Aaron Sloman
Steven Hardy

GIVING A COMPUTER GESTALT EXPERIENCES

POPEYE is a vision program currently being developed by a small group at Sussex University. The aim is to explore the problems of interpreting messy and complex pictures of familiar objects. Familiarity is important because knowledge of the objects helps to overcome the problems of dealing with noise and ambiguities. Figure 1 gives an example. Pictures are presented to POPEYE in the form of a two-dimensional binary array, representing scenes containing overlapping letters made of "bars". Pictures are generated by programs either from descriptions or with the aid of an interactive graphics terminal. We are using POP2, the programming language developed for A.I. at Edinburgh University. However, we have found it useful to extend the language, and this paper describes some of the extensions. POPEYE's domain-specific knowledge will be described on another occasion.

POPEYE should process the picture in a sensible, flexible way, so that the main features to have emerged at any time can redirect the flow of attention. This applies at all levels. For instance, instead of doing the usual complete scan of a picture before starting higher-level processes, POPEYE builds histograms while doing the scan, and uses them to indicate the need to switch to something more important. But if higher level processes run out of things to do, the scan is continued. Similarly, the higher-level processes themselves may generate something still more important, in which case they have to be suspended. So, many different kinds of processes need to be able to coexist. Starting higher processes before lower level surveys are complete generates horrific problems to do with not jumping to conclusions too quickly, and being prepared to undo decisions, occasionally. However it seems that the human ability to make something meaningful emerge quickly out of a mess of data depends on this. To make it easier to write programs which generate and control many different processes we have implemented a "process" package, including scheduling facilities. These will be described below.

Another kind of problem is concerned with storage and retrieval of large amounts of information. For instance, the same item may be relevant in different ways in different contexts: sometimes a line-segment is relevant to a subproblem because of its orientation, and sometimes because of its loca-

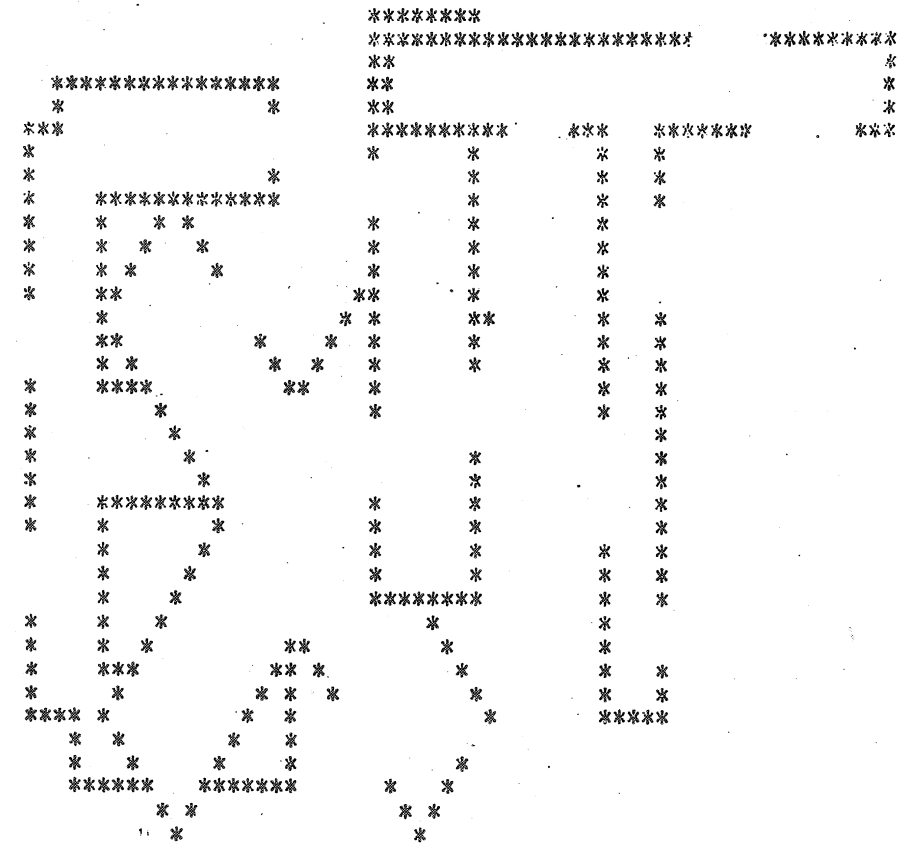

Figure 1.

