Computing with Text-Graphic Forms

Fred H. Lakin *

Stanford Artificial Intelligence Laboratory

Abstract

Computing with text-graphic forms occurs when text-graphic patterns are used to direct the processing of other text-graphic patterns. The PAM graphics system was designed for just this kind of computation; PAM stands for *PA*ttern *M*anipulating -- PAM is a generalization of LISP (McCarthy [0]) from computing with (textual) symbolic expressions to computing with text-graphic forms.

Like LISP, PAM acheives processing power by providing atomic objects, means of structuring them into complex objects and taking them apart, and equality tests for objects. *Text-graphic engines* can then be defined, allowing text editors, text-graphic editors, circuit design aids and visual language processors to all be developed in the same LISP-like environment. Examples discussed in the paper are:

handPAM is an agile environment for the *manual* manipulation of text-graphic objects (described briefly).

writtenPAM provides programmatic manipulation of visual objects. *Pattern processing* is demonstrated by functions which translate a LISP sexpr to the visual name-shape synonyms of the OUTLINE notation system, and then spatially lay it out. writtenPAM also permits definition of *pattern evaluating*, enabling actual *computation* with text-graphic forms. An *evaluate* function for text-graphic objects is given which can execute OUTLINE expressions.

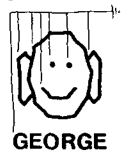
An experimental version of the PAM system has been implemented in MACLISP at the Stanford Artificial Intelligence Lab.

* now visiting Xerox Palo Alto Science Center, 3333 Coyote Hill Road, Palo Alto, CA 94304

Introduction

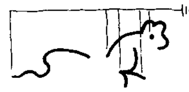
In order to set the stage for writtenPAM, the subject of this paper, a bit of background needs to be given on handPAM, the manual environment in which evaluation occurs. The PAM system as a whole was designed for the agile manipulation of text-graphic patterns -- *first* manually, and then, *later*, programmatically. The viewing screen for handPAM offers a static display (it doesn't scroll) with discretionary evaluation (objects first are created simply as visual entities, and then later submitted for evaluation). Objects on the display follow LISP structure -- either atomic or tree structured. This structure for text-graphic images was chosen because it most directly satisfied the needs of manual manipulation.

Initially concerned with manual manipulation, the user soon finds that the *selective* creation, moving and erasing of text-graphic objects necessitates some way of structuring complex objects and their parts; trees provide a simple and obvious way to structure visual objects, e.g.

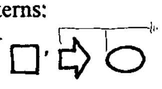
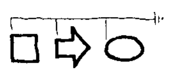


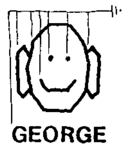
Complex objects with such a stree structure are called *text-graphic patterns.* handPAM is defined by a function called *handle*, and is actually a generalization of Warren Teitelman's LISP editor to text-graphic patterns on a static display (Teitelman credits Peter Deutsch for the original idea of a structure editor [6]). Thanks to the simplicity and power of LISP functions and structures, some of the ambiguities of text-graphic manipulators (Sutherland [1], Futrelle [2]) are clarified, in particular the one characterized by Sutherland as the *Structure of Drawings Problem.* handPAM is described in more detail in Lakin [7], [11]. Among the editor operations available on the 'current form' is *evaluation*, thus permitting writtenPAM forms to be created and evaluated from within the handPAM environment.

writtenPAM allows people to talk directly about text-graphic objects and manipulations on them, as for example in setting the value of variables: *x*←

 ,

constructing patterns:

patcons[ *]* =  ,

taking them apart: *first*  = GEORGE , and making equality tests: *equal?[x,* *]* = **FALSE**.

In short, writtenPAM's functions and predicates operate on visual objects and return them as values. As a matter of fact, this language is 'intrinsically graphic' (Futrelle's term [2]) and only later textual, but we still speak of writing programs, so we call it writtenPAM for convenience.

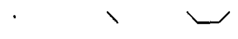
The remainder of this paper deals with writtenPAM. In particular are examined data types and structures (Figure 1), basic functions and prediates, examples of pattern processing, and examples of pattern evaluating.

A Structure from Manipulation for Text-Graphic Objects

A *text-graphic object* is either a line or a pattern.

A *line* is a drawline or a character or a textline.

A *drawline* is a line drawn through none or more locations.



A *character* is one or more drawlines.

A *textline* is one or more characters.

A *pattern* is a group of none or more lines and/or patterns.

