) and inversely related to the number of moves separating the node from the end of the game: next-move mates and stalemates are fully discernible even to the beginner, but next-move expected utilities obtained by backing up are less so. Reflecting these considerations, we shall define the discernibility from a board state s_o of the expected utility of a given successor state s_j as:

$$d_i = (M+1)^{\frac{3(r_j+3)}{r_j+\varepsilon}} \qquad \text{..... (1)}$$

where M is the merit of the player in kilo-points of the U.S. Chess Federation scale, so that $0 \leq M$, and r_j is the number of moves that the value associated with s_j has been backed up. The symbol ε denotes an arbitrarily small quantity introduced to avoid the expression becoming infinite for $r_j = 0$.

The expected utilities themselves are real numbers lying in the range from -1 through 0 to +1. They are interpreted as being in logarithmic measure, to base d. Using this base, we take the anti-logarithms of the expected utilities associated with the n successors of a given position as giving the relative probabilities with which a player of merit M who has reached s_o selects the corresponding moves. Thus, for the transition $s_o \rightarrow s_j$,

$$p_j \propto d^{u_j} \qquad \text{(2)}$$

Normalising these so as to obtain actual probabilities, p_1, p_2, ... p_n, the expected utility of a position is evaluated as $\sum_{i=1}^{n} p_i u_i$, where u_i is the expected utility of the position generated by the i-th member of the set of available moves. Starting at the terminal positions, this gives a method for assigning expected utilities to successively higher levels of the game tree until every position has been labelled.

A sample computation

Consider the terminal fragment of game-tree shown in Figure 1. We shall illustrate step by step the calculation of expected utilities so as to label every node in the diagram. First we make assumptions for/

for the playing strengths K_W and K_B of White and Black respectively. If we are to extract examples of the broad range of evaluative concepts from so ultra-simplified a game tree we must set these strengths very low. Let us set $K_W = 0.2$ and $K_B = 1.4$: White is then an abject beginner and Black a weak tournament player. In our model $K = 0$ implies random play. The notation u(s) denotes the expected utility of position s.

H4: All successors have the same value, +1. $u(H_4) = +1$.

H5: There is only one successor, so the move-probability is unity. $u(H_5) = +1$.

G1: Unique successor. $u(G_1) = 0$.

G2: Equivalued successors. $u(G_2) = -1$.

G3: Equivalued successors. $u(G_3) = +1$.

F9: From (2) we have

Move to G1: $d^0 = 1$ = relative probability.

Move to G2: r = 1, so, from (1), $d = 1.2^{12} = 8.915$. Rel. prob. = 1/8.915 = 0.1121.

Move to G3: r = 2, so $d = 1.2^{7.5} = 3.925$ = rel. prob.

Normalized probabilities:

G_1	0.1985
G_2	0.0222
G_3	0.7792

$u(F_9) = 0.1985 \times 0 + 0.0222 \times -1 + 0.7792 \times +1 = +0.757$

E1: Equivalued successors. $u(E_1) = -1$.

E2: r = 0. $u(E_2) = -1$, and similarly for $u(E_3)$ and $u(E_4)$.

E5: Unique successor. $u(E_5) = 0.757$.

D9: Move to E1: r = 1. $d = 1.2^{12}$. Rel. prob. = 1/8.915 = 0.112. Similarly for moves to E_2, E_3, and E_4.

Move to E6: Rel. prob. = 1, and similarly for move to E_7.

Move to E5: r = 4. $d = 1.2^{5.25} = 2.604$. Rel. prob. = 2.0640.

Normalised probabilities:

E_1	0.025
E_2	0.025
E_3	0.025
E_4	0.025
E_5	0.457
E_6	0.222
E_7	0.222
	1.001

$u(D_9) = 0.457 \times 0.757 - 0.100 = 0.246$

C1:/

C1: r = 0. $u(C_1) = -1$, and similarly for $u(C_2)$, $u(C_3)$ and $u(C_4)$.

C5: Unique successor. $u(C_5) = 0.246$.

C6: Equivalued successors. $u(C_6) = 0$, and similarly for $u(C_7)$ and $u(C_8)$.

B1: Move to C1: r = 1. $d = 1.2^{12}$. Rel. prob. = 1/8.915 = 0.112 and similarly for moves to C_2, C_3, and C_4.

Move to C5: r = 6. $d = 1.2^{4.5} = 2.272$. Rel. prob. = 1.2240.

Normalised probabilities:

C_1	0.06703
C_2	0.06703
C_3	0.06703
C_4	0.06703
C_5	0.73190
	1.00002

$u(B_1) = 0.7319 \times 0.246 - 0.2681 = -0.088$.

B2: Equivalued successors. $u(B_2) = 0$.

A: Move to B1: r = 7. $d = 2.4^{4.286}$. Rel.prob. = 1.391.

Move to B2: Rel. prob. = $d^0 = 1$.

Normalised probabilities:

B_1	0.582
B_2	0.418

$u(A) = 0.582 \times -0.088 + 0.418 \times 0 = -0.051$.

In Figure 3 the tree of Figure 1 is shown with expected utilities, calculated as above, attached to the nodes. The expected utility of the root node, A, turns out to be one twentieth of a unit in Black's favour, - a "slight plus" for Black. The analysis of Black's "plus" is worth pursuing, for it illustrates certain fundamental concepts to which our theory is directed, in particular the idea that a losing move (in the game-theoretic sense of a transition for White to value -1 or for Black to value +1) can also be the "best" move against a fallible opponent.

Note that Black can secure a certain draw by moving to B_2. Note also that the move to B_1 is a losing move in the game-theoretic sense, for White can then win by the sequence $B_1 \rightarrow C_5 \rightarrow D_9 \rightarrow E_5 \rightarrow F_9 \rightarrow G_3$, as shown by the heavy line in Fig. 2. Yet the expected utility of the move, -0.088, is marginally better for Black than that of the "correct" move (expected utility = 0), and our model of Black, possessed of a weak tournament player's discernment, shows/

shows a 58% preference for the move. The statistical advantage arises, as can be seen by inspecting the diagram, from the fact that play is switched into a subtree where the error-prone White has numerous opportunities for error presented to him. He has to find the needle of sound play in a haystack of hazards. In such a situation we sometimes say that Black sets "traps" for his opponent. If the aesthetic features of the move to B_1 appeal to the commentator, he may even use the annotation "!?", which we take to mean "brilliant but unsound". A sufficient increase in the strength of White could give cause to remove the "!" or even to convert it into a second "?". To illustrate this point we have re-calculated the entire diagram after setting $M_W = M_B = 1.4$, shown in Figure 4. Here the move to B_1 does not appear as "best", nor even as a mistake, but as a blunder, and correspondingly our model of Black shows a preference of approximately 40:1 for B_2.

Returning to the list of specimen evaluative comments introduced earlier, we can now derive explications for them. Wherever possible, an explication is expressed in terms of two functions of a board position, namely its game-theoretic value $\underline{v}$ and its expected utility $\underline{u}$. Where a move, rather than a position, is described, we use the notation $\Delta\underline{v}$ and $\Delta\underline{u}$ to denote the changes in the corresponding quantities effected by the move. We denote by $\mathbf{S}_1$ the position from which the move is made and by $\mathbf{S}_2$ the position which it generates. Some items of the original list have for completeness been differentiated into sub-concepts. Some of these would never appear in a chess book although under assumptions of very low playing strength they are generated by our model. Case 2 of (6) is an example of this: a "decisive advantage" of this kind would arise, for example, in the initial position if Bobby Fischer gave Queen odds to a beginner.

	Comment	Explication
(1)	A dead draw.	$v = 0$ and $u = 0$.
(2)	$\mathbf{S}$ is complicated.	the first few levels of the tree rooted in B have high branching ratios.
(3)	=, $\mathbf{S}$ is balanced. Case 1:/	$v = 0$ and $u \simeq 0$.

Comment	Explication
Case 1: $\mathfrak{S}$ is lifeless	$\mathrm{var}(v_t) \simeq 0$ } - see text
Case 2: $\mathfrak{S}$ has high tension	$\mathrm{var}(v_t) \gg 0$ }
(4) $\pm$, White has a slight advantage.	$v = 0$ and $u > 0$.
(5) $\pm$, White has a clear advantage (good winning chances).	$v = 0$ and $u \gg 0$.
(6) +−, White has a decisive advantage.	$u \simeq +1$.
Case 1: White has excellent winning chances.	$v = 0$ and $u \simeq +1$.
Case 2: Although White's game is theoretically lost, he is almost bound to win.	$v = -1$ and $u \simeq +1$.
Case 3: An easy win for White.	$v = +1$ and $u \simeq +1$.
(7) A certain win for White.	$v = +1$ and $u = +1$.
(8) $\mathfrak{S}$ is difficult.	$v \gg u$.
Case 1: White needs accuracy to secure the draw.	$v = 0$ and $u \ll 0$.
Case 2: White needs accuracy to secure the win.	$v = +1$ and $0 < u \ll 1$.
Case 3: Although theoretically won, White's position is so difficult for him that he should offer a draw.	$v = +1$ and $u < 0$.
(9) A losing move.	$v(\mathfrak{S}_2) = -1$ and $v(\mathfrak{S}_1) > -1$.
(10) An inaccuracy: White's move weakens his position.	$\Delta v = 0$ and $\Delta u < 0$.
(11) White's move strengthens his position	$\Delta v = 0$ and $\Delta u > 0$.
(12) ?, a mistake.	$\Delta v = -1$ and not $(\Delta u \ll 0)$.
(13) ??, a blunder.	$\Delta v < 0$ and $\Delta u \ll 0$.
(14) !, a strong move.	$\Delta v = 0$ and $\Delta u > 0$ and $\mathfrak{S}_1$ is difficult.
(15) !!, a very strong or brilliant move.	$\Delta v = 0$ and $\Delta u \gg 0$.
(16) !?, a brilliant but unsound move.	$\Delta v < 0$ and $\Delta u \gg 0$.
(17) Best move.	Δu is max.
(18) (!), best move in difficult circumstances.	Δu is max and $\mathfrak{S}_1$ is difficult.
(19) A safe move.	$\Delta v = 0$ and $\mathfrak{S}_2$ is lifeless.

(20)/

Comment	
(20)	"White should press home his advantage." The rationale for trying to shorten the game when ahead can be understood by noting in Figure 3 how the advantage decays as we move backwards from the terminal positions. In Figure 5 White, in moving from B_1, has been given an additional option in the form of a move to $C_{5.1}$, from which Black is forced to move directly to F_9 (S-shaped arc in Fig. 5). Game-theoretically the choice between moving to C_5 and moving to $C_{5.1}$ is equally balanced since they are both "won" positions for White. But the expected utilities, +0.246 against +0.757, tell the true story, that if he incurs needless delay in a won position, especially if it is a complicated position (high branching ratio of immediately dependent tree), he multiplies his chances of error. Our model selects the move to $C_{5.1}$ with 1.7 times the frequency of C_5, with a corresponding increase of $u(B_1)$ (see Fig. 5).
(21)	"Black should play for time" is the complementary advice one should give to the other player in the foregoing situation. If our hypothetical node $C_{5.1}$ had a second branch leading to D_9 (shown as a broken line in Fig.5), then Black should prefer it to F_9.

We exhibit systematically in Table 1 various combinations of v and u, entering in each case the evaluative comment which seems most appropriate.

"Tension"

The minimax value of s can be regarded as in some sense summarising the values of the terminal nodes of the tree rooted in s. More obviously, the expected utility of s, which has the form of a weighted mean, constitutes a summary of a different kind of this same set of quantities. It seems natural to proceed to statistics of higher order, i.e. from representative values and means to variances. Might such second-moment statistics also possess recognisable meaning in terms of the chess commentator's vocabulary?

I.J. Good (loc. cit.) discusses a property of chess positions which he calls "agitation". He defines it by considering how sharply the estimated utility of a position is changed by investing a further unit of work in deepening the forward analysis. This quantity/

quantity will necessarily be positively related to the variance of the distribution of u-values over the dependent sub-tree, and hence to the measure which we develop below for the "tension" of a position. The former British Champion, C.H.O'D. Alexander, uses this term in an introductory chapter to "Fischer v. Spassky Reykjavik 1972". He writes (see Figure 6)

> "Let me illustrate (a little crudely) this question of tension by comparing two openings:
>
> A. (Giuoco Pianissimo) 1. P-K4, P-K4; 2. Kt-KB3, Kt-QB3; 3. B-B4, B-B4; 4. P-Q3, P-Q3; 5. Kt-B3, Kt-B3.
>
> B. (Gruenfeld Defence: see the Siegen game Spassky v. Fischer) 1. P-Q4, Kt-KB3; 2. P-QB4, P-KKt3; 3. Kt-QB3, P-Q4; 4. P x P, Kt x P; 5. P-K4, Kt x Kt; 6. P x Kt, B-Kt2; 7. B-QB4, P-QB4. The moves in example A are perfectly correct - but after five moves the game is as dead as mutton; it is too simple, too balanced, and is almost certain to lead to an early and dull draw. The moves in example B are objectively no better - but the position is full of tension; White has a powerful Pawn centre but Black can exert pressure on it and, if he survives the middle game, may stand better in the ending - the players are already committed to a difficult and complex struggle in which a draw is not very likely."

A simple way of capturing the spirit of Alexander's definition within the framework of our theory is to use the weighted mean square of the terminal values of the tree rooted in $\mathbf{3}$, i.e.

$$\mathrm{var}(v_t) = \sum_{t \in T} p_t v_t^2$$

where T is the set of terminal positions and p_t is the probability of arriving at the t-th member of this set starting at $\mathbf{3}$. A value of unity corresponds to maximal tension and a zero value to minimal tension (the latter can only be attained by a "dead draw"). The tension of the root node of Figure 3 is estimated by this method as .659/ Referring to comment no.(3) above we assign this root node to Case 2 rather than to Case 1 of the category "balanced". Note that although "tension" is calculated from game-theoretic values, v_t, use is made of the u_t's in the calculation of the probabilities/

probabilities, p_t, and hence the measure is affected by variation of the merit parameters M_W and M_B. As soon as we postulate greater playing strength on the part of White some of the tension of the position is reduced. The tension of node A in Figure 4 is only .024 reflecting the fact that the Black is almost certain to steer play into the "dead draw" sub-tree.

Note that $\sum_{t \in T} p_t v_t^2$ is equal simply to the probability of a non-drawn outcome. But we have preferred to formulate the expression explicitly as a variance, since in realistic cases game-theoretic values are not likely to be available, or calculable in practice. The approximating formula $\sum_{t \in U} p_t u_t^2$ may then prove useful, where the u_t's have been assigned by some evaluation function (or by human intuition) to the members of U, the set of states on the lookahead horizon.

Concluding remarks

Our object has been to extend the strict game-theoretic model of chess, which assigns to board positions only three values: +1, 0 and -1. A good model should do justice to the profusion of chess commentators' evaluations. Specimen evaluative comments have been displayed as bench-marks against which the extended theory may be assessed. We have illustrated with worked examples a simple model based on the notions of utility and statistical expectation. Our model finds no particular difficulty in explicating the specimen evaluative comments. It also reduces to the game-theoretic model in the special case of perfect play.

Chess programs might benefit from using such a model, rather than the minimax model. The point could be tested experimentally. Another worth-while study would be to explore parts of a non-trivial sub-game of chess of which virtually complete game-theoretic knowledge exists (as in S. Tan's (1974) program for K + B versus K + P end-games) in search of illustrative tree fragments to replace our concocted examples. The numerical explication of concepts could then be used to make the program print out its own comments on sample end-game play. These could be compared with the intuitions of experienced players.

References

Alexander, C.H.O'D. Fischer v. Spassky Reykjavik 1972. Penguin.

Good, I.J. (1968) A five-year plan for automatic chess. Machine Intelligence 2 pp. 89-118 (eds. E. Dale and D. Michie). Edinburgh: Edinburgh University Press.

Tan, S.T. (1974) Kings, pawn and bishop. Research Memorandum MIP-R-108. Edinburgh: Department of Machine Intelligence.

Acknowledgement

This work was done during two months' study leave granted me by the University of Edinburgh.

	u = 0	u = −1	u = +1	u ≃ 0
v = −1	s is virtually impossible (because of the unlikelihood that u should be identically zero).	s is a certain win for Black.	s is impossible.	White has excellent drawing chances. Black needs accuracy to make sure of his win.
v = 0	s is a certain draw ("dead draw")	s is impossible.	s is impossible.	s is a balanced position.
v = +1	s is virtually impossible (because of the unlikelihood that u should be identically zero).	s is impossible.	s is a certain win for White.	Black has excellent drawing chances. White needs accuracy to make sure of his win.
	1	2	3	4

Table 1. Evaluative comments on positions (comments on moves are not shown here) corresponding to various combinations of game-theoretic value, v, and expected utility, u.

$u \simeq -1$	$u \simeq +1$	$-1 \ll u \ll 0$	$+1 \gg u \gg 0$	$-1 < u \ll 0$	$+1 > u \gg 0$
An easy win for Black (decisive advantage).	Black has a theoretical win but is almost bound to lose.	Black has a mildly difficult win.	Black needs extreme accuracy to make sure of his win (a very difficult win for Black).	Black has a clear advantage.	Black has a theoretical win but is likely to lose.
Black has excellent winning chances. White needs great accuracy to make sure of the draw.	White has excellent winning chances. Black needs great accuracy to make sure of the draw.	Black has a slight advantage. White needs care to make sure of the draw.	White has a slight advantage. Black needs care to make sure of the draw.	Black has good winning chances. White needs accuracy to make sure of the draw.	White has good winning chances. Black needs accuracy to make sure of the draw.
White has a theoretical win but is almost bound to lose.	An easy win for White. (decisive advantage).	White needs extreme accuracy to make sure of his win (a very difficult win for White).	White has a mildly difficult win.	White has a theoretical win but is likely to lose.	White has a clear advantage.
5	6	7	8	9	10

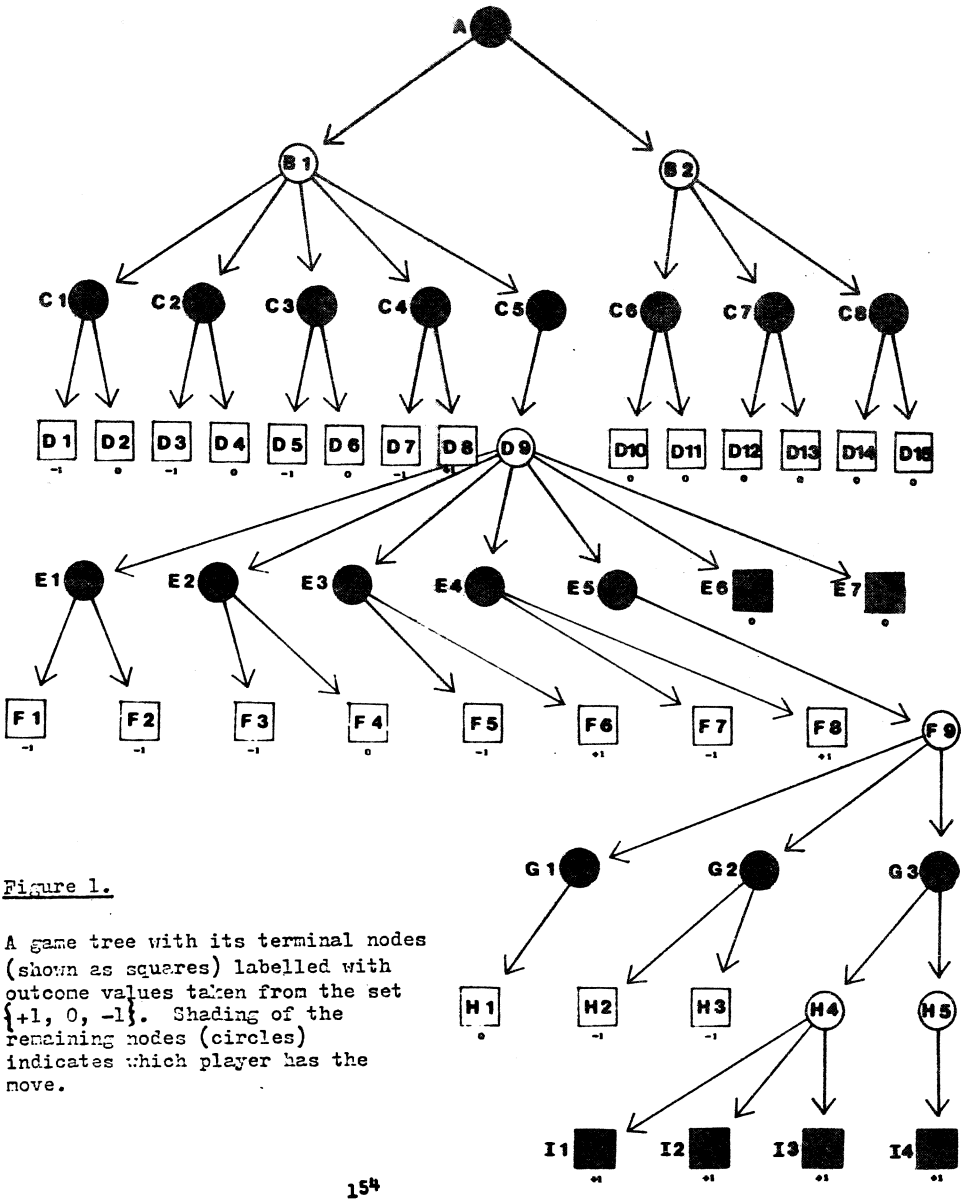


Figure 1.

A game tree with its terminal nodes (shown as squares) labelled with outcome values taken from the set $\{+1, 0, -1\}$. Shading of the remaining nodes (circles) indicates which player has the move.

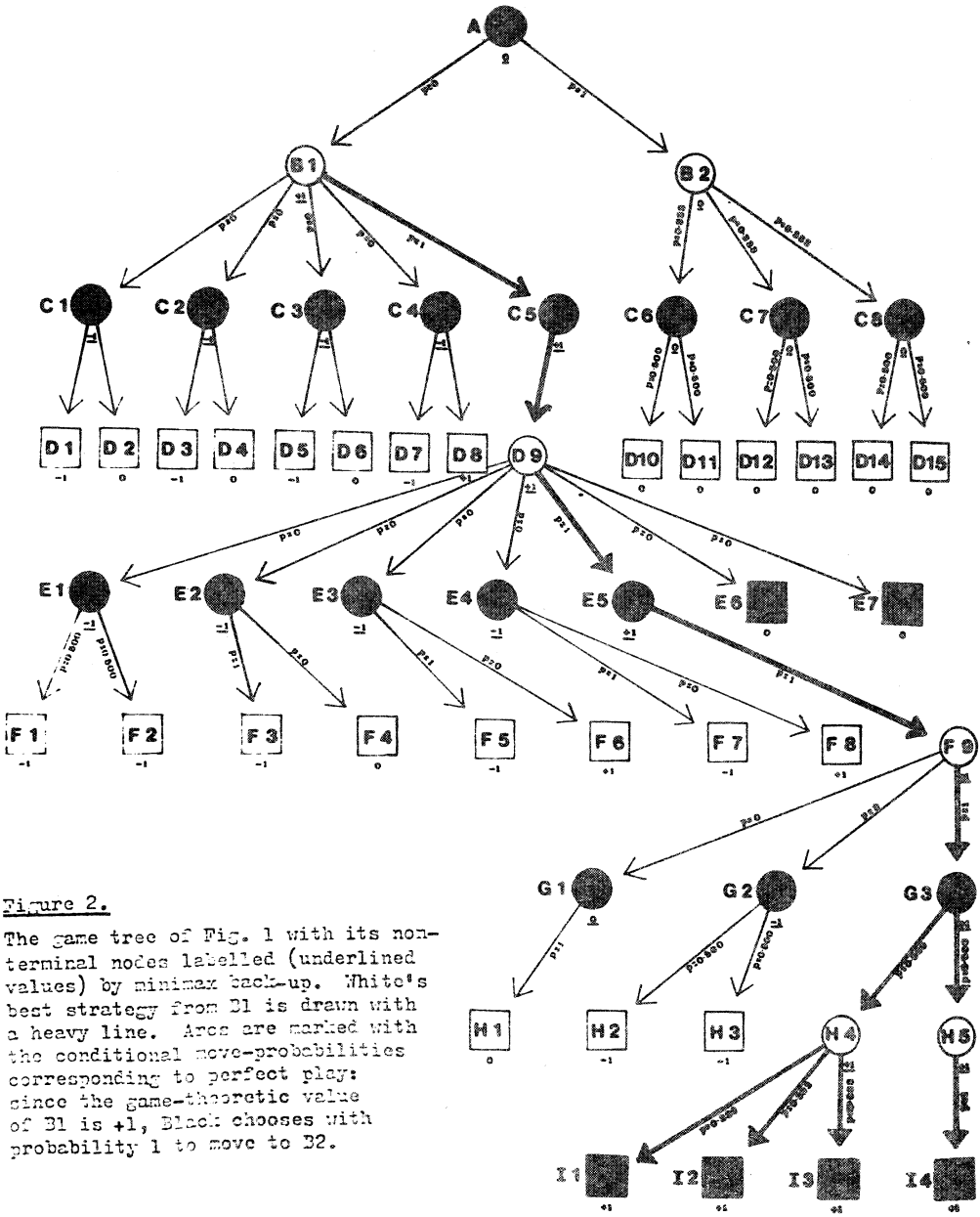


Figure 2.

The game tree of Fig. 1 with its non-terminal nodes labelled (underlined values) by minimax back-up. White's best strategy from B1 is drawn with a heavy line. Arcs are marked with the conditional move-probabilities corresponding to perfect play: since the game-theoretic value of B1 is +1, Black chooses with probability 1 to move to B2.

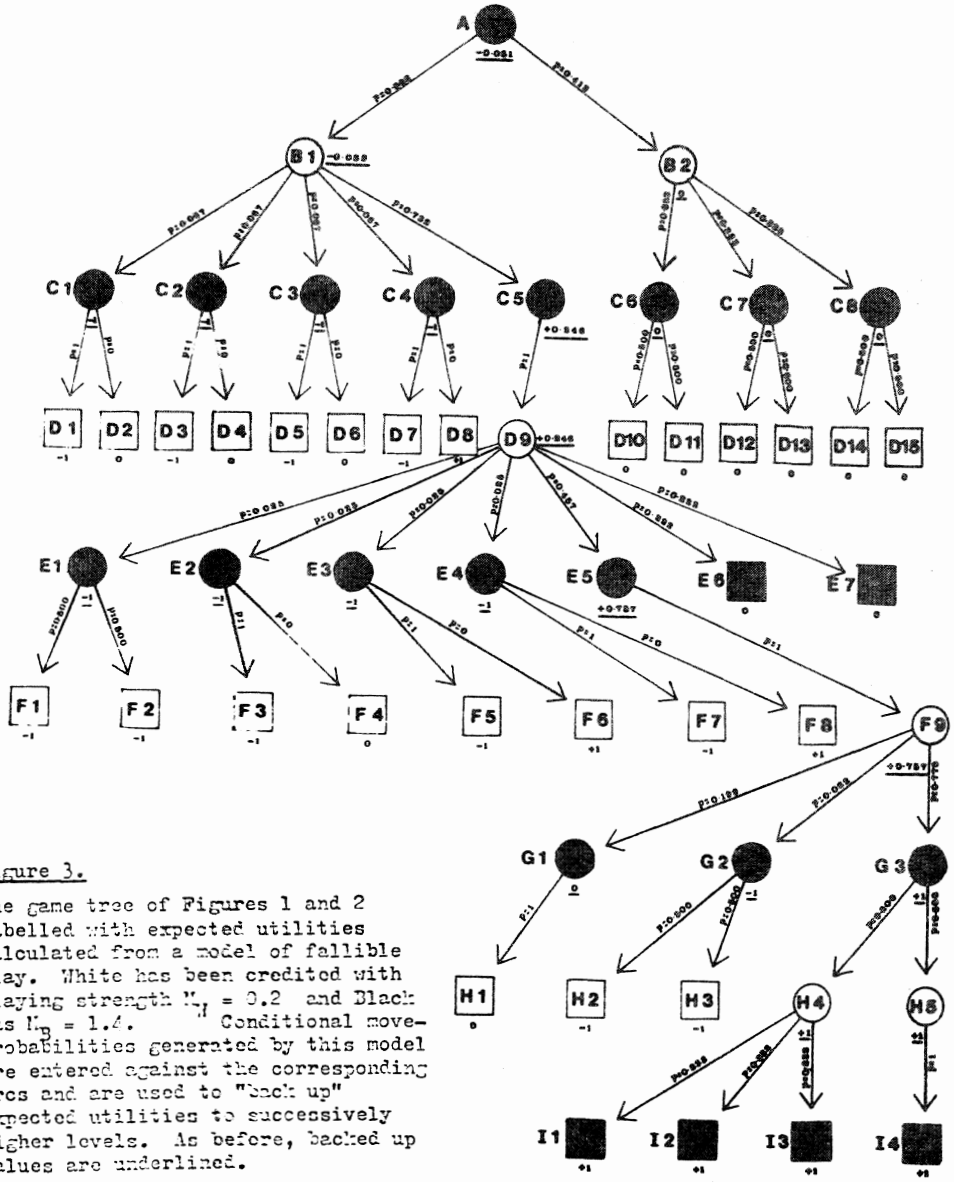


Figure 3.

The game tree of Figures 1 and 2 labelled with expected utilities calculated from a model of fallible play. White has been credited with playing strength $W_f = 0.2$ and Black has $W_f = 1.4$. Conditional move-probabilities generated by this model are entered against the corresponding arcs and are used to "back up" expected utilities to successively higher levels. As before, backed up values are underlined.

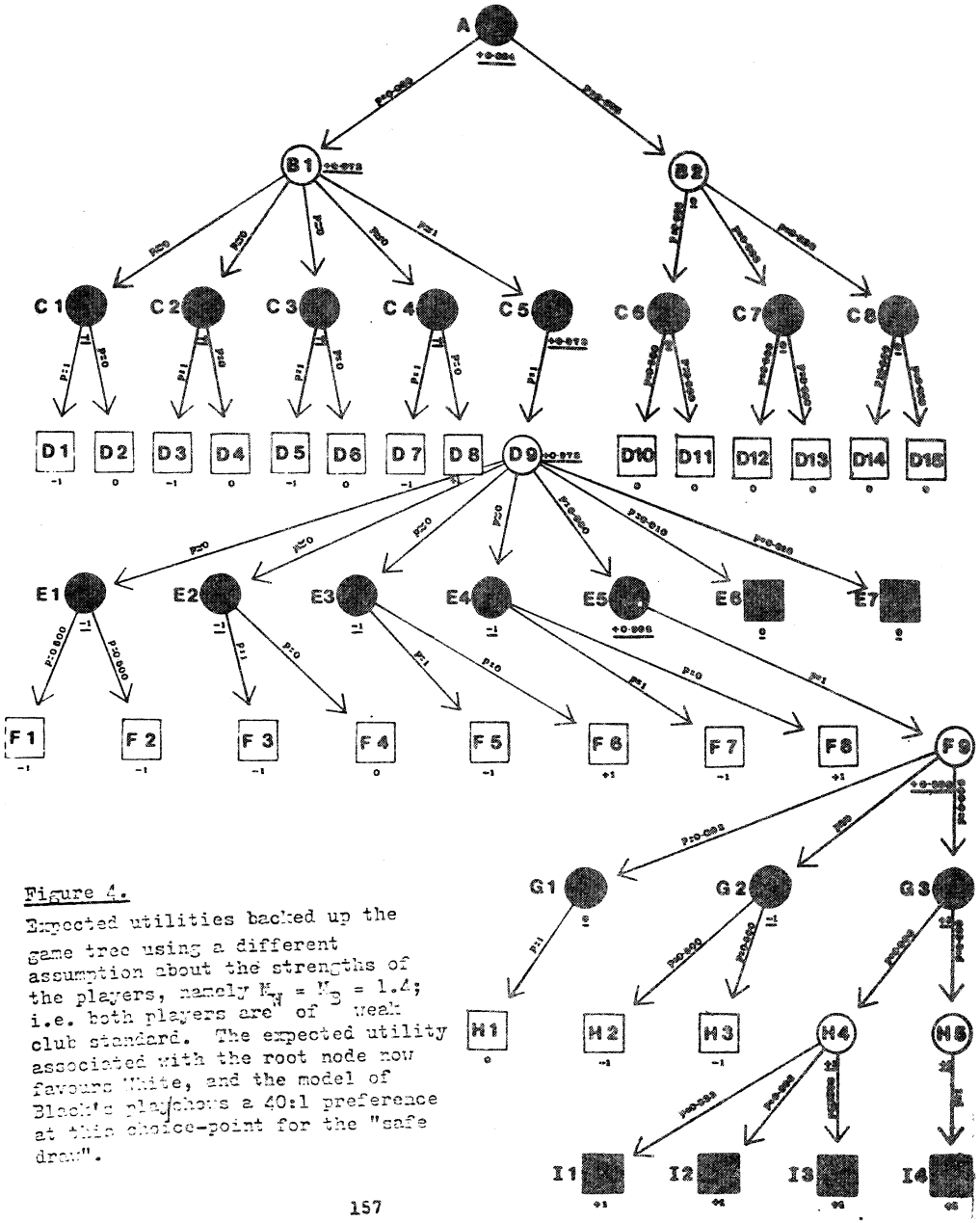


Figure 4.

Expected utilities backed up the game tree using a different assumption about the strengths of the players, namely $M_W = M_B = 1.4$; i.e. both players are of weak club standard. The expected utility associated with the root node now favours White, and the model of Black's play shows a 40:1 preference at this choice-point for the "safe draw".

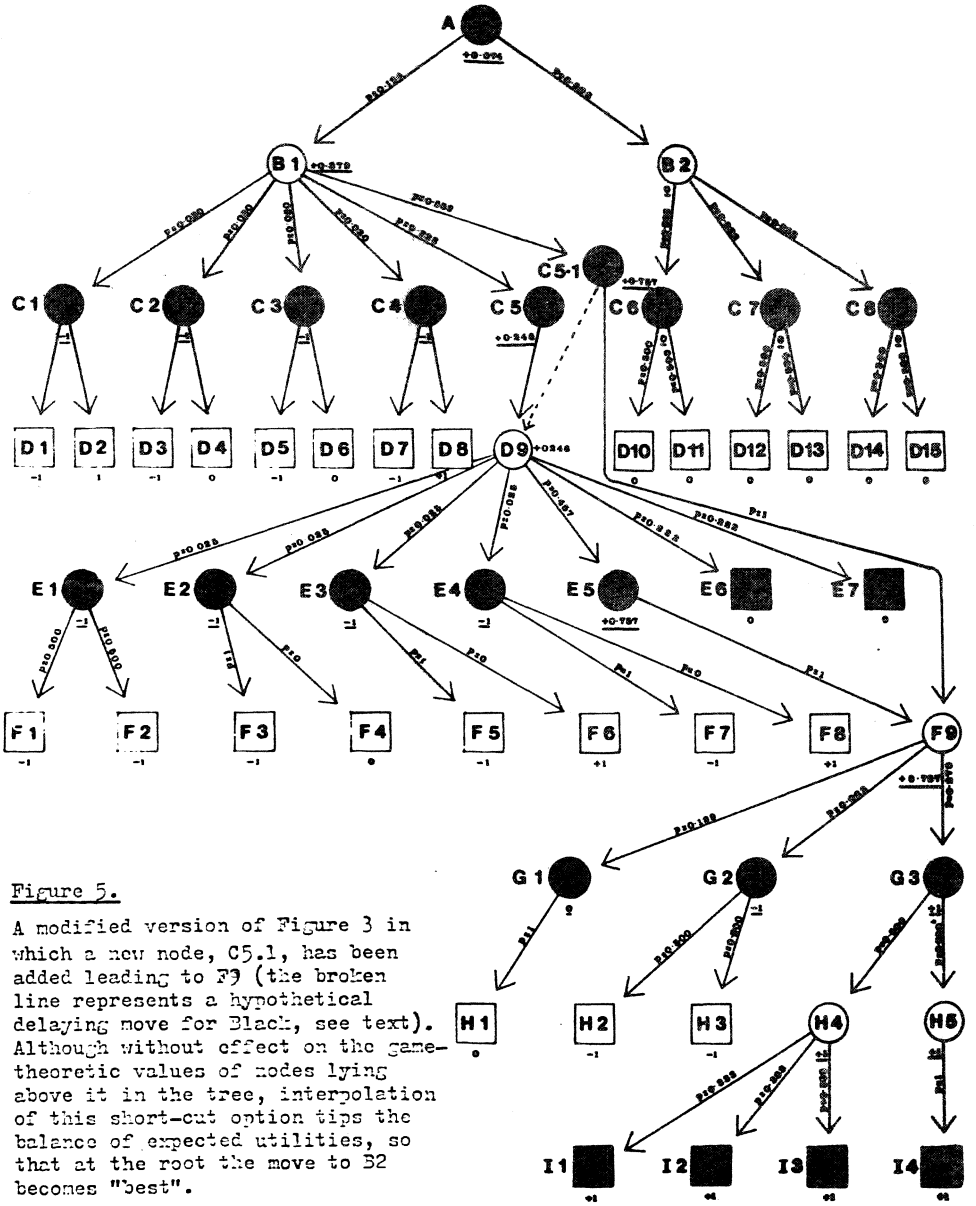
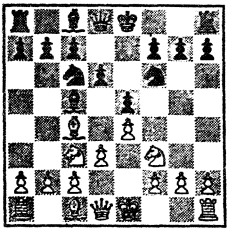
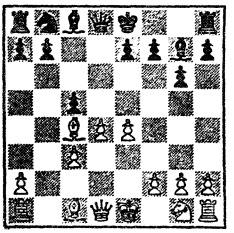


Figure 5.

A modified version of Figure 3 in which a new node, C5.1, has been added leading to F9 (the broken line represents a hypothetical delaying move for Black, see text). Although without effect on the game-theoretic values of nodes lying above it in the tree, interpolation of this short-cut option tips the balance of expected utilities, so that at the root the move to B2 becomes "best".