This is a simple example of the kind of picture the program will be expected to analyse and interpret. Most people find that familiar letters, and possibly also a word, emerge very quickly.

tion in the picture. So a lot of cross-indexing is needed. Moreover, the program will eventually have to have a very large number of different concepts, relating to different kinds of structures and relationships which can be found in the picture and the scene, so an efficient retrieval system is needed for matching picture-fragments and scene-fragments with previously stored structural descriptions. To deal with this problem we are using a pattern-directed data-base package, with demons which can react to attempts to access or modify the data-base.

THE DATA BASE

The data base contains many features found in PLANNER-like languages. It is a modified and extended version of the HBASE system designed by Harry Barrow as a POP2 library package. The data base consists of a network of patterns, with an indexing mechanism for finding a pattern quickly, given a template for it. There is a POP2 library package for associating different values with the same data base item in different "contexts". The main extension to HBASE has been the provision of a "demon" mechanism. A demon is a procedure invoked by the occurrence of a specific type of event involving a pattern, e.g. when ASSERT, DENY or RETRIEVE is applied to the pattern. A "possibilities" list mechanism is provided, so that real or simulated data-base items may be generated and tried one at a time, if necessary. Generators use the process package, described below. Possibilities lists are themselves generators, so that explicit lists of possibilities don't have to be built up.

Eight standard types of pattern-invoked demons are provided, whose names should indicate their functions: PREASSERTED, IFASSERTED, PREDENIED, IFDENIED, PRENEEDED, IFNEEDED, ALREADY-ASSERTED, ALREADYDENIED. The last two types are for trapping redundant data-base alterations - especially useful for debugging. The system makes it easy to define and use further types of demons. For instance, the function RUNALL takes a pattern, and a list of types of demons, and causes all demons of the types listed, whose patterns match the given pattern, to be executed. This is used in the processes package. Processes may be sent "messages" in the form of patterns. Each class of processes uses a specialised set of demons to react to such messages. Later we shall illustrate the use of the data base in combination with the process package.

THE PROCESS PACKAGE

The process package consists of (a) a scheduler, (b) process-creating facilities and (c) a message-broadcasting system, which uses the data base. These will be described in that

order.

The scheduler is our attempt to overcome the following problem. POPEYE will be analysing a wide variety of picture fragments, yet wanting to switch attention to the most important features as they emerge. This suggests the need for some central intelligent process with an overview of everything happening, so that informed decisions can be taken. Yet no central administrator can manage all the complex different kinds of knowledge required for comparing different possible developments. We have therefore tried to decompose the scheduling task as follows.

The central scheduler knows only about the major categories of jobs which may be ready to run. These categories are listed in SCHEDULE. Lower-level processes know how to decide, on the basis of detailed evidence available to them, when to create a job, and which category to assign it to, or more generally which job-description to associate with it. For instance, processes concerned with linking line-fragments into larger clusters might be able to decide to create a hypothesis that a junction between two bars has been found. That hypothesis will be a process which may collect further evidence, and perhaps link itself with other bar-hypotheses to generate a letter hypothesis.

As each process is either initially created, or given some new evidence, it can put itself into an appropriate category of runnable jobs, leaving it to the central scheduler to decide which category is currently most important. So all the scheduler has to do is find the most important non-empty category, and select a job from it, either arbitrarily or by asking a manager for that category, By allowing schedulers themselves to be processes which can run for a while, then halt, then continue, we can allow some of the jobs to have their own local versions of the scheduler, with their own local SCHEDULE. Since POP2 does not provide a time-sharing facility, the programmer has to take care to design modules so that they don't run for too long. For instance, whenever a process does something which may have created or reactivated more important processes, then it should suspend itself soon after. The DETACH facility, described below, makes this possible.

The use of job-descriptions instead of numerical priorities has a number of advantages. For instance if the task changes (e.g. "find all vertical bars"), then the order of importance represented by the schedule can be altered on the basis of knowledge of which kinds of processes are relevant to the task. Further, if a relatively important process needs a specific low-level process to be run (e.g. a bar process needs a line-segment process to get information about junctions at its ends), then instead of juggling with priorities the first process can look in the appropriate job category, find what it

needs and run it.