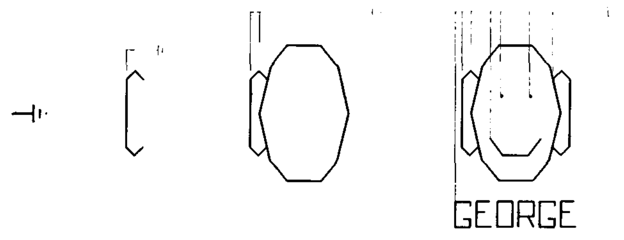
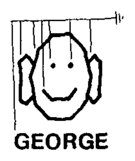
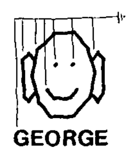
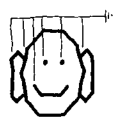


Figure 1

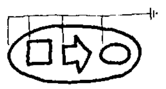
The Basic Functions and Predicates of writtenPAM

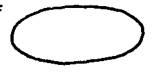
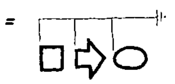
The functions *first* and *rest* operate on text-graphic patterns. *first* returns a copy of the first text-graphic object in a pattern, and *rest* returns a copy of the remainder of the pattern.

first  = **GEORGE**

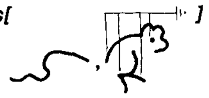
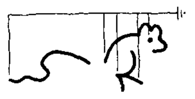
rest  = 

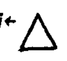
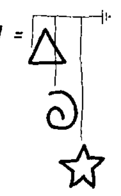
A word about the notation conventions used here. Text-graphic objects themselves are drawn with thick black lines, eg **GEORGE** and . Variables standing for text-graphic objects are shown in bold italic lowercase, et ***tg*** and ***obj***. Function names like *first*, *rest* and *patcons* and reserved text-graphic forms like ←, *[ ]* and are shown in thin italics.

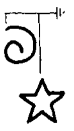
tg← 

first ***tg*** =  *rest* ***tg*** = 

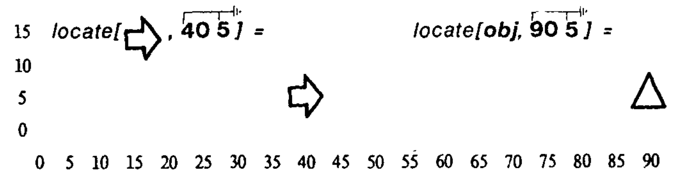
patcons is the *pattern constructor.* It takes two arguments, the second of which must be a pattern. *patcons* returns a new pattern constructed by putting its first argument onto the front of its second argument.

patcons[ *]* = 

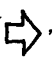
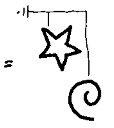
obj←  *patcons[****obj****,****pat****]* = 

pat← 

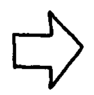
There there are the three spatial functions. *locate*, *rotate* and *scale*. *locate* takes two arguments, an object and a new location. It returns a copy of the object at the new location.



rotate also takes two arguments, an object and the number of degrees the object is to be rotated. It returns a copy of the object rotated clockwise through the angle specified.

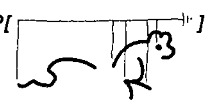
rotate[ *, 90]* = *rotate[****pat****,180]* = 

And *scale* takes two arguments, an object and the factor by which the object is to be scaled. It returns a copy of the object scaled by that factor.

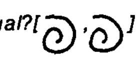
scale[*,2.0]* =  *scale[****obj****,0.5]* =

Finally there are the predicates. *line?* returns **TRUE** if its argument is a line (a drawline or a character or a textline) and **FALSE** otherwise.

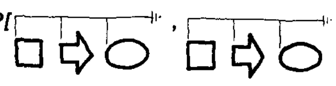
line?[*]* = **TRUE**   *line?[*GEORGE*]* = **TRUE**

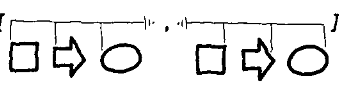
line?[*]* = **FALSE**

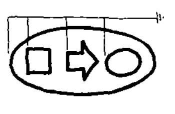
equal? is the general equality test for text-graphic objects. In order to be equal, two objects must appear the same visually *irrespective* of their location, orientation or size in relation to each other. And, if the two objects are characters, textlines, or patterns, then they must also have parallel tree structures. *equal?* will often be written as the infix =.