Giuoco Pianissimo



Gruenfeld Defence

Figure 6. Two chess positions illustrating the concept of "tension" (from Alexander, 1972). The upper position has low tension, and the lower has high tension (see text).

CORTICAL EMBODIMENT OF PROCEDURES

P. D. SCOTT,
DEPT. EXPERIMENTAL PSYCHOLOGY
UNIVERSITY OF SUSSEX.

Introduction.

Those of us who work on neural nets can hardly fail to be aware that many workers in other branches of artificial intelligence tend to regard such models as uninteresting, on two counts - They have very limited capabilities and, even if one were built which performed a complex task, it would not be clear how it did so. The present paper proposes a model of the structure of cerebral cortex which it is believed removes both of these objections. In addition the model retains one of the most important characteristics of earlier neural nets - it learns to do whatever it does.

The current lack of interest in neural nets is largely a reaction against the extravagant claims made about the potential of passive hierarchical networks (Fig 1) such as the Perception by some workers in this area. Such networks do perform certain tasks very well and furthermore they will learn to perform them. Other work has developed these capabilities to the maximum without claiming that they provided a complete model of the whole of perception (Uttley, 1,2). Minsky and Papert have pointed out the limitations inherent in all systems of this type (Minsky and Papert 3). If we wish to build models capable of more complex behaviour we must therefore either abandon neural nets altogether or else find some system more powerful than a passive hierarchy. Most workers chose the former course. I opted for the latter, partly because the former means abandoning the learning capabilities of neural networks but also because it seems reasonable to believe that the neurone is the functional sub-unit on which the brain is based.

Measuring Computational Power.

One way of deciding what the computational limitations of a device are is to demonstrate its equivalence to a specific class of automata. For example, one might prove a device is equivalent to a Turing machine and is thus able to perform any computation that any machine can. Alternatively one could demonstrate directly that certain classes of computation are outside the capabilities of the device. Both methods put rigorous limits on the range of things that can be done. Unfortunately they tell us very little about how

easily these things are done. Even the most persuasive of computer salesmen would have to buy many rounds before he could sell a machine as "capable of anything" on the grounds that it is equivalent to a Turing machine. He is much more likely to try and show that his machine is better than an existing machine whose 'power' is known to his potential customer. Similarly one almost always discusses the 'power' of a programming language by comparing its features with those of other languages. I propose to extend this idea and discuss neural nets in terms of what they have in common with programming languages. Certain difficulties will arise because neural nets operate in parallel while the programming languages considered operate sequentially. Nevertheless I think the comparison will be useful.

Neural Nets Compared With Other Programming Languages.

The instructions in the assembler language of any computer may conveniently be classified into three principal categories

1. Information-moving instructions
2. Transformational instructions
3. Control transfer instructions.

Suppose we were to try and write a program in such a language without using any of the instructions in category 3. This means we would not be allowed any form of jumping. Control would always pass to the next instruction in sequence. This is just the situation we find in the passive hierarchical neural net if we liken the computation of each layer to categories 1 and 2. Incoming patterns are processed by each layer in turn. There is no facility enabling control to be transferred elsewhere. We do not usually apply the term computer to a machine so limited. Indeed Babbage is credited with the invention of the computer largely because he appreciated the necessity of control transfer instructions and so introduced what we call a conditional jump. This means that control need not simply pass to the next instruction but may be transferred elsewhere depending on the outcome of a particular test. It would seem reasonable if we are to build more powerful neural nets to look for a way to introduce such conditional branching.

Control Transfer In A Neural Net.

A conditional branch instruction tests a specified predicate then transfers control to one place if it is true and to another if it is false. Our neural equivalent must incorporate these essential features. Since two distinct outputs will be required it will consist of at least two neurones. It will also require at least two inputs, one which transfers control to it and one which transmits the appropriate predicate. Such a unit is shown in Fig. 2.

The neurone labelled 'Do cell' output has a threshold level such that it acts as an 'and' gate on its inputs from the 'Try cell' and the predicate. The 'Do cell' output powerfully inhibits the 'Try cell'. If the predicate is true then control is transferred along output 1. If on the other hand it is false then control is transferred along output 2. The reasons for choosing this particular two-state structure will be demonstrated below. The whole is referred to as a 'Try-Do' unit.

It is unlikely that a programmer will be satisfied with the straightforward conditional branching capabilities. He will probably wish to write sub-routines. This means he will want not only to transfer control to another piece of the program but also to transfer it back again to the calling point when that piece has been executed. How would it be possible to incorporate such procedure calling facilities into the neural net?

Consider what happens to the Try-Do unit if the predicate is initially false but later becomes true. Control will then be transferred from output 2 to output 1. Thus if in some way the predicate indicated that the relevant 'procedure' had been completed the 'Try-Do' unit would provide a convenient procedure calling facility.

Some Examples Of Control Transfer Motor Control Networks

To demonstrate the power of the system we shall consider Try-Do units located in motor cortex. Output 1 of each unit will result in the animal performing a specific action.

In the first examples we shall consider a new born baby's behaviour at the breast (Piaget 4). Fig. 3A illustrates a single unit which exhibits a baby's behaviour

immediately after birth. It also introduces the graphical symbol for a Try-Do unit. The action caused by an output from the Do cell is sucking. The predicate which must be true before it can be done is the simple tactile sense of the nipple in the mouth. The unit is activated by the infant's hunger drive. There is no second output so if the nipple is absent the child has no way to remedy this.

The next example (Fig 3B) provides a solution. We have added another Try-Do unit which controls the action crying. In this case the child will cry in the absence of the nipple so that his mother may remedy this. It may incidentally appear that he will cry briefly anyway. However if try cells require the temporal summation of several input signals before they fire this will not occur. Notice that we transfer control to the crying unit if the nipple is absent and return control to the sucking unit when it is presented.

Fig. 3C illustrates another possible alternative. Here we introduce two Try-Do units controlling head movements to right and left. If the baby is at the breast but the nipple absent then he will call both head moving procedures but only the appropriate one will result in an action. Of course if the baby is not at the breast he must resort to crying again (Fig 3D).

A Try-Do unit may thus be viewed as a procedure. This procedure is called either by a basic drive or by one or more other Try-Do units. It can itself call one or more other units and also initiate a specific action. Obviously vastly more complex pieces of behaviour could be programmed in such a way. Another way of looking at such devices is to view them as machines which traverse directed graphs of sub-goals to reach a particular goal state i.e. a reduced basic drive. The directed graphs of examples 3A - 3D are all cycle free. Fig. 4 shows a network which attempts to cope with the "Hole-in the bucket" dilemma. The directed sub-goal graph of this is a cycle which can only be traversed if one of the goals is already satisfied. The network in Fig. 4 will do nothing if this is not so. As soon as it is it will execute just those actions needed to mend the bucket.

Since the examples shown have included diverging and

converging paths as well as a cycle we can clearly build a machine which will traverse any particular directed graph of subgoals. Notice incidentally that no actions are performed until a path has been found. In fact provided we have enough paper one can very easily build a Turing machine out of Try-Do units.

Try-Do Units In Perception.

So far we have considered the Try-Do unit as a functional unit of motor cortex. The cerebral neocortex however has a suggestive structural uniformity which leads us to the possibility that the same functional sub-unit might find equal application in sensory and cognitive processing. Lashley (11) has argued that the problem of serial order is central to the understanding of complex behaviour. In recent years psychologists have come to view perception not as a passive response taking its organisation from the external stimuli but as an active constructive and inferential process (e.g. Neisser (7), Bartlett (8)). A full discussion of the structural and functional evidence for Try-Do units will be found in Scott (9).

These facts suggest that it might be fruitful to apply the Try-Do units to perceptual tasks. To do this we add an additional cell which indicates if the Do output has just fired (Fig 5). This was not necessary in the examples of motor control because the consequences of an action on the world served such a role. (Nevertheless such additional cells may be useful in motor cortex to provide smoothly integrated movements). Fig. 6 shows a network for finding right angles of a certain orientation. Clearly such a procedure could be called by several higher procedures which found for example squares or right-angled triangles. Notice that the procedure has one input and one output. We could conceptually replace it with any procedure which finds right angles. Thus although the network to recognise a complex object might involve a great number of units it will always be possible to reduce it to relatively simple functional components which we may regard as procedures performing specific tasks.

Learning In Try-Do Networks.

Learning has gone out of fashion in A.I. In the first decade or so, the ghost of Lady Lovelace haunted all those whose programs only did what the programmer told them to so that learning was considered a measure of intelligence. In these days of metrication we use a new yardstick - how far the machine exploits knowledge of the world. With few exceptions A.I. workers seem to have shelved the learning issue. It is a problem to be tackled later.

My conviction is that you cannot fully understand how we do something until you understand how we come to do it. Neural nets models have often exploited the mechanism of varying synaptic weight in order to alter the net's structure as a consequence of experience. The procedural nets described above retain this feature in the following way.

A machine consists of a set of sensory predicate cells, a set of drives and a set of Try-Do units associated with specific actions. The pathways coupling Try and Do units are fixed. The others are adaptive and fall into two groups. The first consists of paths from all the drives to all the Try cells and paths from all the Try cells to all the other Try cells. These are the pathways along which control is transferred to sub-goal seeking procedures. The weight of such a pathway is made proportional to the Shannon mutual information between performance of the action and reduction of the drive. Thus in Fig. 3B the path between Try Suck and Try Cry is rewarded because crying results in nipple presentation which switches the Suck unit into state 'Do' and the output from 'Try Suck' is thus reduced. The other group of adaptive pathways are those from all the senses to all the Do cells. These are rewarded when a reduction in the input to the Try-Do unit correlates with activity in the Sense - Do pathway. Detailed discussion of the learning equations appears in Scott (9).

In this way a device whose behaviour is initially random gradually programs itself into a network of neural procedures.

Implementation.

Both fixed and adaptive pathway versions of several Try-Do networks have been demonstrated by simulation in Algol 68

on an ICL 1904 computer (Scott (9)).

Further Comparison With Programming Languages.

We now return to the comparison of neural nets with programming languages. The level we have brought neural nets up to is that of an assembler code with subroutine call facilities. There is still a long way to go before the exalted heights of Planner or Algol 68 and yet in certain ways we may be further on.

Many of the advances in programming languages since assembler code can be placed into two categories. The first, usually termed 'syntactic sugar', consists of more natural syntax in which to write programs. We have to sacrifice this facility and stay at the 'machine-code' level if we wish to study learning. Since we are repaid by having our programs self-writing this seems a reasonable bargain. If we abandon learning it is a relatively easy thing to provide 'syntactic sugar' in much the same way as Fortran does for assembler code. The other category of programming advance consists of enhancements of data structuring facilities. Here the analogy breaks down because the distinction between data and procedure is not clear in a neural net. The knowledge that any neural net has is embodied in the form of connections between units. In the examples we have discussed this has amounted to storing which procedures to call in a particular situation. This is of course a simplistic statement of the 'thesis of procedural embedding' (Hewitt (10)). It was to make possible such procedural embedding of knowledge that the language Planner was developed.

This is not the only resemblance between the proposed neural network and Planner. The control structure has the same power as that of a highly multi-processed Planner implementation. Consider for example, the 'Hole in the bucket' problem. If while seeking a stone we happen to find a straw of the right length we immediately backtrack and mend the bucket. There are however marked differences. In particular Planner possesses a data base of declarations and imperatives. Much of Planner is built around operations on this data base. Probably the fairest comparison would be to liken the Try-Do network to a machine on which the procedural

component of Planner could very easily be implemented.

Summary.

The computational power of neural networks has been measured by comparing them with programming languages. In the light of these comparisons a two-state neural unit has been proposed as a building block for cerebral cortex. It is argued that such units allow the cortical implementation of procedure calls. The resulting networks are self-programming.

This work forms part of a D.Phil. thesis to be submitted in 1974. It was carried out with the supervision of Professor A. M. Uttley, University of Sussex and the financial support of the S.R.C.

References

1. Uttley, A.M. 1966 Brain Research 2, 21.
2. Uttley, A.M. 1970 Journ. Theor. Biol. 27, 31.
3. Minsky, M. and Papert, S. 1969 Perceptrons.
4. Piaget, J. The Origins of Intelligence in Children.
7. Neisser, U. Cognitive Psychology.
8. Bartlett, F. 1932 Remembering.
9. Scott, P.D. Doctoral thesis to be submitted Summer 1974, University of Sussex.
10. Hewitt, C. Proc. 2nd International Joint Conference on Artificial Intelligence 1971, p. 167.
11. Lashley, K. The Problem of Serial Order in Behaviour Hixon Symposium, 1951.

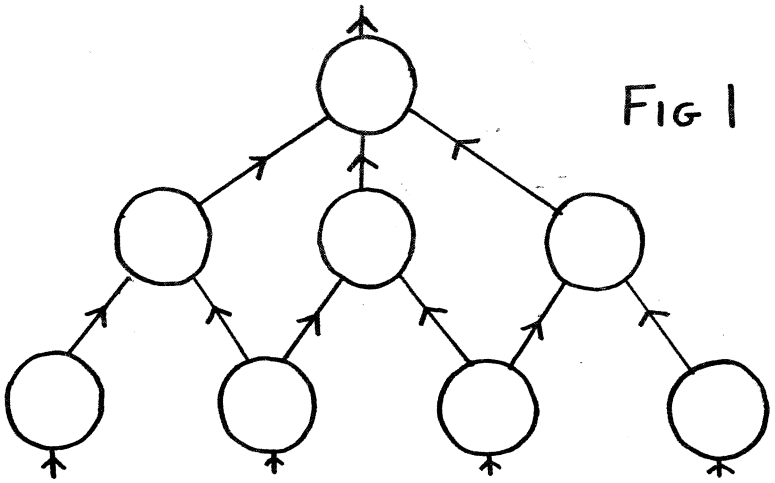

INPUT

FIG 1

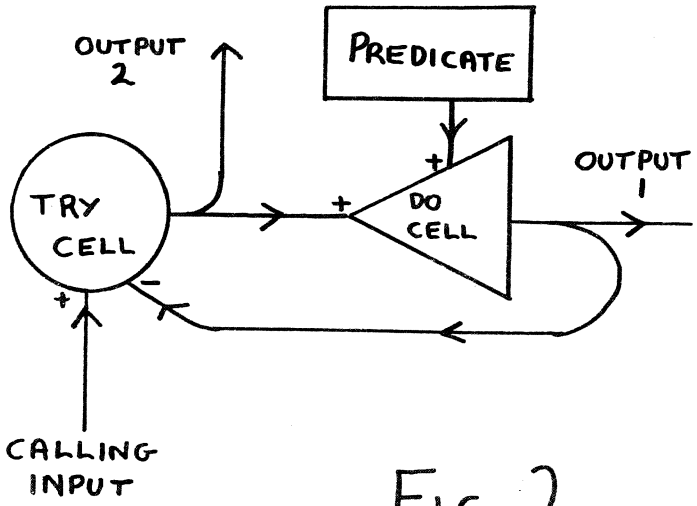
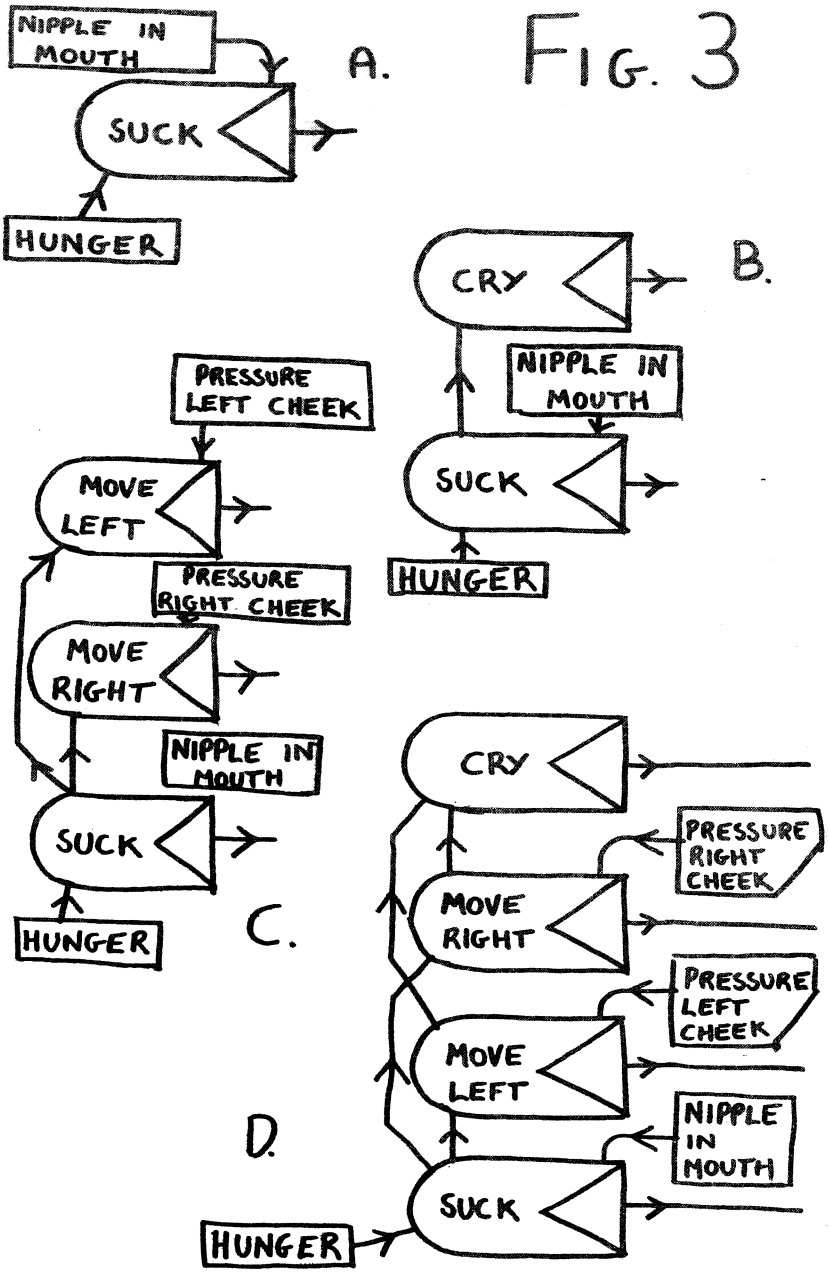

OUTPUT 2
PREDICATE
TRY CELL
DO CELL
OUTPUT 1
CALLING INPUT

FIG 2

FIG. 3


NIPPLE IN MOUTH
SUCK
HUNGER
A.
CRY
NIPPLE IN MOUTH
SUCK
HUNGER
B.
PRESSURE LEFT CHEEK
MOVE LEFT
PRESSURE RIGHT CHEEK
MOVE RIGHT
NIPPLE IN MOUTH
SUCK
HUNGER
C.
CRY
PRESSURE RIGHT CHEEK
MOVE RIGHT
PRESSURE LEFT CHEEK
MOVE LEFT
NIPPLE IN MOUTH
SUCK
HUNGER
D.

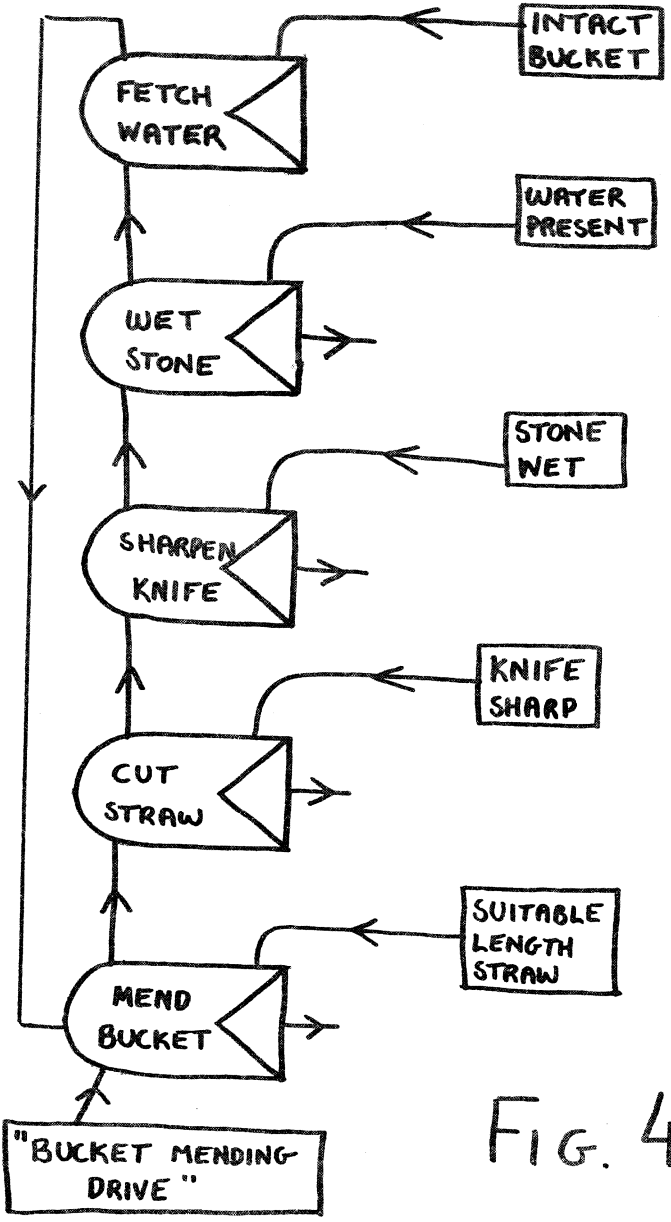

INTACT BUCKET
FETCH WATER
WATER PRESENT
WET STONE
STONE WET
SHARPEN KNIFE
KNIFE SHARP
CUT STRAW
SUITABLE LENGTH STRAW
MEND BUCKET
"BUCKET MENDING DRIVE"

FIG. 4

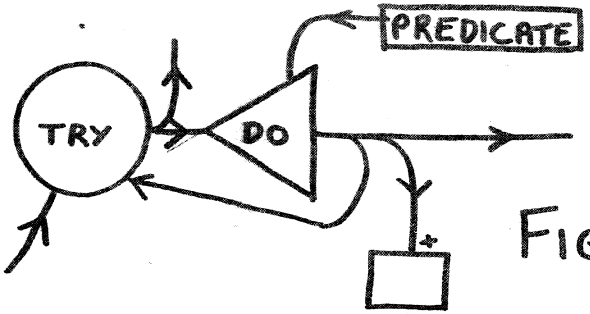

PREDICATE
TRY
DO
ADDITIONAL CELL
INDICATING THAT 'DO' CELL
HAS JUST FIRED

FIG. 5

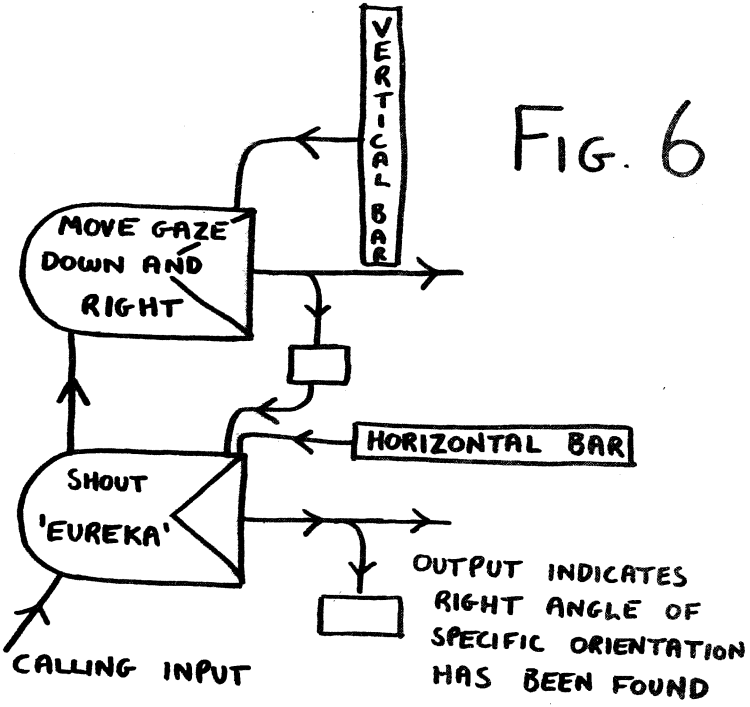

VERTICAL BAR
MOVE GAZE
DOWN AND
RIGHT
HORIZONTAL BAR
SHOUT
'EUREKA'
OUTPUT INDICATES
RIGHT ANGLE OF
SPECIFIC ORIENTATION
HAS BEEN FOUND
CALLING INPUT

FIG. 6

ON LEARNING ABOUT NUMBERS
(Some problems and speculations.)

By Aaron Sloman, School of Social Sciences, University of Sussex.

Abstract

The aim of this paper is methodological and tutorial. It uses elementary number competence to show how reflection on the fine structure of familiar human abilities generates requirements exposing the inadequacy of initially plausible explanations. We have to learn how to organise our common sense knowledge and make it explicit, and we don't need experimental data so much as we need to extend our model-building know-how.

oooooooooo

Introduction

Work in A.I. needs to be informed by accurate analysis of real human abilities if it is to avoid exaggerated claims, and excessive concern with toy projects. The reflective method advocated here has much in common with the approach of some linguists and with philosophical analysis of things we all know, as practised by Frege, Ryle, Austin, Wittgenstein and others. Philosophers' analyses are distorted by their preoccupation with old puzzles and paradoxes, and by their failure to think about the problems of designing symbol-manipulating (information processing) mechanisms. Psychologists, with a few exceptions (e.g. Piaget, Wertheimer, Heider) miss out on the analysis altogether, partly because they confuse it with introspection, partly because they are driven by the myth that to be a scientist is to collect new data, and partly because the technique is hard to learn and teach.

The analysis of elementary number competence, given below, is mixed up with speculation about mechanisms. A metaphor now taken for granted, though perhaps one day it will have to be abandoned, is that acquiring and using knowledge requires a memory containing vast numbers of "locations" at which symbols of some kind can be stored. They need not be spatial locations, since points in any symbolic space will do, such as frequencies of radio waves, or structures of molecules. So my remarks below about locations and addresses which identify them make no assumptions about the medium used, except that it provides enough locations at which symbols can be stored, including symbols which identify locations in memory, i.e. "pointers". I make no assumptions about the mechanisms making addressing possible except that explicit addressing takes a negligible amount of time. It makes no difference for present purposes whether the locations are brain cells, molecules, frequencies of brain waves, or parts of some spiritual mechanism. Physiology is irrelevant to many problems about the structures and functions of mental mechanisms.

The main problem to be discussed here is: What is elementary number competence and how is it possible? The first task is to make explicit our common sense knowledge about what sorts of things are possible. (Not laws of behaviour, but possibilities are what we first need to explain. There are very few laws of human behaviour, but very many possibilities.) By thinking about the mechanisms required to explain these possibilities we begin to reveal the poverty of most philosophical and psychological theories about the nature of mathematical concepts and knowledge: they hardly begin to get to grips with the details we all know.

Number concepts aren't simple things you either get or don't get, but complex extendable structures built up gradually. Reflecting on even the

simplest things we know children can learn, shows that children somehow cope with quite complex computational problems. Some of these problems are common to many forms of learning, others peculiar to numbers and counting. For any small subset of the problems, any competent programmer could suggest several possible explanatory mechanisms. The difficulty lies in understanding what sorts of mechanisms might not only solve a few specific problems, but could form part of a larger mechanism explaining much more. There is a serious need to extend our knowledge of varieties of possible computational mechanisms.

The particular problems to be discussed here are concerned with knowing number words, knowing action sequences (like counting), and enriching one's understanding of a previously learnt sequence. Many more questions will be asked than answered.

Knowing number words

A child learns to recognise sounds like "one", "two", "number" and "count". An untutored view is that repeated exposure causes the sound to be stored, so that new occurrences can be recognised by matching. Immediately all sorts of questions can be asked. In what form is the sound represented - is it analysed into recognisable fragments, such as phonemes? How are experiences selected as worth storing? How is a matching item found in the vast store of memories when a word is recognised? Is an index used for finding items, and if so how does a child know about index construction? How is the matching between perceived and stored items done? Are variations coped with by storing variant forms or by using a flexible matching procedure or both? In the first case, how is the equivalence of stored variants represented? Why is repeated hearing sometimes needed for learning - is it because the child needs to experiment with different modes of analysis, representation and matching, in order to find a good way of dealing with variations? If so, how are the experiments managed? Why is repetition sometimes not needed for learning? When a new word is learnt how is new storage space allocated? How is the ability to say the word represented? Is output controlled by the same representation as recognition? How are different output styles associated with the same item, such as English and French number names, Arabic and Roman numerals? Does being able to count in different languages/explicit storage of different sequences, or is the same sequence used with a decision about output style at each step? Or can both methods be used?

Using only 26 letters we can construct thousands of words. A frequently used principle of computation is that if a small set of symbols is available and quickly recognisable (e.g. because the set is small and the matching simple), then a very much larger efficiently usable set of symbols can be made available, each consisting of some combination of symbols from the small set. By imposing an arbitrary order on the original set of symbols, we can make processes of storing and retrieving large numbers of the new symbols look like fast parallel searches, for instance in the way we use alphabetical order to find a name in a directory without exhaustive search. Alternatively, recognition of a complex item may take the form of computing a description, using recognition of the components, as in parsing a sentence or finding the average of a set of numbers (constructive recognition). So perhaps analysis of words into syllables, phonemes, or other sub-structures is used by children to facilitate storage and recognition of the thousands of words they learn. This attributes to toddlers sophisticated but unconscious computational abilities (e.g. the construction and use of indexes, decision trees, parsers). What do we know about possible mechanisms?

It is often suggested that some of the remarkable efficiency of human memory could be explained by a content-addressable store, i.e. a large collection of storage units each capable of comparing its contents with a broadcast pattern, and shouting "Here it is" to a central processor. However, this leaves problems about explaining our ability to cope with items varying enormously in size and complexity, such as letters, words, phrases, sentences, poems, plays, the number sequence, etc., and our ability to retrieve on the basis of elaborate inferences rather than simple matches: e.g. "What's the smallest three-digit number which rhymes with 'heaven' and contains a repeated digit?". The central processor would need to be able to transform questions into forms likely to produce responses from relevant storage units. This requires some kind of index or catalogue of the contents of those units, which would make their content-addressability redundant! Most of this paper is concerned with problems of indexing.

Associations between learnt items

Merely being able to tell whether an item has been met before is not of much use. More must be known about it: such as how to produce it, in what forms it may be experienced, that it is a word, that it belongs to a certain syntactic class, that it has certain uses, that it is one of a group of words with related meanings or uses (a semantic field), that various objects and procedures are associated with it, and so on. Associationist psychologists and empiricist philosophers are obviously right in claiming that much knowledge depends on learnt associations. But they have been so concerned with the external conditions for establishing such associations that they have hardly begun to think about the problems of how such knowledge might be represented, stored and manipulated so as to be accessible, usable, and if necessary modifiable. (Explanations which convince one's colleagues are sometimes seen to be inadequate only as a result of attempting to design a mechanism actually able to do these things.)

Any one item may have to be associated with very many others. The word "word" is somehow linked to thousands of instances, and the item representing one's home town linked to very many facts known about that town. Similarly, we expect children to pick up many facts about an individual number, such as that it is a number, that it is used in counting, what its successor is, what its predecessor is, whether it is odd or even, whether it is prime and if not then what its factors are, which pairs of numbers add up to it, the result of adding or multiplying it with various others, how to say it, how to write it, how to recognise it when said or written in various styles, how to bypass counting by recognising spatial patterns corresponding to it, what it can be used for, how to count forwards from it, how to count backwards from it, where it lies in relation to various "landmarks" in the number sequence, and so on. (See figure 1.) Why should we expect children to pick up so many associations? The process of building up those associations is a long one and involves many mistakes which get corrected. An explanatory theory must specify a mechanism which is not merely able to hold the finished structure in an efficiently accessible form, but is also capable of explaining how such structures can be built up, how they are modified, how they are used, etc. I do not believe educational psychologists have the foggiest notion of what such a mechanism might be like. Yet gifted teachers have some intuitive grasp of how it works.

Take the question "What's after three?". The problem is not merely to find something associated with "three" and "after". Besides "four", "two" will be associated with them, and so may lots of pairs of numbers be, e.g. pairs N and K for which it is known that N is K after three: five is

two after three. So getting to the required association requires the ability to analyse the question (which may be ambiguous) and use the analysis to control the search for relevant links in the store of associations. (E.g. in figure 1, find the node representing three then search for a link from it labelled "successor". Do children learn to translate the original question into this kind of internal procedure? How?)

There are many ways in which associations can be stored, and different structures require different procedures for their use. A common method in computing is to use "property-lists" or "association-lists", as in figure 2, which shows a chain of links where each link contains two storage cells treated as an association by the memory mechanism. A chain may be attached to some item, e.g. to the concept "numbers", and related items are "hung" from the chain by means of pointers giving their addresses. As figure 3 shows, the items hung from the chain may themselves be associations, corresponding to the labelled links of figure 1. Thus in the context of the chain attached to "three", there is an association between "predecessor" and "two", whereas in a chain attached to "four" (not shown) there would be an association between "predecessor" and "three". Associations are relative to context.

Stored structures are not enough. Procedures are required for creating and finding associations in them. For instance, the following procedure will generate a search down a chain starting at LINK, looking for an association of type LABEL, in a structure like figure 3, and will return the associated item as its result.

```
PROCEDURE FINDASSOC (LINK,LABEL);
   WHILE ( HD(HD(LINK)) ≠ LABEL ) REPEAT
     (ASSIGN  TL(LINK) TO LINK);
   RETURN(TL(HD(LINK)));
END
```

So FINDASSOC(THREE, TYPE), yields a pointer to NUMBER as its result, in figure 3. A more complex procedure is required for adding a new association: it will have to get a free link (how?) and insert it at a suitable place in the chain, with its HD pointing to the new association and its TL pointing to the next link in the chain, if any. If children do anything like this to store and use associations, then how do they build up such chains, and how do they come to know the procedures for finding required associations? Are these inborn mechanisms? Clearly not all procedures for getting at stored information are innate. For instance, children have to learn how to count backwards or answer "What's before 'four'?" even though they may already know the order of the numbers. More on this later.

Learning a sequence

In this paper I shall not consider the more advanced stage where a child grasps a rule for generating indefinitely many number names, e.g. using decimal notation. An earlier stage involves learning to recognise not only isolated words, but also a sequence "one", "two", "three", etc. This is common to many things children learn. Some learnt sequences are made up of already meaningful parts which combine (how?) to form a new meaningful whole, like "Mary had a little lamb...", whereas other sequences, like the alphabet and numbers used in early counting games, are arbitrary, when first learnt. Sequences with varying amounts of significant structure include: the days of the week, the letters used to spell a word, the sounds in a spoken word, the sequence of intervals in a song, the steps required to assemble a toy, routes frequently travelled, recipes, and various games and rituals. An adequate explanation of how the simple and arbitrary

sequences are learnt, or stored or produced should also be part of an explanation of the ability to cope with more complex structures containing simple sequences as parts, such as nursery rhymes which have many levels of structure, and action procedures which, besides simple sequences, also contain loops, conditional branches, sub-procedures, gaps to be filled by decision-making at execution time, and other forms of organisation.

All this points to the old idea (compare Miller, et al.) that human abilities have much in common with computer programs. But further reflection on familiar facts shows that programs in the most common programming languages don't provide a rich enough basis for turning this from a thin metaphor into an explanatory theory. For instance, people can excute unrelated actions in parallel. Moreover, they apparently don't require their procedures to have built-in tests to ensure that conditions for their operation continue to be satisfied, with explicit instructions about what to do otherwise, like instructions for dealing with the end of a list. All sorts of unpredictable things can halt a human action at any stage (like learning one's house is on fire) and a decision about what to do can be taken when the interruption occurs, even if no explicit provision for such a possibility is built into the plan or procedure being executed. These points suggest that models of human competence will have to use mechanisms similar to operating systems for multi-programmed computers. For instance, an operating system can run a program, then interrupt it when some event occurs even if the program makes no provision for interruption. Similarly, if something goes wrong with the running of the program, like an attempt to go beyond the end of a list, the program breaks down, but the operating system or interpreter which runs the program can decide what to so, e.g. send a message to the programmer, so that there is not a total breakdown. Of course, the operating system is just another program. So the point is simply that to make the program metaphor fit human abilities, we must allow not merely that one program can use another as a subroutine, but that some programs can execute others and control their execution, in a parallel rather than a hierarchic fashion. (These arguments are familiar to many people in A.I.)

In counting objects, a child has to be able to generate different action sequences in parallel, keeping them in phase. Thus the process of saying number names, controlled by an internal structure, and the process of pointing in turn at objects in some group, controlled by the external structure, have to be kept in phase. In a suitable programming language one could keep two processes in phase by means of a procedure something like

```
PROCEDURE COEXECUTE (PROCESS1, PROCESS2, STOPPING-CONDITION);
 START:
  STEP(PROCESS1);
  STEP(PROCESS2);
  IF NOT(STOPPING-CONDITION) THEN GOTO START;
END
```

Unfortunately, this is not an acceptable model in view of the familiar fact that children (and adults doing things in parallel) sometimes get out of phase when counting and (sometimes) stop and correct themselves. This suggests that keeping the two sequences in phase is done by a third process something like an operating system which starts the processes at specified speeds, but monitors their performance and modifies the speeds if necessary, interrupting and perhaps restarting if the sequences get out of phase, which would be impossible with the procedure COEXECUTE. It is as if we could write programs something like:

```
PROCEDURE RUNINPHASE(PROCESS1, PROCESS2);
DO IN PARALLEL (a) to (d):
   (a) RUN PROCESS1;
   (b) RUN PROCESS2;
   (c) IF PROCESS1 AND PROCESS2 BEGIN TO GET OUT OF PHASE THEN
         MODIFY SPEED OF PROCESS1 OR PROCESS2 TO KEEP IN PHASE;
   (d) IF PROCESS1 AND PROCESS2 GET OUT OF PHASE THEN RESTART THEM;
END
```

The computational facilities required for this kind of thing are much more sophisticated than in COEXECUTE and are not provided in familiar programming languages. (Monitoring interactions between asynchronous parallel processes may be an important source of accidental discoveries (creativity) in children and adults.)