The process-creating facilities make extensive use of "partial application", one of the features of POP2. A POP2 function can be partially applied to a set of arguments, forming a "closure". For example, the following function assigns X to be the second element of L.

```
FUNCTION PUTSECOND X L;  X -> HD(TL(L)); END;
```

If L1 is a list, we can make a closure which behaves like a function whose execution assigns the word "cat" to be the second element of L1, thus:

```
PUTSECOND(%"CAT",L1%) -> PUTCAT;
```

Every time PUTCAT is executed, it runs the function PUTSECOND in an environment in which X and L are given the two "frozen values" specified. Thus PUTCAT has both program and data stored within it. It can be executed, like a function, but it can also be accessed, like a data structure. Both the frozen values and the frozen procedure may later be altered. By giving a POP2 closure a pointer to itself, we can allow it to update its own frozen values. So it can behave like a process with a memory, unlike ordinary functions. Doing this in POP2 requires defining functions which are ugly and obscure. So we have defined POP2 "macros" which alter the syntax, hiding the ugliness. The central ideas come from Steve Hardy's POPCORN system, currently under development.

The most basic mechanism is the creation of a "process" consisting of a function and a variable-binding environment which, unlike an ordinary POP2 closure, will remember assignments to its variables. If FOO is a function in which the variables P Q and R are not declared as locals, then the brackets "/::" and "::/" can be used to create a process whose "manager" is the function FOO and whose environment uses the values of P Q and R at the time of creation.

```
FOO /:: P Q R ::/  -> PROC;
```

PROC is now a process which will run FOO each time it is executed. But if FOO assigns new values to P, Q or R, then they will be remembered for subsequent executions.

Such processes can be accessed from outside. For instance PROC("P") will produce the current value of P in PROC. Processes automatically get updaters, so that `4 ->PROC("P");` will change the stored value of P. ENVEVAL can be used to evaluate a function inside a process. Each process gets a local variable SELF, so the code for the manager function can include instructions involving SELF. The function SAVESELF allows a process to make a copy of its current state. The function SHOWSELF enables any process to print itself out in a neat format. The operation SETMANAGER may be applied to a

function or lambda expression and will give the current process a new manager function.

Similar effects could have been obtained using the state-saving functions of POP2. However, this would have been much more awkward to use, and much less efficient.

We have provided additional macros and functions, using the process-creating brackets, to provide features analogous to the "classes" of SIMULA67. We use the term "processmaker". This denotes a function which creates a process, does some initialising computations, such as setting up data-structures for the process, then returns the process. The process has the features mentioned above. Further, the macro DETACH, illustrated below, makes it possible to write code which can be executed, left for a while, then continued. The process-initialising instructions are distinguished by occurring between the words "INITIALISE" and "BEGIN".

Here is a simple example: a processmaker which produces generators for pairs of numbers.

```
PROCESSMAKER PAIRS XMAX YMAX;
 INITIALISE;
   VARS X Y;
 BEGIN;
   FOR X FROM 1 BY 1 TO XMAX DO
    FOR Y FROM 1 BY 1 TO YMAX DO
        CONSPAIR(X,Y)
        DETACH;
    ENDDO;
   ENDDO;
 TERMIN;
END;

VARS PAIRGEN;
PAIRS(5,7) ->PAIRGEN;
```

PAIRGEN is now a process which will generate a new pair each time it is executed, until it is exhausted, in which case it produces the POP2 "terminator" TERMIN.

A macro GENERATE, which uses DETACH, is provided to enable one generator to create and use others, on the assumption that an exhausted generator will produce TERMIN as its result. The following is a recursive processmaker which creates a generator for the atoms of a tree. (In POP2, HD and TL correspond to CAR and CDR of LISP).

```
PROCESSMAKER FRINGE TREE;
 INITIALISE;
 BEGIN;
  UNLESS TREE=NIL THEN
   IF ATOM(TREE) THEN TREE; DETACH;
   ELSE
        GENERATE FRINGE(HD(TREE));
        GENERATE FRINGE(TL(TREE));
   CLOSE;
  CLOSE;
 TERMIN;
END;

VARS TREEGEN;
FRINGE([A [B C] [[D] E F]])->TREEGEN;
```

TREEGEN is now a process which when first called will produce "A", then the next time "B", and eventually TERMIN. GENERATE used like this is rather inefficient, but illustrates the facilities.