equal?[*]* = **TRUE**   *equal?[*B, *]* = **TRUE**

*equal?[*B, *]* = **FALSE**

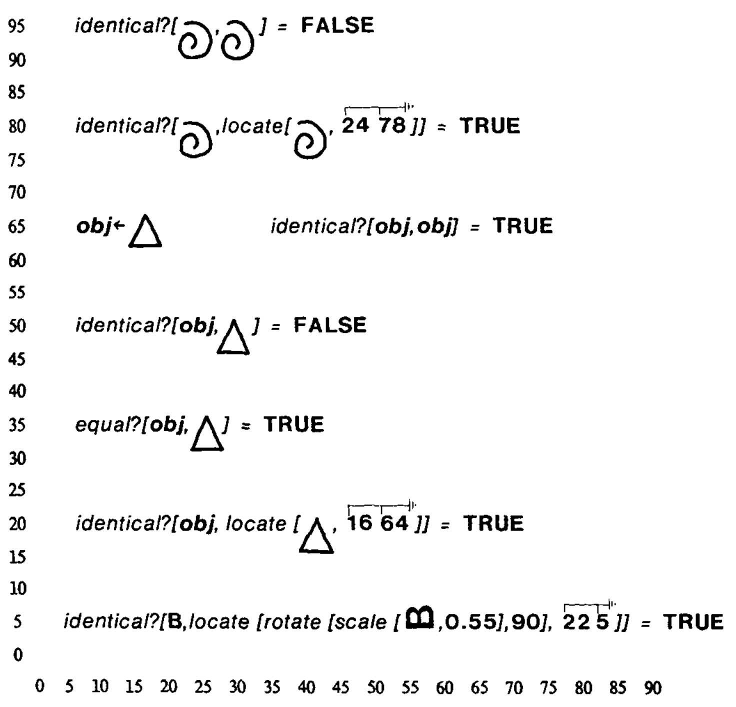
equal?[*]* = **TRUE**

equal?[*]* = **FALSE**

tg←    *equal?[first* **tg**, *]* = **FALSE**

equal?[first rest **tg**, *]* = **TRUE**

And *identical?* is the location, orientation and size *dependent* equality test. Identical objects not only appear the same and are built the same, they are also located, oriented and sized to each other exactly; two identical objects displayed simultaneously will be precisely on top of each other (all their lines will be coincident).



identical?[, *]* = **FALSE**

identical?[,*locate[* , 24 78*]]* = **TRUE**

obj←

*identical?[***obj**,**obj***]* = **TRUE**

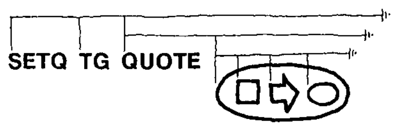
*identical?[***obj**, *]* = **FALSE**

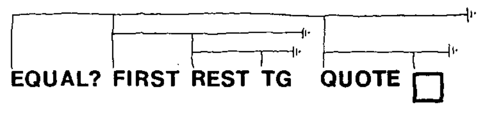
*equal?[***obj**, *]* = **TRUE**

*identical?[***obj**, *locate [* , 16 64*]]* = **TRUE**

*identical?[*B,*locate [rotate [scale [* ,0.55*]*,90*]*, 22 5*]]* = **TRUE**

To conclude this section, mention should be made of the other way PAM functions can be denoted visually. The notation used above for representing PAM functions on text-graphic objects is derived from McCarthy's 'blackboard notation' for LISP functions on symbolic expressions (McCarthy and Talcott [10]). In addition (again following McCarthy) there is also an alternative reflexive notation in which PAM function calls themselves are represented by text-graphic patterns, e.g.



and 

This more basic notation allows PAM function calls, as text-graphic patterns with explicit tree structure, to be conveniently manipulated either manually (by the human using handPAM) or programmatically (by other PAM functions). And, for practicality, the more basic notation was implemented first when it came time to actually evaluate visual objects on the screen, as can be seen from the illustrations in the next two sections.

Speaking of implementation, the relationship between LISP and writtenPAM in the current system should be made clear. All the primitive PAM functions are simply LISP functions which take LISP lists as arguments and return LISP lists which can then be displayed by the display processor *AddToScreen*. Thus the actual code is LISP, using the PAM names like *first* and *rest* so that *car* and *cdr* still work, and adding question marks for predicates so that the PAM *null?* is distinguised from *null*. For example, shown below is the code for *treverse*, which reverses the top level tree structure of a pattern (but not its spatial structure); and the code for *LISPtoTextTran*, which translates LISP sexprs into patterns having the same tree structure with a graphic text object for each textual LISP atom:

```
(define treverse (pat) (trev1 pat nilpat))

(define trev1 (pat result)
    (cond ((null? pat) result)
          (t (trev1 (rest pat) (patcons (first pat) result)) ) ) )


(define LISPtoTextTran (sexpr)
    (cond ((null sexpr) nilpat)
          ((atom sexpr) (MakeGraphicText sexpr))
          (t (patcons (LISPtoTextTran (car sexpr))
                      (LISPtoTextTran (cdr sexpr)) ) ) ) )
```