Further, the child has to be able to apply different stopping conditions for this complex parallel process, depending on what the task is. So it should be possible for yet another process to run the procedure RUNINPHASE, watching out for appropriate stopping conditions. For instance, when the question is "How many buttons are there?" use "No more buttons" as main stopping condition, whereas in response to a request "Give me five buttons", use "Number five reached" as main stopping condition. I say main stopping condition, because other conditions may force a halt, such as getting out of phase or running out of numbers or (in the second case) running out of buttons. How do children learn to apply the same process with different stopping conditions for different purposes? How is the intended stopping condition plugged into the process? This would be trivial for a programmer using a high-level language in which a procedure (to test for the stopping condition) can be given as a parameter to another procedure - but do children have such facilities, or do they use mechanisms more like the parallel processes with interrupt facilities described here? These parallel mechanisms might also explain the ability to learn to watch out for new kinds of errors. E. g. having learnt to count stairs where there is no possibility of counting an item twice, learning to count buttons or dots requires learning to monitor for repetition and omission. There are many ways this could be organised.

If we consider what happens when a child learns to count beyond twenty, we find that a different kind of co-ordination between two sequences is required, namely the sequence "one, two, three ... nine" and the sequence "twenty, thirty, ... ninety". Each time one gets round to "nine" in the first sequence one has to find one's place in the second sequence so as to locate the next item. A programmer would find this trivial, but how does a child create this kind of interleaving in his mind? And why is there sometimes difficulty over keeping track of position in the second sequence "... fifty eight, fifty nine, ... um .. er, thirty, thirty one..."? Clearly this is not a problem unique to children: we all have trouble at times with this sort of book keeping. But how is it done when successful? And what kind of mechanism could be successful sometimes yet unsuccessful at others? My guess is that human fallibility has nothing to do with differences between brains and computers as is often supposed, but is a direct consequence of the sheer complexity and flexibility of human abilities and knowledge, so that for example there are always too many plausible but false trails to follow. When computers are programmed to know so much they will be just as fallible, and they'll have to improve themselves by the same painful and playful processes we use.

We have noted a number of familiar aspects of counting and other actions which suggest that compiled programs in commonly used programming

languages don't provide a good model for human abilities. A further point to notice is that we not only execute our procedures or programs, we also build them up in a piecemeal fashion (as in learning to count), modify them when they seem inadequate, and examine them in order to anticipate their effects without execution. We can decide that old procedures may be relevant to new problems, we can select subsections for use in isolation from the rest, and we may even learn to run them backwards (like learning to count backwards). This requires that besides having names and sets of instructions, procedures need to be associated with specifications of what they are for, the conditions under which they work, information about likely side-effects, etc. The child must build up a catalogue of his own resources. Further, the instructions need to be stored in a form which is accessible not only for execution but also for analysis and modification, like inserting new steps, deleting old ones, or perhaps modifying the order of the steps. Such examination and editing cannot be done to programs as they are usually stored.

List structures in which the order of instructions is represented by labelled links rather than implicitly by position in memory would provide a form of representation meeting some of these requirements (and are already used in some programming languages). Thus, figure 2 can be thought of either as a structure storing information about number names (an analogical representation of their order), or else as a program for counting. The distinction between data structures and programs has to be rejected in a system which can treat program steps as objects which are related to one another and can be changed. We explore some consequences of this using counting as an example.

Learning to treat numbers as objects with relationships

There are several ways in which understanding of a familiar action sequence may be deficient, and may improve. One may know a sequence very well, like a poem, telephone number, the spelling of a word, or the alphabet, yet have trouble reciting it backwards. One may find it hard to start from an arbitrary position in a sequence one knows well, like saying what comes after "K" in the alphabet, or starting a piano piece in the middle. But performance can improve. A child who counts well may be unable easily to answer "What comes after five?". Later, he may be able to answer that question, but fail on "What comes before six?", "Does eight come earlier or later than five?" and "Is three between five and eight?". He doesn't know his way about the number sequence in his head, though he knows the sequence. Further, he may understand the questions well enough to answer when the numbers have been written down before him, or can be seen on a clock. (There are problems about how this is learnt, but I'll not go into them.) Later, the child may learn to answer such questions in his head, and even to count backwards quickly from any position in the sequence he has memorised. How? To say the child "internalises" his external actions is merely to label the problem: moving back and forth along a chain of stored associations is quite a different matter from moving up and down staircases or moving one's eye or finger back and forth along a row of objects.

There are at least two kinds of development of knowledge about a stored structure (which may be a program), namely learning new procedures for doing things with the structure, and extending the structure so as to contain more explicit information about itself. The former is perhaps the more fundamental kind of development of understanding, while the latter is concerned with increased facility. A very simple procedure enables a chain like that in figure 2 to be used to generate a sequence of actions, for example:

```
PROCEDURE SEQUENCE (LINK);
 START:
  OUTPUT(HD(LINK));
  ASSIGN TL(LINK) TO LINK;
 GOTO START;
END
```

or

```
PROCEDURE SEQUENCE (LINK);
 OUTPUT(HD(LINK));
 SEQUENCE(TL(LINK));
END
```

Going down the chain starting from a given link is thus easy, and a procedure to find the successor of an item would use a similar principle. But answering "What's before item X?" is more sophisticated, since on getting to a particular location (e.g. the link whose HD points to X), one does not find there any information about how one got there, so the last item found must be stored temporarily. One method is illustrated in the following procedure.

```
PROCEDURE PREDECESSOR (X,LINK);
 LOCAL VARIABLE TEMP; ASSIGN "NONE" TO TEMP;
  START:
   IF HD(LINK)=X THEN RETURN(TEMP)
    ELSE ASSIGN HD(LINK) TO TEMP AND ASSIGN TL(LINK) TO LINK AND
    GOTO START;
END
```

How could a child learn to create a procedure like this or the more elegant versions a programmer would write? Does he start with something more specialised then somehow design a general method which will work on arbitrary chains? Perhaps it has something to do with manipulating rows of objects and other sequences outside one's head, but to say this does not give an explanation, since we don't know what mechanisms enable children to cope with external sequences, and in any case, as already remarked, chains of associations have quite different properties. For a child to see the analogy would require very powerful abilities to do abstract reasoning. Maybe the child needs them anyway, in order to learn anything.

In any case, merely being able to invent procedures like PREDECESSOR is not good enough. For some purposes, such as counting backwards quickly, we want to be able to find the predecessor or successor of an item much more quickly than by searching down the chain of links until the item is found. If a child knew only the first four numbers, then he could memorise them in both directions, building up the structure of figure 4 instead of figure 2. Notice that this use of two chains increases the complexity of tasks like "Say the numbers", or "What's after three?", since the right chain has to be found, while reducing the complexity of tasks like "Say the numbers backwards," and "What's before three?" However, when a longer sequence had been learnt, this method would still leave the need to search down one or other chain to find the number N in order to respond to "What's after N?", "What's before N?", "Count from N", "Count backwards from N", "Which numbers are between N and M?", etc., for there is only one route into each chain, leading to the beginning of the chain. For instance, when one has found the link labelled X (figure 4) one knows how to get to the stored representation of "three", but it is not possible simply to start from the representation of "three" to get to the links which point to it in the two chains. So we need to be able to associate with "three" itself information about where it is in the sequence, what its predecessor is, what its successor is, and so on.

A step in this direction is shown in figure 5, where each number name is associated with a link which contains addresses of both the predecessor and the successor, like the link marked V, associated with "two". The information that the predecessor is found in the HD and the successor found

in the TL would be implicit in procedures used for answering questions. However, if one needed to associate much more information with each item, and did not want to be committed to having the associations permanently in a particular order, then it would be necessary to label them explicitly, using structures like those in figure 1 and figure 3, accessed by a general procedure like FINDASSOC, defined previously.

To cut a long story short, the result of explicitly storing lots of discoveries about each number, might be something like figure 6, which is highly redundant. The structure may look very complex, yet using it to answer questions requires simpler procedures than using, say figure 2, for, having found the link representing a number, one can then find information associated with that number by simply following forward pointers from it, e.g. using FINDASSOC, whereas in figure 2 or 5 finding the predecessor and successor of a number requires using two different procedures, and each requires a search down a chain of all the numbers to start with. Of course, a structure like figure 6 provides simple and speedy access at the cost of using up much more storage space. But in the human mind space does not seem to be in short supply!

If an item in a structure like figure 6 has a very long chain of associations, it might be preferable to replace the linear chain with a local index to avoid long searches. This would require the procedure FINDASSOC to be replaced by something more complex. Alternatively, one could easily bring a link to the front of the chain each time the association hanging from it is used: this would ensure that most recently and most frequently used information was found first, without the help of probabilistic mechanisms.

Notice that in a structure like this, normal "part-whole" constraints are violated: information about numbers is part of information about "three", and vice versa. So by using pointers (addresses) we can allow structures to share each other. In a rich conceptual system circular definitions will abound. If knowledge is non-hierarchic, as this suggests, then perhaps cumulative educational procedures are quite misguided. Further, this kind of structure does not need a separate index or catalogue specifying where to look for associations involving known items, for it acts as an index to itself, provided there are some ways of getting quickly from outside the structure to key nodes, like the cells containing "three" and "number". (This might use an index, or content addressable store, or indexing tricks analogous to hash coding, for speedy access.) The use of structures built up from linked cells and pointers like this has a number of additional interesting features, only a few of which can be mentioned here. Items can be added, deleted, or rearranged merely by changing a few addresses, without any need for advance reservation of large blocks of memory or massive shuffling around of information, as would be required if items were stored in blocks of adjacent locations. The same items can occur in different orders in different structures which share information (see figure 4 for a simple example). Moreover, the order can be changed in one sequence without affecting another which shares structure with it. For instance, in figure 4 the addresses in links W, X, Y, and Z can be changed so as to alter the order of numbers in chain labelled "reverse" without altering the chain labelled "forward".

As we saw in connection with figure 2, when the rest of the mechanism is taken for granted, a structure of the kind here discussed looks like a program for generating behaviour, but when one looks into problems of how the structure gets assembled and modified, how parts are accessed, how different stopping conditions are applied, etc., then it looks more like a

data structure used by other programs. If the distinction between programs and data structures evaporates, then don't some A.I. slogans about procedural knowledge have to be retracted, or at least clarified? (Compare Hewitt 1971.).

Conclusion

Further simple-minded reflection on facts we all know reveals many gaps in the kinds of mechanisms described here. For instance, very little has been said about the procedures required for building, checking, modifying, and using a structure like figure 6. Nothing has been said about the problems of perception and conception connected with the fact that counting is not applied simply to bits of the world but bits of the world individuated according to a concept (one family, five people, millions of cells - but the same bit of the world counted in different ways). Nothing has been said about recognition of numbers without explicit counting. Nothing has been said about how the child discovers general and non-contingent facts about counting, such as that the order in which objects are counted does not matter, rearranging the objects does not matter, the addition or removal of an object must change the result of counting, and so on. (Philosophers' discussions of such non-empirical learning are so vague and abstract as to beg most of the questions.) I cannot explain these and many more things that even primary school children learn. I don't believe that anybody has even the beginnings of explanations: only new jargon for labelling the phenomena.

I have offered all this only as a tiny sample of the kind of exploration needed for developing our abilities to build theoretical models worth taking seriously. In the process our concept of mechanism will be extended and the superficiality of current problems, theories and experiments in psychology and educational technology will become apparent.

Philosophers have much to learn from this sort of exercise too, concerning old debates about the nature of mind, the nature of concepts and knowledge, varieties of inference, etc. Consider answers they have given to the question "What are numbers?", namely: numbers are non-physical mind-independent entities (Platonists), numbers are perceivable properties of groups of objects (Aristotle?), numbers are mental constructions whose properties are found by performing mental experiments (Kant and Intuitionists), numbers are sets of sets, definable in purely logical terms (Logicists), numbers are meaningless symbols manipulated according to arbitrary rules (Formalists), they are whatever satisfy Peano's axioms (Mathematicians) or numbers are simply a motley of things which enter into a variety of "language" games" played by different people (Wittgenstein). (These descriptions are too brief for accuracy or clarity; for more detail consult books on philosophy of mathematics, e.g. Korner's.) Further work will show that each of these views is right in some ways, misleading in others, but that none of them gets near an accurate description of all the rich structure in our number concepts.

I believe the old nature-nurture (heredity-environment) controversy gets transformed by this sort of enquiry. The abilities required in order to make possible the kind of learning described here, for instance the ability to construct and manipulate stored symbols, build complex networks, use them to solve problems, analyse them to discover errors, modify them, etc., - all these abilities are more complex and impressive than what is actually learnt about numbers! Where do these abilities come from? Could they conceivably be learnt during infancy without presupposing equally powerful symbolic abilities to make the learning possible? Maybe the much

discussed ability to learn the grammar of natural languages (cf. Chomsky) is simply a special application of this more general ability? This question cannot be discussed usefully in our present ignorance about possible learning mechanisms.

Finally a question for educationalists. What would be the impact on primary schools if intending teachers were exposed to these problems and given some experience of trying to build and use models like figure 6 on a computer?

Acknowledgements

Some of these ideas were developed during tenure of a visiting fellowship in the Department of Computational Logic, Edinburgh. I am grateful to Bernard Meltzer and the Science Research Council for making this possible, and to colleagues in Edinburgh and at Sussex University for helping to remove the mysteries from computing. Alan Mackworth's criticisms of an earlier draft led to several improvements.

Bibliography

Austin, J. L. 'A plea for excuses', in his Philosophical Papers Oxford University Press, 1961, also in Philosophy of Action ed. by A. R. White, Oxford University Press, 1968.

Chomsky, N. Aspects of the Theory of Syntax, chapter 1. M.I.T. Press, 1965.

Frege, G. Philosophical Writings, translated by P. T. Geach and M. Black. Blackwell

Hewitt, C. 'Procedural embedding of knowledge in PLANNER', in Proc. 2nd IJCAI, British Computer Society, 1971.

Korner, S. Philosophy of Mathematics, Hutchinson, 1960.

Lindsay, P. H. and Norman, D. A. Human Information Processing, Academic Press 1972. Chapters 10 and 11 give a useful introduction to semantic networks.

Miller, G. A., Galanter, E. Pribram, K. M. Plans and the Structure of Behaviour, Holt Rinehart and Winston 1960.

Minsky, M. 'Form and Content in Computer Science', part 3, J.A.C.M. 1970.

Quillian, M. R. 'Semantic Memory' in Semantic Information Processing, ed. by M. Minsky, M.I.T. Press, 1968.

Ryle, G. The Concept of Mind, Hutchinson 1949. also Penguin Books.

Winston, P. Learning Structural Descriptions from Examples M.I.T. A.I. Lab. AI TR-321, 1970.

Wittgenstein, L. Philosophical Investigations, Blackwell, 1953. Remarks on the Foundations of Mathematics, Blackwell, 1956.

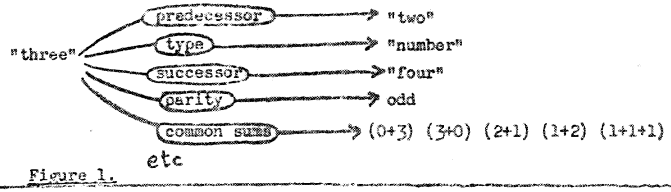
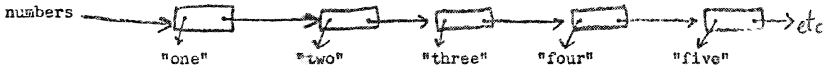


Figure 1.



The use of chains. Each link contains two cells, the first called HD the second TL, containing symbols which may be pointers to other locations. A cell points to a location in memory by containing an address of that location.

Figure 2.

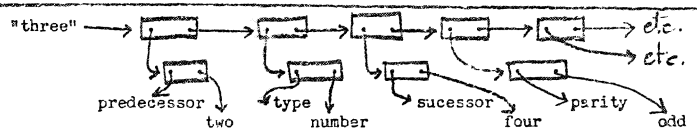


Figure 3. (This shows a translation of figure 1 using list structures.)

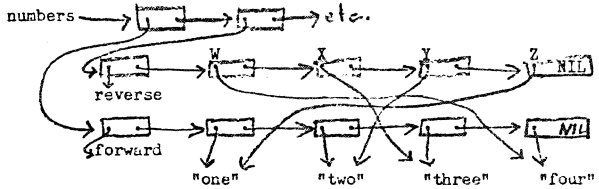
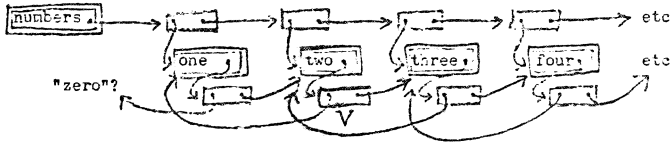


Figure 4. (The ability to count backwards from four.)



Here, writing a symbol inside a link represents the fact that that link may be addressed by means of the symbol, e.g. using an index, and the HD of the link contains a pointer back to the symbol.

Figure 5.

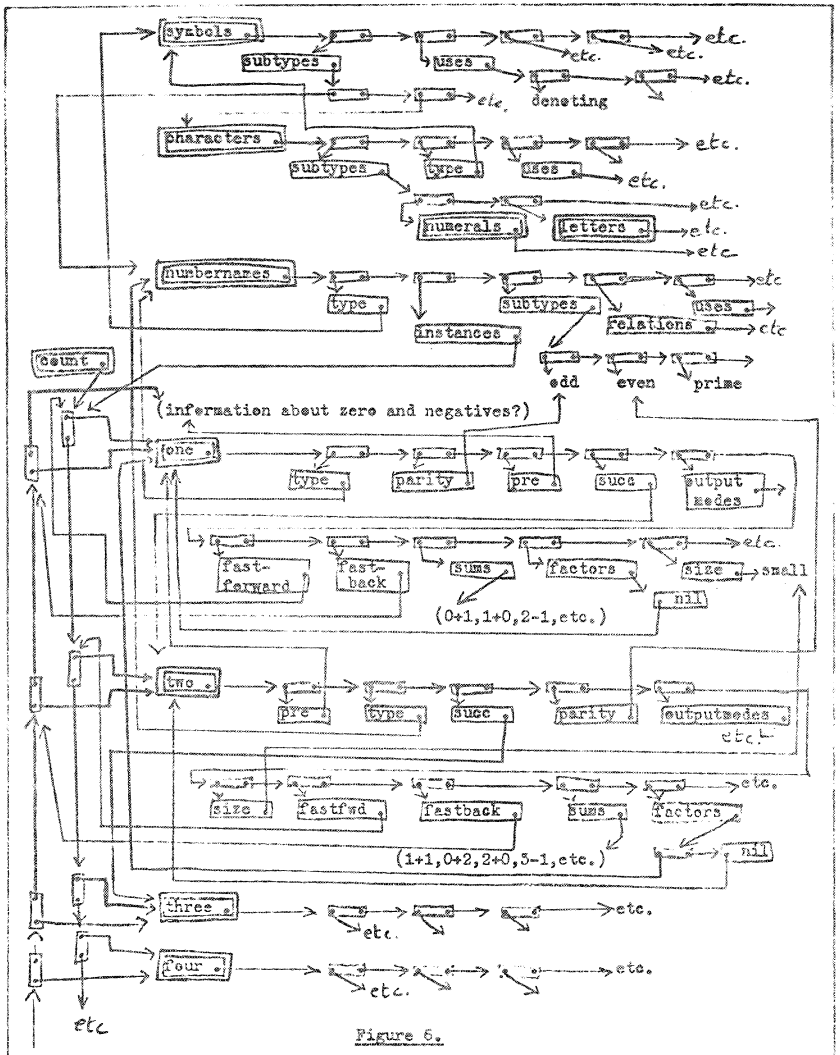


Figure 6.

This represents, with many oversimplifications and omissions, some of the information about the number sequence learnt by children. To save space and improve intelligibility, relations between numbers, number-names, and numerals (e.g. two, "two" and "2") have not been represented. Doubly-bored nodes are meant to be directly addressible from outside the network (compare figure 5). The two vertical chains on the left represent the ability to count forwards or backwards at speed, starting at any known number.

Towards a Programming Apprentice

Brian Smith
Carl Hewitt

Abstract

The Planner Project is constructing a Programming Apprentice to assist in knowledge based programming. We would like to provide an environment which has substantial knowledge of the semantic domain for which the programs are being written, and knowledge of the purposes that the programs are supposed to satisfy. Further, we would like to make it easy for the programmer to communicate the knowledge about the program to the Apprentice. The Apprentice is to aid in establishing and maintaining consistency of specifications, validating that modules meet their specifications, answering questions about behavioral dependencies between modules, and analyzing the implications of perturbations in modules and their specifications.

A tenet of the apprentice project is that programming is a multi-level activity: as well as writing code, programmers communicate in terms of comments and models. Our goal is to elucidate and formalize some of these interactions. The first level of description we have attacked is the level of abstract descriptions of what programs do, rather than how they do it. The contracts and intentions discussed in this paper are an attempt to embody this kind of knowledge in a formal and yet intuitive and useful way. A process known as *meta-evaluation* is presented which can justify why a program fulfills its contract. Further research is being carried out into the role of models, background knowledge, and commentary relating these different levels of description.

This work is presented using Actors, a semantic concept in which no active process is ever allowed to treat anything as an object; instead a polite request must be extended to accomplish what the activator desires.

Procedural Embedding of Specifications

To prove a thing is not enough;
you have to seduce people to accept it.
Nietzsche

We believe that procedures are a good formalism in which to write specifications for tasks, where by the *specifications* for a procedure we mean a statement of what the procedure is supposed to accomplish as opposed to how it does it. If you have a typical specification expressed in any formalism [e.g. the first order quantificational calculus], we believe that there is a procedure that elegantly and naturally tests to see if the specification is satisfied. Programming a task should therefore involve the creation of at least two processes: one that judges whether or not the task has been properly accomplished, and at least one that knows how to accomplish it.

There has been a great deal of research in the past few years [Floyd, Naur, King, Green, Manna, Waldinger, Hoare, Deutsch, Luckham, Good, Dijkstra, etc.] which has attempted to express specifications for procedures in the first order quantificational

calculus. Typically expressing the specifications for a procedure f in the quantificational calculus takes the form of attempting to find formulas P and R such that if P[x] is true then f(x) converges and R[x,f(x)] is true. We believe that the first order quantificational calculus is not in fact a very good language for writing specifications for procedures [particularly for ones involving parallelism, side-effects, and histories].

For example consider the problem of writing specifications for a time-sharing file system. We shall suppose that the system maintains a Track Usage Table which records whether or not each track on the disk is in use. In addition each user of the time-sharing system has a directory of the tracks that are used by each of his files. We wish to specify that no two files both attempt to use the same disk track and that the Track Usage Table is "always" consistent with the user directories.

It is not too difficult to write a procedure to check that the above condition is met at any given instant. We would like to develop a coherent methodology for substantiating that the procedures of the time-sharing system are such that the directories will never fail to pass the consistency check. To this end we have developed a technique called *meta-evaluation* which attempts to implement the process that good programmers go through when they "symbolically" execute their code in order to demonstrate to themselves that it meets its specifications. Currently we do not know how to translate the meta-evaluation process into a proof in the first order quantificational calculus. In fact we do not even know whether it is possible or not! The extent to which we should be concerned [in other than an academic sense] is moot. The meta-evaluation process [and the underlying Actor Induction principle] carry a good deal of conviction and deserve to be analyzed in their own right independently of any reduction to another formalism.

Contracts

When a routine or program is written, the programmer has a notion of what it does. One of the first types of comments to formalize is this description, which is like an advertisement to the world at large of a program's capabilities. Not only might it be useful to other parts of the system to have an abstract description of the routine and therefore not have to look at its definition, but with a careful statement of the conditions of applicability and a good description of the behavior, it may be possible to check the code to see whether it does what it claims to do.

We call such an advertisement a *contract:* a statement of "what" a program does under what conditions. No attempt is made to describe how the code achieves the desired behavior, nor is anything said about when or why this routine might be called. A contract references neither its code nor the world of its use. It is a generalization of a pre-requisite mechanism (with no limit on the complexity of pre-requisite computation possible) coupled with a parallel post-requisite mechanism.

A contract consists of roughly two things: a statement of in-coming assumptions which it expects (but has no reason to presume) to be true when it is called, and a statement of the conditions that will be obtained which it claims (presumably with some justification) will be achieved by invoking the contractor [the actor which fulfills the contract]. Many of our contracts are self-enforcing in the sense that they effectively specify the tests that have to be run to verify that any given invocation of the contractor has fulfilled the contract. There are several advantages in having contracts: contracts can be checked each time the associated routine is called; the routine can be

abstractly verified in some programming context (i.e. the out-going restrictions will be satisfied if the in-coming ones are satisfied), and the contract can be read by other routines who want to know what the routine does.

For example, the contract of a routine to clear off a block might be: if called with the name of a (flat) block, it will return only when there are no other blocks being supported by that block. The contract for a matrix inversion routine might be that if it is called with a regular real-valued matrix A, it will return a matrix B such that A*B = the identity matrix. The contract for a square root routine could require that the output times itself be equal to the input. An elevator's contract might be that if it is called by a person on some floor who wants to go to some other floor in that building, it will pick up the person in a "reasonable amount of time" and deliver her to her destination floor without changing direction. A match-maker's contract might be that if called with the names of two people it will return only when those two people are married.

Sometimes contracts seem difficult to express. Precisely stating the specifications of a time-sharing system including the requirements for protection, reliability in the face of hardware malfunctions, and level of performance under specified load is currently beyond the state of the art. Formalizing the requirement that an elevator pick up a person in a "reasonable amount of time" is difficult although formulations like "before passing any other floor twice in the same direction" capture some of the flavor of what is wanted. In some cases, such as for factorial, an abstract statement of what the routine returns seems possible only in terms of a procedure for calculating it. An iterative factorial routine, however, might have a recursive contract. If the system could show in general that the two routines are equivalent, they will almost certainly both be right. Often the contract of a routine will not completely characterize the behavior of the routine. In fact it is undesirable to put irrelevant details about the desired behavior of the routine in the contract for the routine because other routines might make a practice of relying on them which will make it more difficult to later modify the system if the details prove to be undesirable. A routine may also have only a partial contract, just saying that it expects to be called with a non-negative integer or that it returns a list. There is nothing immoral about not advertising everything about yourself -- you are just liable to be ignored and not well understood.

We formalize contracts with the aim of developing a system capable (with help) of reading a programmer's code and formulating questions where it is not clear that the code satisfies its contract, intentions, and other commentary. This process of meta-evaluation will be explained in detail below.

A Brief Introduction to PLANNER-73 Syntax

Before introducing any formal examples, we must briefly introduce some of the common PLANNER-73 syntax which is used in the rest of the paper. In particular there are six constructs which need to be explained at this point. We note initially that [A_1 A_2 ... A_N] means a sequence of the elements A_1 through A_N.

1. Reminiscent of the LISP lambda expression is the ACTOR message-receiver:

```
(=> pattern
    body)
```

where "=>" is read "receives". This means that if an actor with this definition is sent a message which matches pattern it will evaluate body in the environment resulting from the pattern match. Patterns will often use a notation by which names are bound in an environment. For example, if 3 is matched against the pattern =x, then x will be bound to 3.

For example, the following routine adds one to any message it is passed:

```
(=> =x
   (x + 1))
```

2. ACTOR definitions are usually given labels by using:

```
[a-label <= a-definition]
```

which means that a-label is taken as the name of the procedural fixed point of a-definition. For example, we can define increment to be the function defined in #1 above:

```
[increment <=
   (=> =x
      (x + 1))]
```

3. Another way of binding names is in the use of the let statement, which takes the form:

```
(let
  {[name1 = binding1]
   [name2 = binding2]
          ...
   [namen = bindingn]}
  body )
```

which evaluates body in an environment with all the names bound. For example:

```
(let
   {[x = 3]
    [y = 5]}
   (x * y))
```

will return 15.

4. An often-used mechanism for applying a predicate to a supply of elements is:

```
(for-each
   a-supplier
   a-predicate)
```

which repeatedly asks **a-supplier** for another element, applying **a-predicate** to each one.

5. Occasional use is made of the unpack operator, which is abbreviated as an exclamation point. **!expression** is always equivalent to writing out all of the elements of the expression individually. Thus if **S** is bound to the sequence **[3 4 5]**, then the value of **[1 2 !S]** is **[1 2 3 4 5]**. Also if **[10 20 30 40 50]** is matched against the pattern **[=x =y !=z]**, **x** will be bound to 10, **y** will be bound to 20, and **z** will be bound to **[30 40 50]**.

6. Conditionals take two standard forms. The first is known as the rules expression and has the form:

```
(rules an-expression
   (=> pattern1 body1)
   (=> pattern2 body2)
      ...
   (=> patternn bodyn))
```

The expression is matched against the successive patterns until it matches one of them; then the corresponding body is evaluated in the environment resulting from the pattern match.

Thus:

```
(rules (3 + 4)
   (=> (even) (yes))
   (=> (odd) (no)))
```

evaluates to **(no)**.

A similar construct, more convenient at the outer level of message reception in an ACTOR definition, is the **cases** statement:

```
(cases
   (=> pattern1 body1)
   (=> pattern2 body2)
      ...
   (=> patternn bodyn))
```

in which the incoming message is matched directly against the successive patterns until a match is found, whereupon the corresponding body is evaluated in the resultant environment.

Contract Examples

We introduce the following syntax to express a contract (for an actor A) (similar to a more standard actor definition):

```
(=> pattern-for-incoming-message
    (require: assumptions)
    (use: relevant-knowledge)
    (rider-on-the-continuation: entailments)
    (rider-on-the-complaint-department: entailments-on-complaints))
```

The require: clause specifies what the programmer expects to be true about the incoming message that has matched the pattern. The use: clause of a contract is a place for the programmer to specify such information as what type of knowledge is likely to help in understanding why this contract is true (such as mathematical theorems or knowledge about the blocks world), and to give background information such as to why the programmer expects the routine to converge.

The entailments, or what the programmer expects to be true about the results of the behavior in light of the incoming assumptions, are expressed in what are known as *riders*. A rider behaves like a "one-time contract" that holds only for the current invocation. Its syntax is the same as for a regular contract. A contract often specifies two riders; one to be wrapped around the normal continuation, and one around the complaint-department, should the actor call there with an error.

For example a contract for a divide routine might goes as follows:

```
[contract-for-divide <=
    (=> [=the-numerator =the-divisor]
        (require: (the-divisor > 0))
        (use: ordinary-arithmetic)
        (rider-on-the-continuation:
            (=> [=the-quotient =the-remainder]
                (require:
                    (the-numerator =
                        (the-remainder +
                            (the-quotient * the-divisor)))
                    (the-remainder < the-divisor))))) ]
```

A more interesting example is the contract for an algorithm that sorts the elements of a sequence. In particular, given a sequence of integers, it will return a sequence with the elements arranged in increasing order. This example is significant because it demonstrates the convenience of expressing specifications in a procedural language. A specification of this requirement in the first order predicate calculus is exceedingly cumbersome.

We define the contract for the sequence sorter:

```
[contract-for-sequence-sort <=
    (=> [=the-sequence]
        (require: (sequence the-sequence)
                  (for-each
                      (elements the-sequence)
                      (=> =a-member
                          (integer a-member))))
        (use: knowledge-about-linear-data-structures)
        (rider-on-the-continuation:
            (=> =the-sorted-sequence
                (require:
                    (sorted the-sorted-sequence)
                    (permutation
                        the-sorted-sequence
                        the-sequence))))) ]
```

which makes use of the following two definitions:

```
[sorted <=
    (cases
        (=> [] (yes))
        (=> [=the-only-element] (yes))
        (=> [=first =second !=rest]
            (and
                (first < second)
                (sorted [second !rest])))) ]

[permutation <=
    (=> [=seq-1 =seq-2]
        (rules seq-1
            (=> [] (empty seq-2))
            (=> [=first !=rest]
                (rules seq-2
                    (=> [!=initial first !=remaining]
                        (permutation
                            rest
                            [!initial !remaining])))))) ]
```

The contract might be satisfied by many definitions of **sequence-sort**, one of which follows:

```
[sequence-sort <=
    (cases
        (=> [] [])
        (=> [=first !=rest]
            (merge
                first
                (sort !rest)))) ]
```

```
[merge <=
   (=> [=element =sequence]
      (rules sequence
         (=> [] [element] )
         (=> [=first !=rest]
            (rules first
               (=> (< element)
                  [first !(merge element !rest)])
               (else
                  [element !sequence])))))  ]
```

It is important to realize the distinction between these contracts and run-time checks. A run-time check is always going to be evaluated with the specific arguments of each call to the routine, thus causing inevitable inefficiency, and never giving complete confidence that a new set of arguments won't violate it. A contract is a statement of abstract requirements. It would be possible to have the programming system running in a super-cautious mode and have it check the whole contract on each call, but if we meta-evaluate the code we will discover that the contract will always be satisfied if the incoming pre-conditions are satisfied. Checking the incoming assumptions is therefore the most that is required; within the context of specific programs, the system may be able to show that they are always satisfied.

Intentions

An intention is similar to a contract, but rather than being wrapped around a piece of behavior which expects to be applied, it is inserted around something which expects to be evaluated. It is an abstract statement which the programmer expects to be true about the world at the point the statement is reached, rather than an advertisement to the world of a program's capabilities. It is intended to capture the flavor of comments of the sort: "x should be less than 10 here", "the block has been cleared off by now", "doing this will destroy the contents of that cell", etc. More specifically an intention surrounds an actor (routine) with assumptions that should be satisfied before and after the actor is invoked. It is more likely to be found around an application of a function to some arguments, rather than around the function itself. Thus it has no incoming pattern. An intention for a piece of code makes the properties of the local environment in which the code is intended to operate more explicit. There is a very incestuous relationship between the contract for a program and the intentions in the program. The pre-conditions in the contract must be sufficiently strong to guarantee that the intentions are not violated. Conversely the intentions must be strong enough to enable us to easily substantiate the post-conditions of the contract.

Consider an actor which looks up telephone numbers. The actor might have a contract of its own. A given call to the actor, however, may have an intention saying that the name is the name of a doctor, and that the number returned should be his/her office phone number.

The form of an intention statement is:

```
(intention
        (require: in-coming-assumptions)
        the-expression
        (rider: entailments))
```

As in the case of contracts, the entailments take the form of a rider, and the syntax is the same in this case. **Rider:** is just an abbreviation for **rider-on-the-continuation**, and one could express entailments for the complaint department, if necessary.

The above syntax is the most general intention statement, but we will not be using it in its full form very often. Certain restricted senses of it will turn out to be sufficiently common to warrant their own abbreviations. Intentions can be analyzed as being contracts on particular pieces of code. In fact intentions can be defined in terms of contracts. However, the distinction is analogous to the distinction between evaluation and application, and using the actor **intention** keeps us from having to reach down into the lower levels of message passing in order to express intentions.

Types and Constraints

One of the most common intentions to throw around an expression is a statement of the semantic type of its value. We therefore introduce the actor **constraint** with syntax:

```
(constraint x pattern)
```

which is equivalent to:

```
(intention (require: nothing)
        x
        (rider:
            (=> pattern
                (require: nothing))))
```

Constraints have a graphic abbreviation:

x | y **is equivalent to** **(constraint x y)**

Thus if you expect x to be bound to a noun, you could write:

(constraint x (noun)) or **x | (noun)**

Note that this would evaluate to the same thing as x. It is important to remember that specifying such a constraint (or any kind of intention) doesn't necessarily mean that the code will be slowed down in any way.

Post-requisites

Sometimes it is convenient to specify requirements dealing with the side-effects of the evaluation of an expression. **Post-requisite** statements are like constraints, but use the **require:** clause of the rider. The expression:

(post-requisite x out-going-requirements)

which is an abbreviation for:

```
(intention:
      (require: nothing)
      x
      (rider:
            (=> ?
               (require: out-going-requirement))))
```

Post-requisites have their graphic abbreviation:

x || z is equivalent to: (post-requisite x z)

For example, we might write

(eat your-dinner) || (empty your-plate)

Post-Conditions

Sometimes it is convenient to bypass the incoming pre-conditions but to specify a full out-going rider, which requires more power than the simple constraint or post-requisite statements give. Therefore we introduce the **post-condition** actor, which has the form:

(post-condition x out-going-contract)

which is an abbreviation for:

```
(intention
      (require: nothing)
       x
      (rider: out-going-contract))
```

The post-condition statement has its graphic abbreviation:

x ||| z is equivalent to: (post-condition x z)

For example, we might write

```
(tidy-up a-room) ||| (=> =the-room
                        (require:
                           (clean the-room)
                           (arranged (furniture the-room))
                           (for-each
                              (furniture the-room)
                              (=> [=a-piece]
                                 (clean a-piece)))))
```

Again it should be noted that this is different from putting in a run-time check that will always take the time to see if all these conditions are met.

Use of Intentions

Intentions should only be written when there is reason to believe that they are always true. Like all comments, they should not affect the operation of the program. Further, if the contracts for all the relevant actors are well-enough specified, the system should be able to show that all the intentions within the code are always satisfied. The prime use of the intention statements is in incorrect code. The intentions can be used as runtime checks but more importantly the system can abstractly evaluate the code and raise questions where it is not demonstrably justified that the code satisfies its contracts.

The following program makes scrambled eggs for n people given a bowl in which to break the eggs and a pan in which to cook the scrambled eggs. It has intentions buried deep down inside the code. (For clarity all intentions are written in capitals.)

```
[scramble-eggs <=
   (=> [=n | (> 0)
       =the-bowl | (CAPACITY: (N/4 CUPS))
       =the-pan | (CAPACITY: (N/4 CUPS))]
       (heat the-pan (temperature: 350))
       (clean the-bowl)
       (start breaking-eggs
          [0]
          (=>> [=i]
             (INTENTION
                (REQUIRE:
                   (HAVE
                      (IN: THE-BOWL)
                      (2*I EGGS)))
             (rules i
                (=> n
                   (done))
                (else
                   (get 2 eggs)
                   (break-into the-bowl)
                   (restart breaking-eggs
                      [(n + 1)]))))))   || (HAVE
                                              (IN: THE-BOWL)
                                              (2*N EGGS))
       (whon ((temperature the-pan) = 350)
          (pour the-bowl
             (into: the-pan))
          (stir the-pan (until: solidly-cooked)))))]
```

The pre-conditions for **scramble-eggs** should be sufficient to insure that the intentions in the module for **scramble-eggs** are satisfied. The reader should note that line oriented formalisms for expressing intentions [such as those proposed by Sussman and Goldstein] are inadequate for dealing with intentions in sophisticated applications such as the above.