Neither TREEGEN nor PAIRGEN takes arguments. However, a process manager may be a function which takes arguments, in which case it is easy for other processes to communicate with it, without using ENVEVAL. For instance we are currently experimenting with a class of processes which all use the following manager function:

```
FUNCTION PROCESSMANAGER MESSAGE;
  IF ISFUNC(MESSAGE) THEN MESSAGE()
  ELSE
   RUNALL(MESSAGE,DEMONTYPES)
  CLOSE;
END;
```

A process with this manager will run only if given an argument, MESSAGE. If it is a function it will be executed in the environment of the process, otherwise it is assumed to be a pattern which will invoke one or more demons of the types specified in DEMONTYPES. The data-base index is used to find relevant demons quickly. Different classes of processes may use different versions of DEMONTYPES. For instance, if some have demons which know how to understand English, then they can talk to one another in English. By having two types of messages, functions and patterns, we cater for two cases. When communicating with a process, if you know exactly what you want it to do, then send it a function to execute. Otherwise send it a pattern and let it use its own expertise. By allowing processes to have their initial versions of DEMONTYPES set up by the relevant processmaker, we allow instances to "inherit" procedural attributes from their species. However, individual processes may modify their own versions.

The concept of a processmaker is still evolving. We are currently experimenting with ways of allowing assignments to process-variables to trigger suitable actions. Similarly, it is possible to attach "exit" demons to a process, which will automatically run whenever control is about to leave the process. Thus a process which needs to be careful about something, for a short time, can give itself such a demon, for as long as is necessary.

A process may be located at an "address" in the data base, or possibly at several addresses. The function STOREAT takes a pattern and locates the current process at the address specified. The address is a public description of some important facts about the process, e.g. its type and location in the picture, or maybe some of its relations to other things. Inside the process is the more detailed information it needs to do its stuff, but which it would be wasteful to have represented in the data-base index. We could have represented everything by data base items, and done without processes, but that would have too many disadvantages, of the sorts documented by Bornat, Brady and Weilinga in their contributions to this conference.

The message broadcasting mechanism is provided to enable processes to communicate with one another, using their addresses in the data base. A message may be either a pattern, which will invoke appropriate frame demons, or a word or a function. In the latter two cases it will be evaluated in the environment of each recipient. The target should be a description of the intended recipients. For instance

```
LAMBDA;
   IF SIZE>10 THEN PR(SEGMENTLIST) CLOSE
END
   --> <<LINE $$DIRECTION ==>>;
```

causes every line whose orientation matches the current value of DIRECTION ($$ means "use current value"), no matter what its location (== will match anything), to print its segment-list if it contains more than 10 points.

```
<<NEWPOINT $$POINT>>
 --> <<LINE $$DIRECTION $$LOCATION>>
```

will tell the appropriate line process that a new point has been found for it. This should activate appropriate demons in the environment of that process. To postpone message sending until there's nothing more important to be done, use the following syntax:

<message> --> (<target description>, <job category>);

This will cause the sending of the message to be a job to be activated by the scheduler. The sender will presumably detach,

and hope for a reply later.

When one process runs another by calling it explicitly, the second can reply by leaving results on the stack, as subroutines do in POP2. But when message broadcasting is postponed, the sender may not be active when the message is received, so the stack cannot be used for replies. If a reply is needed, the message must include some data-structure which the sender can examine later. For instance,

```
VARS LETTERBOX; CONSREF(NIL) ->LETTERBOX;

LAMBDA LETTERBOX;
  IF SIZE > 10 THEN
    SELF :: CONT(LETTERBOX) ->CONT(LETTERBOX)
  CLOSE
END(%LETTERBOX%)
```

uses partial application to create a message which can be sent to a lot of lines. By looking at CONT(LETTERBOX) from time to time, the sender will discover which of the lines has a size greater than ten. Similarly a letterbox can be included as part of a pattern message. Incidentally this shows how processes which at first only know of one another by description can get direct pointers to one another.

This message-sending mechanism, combined with other POP2 features, such as interrupts and incremental compilation, enables the programmer to communicate with processes in much the same way as they communicate with one another. This is indispensable during debugging.