Here the global variable *nilpat* is bound to the empty visual pattern; and the function *MakeGraphicText* returns visual vector characters for LISP characters. These two points are important in that they have been overlooked below for the purpose of clarity. The blackboard notation has been written for an interpreter which supports visual literals like ⊣⊢ and **FIRST** instead of using *nilpat* and *MakeGraphicText['*FIRST*]* (thus being consistent with the examples at the beginning of this section).

Pattern Processing

The processing of text-graphic patterns is illustrated by starting with the LISP sexpr

```
(AND (NOT (NULL? Y))
     (OR (AND (EQUAL? X (FIRST Y)) Y)
         (MEMBER? X (REST Y)) ) ) .
```

Figure 2 shows the results of various PAM functions on this sexpr.

The first is a simple translation to graphic text objects. Since ***LISPtoTextTran*** performs no spatial layout, the result is a pile (object *a.* in Figure 2*)*.

Next, the function ***PrettyLayout*** is called on the pile to spread it out and ***ShowTree*** is used to diagram the tree structure (object *b.*).

Then ***LISPtoTextGraphicTran*** is called on the original LISP sexpr, translating any function names on the ***VizObjAlist*** into their visual name-shape synonym according to the OUTLINE notation system (thus addressing the problem of *Program Instrumentation* as detailed in Sutherland [1]). ***PrettyLayout*** has also been called on the resulting object to spread it out (object *c.*).

And finally (object *d.*), the function ***OUTLINELayout*** (the definition below is simplified) has been used to reformat object *c.* so that recursion is denoted by containment (this spatial layout function is one approach to Sutherland's *Logical Arangement Problem* in [1]).

```
LISPtoTextTran[sexpr]←
    if null sexpr then ⊣⊢
    else if atom sexpr then MakeGraphicText sexpr
    else patcons[LISPtoTextTran car sexpr,
                 LISPtoTextTran cdr sexpr]

PrettyLayout[obj]← if line? obj then obj else PL1[obj,obj]

PL1[obj,RegObj]←
    if null? obj then ⊣⊢
    else if line? first obj then
        |NextDone← LocateNextTo[first pat,RegObj];
        |return patcons[NextDone,PL1[rest pat,NextDone]];
    else
        |NextDone← PL1[first pat,move[RegObj, 0.0  -7.5 ]];
        |return patcons[NextDone,
                        PL1[rest pat,
                            move[NextDone, 4.0  7.5 ]]];
```