Program Verification, Justification, and Meta-evaluation

Meta-evaluation is a process which attempts to show that the contracts of an actor will always be satisfied. Traditionally programmers try a program on several selected examples which they hope will bring out all aspects of its behavior, and once it works on those they assert that the program "works" in general. With some of the mechanisms we have been discussing we can design a procedure which will produce a much more coherent justification of whether or not a routine does what it should do. It is a moot point whether or not the justification produced by meta-evaluation is entitled to be called a "proof".

Meta-evaluation deals with contracts of the actors in a program. It tries to show that if the incoming pre-conditions of an actor are satisfied, then the out-going contracts will always be true. This is done by showing that for every call that this actor makes, the pre-conditions of the actors called are satisfied.

When you try to justify the contract for an actor, the contract asserts its incoming pre-conditions and tries to prove its out-going assumptions. It does this by stepping through the definition of its behavior; at each call out it utilizes the contracts of the actors that it calls by proving their in-going assumptions and then asserting their out-going assumptions. This process of asserting incoming requirements and asserting out-going declarations builds up enough information to enable the system to prove the outgoing assumptions of the main actor.

For example take the simple case of factorial. Its definition and contract are given below. In the contract of factorial we shall use an actor product [π] which can calculate things but can also be used as a model of what factorial produces. We use

```
(π (inclusive a to b)
   (=> [=i]
       (f i)))
```

to mean

A recursive definition of factorial and an appropriate contract are:

```
[factorial <=
   (cases
      (=> [0] 1)
      (=> [=n | (> 0)]
          (n * (factorial (n - 1))))))]
```

```
[contract-for-factorial <=
      (=> [=m]
         (require:
            (integer m)
            (m ≥ 0))
         (rider-on-the-continuation:
            (=> =answer
               (require:
                  (equal
                     answer
                     (π (inclusive 1 to m)
                        (=> [=i] i))))))
         (use:
            (knowledge: bag-of-mathematical-knowledge)
            (convergence-ordering: less-than)))]
```

The **use:** statement is a recommendation from the programmer to the system to use algebraic and mathematical knowledge to prove the equivalence of the resultant products.

The justification process procedes as follows: **contract-for-factorial** is asked to justify itself. It asserts that its message is a tuple of one element, and that that element is an integer greater than 0. It then tries to meta-evaluate the code for factorial in this world of assertions. The cases statement causes the world to fork into different extensions. The first branch of the cases is meta-evaluated in the first extension-world. The receive statement adds an assertion to the world that the tuple-element is zero, and that the result is 1. This returns up to the contract, which is able to prove, using mathematical knowledge, that 1 is indeed equal to the product from i = 1 to 1 of i. More formally we are using the following PLANNER-73 plans with

```
            k bound to 1 and
            f bound to (=> [=i] i)
      [simplify-singleton-π <=
         (to
            (simplify
               (π {=k}
                  =f))
            (f k))]

      [simplify-singleton-inclusive <=
         (to
            (simplify (inclusive =k to =k))
            {k})]
```

where in general

```
      (to (simplify
            pattern)
          expression)
```

means: if you want to simplify a clause which matches **pattern** you can use **expression**.

The cases statement then meta-evaluates the second receive statement in the other extension world in which it is asserted that the tuple-element is non-zero. This causes the contract of * to be read, which in turn requires the contract of factorial to be read (nothing peculiar is caused by the fact that we are now using the same contract that we are trying to justify). The contract for factorial successfully tries to prove that (n - 1) is a non-negative integer (this requires reading the contract of minus, obviously). The contract of factorial contract then asserts that

```
(π (inclusive 1 to (n - 1))
   (=> [=i] i))
```

is its value. Times can now assert that it returns

```
(*
   n
   (π (inclusive 1 to (n - 1))
      (=> [=i] i)))
```

which propagates up through the cases to the contract at the top level. The out-going-assumption that this equals

```
(π (inclusive 1 to n)
   (=> [=i] i))
```

is now proved using the following mathematical fact about products with

```
y bound to n
A bound to (π (inclusive 1 to (n - 1)) (=> [=i] i))
f bound to (=> [=i] i)
remaining-terms bound to an empty bag
x bound to n

[simplify-*-π <=
   (to
      (simplify
         (*
            =y
            (π =A
               =f)
            !=remaining-terms))
      (find (y = (f =x))
         (using: arithmetic-equation-solver)
         (then:
            (*
               (π (U A {x})
                  f)
               !remaining-terms)))))]
```

```
[simplify-U-inclusive-singleton-class <=
   (to
      (simplify
         (U (inclusive =a to =b) {(+ =b 1)}))
      (inclusive a to (b + 1)))]
```

Justifying a contract should also show that it converges, and this is done by specifying a partial order on messages to **factorial** which is included in the **convergence:** clause of the contract. In general programmers have an idea of why any loops or recursive routines they write should halt. Mathematicians have devised a very general method for doing this which requires the specification of a partial order R where R must have the property that there are no infinite descending chains of transmissions T1, T2, T3, ..., with the property that (i > j) implies Ti R Tj. The contract has two jobs: to show that the partial order specified by the programmer has the correct properties, and that the definition indeed satisfies the restrictions of the partial order.

Examples of Bug Detection

This section is based on a term project paper by Brian Smith, Dick Waters, and Henry Lieberman entitled "COMMENTS ON COMMENTS or the Purpose of Intentions, and the Intentions of Purposes" which was done for the M.I.T. course "Automating Knowledge Based Programming and Validation Using ACTORS" in the fall of 1973.

Here we will see how the above meta-evaluation would have found several different kinds of bugs that might have been in **factorial**. The specific code that is in error is underlined.

Bug-1: Suppose factorial had been written as:

```
[factorial <=
      (cases
         (=> [1] 1)
         (=> [=n | (> 0)]
            (n * (factorial (n - 1))))))]
```

This is a rather tough bug in a way because factorial will work correctly on every input but 0. The meta-evaluation catches the bug because it is now unable to prove that the input (n - 1) to factorial is ≥ 0. The proof fails because n may = 0. At this point the system may well ask the programmer why it thinks (n - 1) should be ≥ 0 at this place in the code.

Bug-2: Suppose factorial had been written as:

```
[factorial <=
      (cases
         (=> [0] 1)
         (=> [=n | (> 0)]
            (n * (factorial - n 1)))))]
```

This is a syntactic error: The input to factorial must be a 1-tuple but [- n 1] is a three-tuple. The meta-match fails.

Bug-3: Suppose factorial had been written as:

```
[factorial <=
      (cases
         (=> [0] 1)
         (=> [=n | (> 0)]
            (*
               (n - 1)
               (factorial (n - 1))))))]
```

This error is mathematical: The contract-for-factorial is unable to prove that

```
(=
   (*
      (m - 1)
      (π (inclusive 1 to (m - 1))
         (=> [=x] x)))
   (π (inclusive 1 to m))
      (=> [=x] x))
```

since is is not true.

Bug-4: Suppose factorial had been written as:

```
[factorial <=
      (cases
         (=> [0] 1)
         (=> =n | (> 0)
            (n * (factorial (n - 1))))))]
```

This is again a syntactic error: The contract-for-> is unable to prove that m is greater than 0 because it is not; it is a 1-tuple.

Now lets look at some problems in the contract that meta-evaluation will find. (Whether any problem found is considered to be in the code, or the contract is a matter of taste. Meta-evaluation just finds where they disagree.)

Bug-5: Suppose the contract had been written as:

```
[contract-for-factorial <=
      (=> [=k]
          (require:
              (integer k)
              (k ≥ 0))
          (rider-on-the-continuation:
              (=> =r
                  (require:
                      (equal
                          r
                          (π (inclusive 1 to k)
                              (=> [=xi] x))))))
          (use: general-mathematical-knowledge))]
```

This is not really a bug, but it does not go with factorial as it is written. The error is detected because now the whole first clause of the cases is vacuous. N can never be 0. Note that if the bug had not been found here, it would have been found in attempting to satisfy the pre-conditions for **factorial** when the expression **(factorial (n - 1))** is asked to meta-evaluate itself.

Bug-6: Suppose that the contract had been written as:

```
[contract-for-factorial <=
      (=> [=k]
          (require:
              (integer k)
              (k ≥ 0))
          (rider-on-the-continuation:
              (=> [=r]
                  (require:
                      (equal
                          r
                          (π (inclusive 1 to k)
                              (=> [=x] x))))))
          (use: general-mathematical-knowledge))]
```

This bug is sort of syntactic, it is caused by forgetting the normal conventions on returning a result. It is also interesting in that this bug was made by the first author in writing this section, and was not discovered until the meta-evaluation of (=> [0] 1) was complete and the contract asked the pattern [=r] to meta-match 1. The meta-match failed because 1 was not a 1-tuple.

It should also be pointed out that if all 6 of these errors were present together, there would be no added problems with meta-evaluation; it would just complain about the first one it encountered (bug 5) and call out to the repairman or the programmer. It would go on to find the others if it had a chance.

Overview of Meta-evaluation

Meta-evaluation is the process of binding actors to their contracts and then evaluating the actors abstractly on abstract data. Using actor induction we can show that if the meta-evaluation of a configuration of actors succeeds then the contracts of the actors will all be satisfied for all concrete inputs. If the meta-evaluation cannot proceed it will stop at the point in the program where it cannot confirm that a module satisfies its contract [intention] and ask for help. At this point there are several possibilities:

There really is an inconsistency:

The inconsistency is between the intention of the actor sending the message and the contract of the actor being sent the message.

The inconsistency is between the contract of the actor and its actual implementation.

The contracts for a configuration of actors are not mutually consistent.

There is no inconsistency but:

There are hidden assumptions being made about the behavior of certain actors that should be made explicit.

There is hidden domain dependent knowledge that the actor is using which should be made explicit.

The intentions are not being sufficiently explicit as to why they expect to be satisfied.

Of course it can be arbitrarily difficult to decide which one of these circumstances hold. In order for a programming apprentice to be helpful in this regard it must try to formulate its difficulties in concepts that are easily understandable by the programmer.

Benefits of Meta-Evaluation

"Explain all that," said the Mock Turtle.
"No, no! The adventures first," said the Gryphon in an impatient tone: "explanations take such a dreadful time."
--Lewis Carroll

Given that we have to work so hard to meta-evaluate a program we should get some benefit from our labor.

Consistency of Specifications: The successful meta-evaluation of the program for factorial demonstrates that the specifications in the contract are at least consistent. Of course the program which we have exhibited for factorial is not the most efficient. However it is one of the simpliest.

Question Answering: There are many questions which can be easily answered from the above meta-evaluation that are difficult to answer directly from the code for factorial. For example we might ask the question "What is the purpose of the expression (n * (factorial (n - 1))) in the program for factorial?" From the meta-evaluation the following answer can be given. The purpose of (factorial n) is to compute the product of the first n integers; i. e. to compute

```
(π (inclusive 1 to n)
   (=> [=i] i)).
```

This is accomplished by multiplying n with the product of the first (n - 1) integers; i. e. by

```
(*
   n
   (π (inclusive 1 to (n - 1))
      (=> [= i] i))).
```

Perturbation Analysis: Often it is found necessary or desirable to change either the specifications and/or code for a group of modules. The meta-evaluation can help us trace the implications of such changes. For example suppose that it is desired to drop the requirement that the argument of factorial be an integer. So the contract will now read:

```
[contract-for-factorial <=
   (=> [=k]
      (require: (k ≥ 0))
      (rider-on-the-continuation:
         (=> =y
            (require:
               (implies
                  (integer k)
                  (equal
                     y
                     (π (inclusive 1 to k)
                        (=> [=x] x)))))))
      (use: general-mathematical-knowledge))]
```

Note that programs which relied on the old contract will continue to be able to use the new factorial function provided that the new contract subsumes the old contract and the new implementation of factorial satisfies the new contract. There are a variety of reasons that might prompt this change. Some user might have specifically requested it. It might be necessary for other modules which are already written. The system designers might see the need for more generality in the future. In any case we are interested in what the implications are. The major implication that emerges from redoing the meta-evaluation is that the original program for factorial can no longer be demonstrated to converge. In particular it cannot be demonstrated to converge in the open interval between 0 and 1. For this and other reasons the programmers are led to make the following change to the definition of factorial:

```
[factorial <=
   (=> [=n | (≥ 0)]
      (rules n
         (=> 0 1)
         (=> (< 1)
            (integral (interval 0 to 1)
               (=> [=x]
                  (ln (1 / x))^(n - 1))))
         (else
            (*
               n | (≥ 1)
               (factorial (n - 1)))))))]
```

Later it is noticed that another clause can be added to the contract of the new factorial function to the effect that for every n > 0 we have ((factorial (n + 1)) = (n * (factorial n))).

Programming Style and Responsibility

Some authors have advocated top down programming. We find that our own programming style can be more accurately described as "middle out". We typically start with specifications [contracts, intentions, constraints, etc.] for a large task which we would like to program. We refine these specifications attempting to create a program as rapidly as possible. This initial attempt to meet the specifications has the effect of causing us to change the specifications in two ways:

1: More specifications [features which we originally did not realize are important] are added to the definition of the task.

2: The specifications are generalized, specialized, and/or otherwise combined to produce a task that is easier to implement and more suited to our real needs.

At any given point in the programming process, we are confronted with

1: A partial program which attempts to accomplish some task

2: Partial specifications [contracts, intentions, and constraints] which judge whether or not the task is accomplished

3: A partial substantiation which says why the code satisfies part of its contract.

4: A partial collection of the plans for using the background knowledge assumed by the program.

5: A collection of scenarios and models of how the programs are supposed to work in concrete instances.

Current generation software engineering practice borders on the criminally irresponsible in that it does not require that programmers [whether human or machine] substantiate that the code meets its contracts before it is foisted off on an unsuspecting public. Ultimately we would like to automate the process of substantiation; but in the meantime people can perfectly well serve this role. In most cases, current practice does

not even require that rigorous contracts for the code be written down! If civil engineers designed and built bridges and buildings with the same cavalier attitude that has been adopted by current software engineering practice, it would cause serious loss of life. Consider an analogy concerning the development of surgical procedure: Before Lister and Pasteur, surgeons operated without first autoclaving their instruments. Although the surgeons were dedicated and well-intentioned the results often came out badly for the patients.

We expect that simply writing good contracts for what procedures are supposed to accomplish will have large beneficial effects on the way programs are written. A typical current generation program is often both haughty and finicky. It either replies with a complaint about some irrelevant trivial detail or a terse answer with no justification and no opportunity for further interaction to determine the reasons for the answer. For example consider the problem of writing a contract for a chess program. The purpose of a chess program is to make a good move for the position. It is just as difficult to write a program which judges whether a given move [supplied as the answer by the program] is a good one or not as it is to write the chess program which is supposed to find good moves. However, if the program is required to supply a convincing justification why the move that it proposes is a good one then the task of judging the answer is much easier.

Debugging and Validation

A currently popular approach to software production is the "Debugging Paradigm". Debugging can be either high-level and intelligent, using powerful analysis and knowledge of the domain, or it can be uninformed, local, and incompetent, giving no concern to the ramifications of a suggested patch. In either case the debugging process is roughly as follows:

1. Programmers make a first-order attempt, by simply putting together procedures that separately achieve the individual goals. The code is then tested on some sample inputs to see if it works at all and patched until it works on the samples. It is then distributed to users with hopes for the best.

2. When some user [or the whole system] runs into trouble, the programmers are called, who may try to diagnose one of the symptoms as a specific (and undesirable) kind of interaction between two procedures.

3. The programmers may then try to apply a "debugging technique" that, according to a record in memory, is good at repairing that variety of interaction.

4. In any case, the programmers labor until the specific symptom goes away or is declared to be a "feature". The world then drifts tentatively along until step 2 recurs.

The power of this approach lies in the sophistication of the techniques applied and in the sophistication of the analysis of interactions. Until methods are found to incorporate naturally the knowledge that good programmers bring to bear in debugging, a programming system will not be able to provide really useful assistance.

Furthermore, we believe that the debugging paradigm must be integrated with a "Validation Paradigm" in which rigorous contracts are written for the systems and it is substantiated that the software meets its contracts before it is released. This is not to

say that the the debugging paradigm is without merit. But no matter how intelligently the debugging is carried out, it simply does not in itself produce the evidence needed to have confidence in large public software systems. The fact that a program seemed to behave correctly on the last three test inputs that it was given is not a substantial reason to believe that the program will always behave as contracted.

Working with a program on concrete cases has advantages for certain purposes over reasoning entirely abstractly. For example running a new program on a few test cases is a good way to shake out simple bugs from the system [particularly if the system has a way to keep a complete history of the computation and to undo [Teitelman] any or all side-effects when the computation bombs]. Successful histories expressed in EVENT DIAGRAMS are useful in suggesting how to substantiate that the program works in general. Furthermore histories of successful and unsuccessful attempts to solve a certain class of problems can often be variabalized [Hewitt 1969, 1971; Hart, Nilsson, and Fikes 1972; Sussman 1972] to obtain general procedures for that class of problems.

Relation to Automatic Programming

"We base ourselves on the idea that in order for a program to be capable of learning something it must first be capable of being told it."
McCarthy

In his report to ARPA entitled "Automatic Programming", Robert Balzer has identified the four major phases of Automatic Programming as being: Problem Acquisition, Process Transformation, Model Verification, and Automatic Coding. He proposes to investigate whether systems that implement this paradigm can be built to converse with experts [businessmen, doctors, engineers, etc.] who are not programmers to automatically produce programs in their domain of expertise. The extent to which this will be possible within the foreseeable future is unknown.

We are working on a rather different problem: our goal is to construct a Programming Apprentice which can aid expert programmers in constructing large public software systems in such a way that they will be easier to write, debug, and maintain. Furthermore there must be a substantial reason to believe that the programs will behave as contracted. The success of our project is not dependent on the success of the Automatic Programming projects. Indeed, it seems likely that substantial progress is necessary on the programming apprentice problem before Automatic Programming can progress past a certain point. Of course partial successes or useful techniques that are developed for automatic programming stand a good chance of being useful to our Programming Apprentice.

Further Work

Contracts and intentions don't capture everything intelligent to be said about programs. In fact they are just the first level of description beyond the code. We have not yet incorporated more abstract descriptions of behavior like models. In real life programmers almost always have models of what they are implementing and a well-commented program often presents the model first, and then relates the code, line by line, to the model.

We don't yet well understand much of this unexplored area, but are currently investigating it in the world of programs defining data structures. There appears to be a whole class of comments which we call purpose statements which link behaviors. Some that

we are beginning to understand are links between the code and the model, links between parts of the code and other parts affected (often by side-effects), and between the code and the justification of why it works.

Purpose statements within the code come to the fore when there are relations which do not follow the simple flow of control, and in particular when there are side-effects involved. Keeping track of these purposes in the justification process allows the system to monitor the scope of side-effects and to protect them until they are used. Protecting, of course, does not simply mean making sure that a side-effect is never violated; it is closer to meaning that if it is ever violated it should be replaced before it is needed. If a programmer could easily specify in the justification why pieces of code were written, in many cases the system could protect the result until needed; in this way many common bugs could easily be tracked down, if not fixed.

There has been a great deal of work done on achieving and protecting side-effects: Newell and Simon in GPS; Simon in his Heuristic Compiler; Hewitt in development of goal oriented formalisms with ability to delete elements from the data base and to do pattern directed invocation to draw conclusions from the changes; Rulifson in developing a context mechanism for QA4 to attempt to control the scope of changes to the data base; Winograd, Fahlman, and Sussman for the blocks world; Waldinger for simple sort programs; Goldstein for fixed instruction turtle programs; McCarthy, McDermott, Buchanan, and Luckham for simple robot problems; etc. Sometimes having side-effects indicates an unaesthetic program; sometimes it is a very clever thing to do; and sometimes it is the only way to solve a problem.

Advantages of Contracts

Actor based contracts have the following advantages over previously proposed formalisms for expressing what procedures do as opposed to how they do it:

The contract is decoupled from the actors it describes.

We can partially substantiate facts about the behavior of actors without giving a complete formal proof. An actor who is asked can if it chooses make an explicit assumption for some circumstance being the case which the programming apprentice can remember so that it can be dealt with later. At some later time if we require further justification, then we can re-examine the situation.

Contracts of concurrent actions are more easily disentangled.

We can more elegantly write contracts for dialogues between actors.

The contracts are written in the same formalism as the procedures they describe. Thus contracts can have contracts.

Historical contracts in which the behavior of the actor depends on the history of messages which it has received can be easily and naturally expressed. For example it is easy to write a contract for a routine which returns the average of all values it has ever been called with. Furthermore contracts for side effects are expressible without recourse to the notion of a global state by packaging up side-effects in contracts designed that purpose.

The extent to which contracts are checked at execution time as opposed to being verified once and for all [making the execution time check superfluous] becomes at least partially an economic decision. Sometimes [as in type checking] it is cheaper to use an efficient runtime check providing that the possibility of a run time fault is tolerable. There are some applications [e.g. controlling a nuclear reactor or a heart-lung machine] where a run time fault is not tolerable.

Because a basic kind of protection is an intrinsic property of actors, we hope to be able to deal with protection issues in the same straightforward manner as more conventional contracts. We use contracts to express what programs and data structures are supposed to do. In addition we are concerned with expressing and substantiating that programs do not do what they are not supposed to do. For example an actor given access to a data base can be contracted not to write into the data base.

Contracts for data structures are handled by the same machinery as for all other actors.

Conclusions

Every actor can have a ***contract*** which checks that the pre-conditions and the context of the actor being sent the message are satisfied. The contracts of an actor are with the other actors with which it communicates. How an actor fulfills its contract is its own business. A contract for an actor is an absolutely arbitrary monitor on the behavior of the actor except that it is not to affect the behavior of the actor it monitors. If it detects a violation the contract will bring the whole computation to a halt. [In practice what will actually happen is that a Repairman [which is perhaps another program] will be called to attempt to salvage the situation.]

The successful meta-evaluation of actor modules using only the contracts of other modules makes it easier to extend behavior without introducing unpredicted complications. The behavior of any given module can be arbitrarily extended without changing any other modules providing that the contract for the new module can be shown to subsume the contract of the module it replaces. Furthermore if the new module fails to fulfill certain clauses of the contract for the module which it replaces, the modules dependent on the discarded clauses can be identified.

By a ***simple bug*** we mean an actor which does not satisfy its contract. We would like to eliminate simple bugs in programs by the ***meta-evaluation*** of the modules to show that they satisfy their contracts. Eliminating all the simple bugs from a program does not imply that it always behaves as intended. It only implies that the program will fulfill its contract; the fine print in the contract may not be sufficient to imply the intended behavior.

The rules of deduction to establish that actors satisfy their contracts essentially take the form of a high level interpreter for abstractly evaluating the program in the context of its contracts. This process [called ***meta-evaluation***] can be justified by a form of induction. Meta-evaluation captures a large part of the mechanism that a programmer goes through when she reads a piece of code to determine that it will satisfy its specifications. It is a kind of "meta-debugging" in which the code for accomplishing some task is reconciled with the contracts for the task. Meta-evaluation exposes and makes explicit the behavioral dependencies of the programs sufficient to

substantiate all the contracts and intentions of the modules. It exposes both the dependencies within a module and those between modules.

Acknowledgments

This research was sponsored by the MIT Artificial Intelligence Laboratory and Project MAC under a contract from the Office of Naval Research.

We would like to acknowledge other members of the PLANNER Project: Irene Greif, Peter Bishop, Roger Hale, and Richard Steiger who have contributed extensively to the ideas in this paper. Irene Greif is doing theoretical investigations on characterizing behavior involving parallelism and side-effects that provides a mathematical foundation for the meta-evaluation process. The first section of the paper has benefited from extensive conversations with Ben Kuipers and Tom Knight. Many M.I.T. students have served as guinea pigs while this material has been fermenting during the last year. In particular Keith Nishihara and Howie Shrove made valuable comments and criticisms.

> *"These [PLANNER-like] systems have traded increased operational power for loss of awareness. Because the knowledge is represented procedurally, the system is less capable of using it deductively or in determining what the consequence of particular actions may be."*
>
> *Bob Balzer [1973]*

The research reported in this paper is the natural continuation of previous research [Hewitt 1969,1971; Rulifson 1971; Davies 1971; Sussman and McDermott 1972; Hewitt et. al. 1973] for the procedural embedding of knowledge. If all knowledge is to be embedded in actors [procedures], then an intelligent system must have a sound knowledge of programs and programming if it is to understand its own problem solving methods. In this paper we have made extensive use of ideas and techniques developed for the predicate calculus approach to verification of properties of programs [Goldstine and Von Neumann; Floyd; Manna; Waldinger; Milner and Weyhrauch; Green; Deutsch; Luckham and London; etc.]. Boyer and Moore have independently developed a system for proving equivalences between LISP functions. Their work differs from ours in being specialized to proving equivalences and in that their system works on the basis of structural induction where it attempts to automatically guess the right induction principle. Meta-evaluation procedurally embeds Actor Induction as its sole inductive principle. Furthermore, our work includes, but is not limited to, showing that programs satisfy their contracts.

Bibliography

Balzer, R. "Automatic Programming" ISI TR. Jan, 1973.

Balzer, r. "CASAP: A Testbed for Program Flexibility" IJCAI-73.

Boyer, R. S. and Moore, J. S. "Proving Theorems about LISP Functions" IJCAI-73. August, 1973.

Buchanan, J. R. and Luckham, D. C. "On Automating the Construction of Programs" Phd. Stanford. Forthcoming.

Burstall, R. M. "Proving Properties of Programs by Structural Induction" Computer Journal. Vol. 12, pp. 41-48 (1969).

Burstall, R. M. "Some Techniques for Proving Correctness of Programs Which Alter Data Structures" Machine Intelligence 7. 1972.

Cadiou, J. M. "Recursive Definitions of Partial Functions and their Computations" Ph.D. Stanford. 1972.

Cheatham, T. and Wegbreit, B. "A Laboratory for the Study of Automating Programming" SIGSAM Bulletin. Jan. 1972.

Church, A. "The Calculi of Lambda Conversion" Annals of Mathematical Studies 6. Princeton University Press. 1941, 2nd edition 1951.

Davies, D. J. M. "POPLER: A POP-2 PLANNER" MIP-89. School of A-I. University of Edinburgh.

Deutsch L. P. "An Interactive Program Verifier" PhD. University of California at Berkeley. June, 1973. Forthcoming.

Dijkstra, E. W. "The Humble Programmer" CACM. October, 1972.

Floyd, R. W. "Assigning Meaning to Programs" Mathematical Aspects of Computer Science. J. T. Schwartz (ed.) Vol 19. Am. Math. Soc. pp. 19-32. Providence Rhode Island. 1967.

Floyd, R. W. "Toward Interactive Design of Correct Programs" IFIP-71.

Goldstein, I. "Understanding Fixed Instruction Turtle Programs" Phd. M.I.T. 1973.

Goldstine, H. R. and Von Neumann, J. "Planning and Coding Problems for an Electronic Computer Instrument" in in *Collected Works of John von Neumann*. pp. 91-99. Macmillan, 1963.

Greif, I. G. and Hewitt, C. "Behavioral Semantics of ACTOR Systems" Submitted to IFIP-74.

Greif I. G. "Induction in Proofs about Programs" Project MAC Technical Report 93. Feb, 1972.

Hewitt, C.; Bishop, P.; Steiger, R.; Greif, I.; Smith, B.; Matson, T.; and Hale, R. "Behavioral Semantics of Nonrecursive Control Structure" Colloque sur la Programmation. Paris, France. 9-11 April 1974.

Hewitt, C. "The Semantics of ACTIONS and the Semantics of TRUTH" Special Session on Formalisms for Artificial Intelligence at IJCAI-73. August, 1973. Submitted to A. I. Journal.

Hewitt, C. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot" IJCAI-69. Washington, D. C. May 1969.

Hewitt, C. "Procedural Embedding of Knowledge in PLANNER" IJCAI-71. London. Sept, 1971.

Hewitt, C. and Paterson M. "Comparative Schematology" Record of Project MAC Conference on Concurrent Systems and Parallel Computation. June 2-5, 1970. Available from ACM.

Hewitt, C. "Procedural Semantics" in Natural Language Processing Courant Computer Science Symposium 8. Randall Rustin, editor. Algorithmics Press. 1971.

Hewitt, Carl; Bishop, Peter; Greif, Irene; Smith, Brian; Matson, Todd; and Steiger, R. "Actor Induction and Meta-evaluation" Conference Record of ACM Symposium on Principles of Programming Languages. Boston. Oct, 1973.

Hoare, C. A. R. "An Axiomatic Definition of the Programming Language PASCAL" February 1972.

Igarashi, S.; London, R. L.; and Luckham, D. C. "Automatic Program Verification I: A Logical Basis and Implementation" Stanford AIM 200. 1973.

Kaplan, D. M. "Correctness of a Compiler for ALGOL-like Programs" Stanford A.I. Memo No. 48.

Katz, S. and Manna, Z. "A Heuristic Approach to Program Verification" IJCAI-73.

King, J. "A Program Verifier" Ph.D. Thesis. Carnegie-Mellon University. 1969.

Manna, Z.; Ness, S.; Vuillemin J. "Inductive Methods for Proving Properties of Programs" Proceeding of an ACM Conference on Proving Assertions about Programs" January, 1972.

McCarthy, J. "A Basis for a Mathematical Theory of Computation" In Computer Programming and Formal Systems. 1963. North Holland.

McCarthy, J. "Programs with Common Sense" Proceedings of the Symposium on the Mechanization of Though Processes. Teddington. HMSO. London.

Morris, F. L. "Advice on Structuring Compilers and Proving Them Correct" Conference Record of ACM Symposium on Principles of Programming Languages. Boston. Oct, 1973.

Morris, J. H. "Verification-oriented Language Design" Technical Report 7. December, 1972.

Naur, P. "Proofs of Programs by General Snapshots" BIT. 1967.

Park, D. "Fixpoint Induction and Proofs of Program Properties" Machine Intelligence 5. Edinburgh University Press. 1969.

Parnas, D. L. "Information Distribution Aspects of design methodology" IFIP-71.

Parnas, D. L. "A Technique for Software Module Specification with Examples" CMU. 1971.

Rulifson Johns F., Derksen J. A., and Waldinger R. J. "QA4: A Procedural Calculus for Intuitive Reasoning" Phd. Stanford. November 1972.

Scott, D. "Outline of a Mathematical Theory of Computation" Proc. Fourth Annual Princeton Conf. on Information Science and Systems. 1970. pp. 169-176.

Smith, Brian; Waters, Dick; and Lieberman, Henry. "COMMENTS ON COMMENTS or the Purpose of Intentions, and the Intentions of Purposes" Term Project for M.I.T course "Automating Knowledge Based Programming and Validation Using ACTORS" December, 1973.

Snowdon, R. "An Interactive System for the Preparation and Validation of Structured Programs" University of Newcastle upon Tyne. 1973.

Sussman, G. J. "A Computational Model of Skill Acquisition" A. I. TR-297. December, 1973.

Tennent, R. D. "Mathematical Semantics of SNOBOL4" Conference Record of ACM Symposium on Principles of Programming Languages. Boston. Oct, 1973.

Waldinger, R. J. "Reasoning About Programs" Conference Record of ACM Symposium on Principles of Programming Languages. Boston. Oct, 1973.

Wegbreit, B. "Heuristic Methods for Mechanically Deriving Inductive Assertions" IJCAI-73. Vol 39. pp. 253-262.

Weyhrauch, R. and Milner R. "Programming Semantics and Correctness in a Mechanized Logic." First USA-Japan Computer Conference. October 1972.

Winograd, T. "Joshua" draft. 1973.

Wirth, N. "Program Development by Stepwise Refinement" CACM 14, 221-227. 1971.

Vuillemin, J. "Proof Techniques for Recursive Programs" Ph. D. Thesis. Stanford. 1973.

ACTIVE DESCRIPTIONS FOR REPRESENTING KNOWLEDGE

JAMES L. STANSFIELD
Bionics Research Laboratory, School of Artificial Intelligence,
University of Edinburgh.

Introduction.

The representation of knowledge is a fundamental part of any Artificial Intelligence investigation. If we are to have systems which understand, then the knowledge they have must be represented somehow. In this paper we consider some representations for concepts and propose a new method which we call "active descriptions".

What is a concept? If we look in a dictionary we find something like "an idea, a notion, a class of objects". This is not too useful and we do better to examine its root. To "conceive" an idea is "to form it in the mind - to imagine". We see here the idea that a concept is a construct. Again, Bourne, Ekstrand and Dominowski (1971) define a concept as "any describable regularity of real or imagined objects or events". So a concept is a description in some language and constructed from other concepts.

By a description we mean a structure whose parts and the relationships between them give information about whatever is being described. In psychology, mainly class concepts are dealt with. Here the language is propositional calculus and is too simple for most AI programs. If we use English as our language then our descriptions are mainly of concrete and abstract entities such as "the red top of my coffee jar", "walking fast", and "hairiness". Being descriptions they are generally represented by English noun phrases.

This definition of concept seems to equate it with description. There is more to a concept, however, in particular the knowledge of how to use it. A "chair", for instance, besides needing descriptions of its structure needs information about "how to sit down on chairs" and"what kind of shop to buy a chair in". There are thus two poles to the concept "concept", the structural and the procedural. Our main purpose in this paper is to bring these two together, to show that there is a dimension between these poles, and to suggest the kind of system which might deal equally well with both aspects.

The two poles are expressed in two paradigms from AI which are exemplified in...

(1) The procedural representation of Winograd (1971).
(2) The structural descriptions of Winston (1970).

In his program, SHRDLU, Winograd treated meanings as programs. Each word and each grammatical unit had semantic programs associated with it which together built up meanings for English input. The meanings were PLANNER procedures which when evaluated would do what was necessary to respond to the input, e.g. answer questions or take commands. The meaning of a noun group was a program to find an instance of that noun group so the use of the concept was being represented.

Winston's program was based upon structural descriptions used to represent the structure of concepts from a blocks world. Such concepts were "arch", "tower", "bench" and other block constructions. A description of a concept was a relational structure which represented both the structures which should be present in an example of the concept and those which must be absent. The difficulty with representing concepts this way is in representing the procedural information about them. An interpreter for the types of structures used must be written. Procedural embedding can be re-phrased as "why not use the interpreter of a standard programming language".

We try to reconcile these two viewpoints. We feel that the advantage of procedural embedding should not be thrown away since to use concepts easily they should be programs. It turns out that we need structures which sometimes act as programs and sometimes as descriptions. There is one example in SHRDLU where a PLANNER procedure is taken as a structure to be examined. However we find that a new representation is really needed.

The last section of this paper tries to suggest a form for this new representation. We call the representation active descriptions since the objects built up are structures which may be examined and hence used as descriptions but the components of the structures are processes running in parallel. We define a process to be any ongoing activity to which messages can be sent and from which messages emanate. We elaborate this definition in Stansfield (1974). An advantage of this method is that interpretative knowledge can be spread throughout the system - we do not require one processor which can uniformly process any structure. The method is perhaps a step towards some representation where there is no essential difference between descriptions and procedures. This relates it to Hewitt's Actors (Hewitt et al, 1973) since here also there is no difference between data and process. Again we go into this in detail in Stansfield (1974).

Reasons for needing descriptions

1. Referential opacity.

We need descriptions rather than programs to cope with referential opacity. A noun group is referentially opaque if the clause which refers to it refers to it as a description rather than to some items denoted by that description. So in...

(3) "Does Fred know the blocks which are on Block 2?"

... the noun group should not be evaluated to produce certain blocks say b1 and b2. This would leave the sentence equivalent to...

(4) "Does Fred know b1 and b2?"

... which clearly has lost some meaning. Indeed, that sentence may be asked of someone who does not know the particular blocks. The noun group must be treated as a description. Similarly with ...

(5) Fred thinks blocka is a cube.

Blocka might not be a cube so we cannot represent (5) by the two assertions ...

(6) Fred thinks X.
(7) X blocka is cube.

It could be said that the program to find the blocks may be used as the description and we would agree with this. Any structure is a description if it is used as such. Our question is simply, whether a program in a good form for finding examples is in a suitable form for other purposes.

2. Making simple assertions.

We need descriptions to make simple assertions. Winograd mentions that to deal with the assimilation of new knowledge a better representation is needed. In general, new knowledge is not simply stored but is processed and may cause alterations of far-reaching proportions in the knowledge structure. However, even the storing of facts is sufficient for our point. SHRDLU's programs representing noun groups are composed of three types of information linked together in an order which may be thought of as a fourth. The three are ...

(8) A set of patterns expressing features in the description of the noun group, e.g. (red fx), (block fx). Each of these patterns being an examinable structure is a sub-description itself.

(9) The insertion of "thgoal" in front of these patterns to to turn them into PLANNER statements for retrieving and testing.

(10) Statements like "thfind" statements, expressing knowledge of the part quantifiers and numerals play in the actions of retrieving and testing.

In this paper we will be generalizing these features.

To assert simple facts such as ...

(11) AI is a big red block.

we must at least replace the occurrences of thgoal by thassert. This would work so long as we have no numbers or quantifiers. In place of...

```
(12) (thgoal fx block)
     (thgoal fx big)
     (thgoal fx red)
```

we would have ...

```
(13) (thassert fx block)
     (thassert fx big)
     (thassert fx red)
```

It is clearer to have the meaning of (11) to be just the description with "thgoal" or "thassert" being given as an argument ...

```
(14) (arg fx block)
     (arg fx big)
     (arg fx red)
```

We show later that the best way to find an instance of a set of patterns does not necessarily find each in some set sequence. So, we should take "arg" outside each pattern and simply say "arg" of an unordered description as in ...

```
(15) arg( (fx block ),(fx big),(fx red) )
```

The set of patterns is now very like a Winston structural description.