Here is an example showing how data base demons can use process brackets to save a portion of the environment in which they were created. Demons of type IFNEEDED are activated when a RETRIEVE command is unsuccessful. We want an IFNEEDED demon to try to answer a question, and if it fails, to plant an IFASSERTED demon which will watch out for the answer. If the answer turns up later, the second demon will record the fact, which may trigger off other demons, then kill itself. The problem is that the second demon may run long after its creator has exited, so that it needs to save relevant parts of its binding environment, using the process brackets. In the example, "%%" means use the current value of, and "%*" means give this variable a value during matching.

```
IFNEEDED <<SUM $*X $*Y $*Z>> ;
  VARS X Y  Z X1 Z1;
  IF X <1  THEN QUIT(FALSE)
  ELSE
   X-1 ->X1;   Z-1 ->Z1;
   IF RETRIEVE <<SUM  $$X1 $$Y $$Z1>> THEN
        SUCCEED()
   ELSE
        IFASSERTED <<SUM $$X1 $$Y $$Z1>>;
          PR('FOUND THE ANSWER');
          KILLSELF();
          ASSERT <<SUM $$X $$Y $$Z>>;
        END /:: X Y Z ::/
   CLOSE
END
```

A call of RETRIEVE<<SUM 3 5 8>> when <<SUM 3 5 8>> has never been asserted, will activate the IFNEEDED demon. If, in addition, <<SUM 2 5 7>> has never been asserted, which will produce a recursive call of the demon, and it cannot be proved, then the IFASSERTED demon, in the form of a process remembering the values of X Y and Z, will be added to the data base. If it gets triggered later, it will remove itself from the index and store the solution to the problem, which may activate other demons waiting for the solution. Demons can be given names, making it easy for one to access another and kill it if it becomes redundant. Thus chains of demons in the data base can provide some of the functions of a multiple stack mechanism. The lack of efficiency is perhaps compensated for by the ease of inter-process communication.

OVERVIEW

POPEYE sets a number of different processes going in parallel when presented with a picture to interpret, Some collect global statistics about the picture, some search for dot configurations suggesting lines. These may trigger off other processes, some deciding whether parallel lines should be linked to form "tubes", some keeping track of junctions, some trying to link "tube-sections" into larger structures, etc. This kind of "breadth first" approach is required mainly because, with a large amount of information available for analysis and interpretation, it may not be easy to decide what to do next, e.g. which configurations to look for, and where to look for them. Deciding between such alternatives itself requires analysis of evidence, and it will not be obvious what the important clues are, nor where they are. So initially many possibilities are sampled, until items both unambiguous and relatively important begin to emerge, such as a long line, an unambiguous clue to the location of a line, an aspect of the style of the picture,

or a set of linked fragments which uniquely identify a known letter. What counts as important will depend on the stage of analysis. The scheduler will notice the emergence of new important jobs and run them before others. This approach seems to be very similar to that used in the HEARSAY II system at Carnegie Mellon.

We are guided by several principles. One is to use descriptions wherever possible instead of numerical weights or priorities, so that the program has adequate information for taking decisions. Another is to select hypothese not on the basis of their support, or probability, but on the basis of their explanatory power (as recommended in Popper's philosophy of science). For instance, work on a large picture fragment rather than a small one, but work on a scene fragment rather than a picture fragment. But this requires a further principle, which is not to let any hypothesis be generated unless there is good reason to do so and one is not simultaneously generating large numbers of rival hypotheses. When there is no way of choosing between alternatives on the basis of current evidence, don't generate either. Instead there should be a description of what is common to both. Hope that either new detailed evidence will emerge to decide the issue (or look for it if you know it can be found quickly), or else global relationships between ambiguous, intermediate structures will enable larger, unambiguous clusters to emerge without combinatorial searches. For example, as Larry Paul's paper for this conference shows, global relations between a cluster of ambiguous limb-like regions may determine which are arms and which legs, when there's little hope of finding local details to discriminate them.