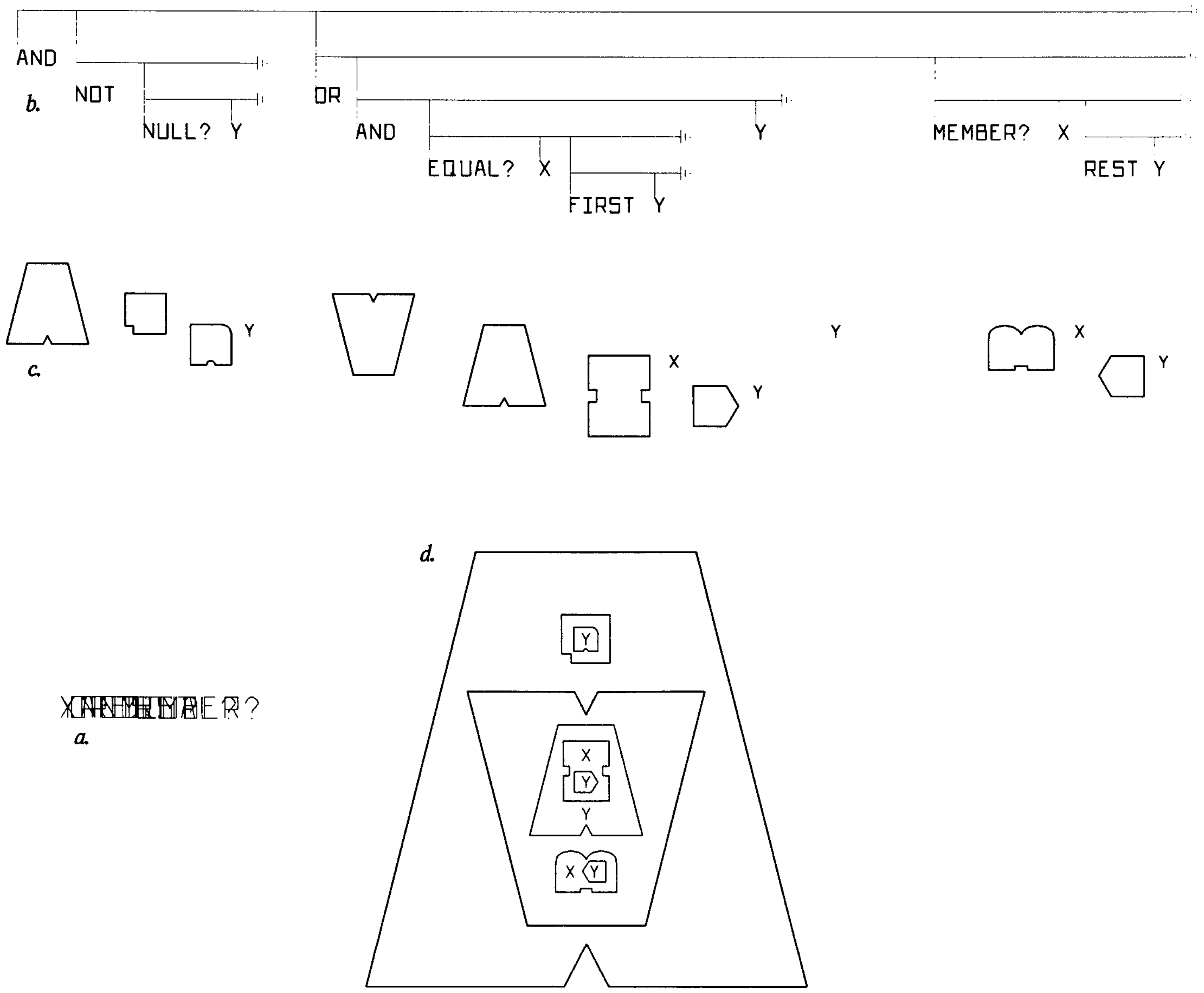


Figure 2

```
ShowTree[pat]← patcons[DrawCrossBar pat,
   mapfirstpat[
      [λ memb:
         if line? memb then
                 DrawVertBar[memb,pat,LONG]
         else patcons[DrawVertBar[memb,pat,SHORT],
                       ShowTree memb] ],
      pat]                                    ]

LISPtoTextGraphicTran[sexpr]←
   if null sexpr then —|I·
   else if atom sexpr then
          LISPtoGraphicAssoc[sexpr, VizObjAlist] or
                    MakeGraphicText sexpr
   else patcons[LISPtoTextGraphicTran car sexpr,
                LISPtoTextGraphicTran cdr sexpr]

OUTLINELayout[obj]← if line? obj then obj
                    else OL1[obj,getloc obj]

OL1[obj,FinLoc]←
   if line? obj then obj
   else |Argsdone←
        |  mapfirstpat[[λ arg: OL1[arg,getloc arg]],rest obj];
        |StackedArgs← locate[stack ArgsDone,FinLoc];
        |return patcons[contain[first obj,StackedArgs],
                    StackedArgs];

contain[container,containee]←
   locate[MaxSizeToObj[container,scale[containee,1.4]],
        getloc containee]
```

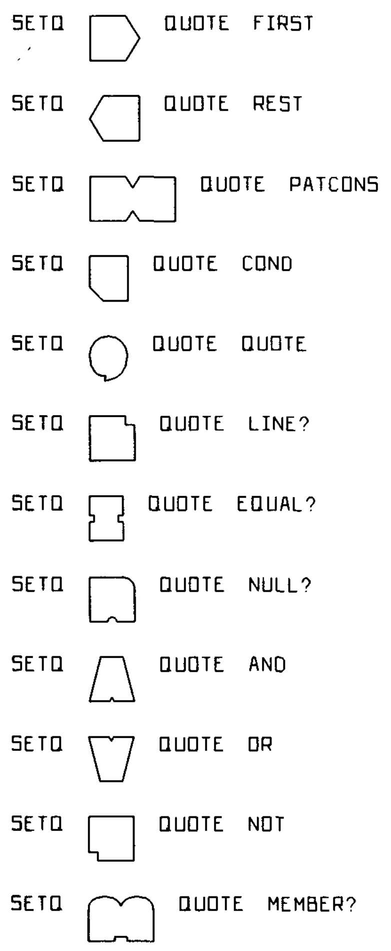


Figure 3

Pattern Evaluating

Computing with text-graphic forms occurs when text-graphic patterns are used to direct the manipulation of other text-graphic patterns. The form chosen to demonstrate this kind of computation is the OUTLINE expression created in the previous example. Before evaluation can proceed, the SETQ forms in Figure 3 need to be executed in order to pack the ***vizalist*** with the visual name-shapes that are the synonyms for each of the textual function names.