3. Assertions and referential opacity.

The question of assertions can be related to referential opacity. Suppose we assert ...

(16) A piano is a box with strings in.

We don't want the separate assertions...

(17) A piano is a box.
(18) The box has strings in.

but require the piano to be the entire object; box and strings together. In other words, rather than ...

"find a box, check it has strings in, match the box to piano"

we need ...

"find a box with strings in and label the whole piano".

This reveals a difference between Winston's method and Winograd's. In Winston's work an "arch" could be defined as a description ...

(19) An arch is a beam with two supporting blocks.

In Winograd's program the definition would be a program ...

(20) Find a beam and check it has two supports.

and the system would say B1 is an arch in Fig. 1.

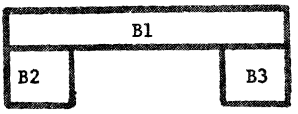


Figure 1.

4. Numerals.

We need descriptions for making assertions involving numerals. Consider telling a program ...

(21) Two blocks are in the box.

If the program cannot see inside the box or work out which blocks are in it then it must store that something satisfying the description "two blocks" is in the box.

Furthermore the program must symbolically manipulate descriptions to answer hypotheticals like ...

(22) If I put two blocks from the table on to the shelf, how many red blocks would be on the shelf?

... where there are no blocks on the shelf to begin with and, say, all blocks on the table are red. We certainly don't want to run the "put" program twice, moving two particular blocks, and then to assume that since these two happen to be red that we will always end up with two red blocks on the shelf. We need to reason about the action and, in so doing, to manipulate a description of it if we are to arrive at a true result.

5. /

/5. Verbs which take descriptions.

Many English verbs take their objects to be descriptions rather than retrieval programs. Below we see that the verb "call" requires a description of some object to work on and only then can it invent a name.

(23) If you had a large shaggy dog what would you call it?

Similarly "look" takes a description in ...

(24) Look at the red sky.

The description "red sky" is more than a program for finding a denotation. There is only one sky so red is entirely unnecessary for this purpose. Instead it points out the feature of the sky to be considered while looking.

PLANNER's use of descriptions for finding examples of concepts

So far we have argued the need for descriptions rather than procedures. Let us now consider arguments based on the methods used in PLANNER and CONNIVER. PLANNER's deduction mechanism for finding instantiations, for asserting objects to be examples of concepts, and for reasoning about concepts in general, has deficiencies when used for the type of reasoning about everyday objects and concepts an artificial intelligence will require. We will show when we consider these that code should be treated symbolically as a description as well as being interpreted.

We can take as a first approximation to a description, a set of patterns perhaps with variables to be instantiated. This could be seen as a local self-contained data-base. The important points about this scheme are ...

(25) The description is not accessed from any particular point.
(26) There are links between the patterns.
(27) The patterns are in no particular order.

Figure 2 shows a possible definition of "arch"

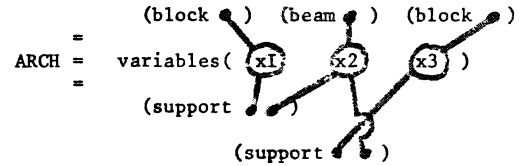


Figure 2.

In PLANNER or CONNIVER the natural way of making such a concept would be to choose a particular ordering of the patterns, envelope each in a thgoal statement, and make the entire structure a procedure for finding and testing for the new concept.

1. Choice of an order for the patterns.

SHRDLU has a very simple method of dealing with this problem in its small world. When it builds the meaning of a noun group it orders the patterns/

/patterns according to weights associated with them by the programmer. This is adequate but for a larger world things can go wrong. There are two problems. One is in the method used to order the patterns and the other is in the time this ordering is done. We don't know that the weights will always work. They may depend on the context at the time of ordering. Even worse, the order of code might depend on the context at the time of execution. (28) is best done in one order at a linguistics conference and the other at a brass band competition.

(28) Look for a linguist who plays the tuba.

2. Problems when already given an order.

Things can go wrong even given a particular order. Suppose we have to find an Egyptian Ephalump, i.e.

(29) thgoal (egyptian fx)
thgoal (ephalump fx).

How can we discover that these goals are inconsistent. Suppose that the first finds an Egyptian object E1. The second goal will fail and another Egyptian object will be searched for. The mistake is that the two goals are considered independently. Now the second goal might instead return a reason for failure since ephalumps are either African or Indian. This can only be used if the second goal knew about the structure of the program it is in. The entire search could then be made to fail with a reason.

There seems to be no simple way of arranging in general that the best advantage is taken of possible interactions between goals without giving up the idea of a particular ordering of goals. Any statement needs to know not only the structure of its immediate containing theorem but also the structure of the deductions made at run time. A more complex example will illustrate this. Consider again ...

(30) thgoal (a fx) ; thgoal (b fx).

The first goal could cause a consequent theorem to be invoked in the course of which (c fx) might be asserted. Thgoal (b fx) might fail because of this assertion. To allow for this, "thgoal (b fx)" must be able to see what part (c fx) played in the execution of "thgoal (a fx)" and what conditions implied it. Having found goals that implied (b fx) they can be failed rather than tried again for new solutions.

Another approach to this problem is to let (b fx) give advice to thgoal (a fx) namely, don't find anything which has (c fx) true about it.

3. There may be no satisfactory order.

The next example shows that no order may be satisfactory. Consider the case ...

(31) A point on line1 and on line2.

in Fig. 3 where line1 and line2 are circles centres O and P respectively. It is foolish to try satisfying these goals independently in either order. There are an infinite number of solutions to each but only two to them both. Instead, some deduction needs to be done first. It is the situation of Fig. 4 where we know our answer is in set A and in set B and where together they imply it is in set C.

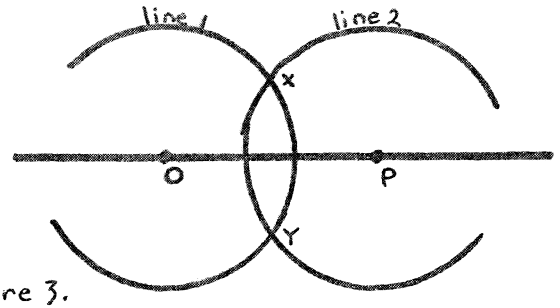


figure 3.

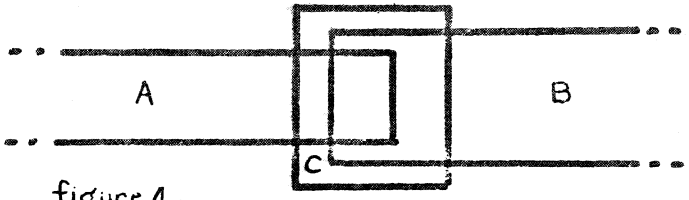


figure 4.

4. PLANNER retrieval relies too heavily on a central data-base.

Finally PLANNER's approach is oriented to the use of a central data base and retrieval is by a simple, perhaps externally directed search.

Let us take two simple cases. Firstly consider ...

(32) Find a lion in London.

Instead of looking at lions in the data-base and seeing if they are in London it is more sensible to realise that if a lion is in a city then it is likely to be in a zoo or a circus. Having found a zoo we can use knowledge of zoos and search for a map on a bill-board. Using our knowledge about maps we are home.

Secondly we consider ...

(33) Find a cup with a chip on it.

If my kitchen were impeccably kept it might, on data-base considerations, be wise to look for something chipped first as there will be fewer of these than of cups. In the real world though it is better to look for cups since we know where to find them. We would look in the kitchen cabinet which could be a data-base in itself. A data-base should be like a shop with a storekeeper. If we want something we ask the storekeeper who knows how his store is organised. He can give us advice by conversation about what we might need for any particular purpose and he might use reasoning to find this advice. Our knowledge of the world should be organised semantically. This reduces the importance of transactions with a syntactic data-base and emphasises the role of deductions.

We have shown that parts of a concept interact allowing deductions which add to the concept. The process is one of construction by deduction. For example, if we have to find some example of a description /

/description we might not search for it immediately but instead consider the description to see what it entails. In doing so we build up a picture of the typical object required. It is unreasonable that such a description can only be refined by mutual interactions between its parts for we must consider its changing relation to the rest of the world model. Any new piece of knowledge might modify a concept already present by allowing further implications about it. In general this is the problem of how concepts can develop, be generalised, enhanced, simplified, etc.. Our picture is of concepts which are active at all times.

Proposed implementation as parallel processes.

We propose that each pattern of a description be a separate process as defined in section one. To ease the problems of co-ordinating these they should be in parallel. With too many antecedent and consequent theorems for co-ordination it is difficult to see how a program will behave. We have a surfeit of demons where the main program is halted to take care of side-effects which in turn take care of more side-effects. The main task is hardly tackled. So much code must be concerned with scheduling that we have a kind of exponential explosion and any linearisation which is not sufficient to approximate to parallelism will give rise to these problems. We saw in (29) and (30) how two goals are often dependent so that discoveries by one may help the other. The order in which these discoveries are made is extremely situation dependent.

The separate processes must have channels along which to communicate. They must know of each other and the results of one or advice from it must be usable by any other. We propose that this should be done by side-effecting. Consider two processes which are part of a description. One is associated with the pattern (a fx) and the other with (b fx). The entire description is asked to find an example of itself so the separate patterns each try to find values for fx which satisfy their own predicate. The variable is treated as a common store so that if one process assigns to it the other can tell. It is also treated as an object since either process may need to assert something about the object. In particular, the predicates (a fx) and (b fx) are both asserted about fx so either process knows both. Because we do away with a global data-base we attach the assertions concerning fx to fx itself. This attachment is reversible so each process can examine fx or add to it. This does not mean that all connections must be made at compile time. During execution, processes may make more acquaintances at first indirectly through their neighbours. A process which wants to find out something may ask a friend to learn where such things can be discovered. In future cases it may then go directly.

The picture of fx built up so far seems very structural rather than procedural. This is not really the case. Suppose we asserted that fx was a "car". Then that assertion should refer to the concept car. One trivial way would be to have car be a structural description itself and to assert all the parts of this description directly about fx. This is inefficient and not very effective, since the assertions would be conglomerated with assertions about other concepts and it would be difficult to use the description as a whole. Besides we may not be able to define car as a structure. Suppose the assertion (car fx) were itself a process running in parallel with the others. Then it could keep /

/keep a watch on fx. If certain other details were asserted of fx such as the number of wheels, engine size, etc., these should "react" with the assertion car to produce a more detailed picture of the car perhaps using these details to make deductions constructively. The use of car must of course only be an instance since we don't want assertions about one car to be transferred to all. The idea of concepts reacting can only be achieved properly by parallel processes.

The possibility for having hypothesis making is very important since hypothesising appears to be important for reasoning. Suppose for instance that one of the patterns in a description finds two possible instantiations of itself. Another pattern might wish to pretend the first is correct and to see the consequences. This is done implicitly in PLANNER and CONNIVER by having separate data-bases for worlds in which the hypothesis is taken differently. PLANNER handles these by the control structure and CONNIVER also by providing contexts. A hypothesis can be made by asserting it as a fact. If it leads to a contradiction a failure can be generated by backtracking, erasing the fact and trying another hypothesis. It is possible to have two separate hypotheses going at once by keeping two contexts and running them in pseudoparallel. There is a third possibility which we find better since it explicitly mentions the hypothesis making. The fact could be "asserted-as-a-hypothesis" i.e. by looking at the fact we could later tell that it was a hypothesis. Any deductions made from it could have attached information saying "I follow from such and such a set of hypotheses". The attachment could be bidirectional so that from any fact we could work back and see what else this implies is false. In a fairly strong sense, deduction and explicitness together replace control structure.

We find many correspondences between the ideas expressed here and those expressed about ACTORS in Hewitt et al (1973). The connections are reported in a thesis draft (Stansfield, 1974).

Summary

To sum up we mention what we believe to be the most important point. This is that a limiting factor on the size of the problems able to be tackled by AI programs has been the inability of programs to form and use concepts. Until programs can use definitions of large units constructed from smaller ones any program covering a large body of knowledge will become messy and suffer from disastrous interactions. Concept building is possible if concepts are considered as relational structures. However these are difficult to interpret. Programs are far more flexibly interpreted but we have shown certain restrictions in using them. The answer lies in active descriptions. To make these work and to allow clusters of expertise we find multi-processing is necessary. This can lead to many anthropomorphisms. The idea of many processes operating simultaneously, giving each other advice, conversing, and side-effecting an environment which is itself a process is extremely suggestive. It can be dangerous if our recursion is not based on anything and we put a homunculus into each process. On the other hand it can be an endless source of ideas and computational possibilities.

Acknowledgment

The author gratefully acknowledges financial support by the Social Science Research Council for the work reported in this document.

BIBLIOGRAPHY

1. Bournce, Ekstrand & Dominic (1971) The Psychology of Thinking, Prentice-Hall.

2. Hewitt, C. (1971) Description and theoretical analysis of PLANNER: A language for proving theorems and manipulating models in a robot. Ph.D. Thesis. M.I.T.

3. Hewitt, C., Bishop, P., Steiger, R. (1973) A Universal Modular ACTOR Formalism for Artificial Intelligence. 3IJCAI. Stanford University.

4. Stansfield, J. L. (1972) [PROCESS 1]: A generalisation of recursive programming languages. Bionics Research Reports: No. 8. School of A.I., University of Edinburgh.

5. Stansfield, J. L. (1974) Programming a Dialogue Teaching System. Draft Thesis Report. Bionics Research Laboratory, School of A.I., University of Edinburgh.

6. Sussman, G. J. & McDermott, D. (1972) Why Conniving is better than Planning. M.I.T. A.I. Memo 255.

7. Winograd, T. (1971) Procedures as a representation for data in a computer program for understanding natural language. A.I. TR.17 M.I.T.

8. Winston, P. M. (1970). Learning structural descriptions from examples. A.I. Tr-231 M.I.T.

The Virtuous Nature of Bugs

by Gerald Jay Sussman

How much time has each of us spent tracking down some bug in a computer program, an electronic device, or a mathematical proof? At such times it may seem that a bug is at best a nuisance, at worst a disaster. Has it ever occurred to you that bugs are manifestations of powerful strategies of creative thinking? That, perhaps, creating and removing bugs are necessary steps in the normal process of solving a problem? Recent research at the MIT AI Laboratory [Sussman 1973] [Goldstein 1973] [Fahlman 1973] indicates that this is precisely the case.

> While "bug" is hard to define, I do not mean those trivial failures of oversight, of manipulation, or typing, that plague us continually. I mean real, conceptual errors.

Recently, I have completed the design of HACKER, a computational model of skill acquisition. HACKER is a problem-solving system whose performance improves with practice. This investigation has elucidated several important aspects of problem solving, including: the relationship of problem-solving to learning; the relationship between imperative and declarative aspects of knowledge; the nature of plans and their teleological structure; and the role of bugs and debugging in the refinement of plans.

A theory of problem solving:

A human problem-solver first tries to classify his problem into a subclass for which he knows a solution method. If he can, he applies that method. If he cannot, he must construct a new method by applying some more general problem-solving strategies to his knowledge

of the domain. In constructing the new method, he is careful to avoid certain pitfalls he has previously encountered and he may use methods he has previously constructed to solve subproblems of the given problem. The new method is committed to memory for future use. If any method, new or old, fails on a problem for which it is expected to work, the failure is examined and analyzed. As a result the method may be modified to accommodate the new problem. Often the analysis of the failure can also be classified and abstracted to be remembered as a pitfall to avoid in the future when constructing new methods.

How HACKER embodies this theory:

Please examine figure 1. HACKER, when attacking a problem (in the Blocks World [Winograd 1971]), first checks to see if he has a program in his Answer Library whose pattern of applicability matches the problem statement. If so, he runs that program. If not, he must write a new program, using some general knowledge of programming techniques applied to his knowledge of the Blocks World. Any proposed program is criticized to avoid certain bugs he has previously encountered. He may use subroutines (in the Answer Library) he has previously constructed to solve subproblems of the given problem. After criticism, the proposed solution program is tried out. The new program is stored in the Answer Library, indexed by an applicability pattern derived from the statement of the problem for which it was written, so that it can be used to solve similar problems in the future. If any program, new or old, manifests a bug when it is applied to a problem which matches its pattern of applicability, general debugging knowledge is used to classify the mode of failure. Often, the nature of the bug can be summarized and remembered as a critic. The program is patched to fix the bug and tried again.

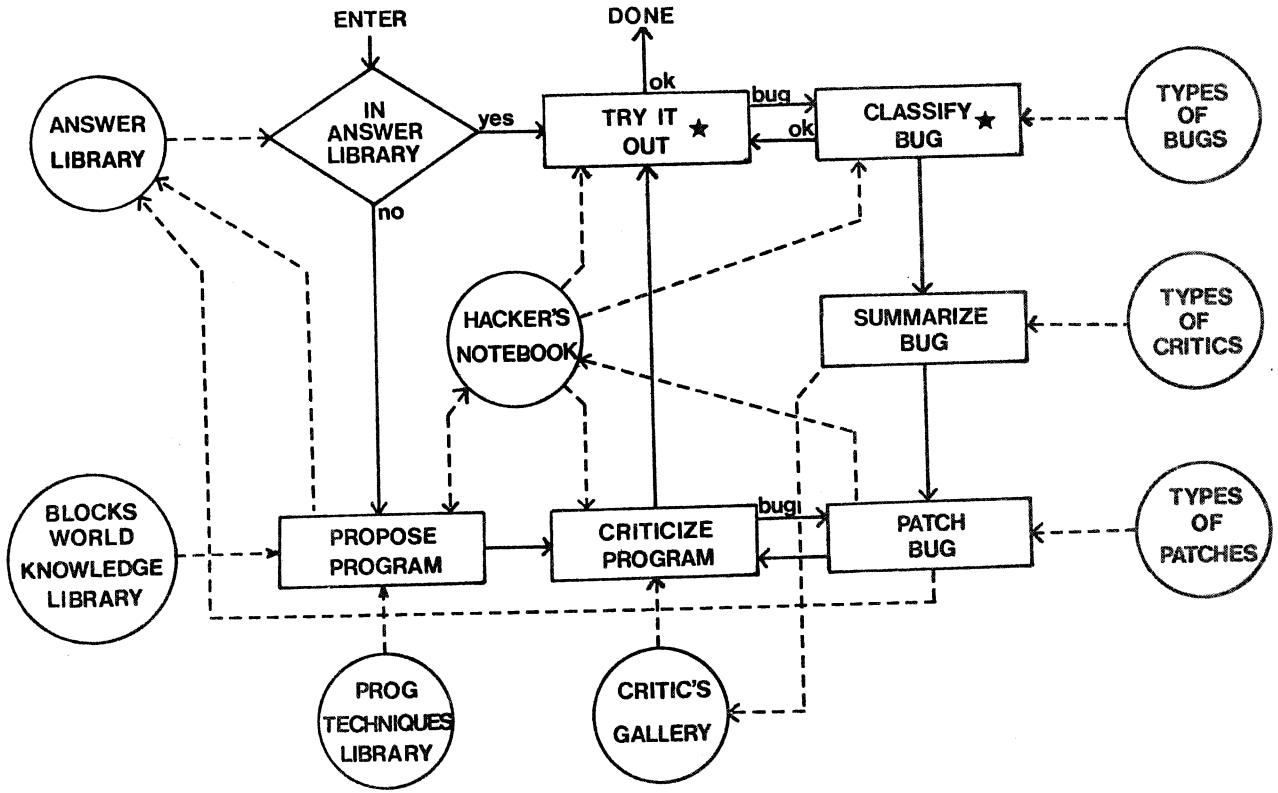

ENTER
DONE
ANSWER LIBRARY
IN ANSWER LIBRARY
yes
no
ok
TRY IT OUT ★
bug
ok
CLASSIFY BUG ★
TYPES OF BUGS
HACKER'S NOTEBOOK
SUMMARIZE BUG
TYPES OF CRITICS
BLOCKS WORLD KNOWLEDGE LIBRARY
PROPOSE PROGRAM
CRITICIZE PROGRAM
bug
PATCH BUG
TYPES OF PATCHES
PROG TECHNIQUES LIBRARY
CRITIC'S GALLERY

figure1

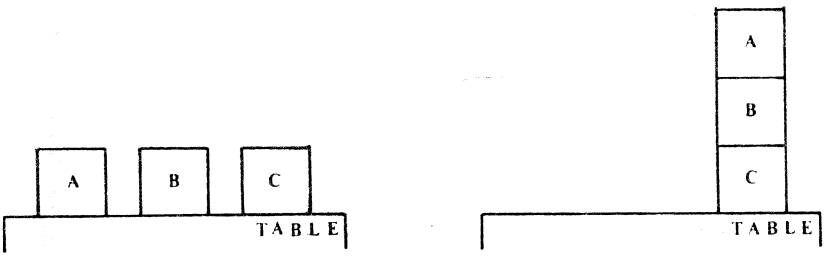
The origins of bugs:

HACKER has ways of repairing bugs when they come up, but how do bugs come up? There are several important sources of bugs. Sometimes, because of generalizations made when a new program is inserted in the Answer Library, a program is applied to a kind of situation which was not anticipated when the program was written. Other bugs result from unanticipated interactions between the steps of a proposed solution. Let us examine the genesis and repair of a bug of this latter kind.

Suppose that one is confronted with a composite goal, in which the problem is to achieve the conjunction of two conditions. In the absence of any further knowledge about the structure of the problem, what is a rational strategy to follow in attempting to solve the problem? The simplest approach, which has had great success in the history of science, is to begin with a "linear theory" -- to assume that the two subgoals can be achieved by independent processes. Thus, the linear theory plan is to break up the conjunction into its components, and then achieve each component independently, with the hope that there will be no interference between the subproblem solutions. Of course, this assumption is often false, and leads to a bug, but it is a place to start. Understanding the nature of the resulting bug will often point out the correct patch to make and may lead to a more fundamental understanding of the problem domain.

Consider, for example, HACKER's behavior on the following problem: Suppose that there are 3 blocks on the table, A,B and C, and we ask HACKER to build a 3-high tower:

(ACHIEVE (AND (ON A B) (ON B C)))



Before After

(Please assume that HACKER has already written a program to (ACHIEVE (ON x y)) for any bricks x,y.) HACKER cannot find any program in his Answer Library which matches the given conjunction problem. HACKER then goes into program proposal mode. He fishes about for a strategy which matches the problem posed. The linear theory for achieving conjunctions is retrieved. It suggests the plan:

```
(TO AND2 (ACHIEVE (AND (ON A B) (ON B C)))
  L3: (ACHIEVE (ON A B))
  L4: (ACHIEVE (ON B C)))
```

That is, in simplified HACKER syntax: first try to get A on B, then try to get B on C. If the subgoals are independent, their order doesn't matter, so the arbitrary order from the problem statement is used. The proposal is then passed by the criticizer (which doesn't know anything about this kind of problem -- yet) and tried out.

Of course, it has a bug. The program, AND2, first puts A on B. Next it tries to put B on C, but that means it must grasp B. It cannot move B with A on it (a physical restriction of the robot's hand), so it removes A from B and puts it on the table. (This is part of that Answer Library subroutine which HACKER has constructed to solve some earlier problem of the form (ACHIEVE (ON x y)) and which is

being used here.) Next, it puts B on C and is done. But it failed to achieve its overall purpose -- A is no longer on B!

Actually, in HACKER, the program would never get this far. Besides proposing the plan, the linear theory also placed the following teleological commentary for that plan into HACKER's Notebook (Figure 1):

```
(PURPOSE L3 (TRUE (ON A B)) AND2)
(PURPOSE L4 (TRUE (ON B C)) AND2)
```

These state that the author of the plan expected that A would be on B starting after line L3 and remain there at least until the program AND2 was done (the fourth position could have contained a line number in a more complex plan where L3 was a prerequisite step rather than a main step) and B would be on C starting after line L4 and remain there until AND2 was done. When a program is executed for the first time, it is executed in CAREFUL mode. In CAREFUL mode these comments are interpreted along with the lines to which they are attached. A daemon was set after L3 to protect the truth of (ON A B) until AND2 is done. This daemon interrupted the execution of L4 at the moment A was lifted off of B. The bug is thus manifest as a PROTECTION-VIOLATION and caught *in flagrante delicto*. Control now passes from the interrupted process to the bug classifier.

Types of Bugs:

We have seen how a bug can be constructed when a powerful but imperfect method of plausible inference is invoked. What do we do when such a bug comes up? Until recently, it was thought that a very good idea would have been to include a combinatorial search mechanism (e.g. backtracking) to unwind the problem solver back to some earlier point where the next most plausible proposal could be selected and

tried out. The hitch with this idea is that this kind of search rapidly leads to a combinatorial explosion -- just what is this "next most plausible" proposal? It might be that the next most plausible proposal will fail in precisely the way that the current one does and that only the one-hundredth most plausible will succeed. Perhaps the program should re-evaluate its plausibilities on the basis of this failure. That is, the program should be able to learn from its mistakes, not only so as not to make the same error again, but to be positively guided by analysis of the structure of the mistake. (See [Sussman 1972] for a more complete argument)

If this conclusion is to be taken seriously it becomes important to better understand the nature of bugs; to classify and name the bugs and repair strategies. The idea of thinking of bugs as important concepts and BUG as a "powerful idea" may seem surprising; but we suspect that isolating and systematizing them may become as important in the study of intelligence as classifying interactions has become in physics!

Now let's see how HACKER understands the above mentioned bug, which has manifested itself as a protection violation. What is its underlying cause? The basic strategy of HACKER in debugging a bug manifestation is to compare (a model of) the behavior of the misbehaving program with various prototypical bug patterns. If a match is found, the program is said to be suffering from a bug which is an instance of the prototype.

What constitutes a model of the behavior of the misbehaving program, and how is it constructed? The details of the construction of a process model are described elsewhere [Sussman 1973], but here is one scheme. At the time of the PROTECTION-VIOLATION interrupt, the bug classifier has access to an essentially complete chronological

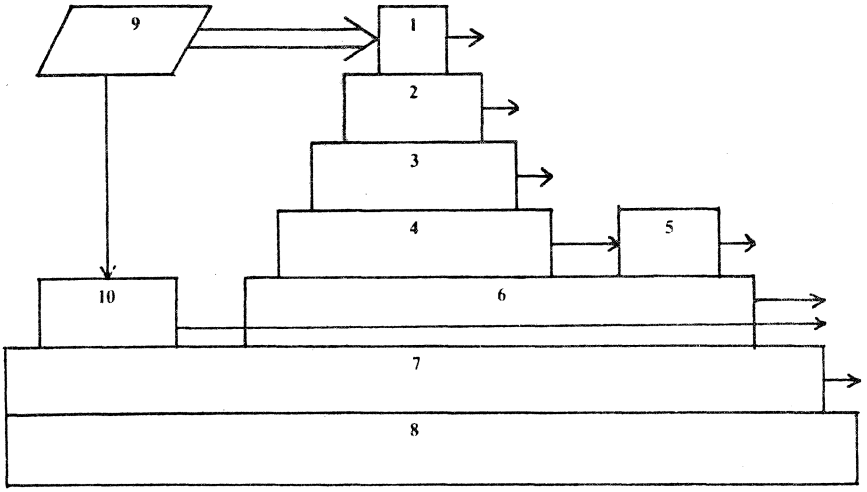
history of the problem-solving process which was interrupted. (A human debugger often uses a "tracer" to help him construct such a history, but special features of CONNIVER [McDermott 1972] provide this and more in CAREFUL mode.; HACKER also has access to a complete teleological commentary of his proposed solution and access to variable bindings and other relevant data.

The bug classifier begins by noting two pointers, the current control point, and the origin of the protection comment whose scope was violated. These pointers are then traced with the help of the relevant teleological commentary and history as follows:

Where was I?	In 1: (PUTON A TABLE)
Why?	Main Step in 2: (ACHIEVE (ON A TABLE))
Why?	Main Step in 3: (ACHIEVE (NOT (ON A B)))
Why?	Main Step Generic in 4: (ACHIEVE (CLEARTOP B))
Why?	Prerequisite Step for 5: (PUTON B C)
Why?	Main Step in 6: (ACHIEVE (ON B C))
Why?	Main Step in 7: (ACHIEVE (AND (ON A B) (ON B C)))
Why?	8: COMMAND
Who complained?	9: Protect (TRUE (ON A B))
Why?	Result of 10: (ACHIEVE (ON A B))
Why?	Main Step in 7: (ACHIEVE (AND (ON A B) (ON B C)))

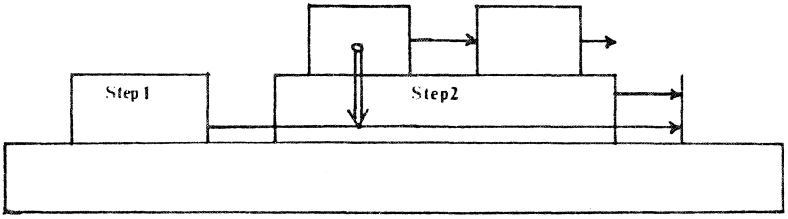
A Main Step is a step in a program whose purpose is to achieve a result which contributes to the overall goal of the program. Its purpose comment states that the result achieved by that step is needed until the program returns to its caller. A Prerequisite Step is one whose purpose is to set up for the execution of some other step. The result of this trace can be summarized in the following schematic

diagram of the buggy process:



Each box in this diagram is a stack frame of the process. The horizontal dimension is its extent in time; the vertical dimension is the depth of functional application. Thus, the blocks labeled 7 and 8 (the AND2 frame and command level frame respectively) exist from the time the command is typed until it returns. Frame number 10 is the frame of line L3:(ACHIEVE (ON A B)) and frame number 6 is the frame of line L4:(ACHIEVE (ON B C)). Frame number 9 is special -- it is the protection daemon on the result of L3. It points at the accused violator. The horizontal arrows indicate the scopes of the purposes of the steps. Arrows which terminate on boxes are prerequisite step scopes. (In this trace there is only one prerequisite scope, from 4 to 5.) Other arrows are main step scopes.

This structure matches a particular prototype bug called PREREQUISITE-CLOBBERS-BROTHER-GOAL (PCBG):



By this I mean a bug which is due to an interaction between two program steps whose purpose scopes terminate at the same time. A prerequisite step (or any number of main steps for a prerequisite step -- the matcher can compress main step scopes but prerequisite step scopes must be explicitly represented -- [Sussman 1973]) for a main step in the code for step 2 clobbered the result of step 1. In this case, the process of achieving (CLEARTOP B), a prerequisite of (PUTON B C), which is a main step in L4:(ACHIEVE (ON B C)), destroys the truth of (ON A B), the result of L3:(ACHIEVE (ON A B)). Since both L3 and L4 are main steps in AND2 their purpose scopes terminate when AND2 returns.

Just how much generality is there in the concept PCBG? Perhaps it is just peculiar to the Blocks World? In fact, PCBG is a very common form of non-linearity.

If, for example, one wants to paint the ceiling, it is simultaneously necessary that the paint be on the platform and that the painter be on the ladder. The linear strategy is to achieve each subgoal independently. The painter can either first lift the can to the ladder platform, and then climb the ladder, which works; or he can

first climb the ladder and then lift the can, which doesn't work. Once he is on the ladder, he has no access to the can on the ground. He must first come down to get the paint (clobbering the previously achieved subgoal of being on the ladder). Climbing down -- to achieve the prerequisite to lifting the paint can -- has clobbered the brother goal of being on the ladder.

In programming, too, one often runs into PCBG's. Consider the problem of compiling the LISP expression (F 3 (G 4)). If the argument passing convention is to load the arguments into successive argument registers and then call the function, we see that the call to function F requires that 3 be in register 1 and the result of (G 4) be in register 2. If we try the obvious order -- first put 3 in register 1, then calculate (G 4) and put it in register 2 -- we find that we must load 1 with 4 to call G, thus clobbering the brother goal of having 3 in register 1.

Fixing the Bug:

Now that the bug is classified, can we come up with a modification to the plan (program) which eliminates the bug? The offending prerequisite must, in any case, be accomplished before its target step. Its scope must extend until that step. But since the first and second conjuncts are brothers (they are both for the same target), their scopes must overlap. Thus, since the scope of the first conjunct and the scope of the prerequisite of a main step for the second step are incompatible, the only way to prevent the overlap is to move the step for the second conjunct ahead of the step for the first. We must assign an order to the plan. Thus, the patcher changes the plan as follows:

```
(TO AND2 (ACHIEVE (AND (ON A B) (ON B C)))
  L4:  (ACHIEVE (ON B C))
  L3:  (ACHIEVE (ON A B)))
```

A new comment is added to HACKER's notebook summarizing this ordering constraint (BEFORE L4 L3). The program is patched and the result works. In this case a critic is compiled which summarizes what has been learned (How this happens is beyond the scope of this paper -- see [Sussman 1973]): If for any blocks a, b, and c we are proposing a program which has lines with the purposes of getting a on b and b on c, we must compile the line which puts b on c before the one which puts a on b. Applied recursively, this advice is sufficient to ensure that any program which piles up bricks will do it in the correct order -- from the bottom-up.

Other bugs:

Of course, not every bug is a PCBG -- not even every bug which manifests as a protection violation. If, for example, we try to build an arch -- (ACHIEVE (AND (ON A B) (ON A C))) -- with a linear theory plan, the bug will manifest as a protection violation but no interchange or other simple modification of the linear theory plan can succeed. This kind of bug is a DIRECT-CONFLICT-BROTHERS (DCB) which can only be resolved using more Blocks World knowledge. In [Sussman 1973] I classify three other types of bugs (but not DCB).

Conclusions:

We can draw the conclusion that to be effective, a problem-solver need not know the precise way to solve each kind of problem. Perhaps a better strategy is to attempt to break a hard problem up into subproblems. Sometimes these subproblems can be solved independently, in which case the linear theory plan will work.

Sometimes the steps of the plan will interact and debugging will be necessary. And sometimes, because of prior experience, we may know that a particular kind of problem may require a particular kind of nonlinear plan, such as the ordered plan required for the problem discussed here.

The appearance of a few bugs need not be seen as evidence of a limitation of problem solving ability, but rather as a step in the effective use of a powerful problem solving strategy -- approximation of the solution of a problem with an almost-right plan. This strategy becomes powerful if the bug manifestation that results from the failure of such an almost-right plan can be used to focus the problem-solver on the source of the difficulty. A problem-solver based on debugging need not thrash blindly for an alternate plan but can be led by the analysis of the failure -- provided that adequate bug classifying and repairing knowledge is available.

Thus, I believe that effective problem solving depends as much on how well one understands one's errors as on how carefully and knowledgably one makes one's initial choices at decision points. The key to understanding one's errors is in understanding how one's intentions and purposes relate to his plans and actions. This indicates that an important part of the knowledge of a problem-solver is in teleological commentary about how the subparts of the performance knowledge relate to each other so as to achieve the overall goals of the system. It also indicates the need for knowledge about how to trace out bugs and about the kinds of bugs that might be met in applying a given kind of plausible plan.

Bibliography

[Fahlman 1973]
Fahlman, Scott
A Planning System For Robot Construction Tasks
AI TR-283 (May 1973) MIT-AI-Laboratory

[Goldstein 1973]
Goldstein, Ira
Understanding Fixed Instruction Turtle Programs
PhD Thesis (September 1973) MIT-AI-Laboratory

[Hewitt 1971]
Hewitt, Carl
"Procedural Semantics: Models of Procedures and the Teaching of Procedures"
Courant Computer Science Symposium 8 (1971)

[McDermott 1972]
McDermott, D.V. and Sussman, G.J
The CONNIVER Reference Manual
AI Memo 259 MIT-AI-Laboratory (May 1972) (Revised July 1973)

[Sacerdoti 1973]
Sacerdoti, Earl
"Planning in a Hierarchy of Abstraction Spaces"
IJCAI-73 (1973)

[Sussman 1973]
Sussman, G.J.
A Computational Model of Skill Acquisition
AI TR-297 MIT-AI-Laboratory (August 1973)

[Sussman 1972]
Sussman, G.J. and McDermott, D.V.
"From PLANNER to CONNIVER - A Genetic Approach"
FJCC (1972)

[Winograd 1970]
Winograd, T.
Procedures as a Representation for Data in a Computer Program for Understanding Natural Language
AI TR-17 (MAC TR-84) MIT-AI-Laboratory (February 1971)

COMPUTER PERCEPTION OF CURVED OBJECTS

Kenneth J. Turner

ABSTRACT

Image-processing techniques are described which reduce TV pictures of curved objects to a line-drawing representation. Details are given of an application of the hierarchical synthesis technique to flexible and imperfection-tolerant recognition using such representations. Extensions of Waltz's methods are outlined which permit the analysis of real scenes of curved objects.

DESCRIPTIVE TERMS

curved objects, curved surfaces, curve-fitting, hierarchical synthesis, image-processing, object recognition, scene analysis, segmentation.

INTRODUCTION

This paper is intended to acquaint the reader with the main results of my thesis (Turner, 1974), but it can do little more than whet his appetite. The aims of this research were twofold : first, to evolve techniques for the perception of curved objects; and second, to experimentally demonstrate the validity and utility of these, both individually and as part of a complex system. The types of object that have been studied are those with curved surfaces which form well-defined intersections (e.g. mugs, cones, and toruses). However, the methods developed for these have been shown to be applicable to other classes of objects (e.g. polyhedra).

Programs have been written which : (a) process TV images of curved objects to line-drawing form; (b) identify objects from such descriptions, despite imperfections; and (c) analyse hand-generated line-drawings depicting scenes of curved objects. By modifying the scene-analysis program to be tolerant to some extent of defects in the picture, it has been possible to integrate a subset of these capabilities in a complete system for analysing simple scenes of curved and polyhedral objects. The following sections describe some of the techniques used by the system.

IMAGE-PROCESSING

Since the image-processing routines are used in a top-down, hypothesis-generation fashion, simplicity and cheapness were the design aims. Edge-detection is by a simple gradient operator whose output is thresholded to obtain spurs, which are elementary boundary steps. Intensity discontinuities are tracked by a myopic "bug" which is directed by the pattern of spurs present in a 5 x 3 window through which it views the picture. Boundaries are chain-coded to form data-structures called curves. Intersections of curves with themselves or with others are detected by the use of a curve membership array, which contains a pointer to a curve for each of its points.