Making all this work requires the program to have a large store of concepts corresponding to various "intermediate" levels of structure, so that it never needs to take large leaps from what it knows to shaky hypotheses. In relation to dots and letters, intermediate concepts include "line segment", "bar", "bar junction". We view Grape's work as illustrating the importance of using intermediate structures between line-junctions and pictures of whole objects. This kind of expertise will work only in a "friendly world". If pictures are too noisy, or objects are piled up so that most things are almost entirely occluded, or if letters are juxtaposed so that gaps between them form too many spurious clues, then POPEYE, like a person, will get confused. However, there are cases where alternative hypotheses need to be coped with, for instance in pictures where there are quite large chunks with globally consistent dual interpretations. Geoffrey Hinton's conference paper suggests a way of using relaxation to deal with this.

Currently POPEYE manages to see important bars, and

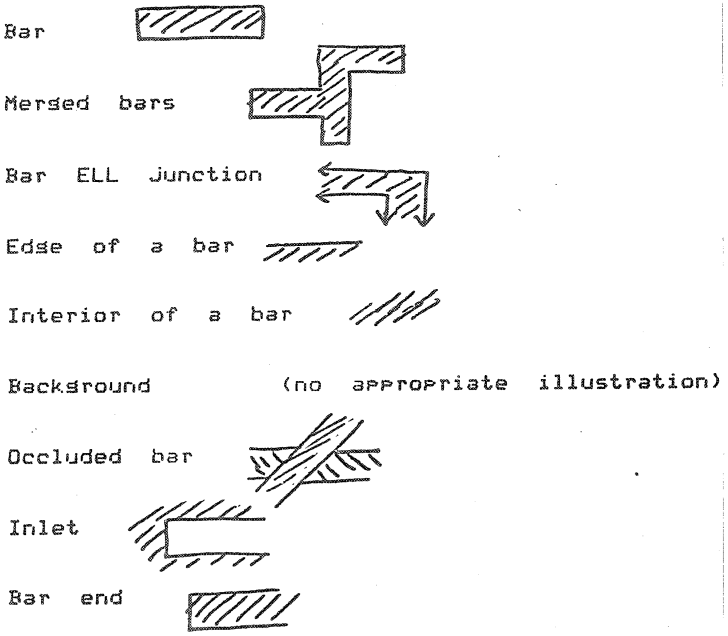

Figure 2

Some concepts from the domain of bar-scenes.