Next, within the programming environment portrayed in Figure 4, the form can be submitted for evaluation. First the form is made into the body of a lambda expression and bound to the OUTLINE nameshape for MEMBER? (since the form represents one defintion of the membership test, returning FALSE or some tail of the second argument). Then the other two assignment forms are evaluated in order to bind the left ear to the mouth and FOO to *george*. And finally, when the function call in the upper left is *evaluated*, the result is returned at the cursor in the upper right. Figure 5 shows the tree structure for the visual forms.

The function *evaluate* is defined below; the evaluation schema is simply a generalization of LISP's *eval* to handle text-graphic forms.

```
evaluate[form,vizalist]←
  if line? form then VizAssoc[second form,vizalist]
  else if line? first form then
    if equal?[first form, FIRST]
      then first evaluate[second form,vizalist]
    else if equal?[first form, REST]
      then rest evaluate[second form,vizalist]
    else if equal?[first form, PATCONS]
      then patcons[evaluate[second form,vizalist],
                   evaluate[third form,vizalist]]
    else if equal?[first form, EQUAL?]
      then if equal?[evaluate[second form,vizalist],
                     evaluate[third form,vizalist]]
           then TRUE else FALSE
    else if equal?[first form, QUOTE]
      then second form
    else if equal?[first form, COND]
      then VizCondProcess rest form
    else if equal?[first form, OR]
      then VizOrProcess rest form
    else if equal?[first form, AND]
      then VizAndProcess rest form
    else if equal?[first form, NOT]
      then if null? evaluate[second form,vizalist]
           then TRUE else FALSE
    else if equal?[first form, NULL?]
      then if null? evaluate[second form,vizalist]
           then TRUE else FALSE
    else if equal?[first form, LINE?]
      then if line? evaluate[second form,vizalist]
           then TRUE else FALSE
    else if equal?[first form, SETQ]
      then PAMset[second form,
                  evaluate[third form,vizalist]]
    else evaluate[patcons[VizAssoc[first form,vizalist],
                          rest form],
                  vizalist]

  else if equal?[first first form,λ]
     then evaluate[
              third first form,
              VizAppend[
                 pairup[mapfirstpat[[λ x: x],second first form],
                        mapfirstpat[[λ x: evaluate[x,vizalist]],
                                    rest form]],
                 vizalist]]
```