Curves are first of all segmented at intersections, that is, at junctions. Further segmentation is carried out by a process which approximates a curve by a sequence of straight and circular arcs. An arbitrary plane curve may be represented using ψ (the angle its tangent at some point makes with a fixed direction, e.g. the x-axis) and s (the arc length measured from a fixed point on the curve). The usefulness of the transformation from x-y coordinates to ψ-s coordinates is that a curve which consists solely of straight or circular segments is mapped onto a ψ-s curve comprising only straight sections.

The segmentation problem is thus reduced to one of finding straight lines. A recursive fitting procedure is employed which is similar to that used by other workers (e.g. Horn, 1971). The point of maximum deviation from a base-line joining the endpoints of the ψ-s curve is found; segmentation takes place at this point if the deviation is above a threshold. The line-fitter then calls itself recursively to deal with the two new segments, a special check being made for a segment which is nearly parallel to a base-line.

The preliminary segmentation into straight and circular arcs is refined by fitting conic sections; this may suggest re-merging segments or breaking them up still further. Segment breakpoints are adjusted so as to optimise the fit of curves. The algorithm given by Rosenbrock (1960) is used to fit conic sections by minimising an error term of the form

$$\sum \left\{ g(x_i,y_i) / \left| \text{grad}(g(x_i,y_i)) \right| \right\}^2$$

where $g = ax^2+bxy+cy^2+dx+ey+f$. This was found to overcome the problem reported by Agin (1972) of fitting excessively flattened curves when using as the error term

$$\sum \left\{ g(x_i,y_i) \right\}^2 .$$

Junctions are derived on the basis of segment endpoint proximity using a segment membership array similar to that maintained for curves. A topological description of the picture is built up in terms of junctions, lines, and areas. A typical picture analysis at this stage is shown in fig. 1. Line parameters are adjusted so that lines pass exactly through junctions. Classification of junctions into types is made on the basis of three features invariant under moderate changes in viewpoint : (a) the number of lines; (b) the relative size of the junction's largest angle, with respect to 180°; and (c) the sizes of the sets of continuous lines*. These features are used to determine the type of a junction; the relative sizes of the angles and the relative curvatures of the lines are used to distinguish a number of variants of each type. Junction classification completes the work of the image-processing routines.

---

* Two lines are said to be continuous if the angle between them is 180°. Every line in a continuous set is continuous with at least one other.

OBJECT RECOGNITION

Recognition is achieved using the technique of hierarchical synthesis (Barrow, Ambler, and Burstall, 1972). The idea is that complex objects should be described in terms of simpler ones and their relationships, all objects ultimately being defined in terms of a set of primitives (e.g. segments or regions).

The heart of the synthesiser is a job scheduler which processes a queue of jobs on a priority basis. Each job has a number of associated parameters which include : an interest-value (used to order the queue); a state component (used to indicate whether the job has been run, is waiting to be run, etc.); a list of pre-jobs (on which the job depends); a list of post-jobs (which depend on the job); a function to be run (the job itself); and the results of the job (generated by its function). As jobs are processed, their results are used to dynamically update the interest-values of others on the queue. Other similar interactions between jobs form part of a promotion scheme which ensures that the most promising jobs are run first.

For recognition purposes a job is allocated to each object in the hierarchy, sub-objects corresponding to pre-jobs. The job function for an object takes an n-tuple of instances of sub-objects and checks their relationships; if the n-tuple meets the specification of the object it is added to the job results list. The hierarchy may be run bottom-up (building all possible descriptions from the primitives) or top-down (looking only for instances of certain objects); a combined top-down/bottom-up mode is also possible. Fig. 2 illustrates a typical hierarchical decomposition of an object.

Object specifications are deliberately made tolerant so as to cope with imperfections in the picture. Each test of a property or relation in the specification is associated with a confidence-value in the range 0 to 1. These confidence values are combined in a weighted linear polynomial which is thresholded to obtain a decision. This scheme has the advantage of being able to handle in a uniform manner imperfections due to noise and occlusion; if part of an object is missing, it merely makes no contribution to the decision polynomial.

A direct comparison was made between the performance of the synthesiser and the object recogniser of Barrow and Popplestone (1971), the only other program which carries out a similar task. Both programs were run on the same set of test pictures - 5 examples of each of an object set comprising : ball, cup, cylinder, doughnut, hammer, mug, pencil, spectacles, tube, and wedge. Barrow and Popplestone's program made a correct identification in 95% of cases in an average time of 270 secs. (137 secs. for region-finding, 133 secs. for description and matching). The hierarchical synthesiser also achieved a recognition success rate of 95%, but in an average time of only 55.4 secs. (48 secs. for line-finding, 7.4 secs. for description and matching). It is noteworthy that the speed-up of description and matching by a factor of 18 was obtained without loss of reliability;

the effective speed difference is even greater because the synthesiser had to match descriptions containing almost four times as many picture elements. The flexibility and imperfection-tolerance of the synthesiser also enables it to cope with situations which Barrow and Popplestone's program cannot properly handle, for example, when there are several occluding objects.

SCENE ANALYSIS

The ideas developed by Waltz (1972) for analysing shadowed polyhedral scenes are of wider applicability. Waltz's principles have been generalised and extended to the analysis of scenes of curved objects. Because general curved objects are too unconstrained to be tractable, certain restrictions were imposed, the most important of these being to disallow surface splines. All the points on the visible portion of a surface must therefore be of the same type : parabolic (e.g. planes, cones, cylinders), elliptic (e.g. spheres), or hyperbolic (e.g. saddles).

The mechanism for generating curved object junction labels is based on the observation that two planes may approximate a curved surface in the vicinity of a corner. A corner composed of both plane and curved surfaces may therefore be approximated by a purely polyhedral one. Note that a convex (concave) surface will give rise to a convex (concave) edge, and that the convexity or concavity of the other edges will be preserved. This process may be applied in reverse, a polyhedral corner being regarded as generating one with curved surfaces. The fact that convexity and concavity are preserved means that the labels of the non-planar corner can be easily derived from those of the planar one. To determine the labels for a certain class of curved objects, the procedure is therefore to obtain the labels for the appropriate degree of polyhedral corner and apply the transformation.

Junction labels must be generated for the cases of : corners, Tees, shadowed corners, shadowed Tees, shadows cast on surfaces, and shadows cast across edges. Junction labels have been derived for certain interactions between planar, conical, cylindrical, and elliptical surfaces; hyperbolic surfaces, being of less interest, have not been considered. Fig. 3 shows some typical labellings.

The illumination over a curved surface may vary from directly-illuminated to self-shadowed. It is also possible for a shadow cast across a convex surface to simply peter out. It is therefore necessary to associate illumination information not with the areas of the picture but with the lines, in the neighbourhood of junctions. This does not eliminate the problem, however, for the nature of the illumination may be different at opposite ends of a line. A related difficulty is that the type of an edge may also vary from one end to another. The solution adopted was to relax the consistency requirement that the interpretation of a line must be the same at all points along it. Instead, transition rules are used which specify how illumination and edge labels may transform into each other along the

length of an edge. The edge-type transition rules are obtained as a by-product of the procedure for producing junction labels. Fig. 4 indicates transitions of this sort and the rules which deal with them.

Fig. 5 is typical of the kinds of line-drawing that can be analysed by these methods. Allowing for differences in implementation, the program takes roughly four times longer to analyse a scene of curved objects than Waltz's program does to analyse a polyhedral scene with a comparable number of junctions. This speed difference stems from the increased size of the label data-base (about 40% of the junctions in fig. 5 have over 3000 independent labellings), the greater complexity of the consistency rules, and the diminished value of illumination information. It was found that illumination labels do not bind the interpretations of separated parts of the picture as strongly as in the polyhedral case. Indeed, it was discovered that ignoring illumination information entirely does not give rise to much ambiguity with curved objects : consistency of surface type is the main cohesive force. A relaxation of the "no splines" rule would therefore probably degrade performance considerably.

By formulating Waltz-like analysis in clique-finding terms, it has been possible to use the same program to interpret real data despite imperfections, the aim being to find the largest consistent subgraphs of the picture. This has made it possible to integrate Waltz-like methods into a complete system for understanding real scenes with shadows, containing both curved and polyhedral objects. Because of computational limitations and the difficulty of obtaining effective feedback, only simple scenes of a few objects have so far been analysed. The same principles, however, ought to be extendable to the analysis of scenes like fig. 5 in actuality.

ACKNOWLEDGEMENTS

This research was carried out in the Department of Machine Intelligence, Edinburgh University, under the able leadership of Prof. Donald Michie. I am considerably indebted to Dr. Harry Barrow for his invaluable guidance and advice during this work. I am grateful to the Science Research Council and the Dalle Molle Foundation for financial support.

REFERENCES

Agin, G.J. (1972) "Representation and description of curved objects" (thesis) A.I. Report AIM 173, Computer Science Dept., Stanford University.

Barrow, H.G., Ambler, A.P., and Burstall, R.M. (1972) "Some techniques for recognising structures in pictures" Frontiers of Pattern Recognition (ed. Watanabe, S.) 1-29, Academic Press, New York.

Barrow, H.G. and Popplestone, R.J. (1971) "Relational descriptions in picture-processing" Machine Intelligence 6 (eds. Meltzer, B. and Michie, D.) 377-396, University Press, Edinburgh.

Horn, B.K.P. (1971) "The Binford-Horn line-finder" Vision Flash 16, A.I. Lab., M.I.T.

Huffman, D.A. (1971) "Impossible objects as nonsense sentences" Machine Intelligence 6 (eds. Meltzer, B. and Michie, D.) 295-323, University Press, Edinburgh.

Rosenbrock, H.H. (1960) "An automatic method for finding the greatest or least value of a function" Comp. J. 3, 175-184.

Turner, K.J. (1974) "Computer perception of curved objects using a television camera" (thesis) School of A.I., Edinburgh University.

Waltz, D.L.(1972) "Generating semantic descriptions from drawings of scenes with shadows" (thesis) A.I. Lab. Report TR-271, A.I. Lab., M.I.T.

Fig. 1 : A typical segmented picture

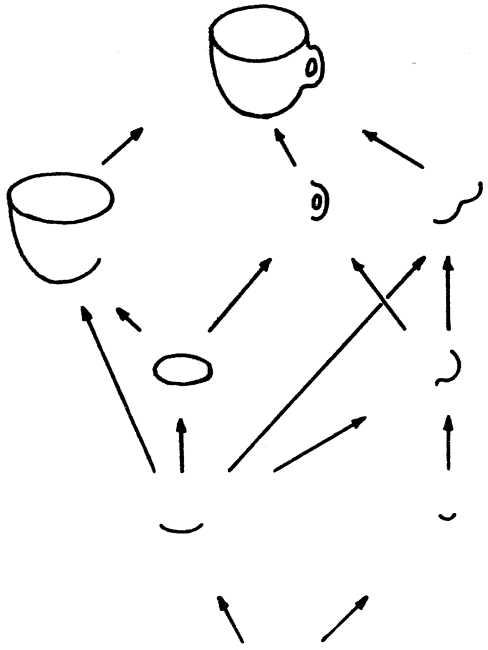
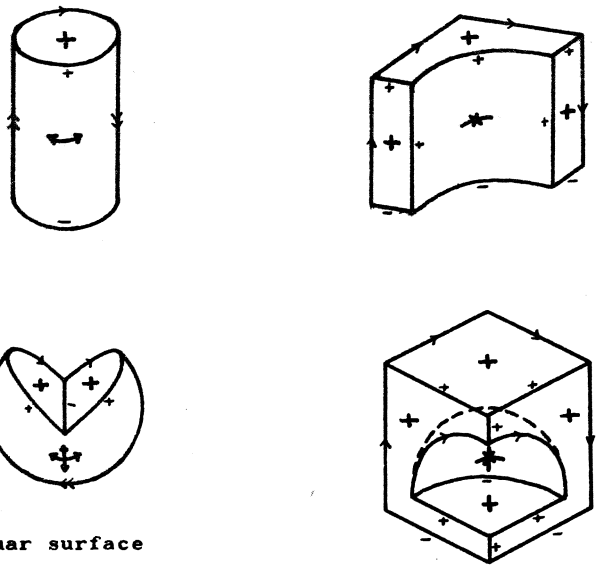


Fig. 2 : A hierarchical representation of a cup



KEY :

+ planar surface

↔/✱ convex/concave conical or cylindrical surface

✣/✱ convex/concave elliptical surface

Fig. 3 : Some representative labellings

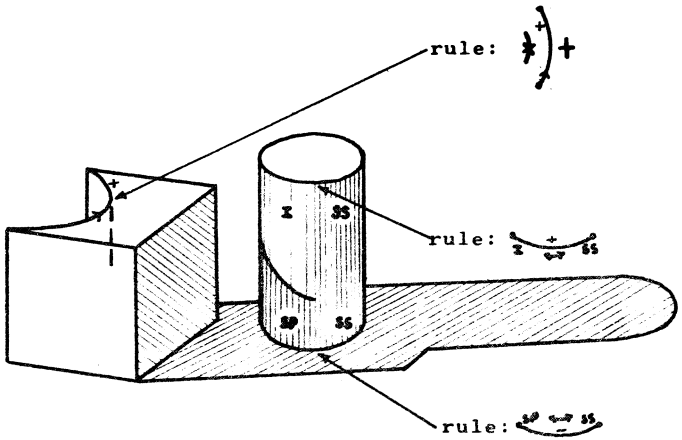


Fig. 4 : Some edge- and illumination-type transition rules

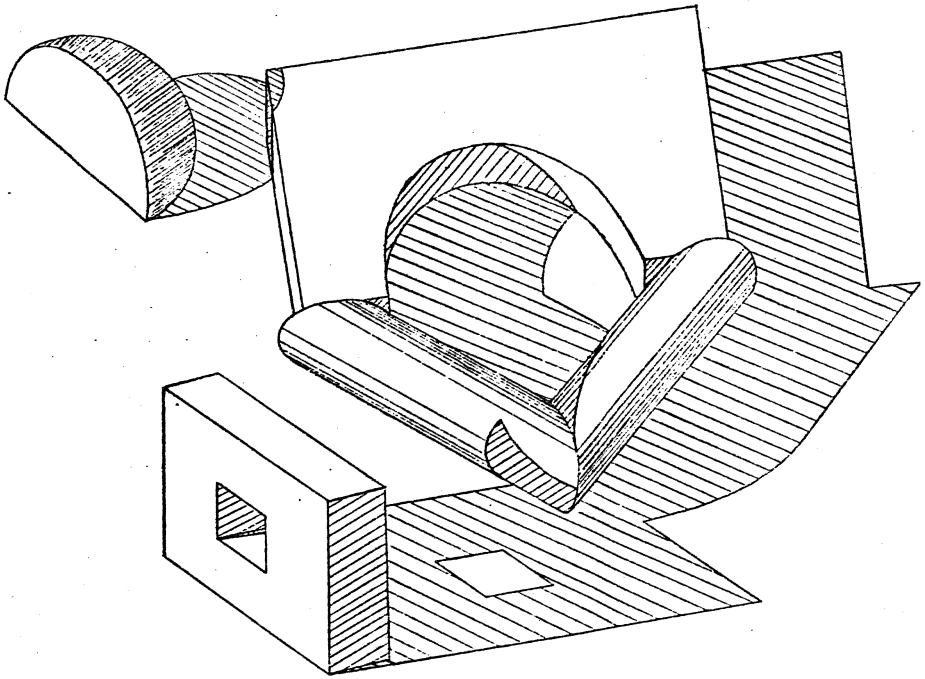


Fig. 5 : Typical line-drawing analysed by the program

ACTION PERCEPTION

Dr. Sylvia Weir, Bionics Research Laboratory, School of Artificial Intelligence, University of Edinburgh.

Recent scene analysis work in Artificial Intelligence assigns a central role to the support relation in making sense of static scenes. We expect pictorial evidence that the laws of gravity are being satisfied. Looking for 'pictorial' evidence for dynamic laws of moving objects might be a way of extending the insights gained in scene analysis to the area of motion perception. We choose to look at a very simple action, namely that of one thing hitting another and setting it in motion, and use the very well documented account of the classical researches by Michotte and his colleagues at Louvain University over the past 40 years or so into the perception of causality (1963) as a basis for a proposed computer model of action perception.

1.0 In the Michotte experiments, the action is presented to the subject in the form of coloured 2-D objects moving across a screen. Michotte uses the verbal responses of his subjects to this (and to a wide variety of other actions) to investigate the principles of structural organisation which govern the perception of kinematic forms, in much the same way as other members of the Gestalt school have studied the perception of static forms. For him the 'causal impression' is

> "... directly experienced (his italics). There is no question of an interpretation nor of a 'significance' superimposed on the impression of movement". (p21)

It seems fruitful to relate his experimental findings and structural analyses to the constructivist scheme underlying contemporary scene analysis work in the Artificial Intelligence field. Clowes (1973) gives an exposition of this approach, using Roberts' (1965) work on machine perception of three dimensional solids as a computational elaboration of the ideas of Bartlett (1932).

> "For all scenes but especially those involving inter-object occlusions the completion of segmentation in Roberts' program is literally based on "a constructive process" in which the 3-D geometry of the model is related to the (incomplete) 2-D geometry of the stimulus pattern by... (the) theory of the picture-taking process. The constructive process is of course the generation of a predicted picture of the selected model, given the point-to-point pairing of picture cue e.g. convex quadrilateral with model property e.g. rectangular face. The picture segment is only accepted if the predictions derived from the model - specifically the picture locations of other vertices of the selected model - are confirmed."

We seek to extend the range of stimulus clues used in static scenes to include movement patterns, and attention directing instructions of the experimenter, and so widen the class of possible models (candidate schemata) used to generate hypotheses about what is going on.

2.0 Description of the Experiments

Michotte uses a remarkably ingenious method of presenting a wide variety of kinetic combinations by rotating a disc bearing appropriately positioned painted coloured stripes, behind a screen which has a/

/a horizontal slit in it (fig. 1). He describes 102 experiments. Impressed by the complexity of the underlying ideas, we restrict our attention to the few described below, and two more described in 3.8.

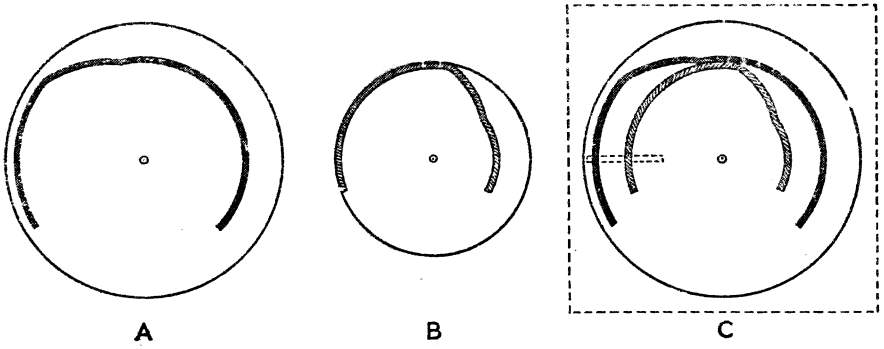


FIG. 1. Combination of discs for use in experiments on launching. Scale 1 : 10.

Experiment 1

Using this apparatus, he achieved "a uniform white background on which stand out two squares of side 5mm. One, a red square, is in the centre of the slit; the other, a black square, is 40mm. to the left of the first. We shall call the black square 'object A', and the red square 'object B'. The subject fixates object B. At a given moment object A sets off and moves towards B at a speed of about 30cm. per sec. It stops at the moment when it comes into contact with B, while the latter then starts and moves away from A, either at the same speed or, preferably, at an appreciably lower one, e.g. 6 or 10cm. per sec. Then it stops, after covering a distance of 2cm. or more, according to the speed adopted. The result of this experiment is perfectly clear; the observers see object A bump into object B, and send it off (or 'launch' it), shove it forward, set in motion, give it a push. The impression is clear; it is the blow given by A which makes B go, which produces B's movement."

Experiment 2 (Michotte's experiment 7)

An entirely different response is elicited by changing only one condition in experiment 1. The subject is asked to fixate a point 7cms. above or below the point of impact so that he is looking at the point of impact indirectly. In these conditions what the subject sees is "a single object A, travelling the whole length of the slit in a continuous movement, and on its way passing over another object which is stationary at the centre". This effect is present whether or not the observer notices the colour change. Sometimes the stationary object is described as having A's colour throughout. Michotte gives no name to this effect. We call it the passing response.

Experiment 3 (Michotte's experiment 29)

If a delay of more than 1/5 sec. is introduced between the arrival of A and the departure of B, an impression of successive independent movements is given.

Experiment 4 (Michotte's experiment 40)

This is the same as experiment 1 except that object A moves at one of the following speeds:- 29, 25, 22, 18, 15cm. per sec., while B moves at a speed of 40cm. per sec.

As long as the ratio of A's speed to B's remained less than 1:1.8 the usual launching effect was seen. When the speed of B becomes noticeably greater than that of A (an A:B ratio of 1:2.7), a triggering effect was reported by experienced observers.

"it is as if A's approach frightened B, and B ran away"
"it is as if A touched off a mechanism in B and set it going".

3.0 Modelling a Michotte experiment

A program is currently being implemented in which a model of the world is represented by sets of items and methods in a version of the CONNIVER programming language (McDermott & Sussman, 1972) implemented in POP-2 by Steve Hardy (1974). Input to the program is in the form of descriptions such as might be produced by a coloured line drawing analyser. For example, the starting situation:

"a red square is seen at the centre of the slit"

gives rise to an input description of the form

[At B midscreen] [[centposition [6 1]][shape square][colour red]]

i.e. an item with an associated property list, which is added to the data base.

3.1 Experimental instructions are input directly. For example, in experiment 1

[fixate midscreen].

This is more than an instruction about where to look. Experimenters, unlike illusionists, are interested in directing attention to where things will happen. Accordingly we would like this instruction to generate the expectation of an event. We use an if-added method triggered by the pattern

[fixate *place] (variable names are proceeded by *)

so that adding an item containing such a pattern to the data base generates

[willhappen *event midscreen]

and sets up a process which looks for participants of the expected event, and evidence for their participant status e.g. active-agent status, passive-object status. Instantiation of the event will depend on the kinetic pattern being built up.

3.2 To provide a way of generating descriptions of the movements taking place, we represent the process continuum as successive time slices, or conceptual snapshots, depicted as a frame sequence rather like a strip cartoon (fig. 2). It is as though the observer takes successive samplings of the movement processes and forms descriptions of each, so that the difference-descriptions between 2 successive frames express the changes which have occurred during a particular time-interval. Such differencing is a pervasive phenomenon occurring in many schemes of analyses from, for example, the low-level 'retinal' differencing of Lamontagne (1974) to the high level difference-descriptions used by Evans in his ANALOGY program (1965) and by Winston's program for LEARNING STRUCTURAL DESCRIPTIONS (1969).

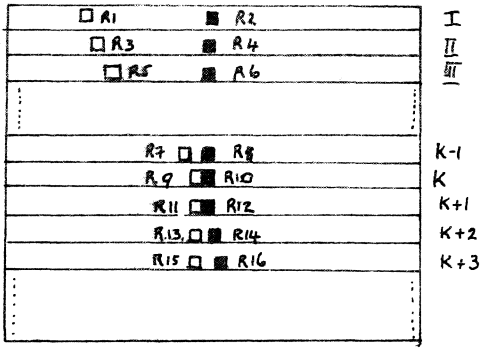


Fig. 2

There will in general be more than one way of pairing picture regions in successive frames and we need a set of rules for deciding which of the possible pairings corresponds to an ENDURING OBJECT IN MOTION. Evidence is weighed in the context of ongoing expectations.

3.3. To illustrate the steps involved in identifying the moving object we consider the first 2 frames in fig. 2. The pair R2.R4 is an obvious candidate for a match since it gives perfect agreement on position and colour, while R1.R3 is a better match than R1.R4. In each case, the presence or absence of differences is interpreted thus:-

The exact match between R2 and R4, i.e. no change in position, generates
[B stationary midscreen]
The mismatch between R1 and R3, i.e. the position difference, produces
[A moves][speed medium]

3.4 Once a movement has begun, we take a different view of what constitutes an appropriate match. In the comparison between two successive frames, we are now looking for that region in the second frame which most closely corresponds to the predicted next position of a region in the first frame and not to its absolute position. So now an exact match, for example that between R3 in frame 2 and R5 in frame 3, generates no surprises, whereas a mismatch at this point, which would correspond to a change in velocity, would require explaining.

3.5 The next point exemplifies a crucial feature of the model, viz. the co-existence of several processes, any one of which is ready to pounce on a new piece of information to use for purposes of its own. The CONNIVER programming language provides facilities for doing just this.

We recall that the fixation instruction in experiment 1 sets up a process on the lookout for participants in some as yet unspecified event to take place at the middle of the screen (a combination of if-needed and if-added methods does this quite naturally). The addition of the item
[A moves]
to the database (3.3) activates this process. In the context of the expected event, 'midscreen' forms the point of reference for A's movement, which becomes
[A movesto B]
and A's approach to a stationary object in its path and at the required place is seized upon to generate
[A agentof *event]
[B objectof *event]

The expected event now begins to look very much like a possible collision. The program makes this assignment

[willhappen collision midscreen]

and starts up a search for evidence of such an impact i.e. adds an if-added method whose pattern is

[*agent nextto *object].

When invoked, this has a twofold effect. It modifies the region-pairing process and sets up a search for consequences of the impact. (See 3.62)

Until this happens, control passes back to and remains with the main frame comparison process, and analysis proceeds through the frame sequence.

3.6 On arriving at frame K (see fig. 2), the program takes in as part of the input description of this frame, an item which corresponds to the interpretation of the T junction and the shared line between R9 and R10 i.e. the fact that R9 and R10 are touching. At this point the difference in the context of expectations in experiments 1 and 2 makes itself felt.

3.61 In experiment 2, in which a point above the screen is fixated, the events being observed are in the periphery of the field of vision. None of the expectations outlined in 3.1 and 3.5 have arisen. The pairs R7.R9 and R8.R10 were the last chosen, and the predicted next position continues to serve as the criterion for an exact match. R9.R12 is the obvious match, leaving R10.R11 as the stationary object. Inevitably R12.R14 and R11.R13 are linked and so we get the single moving object which passes over a stationary object - the passing effect.

In a footnote Michotte refers to an observation which is beautifully explained by the above. Some subjects, he tells us, who "apparently observe in a particularly analytical way" see a small retreat (the pairing R10.R11) of the stationary object as the moving object passes over it.

3.62 In contrast, in experiment 1, A's arrival next to B signals the crucial point in the action sequence for which our monitor set up in 3.5 has been waiting. It takes control, over-rides the prediction concerning the next position of A and instructs the region-pairing routines to treat the next frame comparison as it did the first two frames in the sequence viz. to allow the smallest absolute change in position to win. Accordingly R9.R11 and R10.R12 are paired and both objects are stationary.

Now the description of frame (K + 1) is entered and again the next-to method is triggered. This affects the region-pairing process in the same way and produces the pairings R11.R13 and R12.R14, which yield

[A stationary]

[B moves][speed slow].

B's movement is recognised as the sought-after consequence of the impact. Consequences are items of the form

[consequence *event *eventlist]

In this case the appropriate instantiation is

[consequence collision[[B moves]]].

A limit is set to the number of times this monitor can be called in the same collision sequence. When this limit is reached, or before if a consequence-item is asserted, the monitor kills itself. This suicide embodies the experimental fact that a delay in B's movement prevents its linkage with A's movement. This is the situation in experiment 3. Finally we notice that the onset of B's movement automatically resets the region-pairing routine to use the predicted next position for its nearest region comparison as described in 3.4. Frame comparison now proceeds uneventfully until the end of the sequence.

Notice that the analysis outlined for experiment 1 goes through for experiment 4 too - the difference between the impressions in these two experiments is discussed next.

3.7 At the end of the frame sequence analysis, the database contains a series of items tagged by the frame number in which they occurred, which together form a description of the movements which have taken place. Now the episode-interpreter takes over. This consists of a set of methods incorporating knowledge about "pushing" and "triggering", e.g. constraints on the relative speeds of the participants which allow the interpretation "pushing". A Michotte footnote (p112), informative as ever, refers to "a curious (sic!) agreement between the operation of the laws of perception and the laws governing the physical world". The crucial feature giving rise to the triggering effect of experiment 4, is the speed of B's withdrawal after impact, which is suffiently greater than that of A's approach to evoke a different explanation. Newcomers to the experiments will see "launching" as long as B's speed is less than 5 times that of A; but in the case of experienced subjects,the impression of "launching" changes to "triggering" if B moves more than twice as fast as A.

Michotte describes a variety of triggering effects. Some observers postulate the the existence of a mechanism in B which is set off by A's arrival; others "see" a predator-prey relationship between the participants with B fleeing from A. We have decided not to distinguish subcategories of triggering in the current model.

3.8 Two further experiments are cited briefly to demonstrate the way the dynamic structuring of the movement pattern is sensitive to emerging symmetries, and to emphasise the important of observational attitudes.

We use a graphical reppesentation of the events to bring out the resemblance between finding the structure of a movement pattern and assigning relational bindings to the lines in a line drawing.

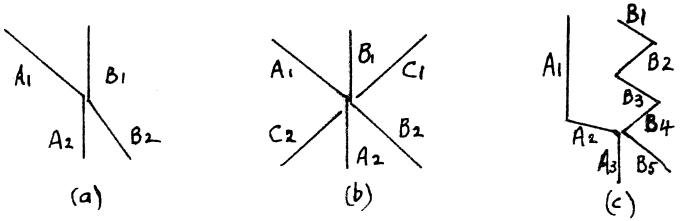


Fig. 3 (a) shows Experiment 1. Linking A1.B2 and B1.A2 gives the passing effect.
(b) shows Michotte's experiment 24.
(c) shows Michotte's experiment 21.

3.81 In 3(b) a third object has been added; this starts from a point the same distance to the right of B as A is to its left, and moves towards B at the same speed as A does. On reaching B, it disappears momentarily, then reappears on the other side of A and continues its journey until it reaches the place where A began. Three interpretations are offered.

(i) /

(i) 2 objects (A1.C2 and C1.B2) are seen to perform "a to-and-fro movement in relation to an object (B1.A2) in the centre of their path".

The latter is seen to change colour and move slightly.

(ii) 2 objects (A1.B2 and C1.C2) "go towards each other and cross at the point where a third (B1.A2) is to be found in the centre of their path".

(iii) 2 objects "rotate in the 3rd dimension around a permanent central object".

The symmetry of the impending collisions so radically affects the analysis that the same objects "continue moving" throughout, and their movement does not affect the "stationary" object. i.e. the consequence of the "collision" is simply the next part of the agent's movement - the passive object is unaffected and no causality is imputed.

3.82 In 3(c) an oscillatory movement of B precedes A's movement, and is timed so that B reaches the centre of the screen just as A does, and then B makes its last journey to the right. In this experiment the oscillating object (the zig-zag pattern) becomes the thing which A approaches and comes to rest beside. No dynamic effect is created unless the subject is asked to concentrate very hard on the point of impact. In this case the last leg of B's journey is not linked to the sequence B1.B2.B3.B4. Instead, this oscillation is seen as a preliminary to the standard launching of B by A.

4.0 Discussion

Michotte argues that the causal impression of launching depends crucially on establishing that the entire movement "belongs" to A even though, after the impact, it is B which is displaced in space. The movement is seen to belong to A because A has been established as the dominant object, since it starts moving first and moves faster than B. The notion of a movement belonging to an object even after that object has stopped moving seems a strange one, especially if we are to take this as literally as Michotte wishes us to.

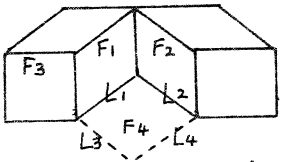
> "This kind of response of course is nothing but a literal translation or accurate report of the retinal stimulation such as could be achieved by an electrical recording device".

In fact, A doesn't always start moving first. The launching quoted in 3.82 (Michotte's experiment 21) occurs in spite of the fact that B moves first - one needs only provide the correct viewing conditions. This passion to exclude all possibility of interpretation in terms of schemata, which Michotte shares with other experimental psychologists, notably J. J. Gibson, seems to rest on some feeling that allowing interpretation to creep in is somehow equivalent to saying that the phenomenon hasn't really been perceived - someone only thought they saw it!

The notion of "belonging to" seems to be exactly a matter of interpretation. As must be apparent, we regard Michotte's patterns of moving squares as having the same correspondence to actual moving objects in the real world as, say, junctions in a 2-D drawing have to the corners of planar solids i.e. a Michotte experiment is a kind of kinetic diagram, and one is continually impressed by the scholarly way in which he has /

/has explored a whole range of such diagrams.

The crucial role of context in assigning a meaning to a picture fragment has its exact counterpart in the case of a kinetic fragment.



Without lines L3 and L4, the line L1 and the face F1 belong to the same cube as F3 does. When L3 and L4 are added, we have to account for face F4, and we see L1 as belonging to the same cube as L2 and F2.

Just in the same way in the oscillating experiment (fig. 3c and 3.82) we saw B5 change allegiance, this time as a result of a change in fixation-attention.

The idea of a context of expected events as elaborated in this report corresponds to the notion of a scenario or structured script used by people working in language understanding and belief systems (Winograd (1973); McDermott (1973); Abelson (1973); Rumelhart & Norman (1973)). Particular events are to be understood as belonging to wholes of identifiable types in terms of which expectations of new events will arise.

Consider the interpretation of a kinetic diagram which is seen as a launching. In the static scene case quoted from Clowes in section 1.0, we noted that the mismatch between the 2-D picture fragment and the 3-D model fragment is interpreted in a series of well understood systematic inferences based on knowledge about the picture taking process. What could provide a more appropriate stimulus cue to invoke a 3-D model of a block than the squares used in these experiments? And what could be more likely than one "block" colliding with another to invoke the concept of mechanical causality? It is extremely easy to SEE the square as a block. However when the square is to be SEEN-AS some animate object being frightened away, the metaphor reaches awareness and is expressed overtly

"it is AS IF A's approach frightened B, and B ran away".

How much would one need to add to the diagram in order to experience a "direct" triggering impression? What hints could one include in the experimental instruction to facilitate the seeing-as process e.g. "I am going to show you a cartoon strip".? How can one separate out the attention-directing element from fixation instructions? (Piaget (1961) summarises many years of work in this area).

Finally how much can these socalled "direct" responses (products of unconscious inference) be manipulated by conscious control. For example, in the series of experiments under review most subjects claim that even when they are aware of the artefactual nature of the apparatus, they continue to see A push B. On the other hand, one or two subjects "observing in an analytical way" see successive movements simply co-ordinated in time. Gregory (1970) gives numerous examples of the compulsive nature of certain perceptual constructs in spite of 'knowing' that the corresponding real world object does not have the ascribed characteristics. On the other hand, in the Johannsen demonstrations (1971) the tendency to construct objects out of points of light moving in particular patterns can be consciously inhibited, when an analytic posture is adopted.

Reaching awareness, we saw above, was connected with the inappropriateness of part of the stimulus cue to the response being given - we NOTICE when things don't go through smoothly. This inappropriateness of response is exactly the characteristic of neurotic compulsions. It is not strange to wash your hands periodically, especially if they're dirty; but it is strange to want to wash your hands all the time. Psychotherapy is largely about tracking down the appropriate metaphors.

The inappropriate part of the stimulus cue in the triggering response was the appearance of the participant. Michotte mentions (p82) that he can "favour" the launching effect by using, for B, a triangle on its side pointing in the direction of the movement. We propose exploring this aspect in greater detail. We have described our frame sequences as rather like strip cartoons. We would like to increase the resemblance.

Acknowledgements

This work was carried out with the support and encouragement of Dr. J. A. M. Howe, and the financial support of the Social Science Research Council.

References

Abelson, R.P. (1973) The Structure of Belief Systems. In Computer Simulation of Thought and Language. K. Colby & R. Schank (eds.) W. H. Freeman & Co.

Bartlett, F.C. (1932) Remembering. Cambridge University Press.

Clowes, M.B. (1973) Lectures given at AISB Summer School, Oxford, 1973.

Evans, T. (1963) A Heuristic Program to solve Geometry Analogy Problems. Ph.D. Dissertation M.I.T. In Semantic Information Processing (1968). Minsky (ed.) M.I.T. Press.

Gregory, R.L. (1970) The Intelligent Eye. Weidenfield & Nicolson, London.

Hardy, S. (1973) The Popcorn Manual. Essex University.

Johannsen, G. (1971) Visual Motion Perception: A model for Visual Motion and Space Perception from changing Proximal Stimuli. Report 98. Department of Psychology, University of Uppsala, Sweden.

Lamontagne, C. (1973) A New Experimental Paradigm for the Investigation of the Secondary System of Human Visual Motion Perception, Perception, Vol. 2 No. 2: p167-180.

McDermott, D.V. (1973) Assimilation of New Information by a Natural Language Understanding System. M.Sc dissertation. M.I.T.

McDermott, D.V. & Sussman, G.J. (1973) CONNIVER Reference Manual. M.I.T.

Michotte, A. (1963) The Perception of Causality. Transl. by T. & E. Miles. Methven.

Piaget, J. (1961) Mechanism of Perception. Transl. Seagrim (1969). Routledge & Kegan Paul..

Roberts, L. (1965) Machine Perception of Three-dimensional solids. In Optical and Electro-optical Information Processing. Tiffet et al (eds.) M.I.T. Press.

Rumelhart, D. & Norman, D. (1973) Active Semantic Networks as a Model of Human Memory. In Proc. Third Internat. Joint Conf. on Art. Intell. (20-23 Aug. '73) Stanford University.

Winograd, T. (1973) Invited Address, Third Internat. Joint Conf. on Art. Intell. (20-23 Aug. '73) Stanford University.

Winston, P. (1970) Learning Structural Descriptions from Examples. Ph.D. Dissertation. A.I. Technical Report 231. M.I.T.