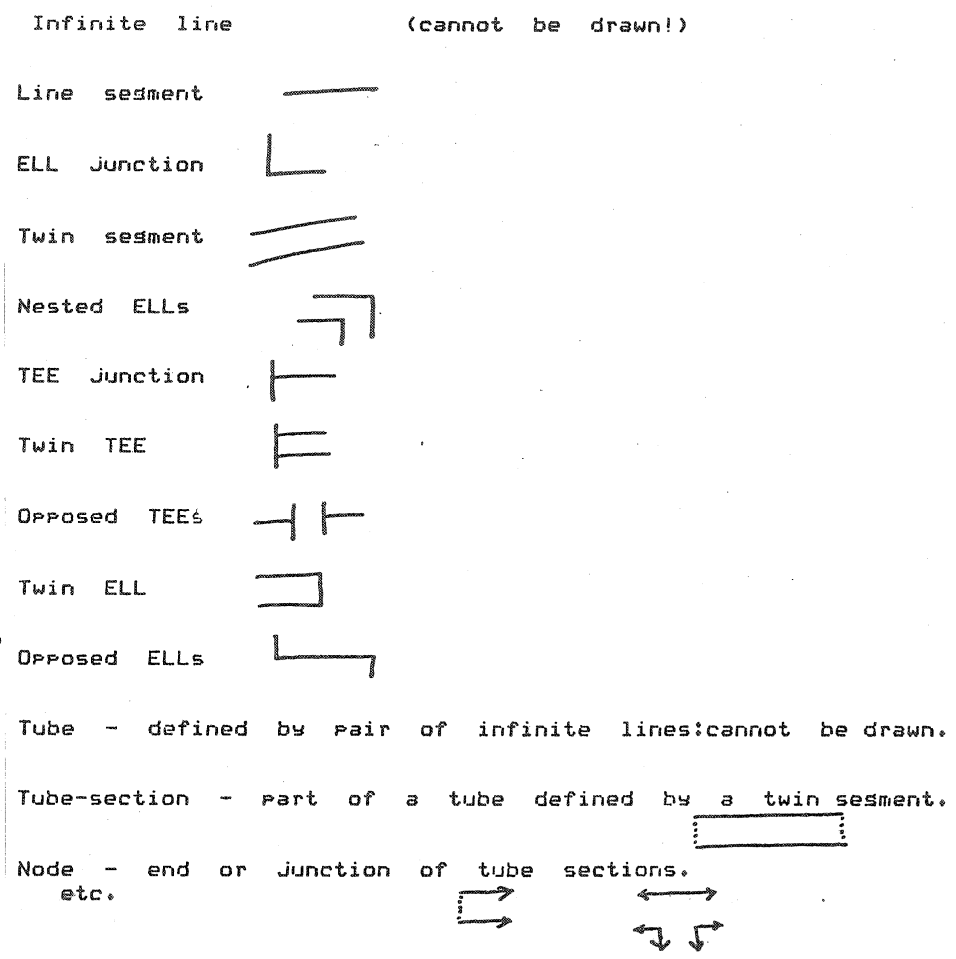

Figure 3

Some concepts from the line-picture domain.

junctions between them, without exploring all points and line-segments in the picture. A fair amount of positive or negative noise (spurious or missing dots) can be added without upsetting the process much. We have begun to work on the concepts required for dealing with significant clusters of bar fragments, so as to enable whole letters to emerge from the mess. The figures illustrate some of the concepts involved.

ACKNOWLEDGEMENTS

The POPEYE project is funded by the Science Research Council. Much of POPEYE'S domain-specific code is being written by David Owen, who joined the project in September 1976, followed by Geoffrey Hinton in January 1975. The idea of using histograms and other global picture descriptions to control processing came from the work of Max Clowes and Frank O'Gorman. The latter has been closely involved in our theoretical discussions. The design of the project also owes much to interactions with Sylvia Weir, Alan Mackworth, Mike Brady and Richard Bornat. The process package is influenced by Carl Hewitt's work on "actors", and the paper by Bobrow and Norman. Instead of standard POP2 we are using POP10, a dialect developed by Julian Davies and maintained on the Edinburgh PDP10 by Arnold Smith.

But for Pat Norton's speed, accuracy and patience this paper would never have been typed on time.

CORRIGENDUM

P246 ninth line from bottom should read ENVEVAL ("P", "PROC") and not PROC ("P").

RELEVANT REFERENCES

Bobrow,D. & A. Collins (1975) Representation and Understanding, Academic Press.

Bobrow,D. & D. Norman (1975) Some principles of memory schemata in Bobrow and Collins.

Clowes,M.B. (1971) On seeings things in Journal of Artificial Intelligence.

Davis,L.S., A. Rozenfeld. & S.W. Zucker (1975) General purpose models: expectations about the unexpected, Computer Science Technical Report, TR347, University of Maryland.

Erman,L.D., R.D.Fennell, V.R.Lesser & D.R.Reddy (1973) System organisations for speech understanding in Proc.3rd I.J.C.A.I. Stanford.

Grape,G.R. (1973) Model based (intermediate level) computer vision, Stanford A.I. Memo AIM-201.

Hewitt,C., P.Bishop & R.Steiger (1973) A universal modular ACTOR formalism for A.I. in Proc.3rd I.J.C.A.I., Stanford.

Hinton,G. (1976) Using relaxation to find a puppet, this conference.

Magee,B. (1974) Popper, Fontana Modern Masters.

O'Gorman,F. & M.B. Clowes Finding picture edges through collinearity of feature points in Proceedings 3rd I.J.C.A.I. Stanford, 1973.

Paul,J.L. (1976) Seeing puppets quickly, this conference.

Shirai,Y. (1975) Analysing intensity arrays using knowledge about scenes in Winston.

Stansfield,J.L. (1974) Active descriptions for representing knowledge in Proceedings AISB summer conference.

Reddy,D.R., L.D.Erman, R.D.Fennell & R.B.Neely The hearsay speech understanding system in Proc.3rd I.J.C.A.I., Stanford (1973).

Turner,K.J. (1974) Computer perception of curved objects using a television camera in Proceedings A.I.S.B. summer conference.

Waltz,D.(1975) Understanding line drawings of scenes with shadows in Winston.

Winston,P. (1975) The psychology of computer vision, McGraw-Hill.

Hardy,S. The POPCORN75 reference manual, forthcoming.