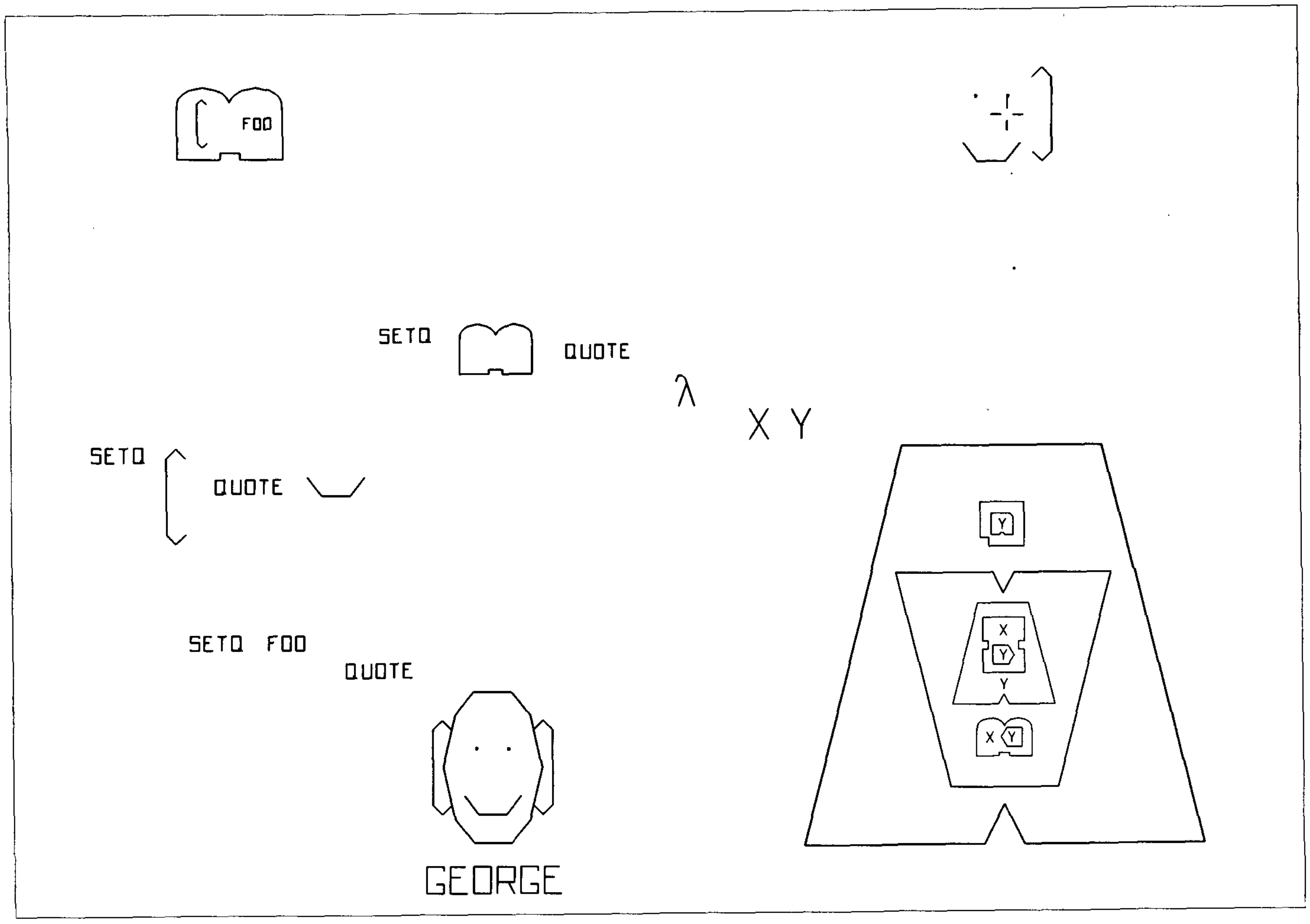


Figure 4

Notes on the Illustrations

Figures 1, 2, 3, 4 & 5 were constructed in the experimental handPAM environment at SAIL. They were printed double size on a Varian Statos and then photographically reduced. Figure 1 is reused with credit to the ACM SIGGRAPH '80 Proceedings.

Acknowledgments

Special thanks to the people without whom this project could not have happened. To John McCarthy, for inventing LISP and giving me access to the computer facilities at SAIL. To Jonathan King, Allan Hayes and Ben Laws for long discussions on the pre-computer PAM system. To Jim Collatz for actually learning handPAM before it was implemented. To Hans Moravec and Rodney Brooks for providing the basic graphics tools at SAIL; and to Dick Gabriel and Paul Martin for showing me the finer points of MACLISP.

Thanks also to the Systems Science Laboratory, Xerox Palo Alto Science Center, for use of their manuscript preparation facilities.

References

[0] McCarthy, John, "Recursive functions of symbolic expressions and their computation by machine," *Comm. ACM*, 1960, vol. 3, no. 4.

[1] Sutherland, Ivan E., "Computer Graphics: Ten Unsolved Problems", *Datamation*, pages 22-27, May 1966.

[2] Futrelle, R.P. and Barta, G., "Towards the Design of an Intrinsically Graphical Language", *SIGGRAPH '78 Proceedings*, pages 28-32, Aug 1978.

[3] Engelbart, D.C., "Augmenting Human Intellect: A Conceptual Framework," SRI International, Menlo Park, California, Oct 1962.

[4] English, W.K., Engelbart, D.C., and Berman, M.L., "Display-Selection Techniques for Text Manipulation", *IEEE Trans. on Human Factors in Electronics*, Vol. HFE-8, No. 1, March 1967.

[5] Sacerdoti, Earl. "Planning in a Hierarchy of Abstraction Spaces", *Adv. Papers 3rd Intl. Conf. on Artificial Intelligence*, Stanford University, August 1973.