A NON-CLAUSAL THEOREM PROVING SYSTEM

by David Wilkins

ABSTRACT: There are reasons to suspect that non-clausal first-order logic expressions will provide a better base for a theorem prover than conventional clausal form. A complete inference system, QUEST, for the first-order predicate calculus using expressions in prenex form is presented. Comparison of this system with SL-resolution shows that clausal techniques can be transferred to prenex form and expected advantages do seem to appear.

KEY WORDS: resolution, clause, prenex form, SL-resolution

1- Introduction

A predicate calculus expression in prenex form is obtained from a given wff by eliminating implication signs, standardizing variables, reducing the scopes of negation signs, skolemizing existential quantifiers, and removing universal quantifiers; see Nilsson(1971) for a precise definition. Prenex form differs from conventional clausal form only in that distributivity is not repeatedly applied to yield an expression in conjunctive normal form. There are a number of reasons for suspecting that prenex form would be superior to clausal form in automatic theorem-proving.

As anyone who has converted large expressions to clausal form knows, the application of distributivity causes a multiplicative explosion in the number of literals so using prenex form will at least save storage and execution time (human or otherwise). Another advantage is that the same information is not spread over a number of clauses. If the expression $A \vee B \vee (C \wedge D \wedge E)$ is necessary in a refutation, a resolution-type system will resolve away A and B and one of C,D, or E. A clausal theorem prover may refute A and B and get stuck on C. It then has to back up and try clause ABD which will involve redoing the refutations of A and B. Current theorem proving systems do not avoid this redoing of work but this would be a natural result of using prenex form. With prenex form we also gain the ability to use subexpressions that are "anded" together. For example, in the expression $A \vee B \vee ((C \vee D) \wedge (\neg C \vee E))$ the $\neg C \vee E$ subexpression can be used to refute the $C \vee D$ subexpression without pulling in the "higher level" information that resolving against the clause $AB\neg C$ would. In this simple example, a clausal system could avoid re-refuting A and B but the ability to use subexpressions becomes valuable in the general case.

Theorem provers are considered inefficient problem solvers, but given an unsatisfiable set of predicate calculus expressions with no meaning attached to them, I would be worse than inefficient in finding a refutation. The theorem prover needs some

kind of knowledge about its input, or must at least be given advice. I will present a few reasons why I think prenex form is more suited for giving advice about than is clausal form. We find non-clausal forms easier to express ourselves in since we write axioms that way. Suppose I have an axiom which represents the fact F. If distributivity shatters this axiom into n clauses, the only plausible interpretation is that these clauses are all the possible cases that can occur. The advice changes from "use this axiom to prove F" to "here is a set of axioms related to F, use as many as are needed". Actually writing out expressions and looking at their clausal and non-clausal forms should convince the reader that clauses are not the best way to conceptualize things.

QUEST is a complete inference system for unsatisfiable sets of expressions in prenex form. It exhibits expected non-clausal advantages and is as computationally efficient as SL-reolution [4], one of the more sophisticated clausal inference systems.

2-Definitions

Prenex expressions are naturally tree-structured so I will use conventional terminology(Knuth 1968) to refer to trees except as noted below. Each tree has one particular node designated as the current node and this node is said to have control. Each node is the parent of the roots of its subtrees and each subtree is a son of the root node. Note that parent and son are not inverses. A node is an ancestor of a node, N, if and only if it is the parent of N or the parent of an ancestor of N. A tree is a descendant of a node N if and only if it is a son of N or a descendant of the root of a son of N. A node is active iff it is the current node or an ancestor of the current node. The cousins of a node, N, in a tree, T, are all those (and only those) subtrees of T that are sons of N or sons of an ancestor of N in T, and in addition are such that their root node is not an active node in T. A cousin of the current node in a tree is said to be a current cousin in the tree. Branch nodes will be either AND nodes or OR nodes while terminal nodes will be either **T**(true), **F**(false), or a literal.

If T is a tree and $\propto$ a substitution, then T*$\propto$ denotes the tree produced by applying $\propto$ to all nodes in T. Two trees, T1 and T2, are unifiable iff there is a substitution, $\propto$, such that T1*$\propto$ and T2*$\propto$ are isomorphic, in the sense that there is an isomorphism, f, from the nodes of T1*$\propto$ to the nodes of T2*$\propto$ such that if M and N are any two nodes in T1*$\propto$ the following three statements are true: 1) if N is an AND, OR, **T**, or **F** node then (N)f is the same type of node; 2)if N is a literal then (N)f is the same literal; 3)if M is the parent of N then (M)f is the parent of (N)f. T1 is said to be the same (sub)tree as T2 iff T1 and T2 are unifiable with the null substitution. The negation of a tree is formed by doing the following three things to the tree: 1)replace all **T** nodes by **F** nodes and vice versa, 2)replace all AND nodes by OR nodes and vice versa, 3)replace all literals by their negation.

I will now define the Truth Value Inference Rules which simply implement the definitions of "and" and "or". A node can be inferred false iff the node is an AND node and the root of one of its sons is **F**, or the node is an OR node and the roots of all its sons are **F**. A node can be inferred true iff the node is an AND node and the roots of all its sons are **T**, or the node is an OR node and the root of one of its sons is **T**.

The expression input to QUEST are assumed to be conjoined together, so an input set is represented by a tree whose root is an AND node and the subtrees of the root are the expressions in the input set.

An unsatisfiable tree, T, is minimally unsatisfiable iff when any subtree of T which is the son of an AND node and not the only son of that AND node is removed from the tree, the resulting tree is satisfiable.

3- Inference rules of QUEST

The truth value inference rules have already been mentioned. Let T be a tree from which we are trying to infer false. Suppose N is the current node in T and let S be the son of N we are currently trying to refute. When N is an OR node, all sons must be refuted but when N is an AND node the search strategy may pick a son to refute. To be complete, the inference system must in general allow any son to be tried although QUEST restricts the choice in some cases without sacrificing completeness. The distinction between rule of inference and operation must be understood. The rules of inference presented here are ways of changing a derivation tree so the validity of the expression it represents is unchanged. QUEST changes these rules into operations (of the same name) by allowing substitutions to be made in order to apply the rule and by placing restrictions on the use of the rule.

To develop the first rule, let O be the set of cousins of N in T which are sons of OR nodes, with S deleted from the set. Rule one considers all members of O to be false and infers whatever it can about S. Intuitively, the validity of this runs as follows. To obtain a refutation of T by working at N, all sons of OR nodes which are ancestors of N, i.e. O, must be inferred false if inferences about N are to help in a refutation. So if S is to be inferred false above N, it is safe to infer S false at N and wait for the refutation above N. This is valid because all inferences at a node are made using the information in a node's ancestors. Therefore, if S is the same subtree as a member of O, it is inferred false, and if it is the same as the negation of a member of O, it is inferred true. This rule is called the factoring rule of inference because it's role corresponds to the role of factoring in clausal inference systems. S is said to be factored on and the member of O is said to be factored against.

Let A be the set of cousins of N which are sons of AND nodes, with S deleted from the set. Rule two considers members of A true and infers whatever it can about S. This simply uses the information provided by the axioms used so far. If S is the same subtree as a member of A, it is inferred true, and if it is the same subtree an the negation of a member of A, it is inferred false. This rule is similar to ancestor resolution in linear resolution systems, but has added aspects because of the non-clausal structure. To avoid confusion, it will be called the smashing rule of inference. S is said to be smashed and the member of A is said to be smashed against. I shall call the combination of smashing and factoring the reduction rule of inference. Reduction and extension (a term used in the next paragraph) are both used in SL-resolution for similar ideas.

The last inference rule is the one corresponding most closely to resolution. It grows the current tree and is called the extension rule of inference. Extension says that members of A (the set defined in the last paragraph) can be grown onto N as follows: if N is an OR node then S can be replaced by a tree whose root is an AND node with its subtrees being S and a copy of a member of A; if N is an AND node then simply add a copy of a member of A as a new son of N. The member of A is said to be extended against and S is said to be extended on.

QUEST is basically these rules with strict restrictions put on them to guide the refutation and prune the search space.

4- Informal description of QUEST

QUEST has five operations. Each operation produces a new tree from an old tree. A QUEST derivation is a sequence of trees where each is produced by applying one of the five operations to its predecessor. The object is to produce the tree whose root is the node **F** from the input tree.

The most trivial operation is diving which simply moves control down one node to the root of a son of the current node. The truncation operation changes the current node to **F** and moves control up to its parent whenever the current node can be inferred false by a truth value inference rule. The reduction operation does inferences from the reduction rule of inference, but only when false is inferred, and then only when it is inferred from a son of the current node. The reduction operation, unlike the other two, may apply a substitution to the tree in order to make this inference. The deletion operation does true inferences from the reduction rule of inference, but only when a son of an AND node that is not the only son of that AND node is inferred true. Deletion may not apply a substitution since true inferences are a sign that something has gone wrong so we don't want to waste effort producing them.

The last operation is the extension operation. Growing the tree without a purpose will probably not get us any closer to a refutation so it will be required that the tree extended against have a subtree which is the negation of S. The number of subtrees in a tree increases with increasing tree size something like factorially or worse, so finding something to extend on may involve a huge number of unifiability tests. The test for unifiability of two trees is not trivial since the nodes at one level can be matched in many different ways with the nodes at that level in the other tree. Therefore QUEST does not allow extension on non-literal subtrees at all. Since only literals are extended on, the subtree extended against will always contain the negation of this literal after a substitution has been applied. This negation is called the literal extended against. The tree is grown as in the extension rule of inference and control is given to the root node of the extended against subtree. The extension operation, by definition, also requires that any AND node in the subtree extended against which is on the path to the literal extended against, must refute the son which contains the literal extended against. This does not destroy completeness and significantly prunes the search space since only one son need be tried at these AND nodes, and also makes the operation more like resolution since the extended against literal must eventually get smashed.

QUEST has six restrictions on the applications of these operations which significantly prune the search space.

1: The current tree can no longer be considered if an active node can be inferred true by the truth value inference rules and the reduction rule of inference. This does all true inferences not done by deletion. This restriction stops processing on trees that cannot lead to a proof.

2: The initial current node must be one that is in some minimally unsatisfiable subtree of the input tree. Thus only one starting point need be considered. This corresponds to support subset restrictions in clausal systems, but here the negation of the theorem can always be expressed in one prenex expression, so one starting point can be picked.

3: At a particular OR node, all reductions must be done before any extensions or dives. Without restrictions like this, the search space will be full of derivations that do the same operations in a different order. This eliminates all derivations which do extensions or dives before reductions at the same level. Hopefully, doing reductions first will instantiate the variables further thus reducing the number of possible unifications later on.

4: If any current cousin can be inferred true or false by the reduction rule of inference, then this is the only allowable operation. If there is a false inference the node should have been inferred false when its parent had control, but instead an extension or dive was done. Thus there is an easier proof than the one we are working on. If there is a true inference then either the first restriction will stop processing or a deletion will be done which simplifies the tree.

5: If the next operation is to be extension or diving then it must be done on the son selected by the selection function over OR nodes. This corresponds almost exactly to the selection restriction in SL-resolution. This also orders the applicable operations. If all nodes have n sons then a system without this restriction would have on the average n! times as many derivations it can produce. As would be expected, QUEST works for any selection function over OR nodes.

6: The same selection restriction is now applied to the sons that are reduced. The same factorial saving is made by not repeating the same reductions in different orders. The restriction is implemented by defining a total ordering of the sons rather than a function that picks one out. A function will not work because we cannot always apply reduction to a selected son. If the next operation is reduction then it can only be done if no son greater than (in the given ordering) the son being reduced has been reduced.

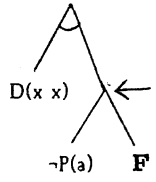
This gives an informal desription of QUEST that is exact as I could make it. The formal definition is fairly short and easy to read but is not within the scope of this paper. QUEST is sound and complete, but again the proofs are too long to present here. The formal definition and proofs can be found in (Wilkins 1973).

5- An Example

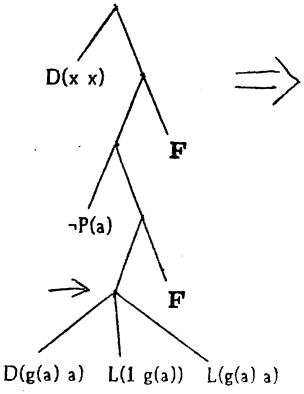
I will present one example of QUEST in action. I will point out places where prenex form is an advantage in the hope the reader will recognize these as general phenomena likely to occur in most problems. To make things readable, I will represent the trees graphically. AND nodes will be distinguished by drawing an arc through their branches. I will leave off the top AND node which has all the input expressions as its sons, but one should remember that it is there. The current node will be desgnated by an arrow.

Theorem: Every integer greater than 1 has a prime divisor. This can be axiomatized as follows: D(x x) means any number divides itself. ¬D(x y)∨¬D(y z)∨D(x z) represents the transitivity of divisibility. P(x)∨(D(g(x) x)∧L(1 g(x))∧L(g(x) x)) says that if x is not prime then a number between 1 and x divides x. Let a be the least counter-example to the theorem. The negation of the theorem is as follows: ¬P(x)∨¬D(x a) says that if x divides a then x is not prime. ¬L(1 x)∨¬L(x a)∨(P(f(x))∧D(f(x) x)) says that if x is between 1 and a then it has a prime divisor.

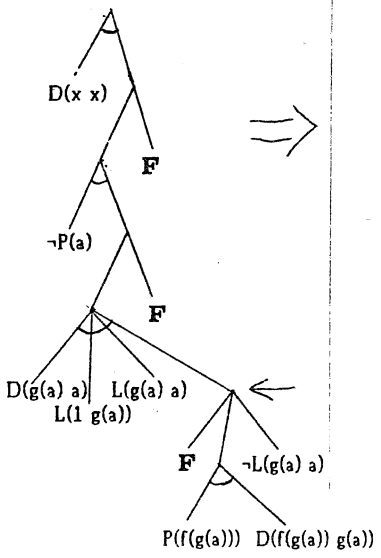
A proof found by a POP-2 program implementing QUEST is presented in the next five diagrams. The meaning easily attached to these diagrams is as follows: 1)Since a divides itself, it is not prime. 2)Thus there is a number, g, between 1 and a which divides a. 3)a was the least counter-example so there is a number, f, which is prime and divides g. 4)f does not divide a, since a is a counter-example. 5)By the transitivity of divisibility, this is a contradiction.



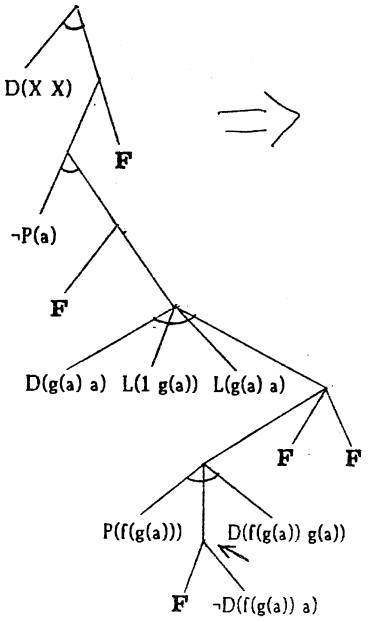
¬D(x a) is immediately smashed against input expression.



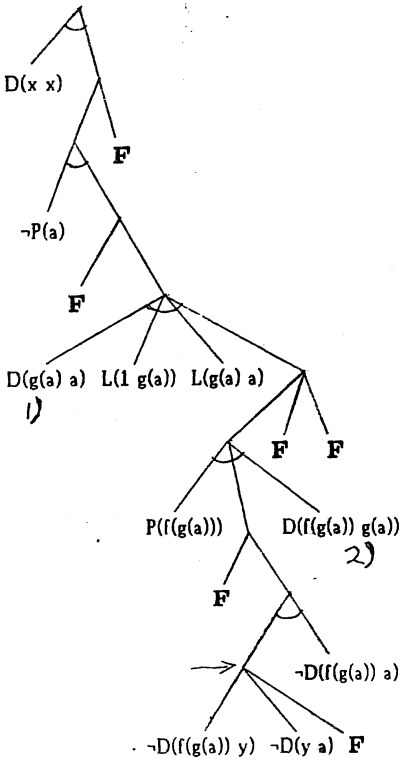
Extension on ¬P(a) against second axiom, followed by smashing the literal extended against. A literal must be chosen to extend on.



Extension on L(1 g(a)) against the fourth axiom, followed by smashing the literal extended against. The next operation must be the smash of ¬L(g(a) a) against the L(g(a) a). In clausal form, if we had just used the clause with L(1 g(a)) in it, we would need another whole clause to get L(g(a) a) and this would involve re-refuting all the literals above this AND node since distributivity would "attach" them to L(g(a) a).



Extension on P(f(g(a))) against the third axiom followed by smashing the literal extended against.



Extension against the first axiom with smashing of the literal extended against. Now y is instantiated to g(a) and both ¬Ds are smashed. Truth value inferences then infer **F** from the whole tree and the proof is complete. Note that the same situation as before arises when we smash the ¬Ds. They are smashed against 1) and 2), both of which are sons of AND nodes different from the son extended on at that AND node. Thus clausal form would have two more extensions against clauses rather than two reductions. I hope the reader may recognize this ability of prenex form as a general advantage and not particular to this problem. In a sense, extension in prenex form sucks in 2 or 3 or n clauses in compact form. Moreover they contain information likely to be relevant since one would usually not expect a single axiom to contain parts irrelevant to each other.

6-Comparison with SL-resolution

Many of the ideas in QUEST come from SL-resolution so it is natural to compare the two. For the reader not familiar with SL, it is presented in (Kowalski and Kuehner 1971). The purpose of this comparison is to show that our clausal techniques can be carried over to the prenex case, and to show how QUEST compares to clausal inference systems in general since SL is currently one of the better ones.

A QUEST derivation tree can be considered as an SL chain. A cousin which is the son of an OR node would be a B-literal, the son of an AND node would be an A-literal, and top to bottom would correspond to left to right. Let us consider the case when clausal input is given to QUEST. All cousins are now literals so I will speak of QUEST trees as if they were SL chains. The only difference between QUEST and SL chains initially is that QUEST chains have the unit clauses tacked on the front as A-literals. This was done because I felt extension against a unit clause is more like reduction than extension. Either system could easily be changed to be like the other. With clausal input, QUEST will never do a dive and will never do a deletion unless the same clause is input twice.

First, let us look at the admissibility restriction of SL. It says that no two literals in the chain may have the same atom unless the next operation is reduction. QUEST has the same restriction since two literals having the same atom is equivalent to being able to infer a cousin true or false by the reduction rule of inference. Let us now consider the operations.

The truncation operation is essentially the same in both systems. There are three differences in the reduction operation. Both systems require reductions at one level to be done before extensions but, as mentioned before, in QUEST this also applies to unit extensions. This is a trivial difference. SL does not allow factoring within a clause or ancestor resolution (smashing) against the rightmost A-literal while QUEST does. SL makes up for this by allowing extension against all factors of the input clauses. Thus SL has fewer reduction choices but more extension choices, but once again either system could easily be changed to be like the other. The third difference in reduction is the ordering of literals to determine the order of reductions. This simply applies the selection idea (the heart of the SL system) to reductions as well as extensions, and I feel it should be included in SL. The only difference in the extension operation is the already mentioned one of SL having more extension choices.

The differences when QUEST is applied to non-clausal input can be thought of as follows. Some links in the chain are now pointers to a tree instead of literals. These trees can be reduced as they are or "expanded in line" by diving operations. There are now A-links in the chain that correspond to sons of AND nodes in the input tree. These provide information which in the clausal case, loosely speaking, is only provided as a new clause and then only with more literals in the clause because of the distributivity applications. The following section gives evidence that the advantages one intuitively expects actually do appear.

7-An implementation

I wrote a POP-2 program implementing QUEST at the University of Essex. The program extends Boyer and Moore's structure sharing techniques (Boyer and Moore 1971) to the prenex case. The purpose of the program is to run on examples in clausal and prenex forms with a breadthfirst search so as to get a fair comparison of the size of the search space. Since QUEST is fairly good on clauses as shown by its comparison to SL, this should be a fair comparison. Three statistics are given: 1) cpu time in seconds, 2) number of extensions against input expressions, and 3) number of derivations being processed in parallel by the breadthfirst search.

For lack of space, I have picked only 3 examples. These demonstrate results found by running other examples. It was also found that the amount of processing needed in clausal cases varied greatly with the clause picked to start with. Problem 1 is the classical Quine-Wang problem P(x a)∨(P(x f(x))∧P(f(x) x)), ¬P(x a)∨¬P(x y)∨¬P(y x). Problem 2 is a variation of 1: G(y a)∨(G(y f(y))∧G(f(y) y)), ¬G(w y)∨(G(y f(y))∧G(f(y) y)∧¬G(y a)). Problem 3 is the example of section 5.

	NON-CLAUSAL			CLAUSAL		
	cpu time	exten.	deriv.	cpu time	exten.	deriv.
Problem 1	1.074	3	2	1.278	8	4
Problem 2	.746	3	2	1.717	11	7
Problem 3	11.296	65	45	28.965	136	74

8- Conclusion

It it may be favorable to abandon clausal form for a form easier to attach meaning to. This paper presents a non-clausal inference system that is complete at the general level and probably as efficient as current clausal systems. Section 7 provides evidence that computational advantages expected with prenex form do in fact appear (I do not wish to argue about judging criteria here). The comparison of QUEST with SL-resolution shows that techniques developed for clausal systems will be applicable to prenex systems.

REFERENCES

1. Boyer, R.S., and Moore, J.S., The Sharing of Structure in Resolution Programs, Metamathematics Unit, University of Edinburgh, 1971.

2. Hayes, P.J., and Kowalski, R.A., Lecture Notes on Automatic Theorem-proving, Metamathematics Unit Memo 40, University of Edinburgh, 1971.

3. Knuth, D.E., Fundamental Algorithms, Addison-Wesley, London, 1968.

4. Kowalski, R.A., and Kuehner, D.G., Linear Resolution with Selection Function, Artificial Intelligence, 2, 1971.

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

6. Wilkins, D.E., QUEST: A Non-Clausal Theorem Proving System, M.Sc. Thesis, University of Essex, 1973.

A COMPUTER SYSTEM FOR MAKING INFERENCES ABOUT NATURAL LANGUAGE

by

Yorick Wilks
Artificial Intelligence Laboratory
Stanford University
Stanford,Calif. 94305, USA.

ABSTRACT: The paper describes the way in which a Preference Semantics system for natural language analysis and generation tackles a difficult class of anaphoric inference problems : those requiring either analytic(conceptual) knowledge of a complex sort, or requiring weak inductive knowledge of the course of events in the real world. The method employed converts all available knowledge to a canonical template form and endeavors to create chains of non-deductive inferences from the unknowns to the possible referents. Its method for this is consistent with the overall principle of "semantic preference" used to set up the original meaning representation.

1.INTRODUCTION

This paper describes inferential manipulations in a computer system for representing the content of chunks of natural language. By inferential manipulations, I mean the drawing of complicated inferences about the course of events in the world that are necessary to understand natural language, and in particular necessary to resolve pronoun references (anaphora), and ambiguities in the senses of words.

To take a simple example : when the system sees the sentences JOHN LEFT THE WINDOW AND DRANK THE WINE ON THE TABLE. IT WAS GOOD, it decides that the pronoun refers to the wine, while if it sees JOHN LEFT THE WINDOW AND DRANK THE WINE ON THE TABLE. IT WAS GREEN AND ROUND, it will decide that it is the table being referred to in the second sentence. "Decide" here must be treated with care, since further text might correct both these decisions, of course, the point is that a hearer or reader, having encountered the amount of text given above , will almost certainly understand in the way indicated, even if the speaker or write intended something different.

The system is programmed in LISP 1. 6 and MLISP, and runs as an analyser of English and a generator of French, on the PDP6/10 at Stanford A. I. Laboratory. This provides a very firm context of verification for a natural language understanding program: in the first example above, if "it" emerges as "il" the French masculine pronoun, it can only refer to "wine" since that is only masculine noun in the sentence. The examples dear to the hearts of those who analyse stories and dialogs can all be reconstructed within a machine translation environment.

The system described here has had its lower level capabilities described elsewhere[6,7,8] : its abilities to cope with complex sentences without a isolable syntax package;its ability to deal with wide areas of word sense ambiguity , and the case ambiguity of prepositions. All these abilities are assumed in the present paper, and not described in detail. Those "front end capabilities" set up very complex semantic objects, called "semantic blocks" : networks of objects called templates , that are themselves complex structures of semantic primitives. The present system is distinguished not only by the more complex objects it handles than other programs (and the greater abilities to handle unrestricted natural language that come from that), but its ability to handle objects representing longer stretches of discourse. The semantic blocks described below , that are these networks of templates, are representations for small paragraphs of text. Again, it must be emphasised, that these complex objects are not merely the result of applying projection rules to dictionaries, as in most contemporary systems [12, 13]. They are built in part from already available complex parts, called templates and paraplates [see 7]that are "fuzzy matched" onto text chunks as wholes.

In this paper then, I am concerned with the manipulation of these complex objects to draw out semantic information, and the application of inference rules to that information, in order to solve concrete reference problems. It is an assumption of this work that these problems cannot be solved independently of a strong representation [1].

I would not defend the details of the semantic codings given in this paper, nor the particular control structure of the program. What is essential in this system , and among its distinguishing features, is (1) the inferential use of partial information , that is, information weaker than that in dictionaries and analytic (always true) rules. i The use of such information constitutes the EXTENDED MODE of the system described below. The second distinguishing feature (2) is the preferring of one representation or inferential chain to another. This is important and a neglected aspect of modern natural language research, where workers often seem to feel that the first representation or inference their system finds MUST be the right one. This is discussed elsewhere [7] , and again below in the context of inferential chains.

The common sense rules of inference used in this system are not deductive consequences about the world, but are likely courses of events which , if and only if they match onto the available explicit and implicit information in the text, may be said to apply, and by applying may enable us to identify mentioned entities and so resolve problems of reference. In the examples above we need to apply at least a rule equivalent to , in ordinary language, IF SOMEONE WANTS AN ENTITY , HE WILL WANT TO CAUSE IT TO MOVE IN SOME WAY. Such a rule is , in this system , in no way contradicted by mention of exceptions ,such as someone wanting some object but doing nothing to move or otherwise affect it. This rule (see below for details) is fuzzy matched onto what we know from the example, and what we know about drinking, including that it is an act of causing a liquid to move. These processes to be described in the paper allow the pronoun to be referred correctly in a way consistent with the common sense inferences a person would make and are reducible to non-deductive forms such as SOMETHING X's AND FOO X's , THEREFORE THE SOMETHING IS FOO.

Such inferences could, of course, be represented in some much stronger system with deductive machinery , given all the missing frame axioms, quantification etc. My point is that nothing would be gained by doing so, because such machinery can never improve the reliability of the partial information being handled. It is the content and applicability of such inferences that should be our concern at present , not the finding of strong systems of logic in which to represent them. I have set out that case in more detail in [8].

Secondly, with regard to what I called preference, it is an important premise of this work that the basic problems of natural language semantics have simply not been solved, either by the linguists or the A.I. people in the field, and that insights about the structure of language are still needed: needed in the same sense in which Papert has often argued that AI must offer simple rule systems

different from the first sledgehammer you thought of. His persuasive example is that of catching a ball, done by a simple algorithm and not at all, as one might have thought, by the solution of complex differential equations. To this end, we avoid the generative grammatical and semantic systems of the linguists, as well as the deductive systems of logicians. The essential part of the present system that aims to offer a little of the missing content is what we call "Preference Semantics".

The key point is that word sense , and structural , ambiguity in natural language will always, in any system, give rise to alternative competing structures, all of which can be said to "represent" whatever chunk of language is under examination. What we mean by "preference" is the use of procedures , at every level of the system, for preferring certain derived structures to others on the basis of their "semantic density", and in this paper we shall be particularly concerned with preferring certain inferential chains to others on that basis.

What we are postulating ,speaking psychologically, is that humans interpret language so as to reduce the conceptual density to a minimum; which can be taken to mean " keeping the amount of new information introduced into the system to a minimum . Without this faculty a language understanding system cannot function. In understanding "Pieces of paper lie about the floor", we will thus choose to interpret it as being about position rather than deception because from the preference information in the system about the concept "lying" we will know that deceptive lying is a concept that prefers an animate agent if it can get it, (here it cannot) while a statement about passive position prefers a physical object as the apparent agent , which is available here. The satisfaction of a preference increases the density of the derived representational network and the densest network will be the one ultimately preferred. But, in understanding "My ideas followed hers closely" ,we want to accept the ideas as the agent, even though our information about the concept of following is that it normally prefers an animate agent if one can be found, since only in that way can the animate sense of "fly" be chosen correctly as the agent in "The fly followed the ladybird into the web". The point is to prefer the normal , but to accept the unusual. A little reflection will show that conventional linguistic rules, with fixed word classes, operating with (unintelligent) derivational rule systems, cannot do this very simple thing.

The preference computations,just sketched above, ,that involve no real world knowledge above and beyond the conceptual knowledge we have about word meanings,I call the BASIC MODE of the system.I want to distinguish the basic from the extended mode that I discuss in detail in this paper in terms of the kinds of anaphora problem the modes can tackle.In the basic mode, the system resolves those anaphoras that depend on the superficial conceptual content of text words. This is does in the course of setting up the initial semantic representation (which I have not yet described at all). I shall call these type A anaphoras. For example, in "Give the bananas to the monkeys although they are not ripe, because they are very hungry", the system in its basic mode would decide that the first "they" refers to the bananas and the second to the monkeys. It does that

by seeing, in the representation for the concept of hunger, that it prefers to be applied to something animate,and that the concept of ripeness prefers to be applied to something plantlike.If every satisfied preference increases the density of the conceptual network ,then we shall get the densest network when the first "they" is tied to "bananas" and the second to "monkeys".

The main part of this paper describes an EXTENDED MODE of the system that tackles two other kinds of anaphora example that I shall call types B and C. Consider the correct attachment of "it" in "John drank the whisky from the glass, and it felt warm in his stomach". It is clear that the pronoun should be tied to "whisky" rather than TO "glass", but how that is to be done is not immediately obvious. Analysis of the example (see below) suggests that the solution requires , among other things, some inference equivalent to the sentence "whatever is in a part of X is in X".

Anaphoras like the last I shall call type B, because the inferences required to resolve them are analytic but not superficial. By analytic I simply mean that the quoted sentence above, about parts and wholes, is logically true , and not a fact about the real world, but rather about the meanings of words like "in". What is meant by "superficial" in the distinction between types A and B will become clear below after some a discussion of the meaning formalism employed

Most importantly,I shall discuss type C anaphoras, which require inferences that are not analytic , but weak generalisations (often falsified in experience) about the course of events in the world. Yet their employment here is not in any sense a probabilistic one. In "The dogs chased the cats, and I heard one of them squeal with pain", we shall, in order to resolve the referent of "one" (which I take to be "cat" not "dog"), need a weak generalisstion equivalent to "animate beings pursued by other animate beings may be unpleasantly affected". Such expressions are indeed suspiciously vague, and a reader who is worried at this point should ask himself how he would explain (say, to someone who did not know English well) how he knew the referent of "one" in that sentence. It can hardly be in virtue of a particular fact about cats and dogs because the same general inference would be made whatever was chasing and being chased. I shall be surprised if he does not come up with something very like the inference suggested, and it may be the nature of natural language itself that is worrying him.

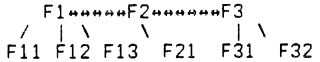
The inferences for type C, then, are general expressions of partial information(in McCarthy's phrase) and are considered to apply only if they are adequately confirmed by the context. What I mean by that will become clear in the course of what follows, but in no case do these expressions yield deductive consequences about the future course of the world, nor is there any assumption here that the event generalised about ALWAYS happen in such and such a way. Indeed, they would be foolish if they did because the world's course cannot be captured in that way. In the whisky example above, it might have been his earlier dinner that in fact made him feel good. Yet, nonetheless, the solution of the anaphora problem for an understander, derived as just described, is definite,

for anyone who writes the sentence about John's stomach will be taken to mean that the whisky was in his stomach, whatever he might have intended in the rare case of a glass swallower.

2.BRIEF RECAP OF THE SYSTEM'S BASIC MODE OF ANALYSIS

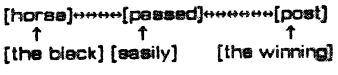
The heart of the basic mode's representation is the template (not to be confused in any way with the usage of that word in character recognition to mean a context-free method of analysis).This is an active frame of complex concepts that seeks preferred categories of concepts to fill its slots,though if its preferences are not satisfied it will accept whatever it finds in default. What Minsky has recently called [2] frames are good first approximations to templates.

The template can be thought of as expressing the gist of a phrase or clause, or even simple sentence, of language.It is a connectivity of FORMULAS,which in turn are complex concepts expressing the senses of words,one formula to a word sense.If F1 etc.stand for formulas,then a template has the following connectivity:



At nodes F1, F2, F3 are the principal formulas of the template and are always agent, action and object (in that order), though any of them may be a dummy in any particular example. (F11, F12, F13)is a list of formulas dependent on main formula F1 etc. Let me give an example of a template structure at this point by using the following simplifying notation: any English words in square brackets [] stand for the meaning representation of those words in the Preference Semantics system. This device is important in the exposition of the material in that the content of the []-abbreviated forms can be seen immediately, whereas the complex coded forms themselves would be as hard to read as ,say , a sentence read a word at a time. But it is important to restate that the rules and formalisms expressed within [] are really formulas and subformulas of structured primitives, and that their tasks could not be carried out, as some still seem to believe, by massaging the English language words, standing for their own meaning representation.

So then, the template connectivity of formulas for "The black horse passed the winning post easily" could be written (ignoring any ambiguity problems for the moment):



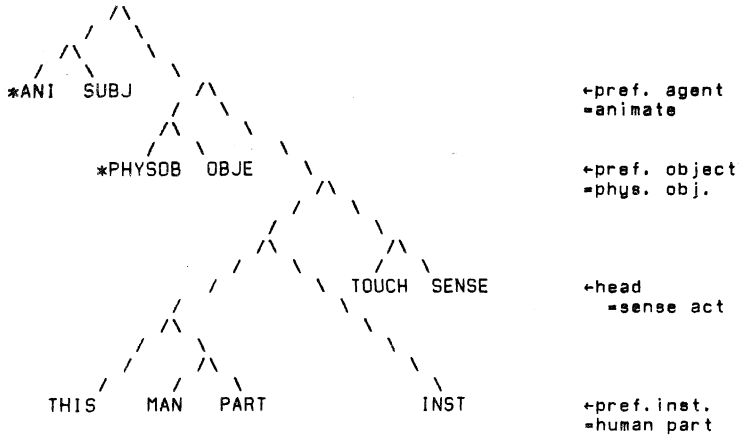
When the system runs,input texts are fragmented into clauses,phrases etc. and templates are matched to each of these,probably a number of templates to each text chunk depending on how potentially ambiguous its words are . The first exercise of preference tries to cut this number down and throw away as many templates as possible . To see how this is done ,we must realise that the formulas at the nodes of the template network are themselves complex objects. Here for example ,are two formulas for the English action "grasp":

"grasp"(action1) → ((*ANI SUBJ)((*PHYSOB OBJE)(((THIS (MAN PART))INST)(TOUCH SENSE)))))

"grasp"(action2)→ ((*HUM SUBJ)((SIGN OBJE)(TRUE THINK)))

There is no space to explain these tree structures of semantic primitives in detail here (see [6 , 7, 8]),nor is there any need to do so.We need only note that the right-most element of each formula is its principal,or head, element. Thus, grasp1 is basically a SENSE action,as in grasping a block, while grasp2 is basically a THINK action, as in grasping a theorem.The case subformulas at the left hand sides of the formulas express the preferences under discussion. The subformula with SUBJ expresses preferred agents (animate things for grasp1 ,and human things for grasp2), while the subformula with OBJE expresses the preferred objects of the actions ,namely physical objects and SIGNS respectively,the latter being thoughts and symbols of thoughts.

This should all become clearer if each formula is thought of as a binary tree,with dependency of all branches to the right. Thus for the first formula above, we have:



So, when analysing "John grasped the idea",the agent preferences of both the templates initially constructed will be satisfied by John ,who is both animate and human.But only grasp2 will have its object preference (for a SIGN-like entity) satisfied.If we think of a satisfied preference as strengthening one of the arrow links in the diagram above, then it is clear that the template with the grasp2 formula at its action node will have the stronger linkage and will be preferred, and the template with the grasp1 formula will be rejected, and never considered again.

The representation of a text(composed of fragments) is then a network of these template networks. The templates are interconnected by case ties. The notion of case is discussed in detail in [7]1, , but for the moment a case can be thought of as a type of link tying one template to some particular node in another template. In the sentence "He lost his wallet / in the subway" (fragmented at the stroke) we might say that the second fragment of the sentence depends on "lost" in the first, and that the dependence is the locative case. Thus in the representation, the template for the second fragment would be tied to the central, action, node of the first, by a link labelled LOCA. The node on the first template to which the case tie ties is called the mark of the second template. Enormous gaps have been left in this brief recapitulation : in particular how this last process is done with the aid of dummy templates and highly structured case objects called paraplates [see 7];

how this superficial template matching is converted to a deeper representation effectively eliminating the dummies, and how the superficial semantic matching and preferring have done the work of a conventional syntactic component.

I described earlier, with the bananas and monkeys example how type A anaphoras are resolved in this basic mode of preference.Once resolved, these type A anaphoras also constitute links between templates, from the pronoun variable to its correct referent. Thus the compressed list form of the whole representation obtained ,for a single fragment of text, from the basic mode is:

(CASE MARK ANAPHORA F1 F2 F3 (F1 dependents)(F2 dependents)(F3 dependents))

The F1 etc., and the lists refer to the nodes on the first diagram above, which was the basic template connectivity. The new,capitalised,nodes can be thought of as other arrows tying the whole template connectivity to nodes in other such connectivities. That is, CASE,MARK and ANAPHORA structure the "semantic block" by setting up a network of templates.And it must be remembered throughout that what is actually at the F1 node is a complex object containing a formula tree, like the ones for "grasp" illustrated above.

So, in the earlier example whose first sentence was "John drank the wine on the table",the compressed list representation for the fent "on the table" would have a dummy agent formula at F1,a pseudo-agent formula for "during" at F2 (since prepositions are treated as pseudo-actions) and a formula for "table" at F3.Ther would be no dependent lists of formulas since "the" is represenrted along with what it qualifies.The CASE tie would be LOCA,since "on the table" has a location case,and the MARK would be a pointer to the node in another template for "John drank the wine"----to the node for the formula for "wine" in fact since that is clearly the point of dependence of the template for "on the table". The ANAPHORA node is unfilled in this example.

These compressed list representations make up the semantic blocks for paragraphs of text in the basic mode.The anaphoras in "The monkeys wanted the bananas although they are not ripe because they are very hungry" are resolved in this way in the basic mode because the resolution demands no more than computations on the preferred posssessors of concepts like ripeness and hunger.

3.QUICK SKETCH OF THE EXTENDED MODE OF INFERENCE

The extended mode of inference ,using common sense inference rules, is called whenever the basic mode cannot resolve a pronoun anaphora, between two or more candidate words, by semantic link density alone. In the example about John and his stomach, density techniques have no way to decide whether the glass or the whisky is in his stomach. On a basis of preferred agents and objects of actions , what I called superficial conceptual information, both are equally good candidates. The extended inference procedure is called and, if it succeeds,it returns a solution to the basic mode which then continues with its analysis.If it too should fail to reduce the number of candidates to one, then the top level of the system tries to solve the problem by default, or what a linguist would call focus. Roughly, that means : assume that whatever was being talked about is still being talked about. So, in "He put the bicycle in the shed and when he came back next week it was gone", neither density criteria, nor the extended inferences to be described here, will help at all. So the system may as well assume ,in this limited context , that the bicycle is still the focus of attention, and hence the reference of "it".

Consider again the following sentence after all the basic mode's routines have been applied:

[1: John drank the* whisky / 2 DIRE : DTHIS from a*glass / 3 : and it felt warm / 4 IN : DTHIS in his*stomach]

Since in is in []-abbreviated form, this object is really four successive list-compressed-templates described above , one for each of the four fragments of the sentence. The slash marks the fragment boundary and the case names DIRE(direction) and IN(containment) indicate the dependencies of templates 2 on 1, and 4 on 3 , respectively. The DTHISs are dummies added to fill out the canonical template triplet in cases of missing agents,objects etc. Further assume that the "his" has been tied to "John" by the basic mode , and presents no problem of analysis, and assume too that the basic mode provided a list of "candidates" for the reference of "it"("whisky" and "glass"). because if there had not been such a list of more than one candidate the routine under description would not have been called into play.

EXTRACTIONS are then made from each template in turn, if and only if it contains a representation of either an answer word or the variable pronoun itself. An extraction is the unpacking of every possible case tie : both those in the action (second)formula of the template and those labelling a link to other templates. In this example we obtain the following extractions: which are template-like forms as follows (where the first digit refers to the fragment #, the second to the number of the extraction from a particular template , and "*" links words with a single formula):

11: [whisky (IN in) John *part]

12: [whisky (DIRE to) John*part]

21: [whisky (DIRE from) a*glass]

41: [?it (IN in) his*stomach]

We can explain how these extractions were made, even in the absence of any detailed knowledge of the structure of formulas: 11, for example, has been derived from the template for "John drank the whisky" because from the structure of the formula for "drink" it follows that the liquid drunk is subsequently inside the drinker. This is because, when making up the formula for the action"drink", we express in it that the action consists in causing a liquid to be inside the agent of the action, as follows:

((*ANI SUBJ)(((FLOW STUFF)OBJE){(SELF IN)(((WRAP THING)FROM)(MOVE CAUSE)))))

This form requires no more to be understood than earlier example formulas, except to note that (FLOW STUFF) denotes liquids,the preferred objects of drinking, and that the action causes to move that liquid into the agent's self,and that it is (FROM implies direction case) liquid moved from a container ,or (WRAP THING).

So, in this informal representation we have acquired new template-like objects that express, in canonical form, new analytic information extracted from the existing templates, and from which new inferences can be made. It is postulated that the generation of this inexplicit information from the deeper levels of the formulas is essential to the process of understanding. These new forms differ from standard templates only in that their second node, or pseudo-action, has had a case name CONSd onto whatever the node was before. Note here that the form (IN in) is not redundant since the case name IN locates the case precisely as containment, while the English preposition can indicate many cases other than containment, as in "in five minutes".

We have now obtained new template items that yield assertive information, but did not appear in the original text. We then try two strategies in turn: first we try a zero-point strategy, which is to try to identify an answer template(or extraction) and a variable template(or extraction) without the use of common sense inference rules [CSIR's] .

The general assumption here, and it is a strong psychological assumption, is that in order to resolve these painful ambiguities the understanding system is going to use the shortest possible chain of inferences it can. And a zero-point strategy will, as it were, have no length at all (in terms of a chain of CSIR inferences) and so if it works, it will always provide the shortest chain. This preference for the shortest

chain is itself a strong psychological hypothesis ,and is ,for example,very different, apparently from the hypothesis about "keeping as many of a frame's terminal satisfied as possible" that Minsky suggests briefly in his recent "Frames" paper[2].The present hypothesis is a "laziness hypothesis" consistent with the general principle in use here of always being prepared to complexify ,or deepen, a representation,but never doing so unless necessary----just as the extended mode is never called unless necessary.This is a very different overall principle from the wide forward inference proposals of Schank [5] and Charniak[1].

This zero-point strategy is adequate for the example under discussion, because we can (under a suitable definition of template and extraction matching) identify extractions 11 and 41, and thus identify "?it" and the whisky, and we are home. This was the solution of a B type anaphora, requiring only analytic, necessarily true, conceptual information.

If the zero-point strategy fails, we bring down all the CSIR rules that contain an action subformula occurring in an answer or problem template form in the pool, and attempt to find the shortest chain that leads from some answer to some variable.

Let us return to the first example of the paper: "John left the window and drank the wine on the table. It was good". Notice already that we can reject all simple solutions based on focus (that the wine is referred to because it is what is being talked about) in view o of the contrasting example whose second sentence is "It was green and round" where clearly it is the table being referred to. Notice that this contrasting sentence pair will be dealt with inside the basic mode, because the preference of concepts of shape for physical object possessors, will reject the wine as referent.

Let us now set out that example, using the informal [] notation, and label original templates from the sentence with T# numbers, and label extracted template forms with E# numbers. We shall have then, after extraction:

T1. [John left the*window]

E1. [John drank the*wine]

E2. [wine (LOCA on) the*table]

E3. [wine (IN in) John]

T2. [?it was good]

All these stay in the inference pool because all contain either the problem variable ?it or one of the possible referents window, wine or table. The extended mode now accesses its CSIRs which are stored under the main action element of their antecedent and consequent . That does not imply however that the action is

only the simple primitive head element---the actions in CSIRs can be as complex sub-formulas as is necessary, which is a different approach from Schank's [5] where the inference rules are strongly classified by the fourteen primitive actions. I consider a much finer discrimination of rules necessary.

Here are two rules I shall call I1 and I2 respectively, given in formal and informal versions:

I1: [animate1 likes 2] → [1 causes-to-move 2]

((*ANI 1) WANT 2) → (1 (MOVE CAUSE) 2)

I2: [1 is good] → [animate2 wants 1]

(1 BE (GOOD KIND)) → ((*ANI 2) WANT 1)

The rules are flexible about expression of restrictions on variables by subformulas or elements . They are unquantified , but analogs of universal and existential quantification can be seen in them: in I2 for example, the appearance of the animate variable 2 can be read as "then there is some animate entity 2 such that. . . etc. "

The strategy searches form both the ?variable and from the potential answer template forms, trying chains of length one first, then of length two. At present it will not attempt to construct a chain longer than two. This could be easily extended to three, but I suspect that understanding of normal situations rarely requires longer chains than that. The preference for the shortest chain is analogous to the use of preference in the basic mode: both uses introduce as little new information into the as is possible.

Again, the consequences drawn are not necessarily true, they resolve ambiguities only where both antecedent and consequent match what we already know or can semantically extract. Much of the effort of the program is in the inexact matching of the template forms to the rules (or "fuzzy matching" as the fashionable phrase now is.) That does not mean the satisfaction of the restrictions on the variables in the rules----that is not fuzzy, but the closeness requirements on subformulas in template forms and rules. This always involves decomposing formulas into case parts , as on the tree diagram earlier, and matching some but not all the branches ---this is a process analogous that sketched by Minsky [2] as "matching frames by matching their terminals".

In the present case a chain with two inference rules is set up as follows:

[?it is good] T2

[animate2 wants ?it] using rule I2

[animate2 causes-to-move ?it] using rule I1

[John drank wine] E1. by fuzzy match to I1.

Hence template node "wine" and ?it are fuzzy matched, because of the fuzzy match of the last two lines of the chain, thus so referring the pronoun "it". It is virtually certain, as always, that there would be chains yielding other possible answer-referents, but none with chains shorter than this one.

4.DISCUSSION

The system described cannot be considered in any way adequately tested , partly because no one has any very clear idea of what constitutes a test in this area. But even to qualify, the basic mode must be shown to be stable under a considerable vocabulary and range of senses for words, and the extended mode must be shown to be determinate with a decent sized inventory of CSIRs. The present(end-1973) vocabulary is 500 words yet, though small, it is to my knowledge the largest of any operating deep-structure semantic analyser. At present, the program swaps in two large core images of 46K and 50K respectively, plus two small ones of 5K each ,all under control of a SAIL program. A trouble-free paragraph of text is processed in about 6 cpu seconds, while a quite simple sentence requiring inference chaining of the sort just described may require that sort of time by itself.

I think the use of the shortest possible CSIR chain can be defended as an extension of semantic preference used in setting up the basic representation. That preference was justified as an opting for the "semantically densest" interpretation which was, I claimed, the one "with the least meaning"(in the sense in which a string of random words carries the maximum possible information). Similarly, the shortest chain of inferences also minimises the information in play, and introduces the least extraneous inductive information into the system. It is clear that such a notion of information based choice is ultimately inadequate. We only have to consider a sentence like "I was named after my father" where it seems clear that we exclude one interpretation simply because it contains virtually no information at all. This alone shows there must be some qualification to a "minimising information" theory.

In this paper, and its predecessors, much emphasis has been placed on the template as a device to be parsed onto real text, because the subject investigated in this paper cannot be treated in isolation from an adequate linguistic base system. The inferring of a correct interpretation is intimately related to the systematic exclusion of competing interpretations, and any system that does not allow realistic ambiguity of sense and structure in at the start can hardly appreciate

this point. I have developed elsewhere [10] an abstract view of meaning along these lines: that to have meaning is essentially to have one meaning RATHER THAN ANOTHER. Or, put another way, having meaning essentially involves procedures for the exclusion of alternative interpretations. This, I believe, is the residual truth lurking beneath the "procedural view of meaning", a thesis which when taken a face value is patently false.

Another important aspect of the system is that it has a uniform representation and inference system at all stages of operation: there is no conventional division into syntactic, semantic and deductive or knowledge packets.

There has been no space in this paper for comparisons with the work of others, though the similarity of the task described here for the extended mode to the work of Charniak [1] will be obvious. There are overall similarities of aim and assumptions ,too, with the work of Schank [5] and Winograd[12]. One main difference of emphasis is the notion of preference. If there is such a notion in those works it is hidden away in the "hacks" and not brought to the fore where it belongs.To my knowledge the only other author who has emphasised the notion, though in a quite different context, is Quillian [4].

ACKNOWLEDGEMENT

This research was supported by the Advanced Research Projects Agency, Department of Defense(SD 183), USA.

REFERENCES

[1] E. Charniak, Jack and Janet in search of a theory of knowledge. Advanced papers of the Third International Joint Conference on Artificial Intelligence, Stanford Research Institute, 1973.

[2] M. Minsky, Frame Systems, unpublished MSS, MIT, November 1973.

[3] S. Papert, The Romanes Lectures, U.C.Berkeley, 1973.

[4] R. Quillian, Semantic Memory, . in M. Minsky, (Ed.) Semantic Information Processing. MIT Press, Cambridge,Mass., 1968.

[5] R. Schank and C. Rieger, Inference and the computer understanding of natural language. Stanford AI Laboratory Memo #197, May 1973.

[6] Y. Wilks, An Artificial Intelligence approach to machine translation, Stanford AI Laboratory Memo #161, March 1971; and in R.Schank and K.Colby, (Eds.) Computer Models of Thought and Language, W.H.Freeman, San Francisco, 1973.

[7] Y. Wilks, Preference semantics, Stanford AI Laboratory Memo #206 , July 1973; and in E. Keenan(Ed.) Formal Semantics of Natural Language,Cambridge U.P., Cambridge, 1974(in press).

[8] Y. Wilks, Understanding without proofs , in Advanced Papers of the Proceedings of the Third International Joint Conference on AI, Stanford Research Institute ,1973.

[9] Y. Wilks, Natural Language Inference, Stanford A.I. Laboratory Memo ,#211, October 1973.

[10] Y. Wilks, Decidability and natural language, Mind, Vol. LXXX, No.320, 1971.

[11] Y.Wilks and A.Herskovits, An intelligent analyser and generator for natural language, to appear in the Communications of the A.C.M.

[12] T. Winograd, Understanding Natural Language, Edinburgh U.P., Edinburgh. 1972.

[13] W. Woods, Procedural Sematics for a question-answer machine. Proc.FJCC , 1968.

PRODUCTION SYSTEMS AS MODELS OF COGNITIVE DEVELOPMENT

Richard M. Young

Bionics Research Laboratory,
School of Artificial Intelligence,
Edinburgh University

Abstract

A form of information processing model known as a "production system" (PS) is described. A PS is a set of rules each of the form C => A meaning that in the circumstances specified by C the subject performs action(s) A. PSs have certain advantages over other forms of model that make them especially suitable for describing cognitive development. This paper discusses their merits, with examples drawn from research into children's seriation behaviour.

Keywords: production systems, cognitive development, Piaget, protocol analysis.

1. Introduction: Production Systems

Comparisons have been drawn between computer programs and aspects of human behaviour since the late 1950s, but in recent years the nature of this relationship has been changing. As Klahr (1973b) points out, information processing models can be viewed at three levels ranging from the metaphoric to the concrete; it would seem that computer models have over the years been migrating downwards through these levels. At first, computer simulation studies merely used the general ideas of programming as a source of metaphors (e.g. Miller, Galanter & Pribram, 1960) or contented themselves with exploring the theoretical possibilities (e.g. Hunt, Marin & Stone, 1966). Since then, however, an extensive theory of human problem solving has emerged which yields information processing models closely tied to the details of the behaviour actually observed (Newell & Simon, 1972). Research in this area typically proceeds by the close analysis of an extended protocol of problem solving behaviour, followed by the construction of a model to reproduce the protocol as faithfully as is practicable.

A central technique in Newell & Simon's theory is the use of "production systems" to capture the regularities in a subject's behaviour. A production system (PS) is a set of rules expressing what the subject does under what conditions. Each rule is a condition-action statement of the form C => A, and means simply that in the circumstances specified by C the subject performs action(s) A. As a simple example, Newell & Simon give the following PS to describe the behaviour of a thermostat intended to keep the temperature of a room between 70° and 72°:

Th1: Temperature $<70^{\circ}$ and Furnace = off => Turn-on [Furnace]

Th2: Temperature $>72^{\circ}$ and Furnace = on => Turn-off [Furnace]

The action on the right hand side of Th1 applies whenever the condition on its left is satisfied, and similarly for Th2.

The /

/The first work to make use of PSs was concerned with the analysis of verbal protocols gathered from adults tackling symbolic problems in cryptarithmetic, formal logic and chess (Newell & Simon, 1972). And more recently, this line of work has led to attempts to model the control of short term memory (Newell, 1973a), the interface to the perceptual system (Newell, 1972; Klahr, 1973a), and the coding of visual information (Baylor, 1971).

However, this paper will focus on a different application of PSs: their use in describing the course of cognitive development. Cognitive development is among the potentially most fruitful topics to which the new information processing techniques are applicable (e.g. Farnham-Diggory, 1972). In part, this is because information processing psychologists have begun to recognise the significance of Piaget's developmental analysis of "genetic epistemology" (Piaget & Inhelder, 1969; Elkind & Flavell, 1969). But Piaget's formulations tend to remain at a rarified level of abstraction and, as has often been noted, it is hard to bring them into close contact with actual behaviour. This is especially frustrating from an information processing point of view, since Piaget seems so nearly to be talking in process terms. He deals with "representations", his "schemata" can be identified at least tentatively as fragments of program, and many of his observations seem to demand a processing explanation. Providing such an explanation consists of more than a mere "working out the details" of Piaget's theories. It involves the challenging task of designing information processing models of cognitive development, based on Piaget's notions (modified where necessary), which serve both to explicate those notions in concrete terms and to square them with the observed facts of human development.

This paper, then, summarises the case for PSs as tools in the investigation of cognitive development. We begin by presenting a PS to model a child's performance on a Piagetian task. After that we will discuss the merits of PSs, making use of examples from the literature to illustrate the main points.

2. Example: a PS for Seriation

All our examples will be drawn from research into the task known as seriation.* In a typical length seriation problem, a child is shown a jumble of wooden blocks of different lengths and has to arrange them in a straight line in order of size (thereby forming a staircase-like pattern). In the analogous problem of weight seriation, the blocks are all of the same size but differ in weight. Usually the difference is not directly perceptible, and the child has to use a balance to compare the blocks.

Seriation was introduced into the literature by Piaget, two of whose books report studies dealing with the task (Piaget, 1952; Inhelder & Piaget, 1964). For Piaget, the ability to seriate is of great importance for the child and underlies many of the other operations of concrete thinking; for instance, seriation and classification together provide the twin supports on which the child's (or the adult's) conception of number is based.

As /

---

* We will therefore not have occasion to deal with Klahr & Wallace's (1972) PS analysis of the Piagetian class inclusion problem.

/As with the other concrete-operational skills, Piaget divides the development of seriation into three main stages. A child at Stage I is unable to construct an ordered line. At first he simply aligns the blocks in an arbitrary order. Later he may construct two or more short series which he cannot combine, or else a single series omitting some of the blocks. Stage II is the level of "empirical seriation". A child at this stage succeeds in building an ordered series, but does so by what Piaget calls "trial and error"; that is, by repeated rearrangement of the blocks in the line being built. "Operational" seriation makes its appearance in Stage III. A Stage III child seriates by choosing the blocks in order of size, and constructing the series step by step from the smallest block, say, to the largest. It is at this stage too that he is first able to insert further, intermediate, blocks into an existing series.

Let us now look at a seriation of six blocks carried out by a five year old boy, Del. Figure 1 shows the layout of the blocks from

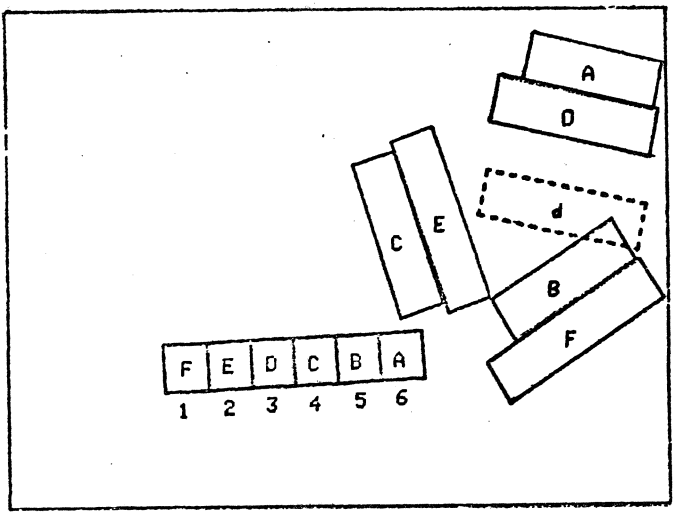


Figure 1: Layout of Del's seriation

Del's point of view. The blocks lying on the table drawn in solid lines show the initial configuration; the box-like structure depicts the final seriated line, with all the blocks standing upright; and the dashed outline represents a block in a temporary intermediate position. Figure 2 presents a summary of Del's behaviour on the task, divided into episodes each concerned with the placing of one block.

Episode	Summary	Line
1. Add F	Scan Pool Reach towards E Get F, put at left	
2. Add C	Get C, put next to F Examine	F C
3. Add E	Get E, put next to C Examine Switch C, E Examine	F C E F E C
4. Add B	Get B, put next to C Examine	F E C B
5. Add A	Move D to d Get A, put next to B Examine	F E C B A
6. Add D	Get D, put next to A Examine Switch A, D Examine Switch B, D Examine Switch C, D Examine Straighten B, A	F E C B A D F E C B D A F E C D B A F E D C B A F E D C B A

Figure 2: Summary of Del's seriation

It is not hard to summarise Del's seriation technique. As far as the choice of blocks is concerned, he starts with the biggest block in Episode 1 but thereafter simply takes successive blocks as they come to hand, regardless of their size. (The one exception is in Episode 5, where Del picks block A instead of the nearer block D). Each new block is added to the right hand end of the line, and is accepted there provided that it preserves the ordering of the line (Episodes 2, 4 and 5). Otherwise the new block is switched with its neighbour and then re-evaluated (Episodes 3, 6), this switching being repeated as often as necessary (Episode 6).

Before we look at the PS to model this performance, we must deal briefly with a couple of points that arise with "real" PSs. First, we assume that behaviour - that is, a production rule - is evoked always in the context of some active goal. These goals are organized into a stack, so that whenever a new goal is set up the old one is saved by being "pushed down" on the stack. And conversely, when the active goal is satisfied it is "popped off" the stack and the previous goal is reinstated.

Second, the matter of conflicts. Running a PS basically consists of repeatedly finding the rule whose left hand conditions are satisfied and then executing the actions on its right. But it can sometimes /

/sometimes happen that two or more rules have their conditions satisfied at the same time, and the question then arises as to which of them to evoke. It is necessary in such cases to have some way of resolving the conflict. The convention adopted here is to give priority to the rule whose conditions are the most restrictive. The usual situation is that the conditions of one rule, R1, are included in those of another, R2; in which case R2 is given precedence when both apply. We will meet several examples of this in just a moment. (Notice that other conventions are possible. For example, Baylor & Gascon (1974) order their rules by priority, and then always choose the highest-priority rule whose conditions are satisfied.)

T1: Goal=SERIATE => Set.goal [ADD.ONE]

S1: Goal=ADD.ONE => Get.block [next nearest]

T2: Goal=ADD.ONE & have.just [Get.block'd] => Change.goal.to [PLACE]

P1: Goal=PLACE => Put.block.at [right]

PG1: Goal=PLACE & have.new.configuration => Examine

PG2: Goal=PLACE & have.just [Examine'd; ] => Goal.satisfied [PLACE]

PG3: Goal=PLACE & have.just [Examine'd: ] => Switch.blocks

---

B1: Goal <new> = SERIATE => Set.goal [ADD.ONE <first>]

B2: Goal=ADD.ONE <first> => Get.block [biggest]

B3: Goal=ADD.ONE <first> & have.just [Get.block'd]
=> Put.block.at [far left]; Goal.satisfied [ADD.ONE]

Figure 3. Production system for Del's seriation

Figure 3 gives a PS to model Del's behaviour. It uses a fairly informal notation, and I hope that most of the rules are self-explanatory. Rules T1 through T2 are concerned with the cyclic behaviour of getting blocks from the pool and adding them to the line; P1 through PG3 govern the actual placing of the blocks, and finally rules B1 to B3 deal with the very first block. Rather than going through the whole PS and explaining the function of the rules one by one, let us watch what happens during a typical episode.

Suppose we plunge into the seriation at the beginning of Episode 3, when blocks F and C have already been placed. The top goal is SERIATE, and /

/and the only rule applicable is T1, so it fires off with the result that the active goal now becomes ADD.ONE. Now rule S1 is evoked, and Del reaches out and gets hold of the nearest block, E. At this point, rule S1 still has its condition satisfied, but so also does T2, which being more specific than S1 is therefore the next rule to fire: the active goal now becomes PLACE. Rule P1 is evoked, and Del moves block E to the end of the line. This results in a "new configuration", so rule PG1 fires (taking precedence over the less specific P1) and Del examines the shape of the line in the vicinity of the block just added. The outcome satisfies the condition for rule PG3, so he switches blocks E and C. That switch yields another "new configuration", so PG1 fires and Del examines the line again. This time rule PG2 is evoked, so the goal of PLACE is satisfied and it is popped off the stack, returning Del to the context of the top goal of SERIATE. And so on. In Episode 6, rules PG1 and PG3 fire in alternation no less than three times.

3. Advantages of Production Systems

Generality across situations

Why should anyone want to use PSs? Their principal methodological attraction is their ability to cope with minor variations of the task (Newell, 1973b). A conventional flowchart model, because of the (psychologically unwarranted) division it imposes between processing and control, typically has to be structured anew for each experimental condition. A PS, on the other hand, to serve as an adequate model of a subject must present a single processing system to handle all variations. This means that the PS an experimenter builds to model a subject's performance in one situation also predicts his behaviour in another. Thus to a far greater extent than is true of conventional processing models, a PS can serve as a miniature "artificial subject" whose reaction to different experimental manipulations can be empirically explored.

It is this property of PSs that makes possible the experimental technique used by Young (1973). In that study, the emphasis was on the possibility of obtaining empirical support for a proposed analysis of a child's seriation ability by examining his behaviour on a variety of problems closely related to the original task. Thus the PS to model a child's performance on a straightforward seriation predicts also what will happen when he is asked to construct the line out of sight behind a screen, for example, or to correct a wrongly seriated line. So the results of these "probe" tasks can be used to aid and support the analysis of the primary seriation task.

A series of studies by Baylor and colleagues at the University of Montreal is also closely concerned with the similarities and differences between related tasks (Baylor & Gascon, 1974; Baylor & Lemoyne, 1973). In addition to the basic length and weight seriations they used a third task, the hidden-length problem. This is again a seriation of length, but with the difference that the blocks used are kept hidden most of the time inside identical tubes. The child is allowed to have no more than two blocks exposed at any time, so in this respect the hidden-length task is similar to weight seriation, in which only two blocks can be compared at once. Baylor & Lemoyne (1973) tackle the issue of the "horizontal décalage" between length and weight seriation; i.e. the fact that although the two tasks have the same logical structure, performance on one lags behind the other by about two years. Their investigation of décalage rests on the possibility of writing a single PS to model a subject's behaviour on all three of the tasks. Figure 4 (adapted from /

Rule	Goal	Common to W, H, L	Specific to weight (W)	Specific to hidden length (H)	Specific to Length (L)
P1:	SERIATE	(PO > 0)			
	=>	(SET-GOAL INSERT)			
P2:	SERIATE	(PO = 0)			
	=>	(SATISFIED SERIATE)			
P6:	INSERT	(PB = 2)	(NEWBLOCK = (PB LIGHTER))	(NEWSTICK = (PB SHORTER)) ((PB RIGHT/SHORTER) > (PF LEFT/LONGEST))	(NEWSTICK = (PB SHORTER)) ((PB RIGHT/LONGER) < (PF RIGHT-OF-HOLE))
	=>	(SATISFIED INSERT)	(MOVE ((PB LIGHTER) (PB HEAVIER)) (PF HOLE))	(MOVE (PB SHORTER) (PF LEFT))	(LEAVE ((PB SHORTER)(PB LONGER)) PF)
P10:	COMPARE	(PF = 0)			
	=>	(SATISFIED COMPARE)	(MOVE (PO LEFT) PB) (MOVE (PO LEFT) PB)	(MOVE (PO RIGHT) (PB LEFT)) (MOVE (PO RIGHT) (PB RIGHT))	(MOVE (PO LONGEST) PB) (MOVE (PO LONGEST) PB)
P11:	COMPARE	(PF > 0) (PB = 0) =>	(LET NEWBLOCK = (PO LEFT)) (MOVE (PO LEFT) PB) (LET NEXT = (PF LEFT))	(MOVE (PF LEFT) (PB LEFT))	(LET NEWSTICK = (PO LONGEST)) (MOVE (PO LONGEST) (PB LEFT)) (LET NEXT = (PF LEFT))

Figure 4: Part of a PS for three tasks (after Baylor & Lemoyne, 1973)

/from their Figure 2) is part of such a PS. It shows how the conditions and actions can be divided into a generic part, common to all three tasks, and a specific part, adapted to the particularities of each. (PO, PB, and PF refer to the Original, Balance, and Final positions).

Role of the environment

Unlike a flowchart, the explicit control structure of a PS makes clear exactly which decisions the child has control over and which others are forced on him by the environment. To see this, consider a fragment of a flowchart of the form shown in Figure 5, which simulates a child performing action A followed by action B. Why are the actions done in the order A then

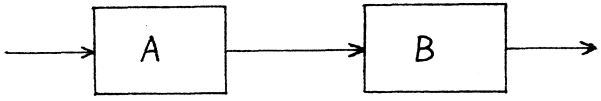


Figure 5. Fragment of a flowchart

B instead of the other way round? The flowchart provides no indication whether the sequence is decided by the child's problem solving processes, and is therefore "arbitrary" in the sense that B could equally well be carried out before A as after it, or whether the order is determined by the very nature of the task, as would be the case if A were the action "pick up the doll" and B were "squeeze the doll".

But a PS exhibits directly the factors that determine what actions are taken. Thus in the PS of Figure 3, we can see how rules like PG2 and PG3 isolate the internal and environmental components. Both rules demand that the child have just performed an Examine, but they respond differentially to the shape of the line. Baylor & Gascon (1974) devote considerable effort to showing how a child's seriation technique uses features of the environment such as the "hole" left in a line of blocks when one of them is removed, and they find that an increased flexibility of response to aspects of the environment is one of the themes characterising a child's growing skill.

So a PS can represent the role of the environment in governing the child's behaviour in a way that a flowchart normally cannot. For a PS presents the set of possible actions that the child can take together with the basis on which he decides between them, whereas a flowchart or algorithm states only the outcome of that decision. This is important, since young children are usually given concrete tasks, such as seriation, which involve a high degree of interaction with the physical world; and also because a pre-operational child relies to a great extent on the environment as an external memory to reduce his cognitive load.

Independence of rules

According to Newell & Simon (1972), "production systems are the most homogeneous form of programming organisation known". One consequence of this homogeneity is that a PS's simple organisation as a collection of rules leads to a corresponding structure in the resulting behaviour. Each /

/Each rule represents a fragment of potential activity that is a meaningful component of the total problem solving process. As a result, individual production rules frequently possess a kind of local plausibility that makes them intelligible in their own right, regardless of which other rules are present. This makes it easier to see how the child could have acquired them individually. In Figure 3, for example, each of the rules can be seen as reflecting some particular aspect of the seriation task. Rule T1 is a specialised version of a general rule saying something like: "If you want to do something to each of a set of objects, do it to one of them"; the rule keeps applying until there are no objects left. Similarly, T2 seems to make sense independently of this particular seriation context. (Namely: if one is trying to extend a line and he has a block in his hand, then he should place it in the line). And so on; the completion of this exercise can be left to the reader.

The structural independence of production rules has two consequences, both of considerable psychological import. First, from the subject's point of view, it removes the need for self-programming: the rules relevant to a situation are simply evoked when their particular conditions are satisfied. But from the investigator's point of view it means that various sets of rules can be combined freely to form working PSs. Young (1973) provides a dramatic example of this flexibility. There a collection of rules is presented, various subsets of which form PSs which reproduce the different seriation methods and pre-seriation phenomena noted by Piaget. Some of the rules are concerned with episodic behaviour of trying the blocks one by one - like rules T1 to T2 in Figure 3 - and they have to appear in any PS for seriation (or pre-seriation). But provided these rules are included, almost any selection of some or all of the remaining rules yields a working, psychologically plausible PS for (pre-)seriation. With the conventional form of model, the idea of providing such a "kit" - which serves to specify an entire space of seriation processes as mere collections of its parts - seems unthinkable.

Incremental growth

Perhaps most important for the understanding of development is the fact that the independence of the individual roles makes it possible to extend a PS incrementally simply by adding new ones. For instance, to return for a moment to the simple thermostat described above, we can easily add a more advanced feature to take special action when the temperature falls too low, merely by adjoining to Th1 and Th2 the new rule:

Th3: Temperature<32^{o}=>Call-repair-man; Turn-on [Electric-heater]

or whatever.

In a similar vein, Figure 6 shows a hypothetical sequence of PSs, each differing from the one before simply by the addition of one or two new rules. It represents a hypothetical child who initially can only arrange the blocks in a line, but then gradually acquires the rules that lead him through one or more of the observed pre-seriation phenomena, on to simple seriation, and finally more reliable seriation.

```
T1:   Goal=SERIATE                                  =>  Set.goal [ADD.ONE]

B1:   Goal <new> = SERIATE                          =>  Set.goal [ADD.ONE <first>]

S1:   Goal=ADD.ONE                                  =>  Get.block [next nearest]

T2:   Goal=ADD.ONE & have.just [Get.block'd]        =>  Change.goal.to [PLACE]

B3:   Goal=ADD.ONE <first> & have.just [Get.block'd]
                    =>  Put.block.at [far left];  Goal.satisfied [ADD.ONE]

P1:   Goal=PLACE                                    =>  Put.block.at [right]      A

-------------------------------------------------------------------------------

B2:   Goal=ADD.ONE <first>                          =>  Get.block [big]           B

-------------------------------------------------------------------------------

PG1:  Goal=PLACE & have.new.configuration           =>  Examine

P2:   Goal=PLACE & have.just. [Examine'd:  <too big>]
                                    =>  Reject.block; Goal.failed [PLACE]         C

-------------------------------------------------------------------------------

PG3:  Goal=PLACE & have.just [Examine'd: [illegible]]=>  Switch.blocks            D

-------------------------------------------------------------------------------

S2:   Goal=ADD.ONE                                  => Get.block [similar to last] E
```

Figure 6. Hypothetical sequence of production systems

Some details have been omitted, but roughly, rules T1 through P1 constitute a PS to build the blocks into a line without regard for size (see location A in Figure 6). Next the child acquires the idea of starting with the biggest - or at least, a big - block; this is represented by the addition of rule B2 (location B). At this point the child is still constructing an unordered series corresponding to Piaget's Stage 1. However, he next begins to satisfy the ordering requirement, at least enough to examine each block after adding it to the line and reject it if it is too big; this is expressed by the addition of rules PG1 and P2 (location C). At this point he is constructing partial seriations with some of the blocks omitted. Perhaps he next acquires a simple correction technique, that of switching an oversize block with its neighbour, by addition of rule PG3 (location D). Like Del, he can now seriate successfully provided that the blocks are not too numerous: his performance would be classified by Piaget as Stage II, trial-and-error seriation. Finally we may suppose that having acquired rule S2 he begins to choose blocks according to their size (location E). His seriation behaviour then appears more or less "operational" depending on the accuracy of his selection. The combination of an at least approximately correct selection (rule S2) with a means for correcting slight errors (PG3) means that he is now able to seriate blocks both harder to /

/to discriminate and more numerous than before.

It is of course true that this sequence does not provide an explanation of development, since nothing has been said about the origin of the new rules. But it is a necessary first step towards such an explanation, for it shows how the description of a child's ability in terms of PSs reveals the gradual, cumulative progression underlying the striking (and discontinuous) changes in his overt performance. The next step is to understand how he acquires these rules from the regularities in his existing behaviour.

Acknowledgement. The author acknowledges the financial support of the Social Science Research Council during the preparation of this paper.

4. References

Baylor, G. W. Program and protocol analysis on a mental imagery task. 2nd International Joint Conference on Artificial Intelligence. London: British Computer Society, 1971.

Baylor, G. W. & Gascon, J. An information processing theory of aspects of the development of weight seriation in children. Cognitive Psychology, 1974, 6 (in press).

Baylor, G. W. & Lemoyne, G. Experiments in seriation with children: Towards an information processing explanation of the horizontal décalage. Université de Montréal, Institut de Psychologie, M.C.P. #15, 1973.

Elkind, D. & Flavell, J. H. (eds.) Studies in Cognitive Development. New York: Oxford University Press, 1969.

Farnham-Diggory, S. (ed). Information Processing in Children. New York: Academic Press, 1972.

Hunt, E. B., Marin, J. & Stone, P. J. Experiments in Induction. New York: Academic Press, 1966.

Inhelder, B. & Piaget, J. The Early Growth of Logic in the Child: Classification and Seriation. New York: Harper & Row, 1964.

Klahr, D. A production system for counting, subitizing, and adding. In W. G. Chase (ed.) Visual Information Processing. New York: Academic Press, 1973.(a)

Klahr, D. An information processing approach to the study of cognitive development. In A. D. Pick (ed.) Minnesota Symposia on Child Psychology, 7, Minneapolis: University of Minnesota Press, 1973.(b)

Klahr, D. & Wallace, J. G. Class inclusion processes. In S. Farnham-Diggory (ed.) Information Processing in Children. New York: Academic Press, 1972.

Miller, G. A., Galanter, E. & Pribram, K. H. Plans and the Structure of Behaviour. New York: Holt, Rinehart & Winston, 1960.

Newell, A. A theoretical exploration of mechanisms for coding the stimulus. In A. W. Melton & E. Martin (eds.) Coding Processes in Human Memory. Washington, D.C.: Winston, 1972.

Newell, A. Production systems: Models of control structures. In W. G. Chase (ed.) Visual Information Processing. New York: Academic Press, 1973.(a)

Newell, A. You can't play 20 questions with nature and win: Projective comments on the papers of this symposium. In W. G. Chase (ed.) Visual Information Processing. New York: Academic Press, 1973.(b)

Newell, A. & Simon, H. A. Human Problem Solving. Englewood Cliffs, N. J.: Prentice-Hall, 1972.

Piaget, J. The Child's Conception of Number. New York: Humanities Press, 1952.

Piaget, J. & Inhelder, B. The Psychology of the Child. New York: Basic Books, 1969.

Young, R. M. Children's seriation behaviour: A production system analysis. Ph.D. dissertation, Carnegie-Mellon University, 1973.