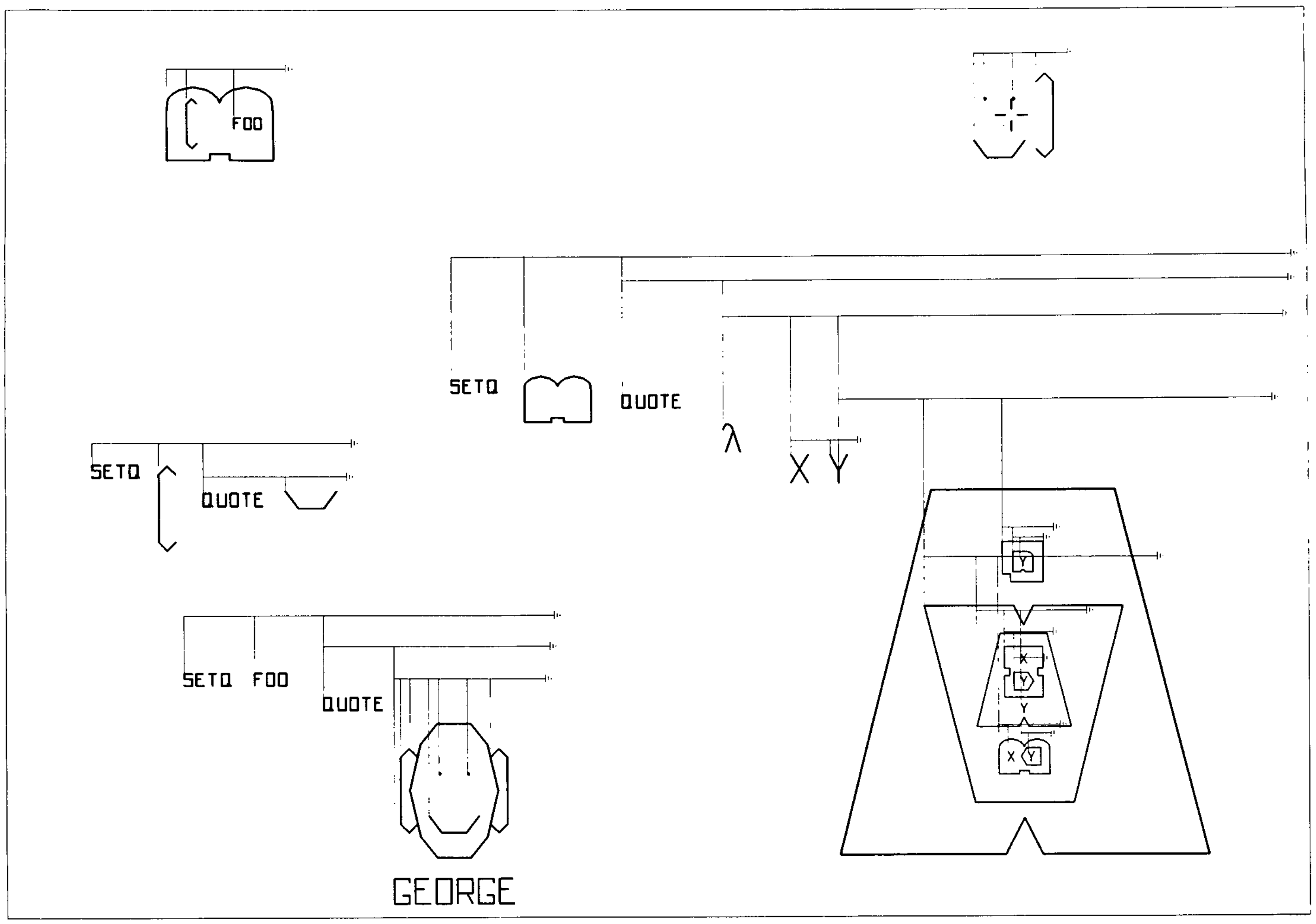


Figure 5

[6] Teitelman, Warren, *InterLISP Reference Manual*, Xerox Palo Alto Research Center, 1978.

[7] Lakin, Fred, "A Structure from Manipulation for Text-Graphic Objects", accepted for publication in SIGGRAPH '80, Seattle, Washington, August, 1980.

[8] Sutherland, Ivan E., "Sketchpad: A Man-Machine Graphical Communication System", *Proceedings—Spring Joint Computer Conference*, pages 329-346, 1963.

[9] Martin, Paul A., *DIP: A Program to Understand Diplomacy Dialogs*, PhD thesis, Stanford University, forthcoming.

[10] McCarthy, John and Talcott, Carolyn, *LISP Programming and Proving*, Class notes CS 206, Computer Science Dept., Stanford University, 1979.

[11] Lakin, Fred, "Indigenous Graphics for LISP", in preparation.