83-02

Logic programming graphics and infinite terms

Paul R. Eggert and Kam P. Chow

June 1983

Department of Computer Science, UCSB

Logic programming graphics and infinite terms

K. P. Chow
P. R. Eggert

ABSTRACT

Logic programming's applications in knowledge based systems will require heavy use of images for effective human interfaces, and future graphic work stations need efficient and easily usable methods of describing images. Logic programming methods provide a promising way for naive users to manipulate images. This promise is demonstrated, and a novel application of infinite logic terms is explored. These terms arise from the lack of an occur check in the fast unification algorithms of most logic programming systems. Infinite terms are not part of traditional logic but are useful in image processing, particularly in describing recursive images. An implementation is briefly described.

Computing Reviews Categories and Subject Descriptors: I.3.6 [**Computer Graphics**]: Methodology and Techniques—***interaction techniques***; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—***representations (procedural and rule-based)***.

General Terms: Human Factors, Languages.

Additional Key Words and Phrases: Logic Programming, Prolog, Infinite Objects.

1. Introduction

Logic programming systems, notably those based on Prolog [1, 11], have received special attention recently owing to their successes in several expert system applications and their selection by the Japanese as a cornerstone of the Fifth Generation Computer Project. Logic programming is easy to explain to users naive to computer programming, and thus promises to have wide application in systems dealing with human experts who are not necessarily versed in computer software. Although at first glance logic programs may look like a compromise between programs and specifications, they execute surprisingly quickly, sometimes faster than efficient compiled Lisp [12, 22]. Current applications are as diverse as architecture and interior design [9, 19], algorithm debugging [18], natural language recognition [24], and compilation [23]. Proposed uses include integration with large knowledge bases, VLSI and robotics design and testing, and speech and image processing [20, 25]. Prolog itself is easy to modify to handle specialized problems [3]. It seems inevitable that logic programming systems will be an important implementation tool in building new knowledge based computing systems.

Logic programming terms have been traditionally displayed in a textual format, but the success of systems such as Smalltalk and Logo show that even good ones can be greatly improved with graphical interaction. Furthermore, several of the potential applications listed above require display of images. The newly developed interactive graphic logic programming system GLOG is based on Edinburgh CProlog [15], and was designed to run on a personal work station with a high resolution bit map display. The design emphasized a simple user interface that is oriented towards logic programming. The desire for simplicity caused the design's primitives to resemble, say, PIC's simple ones [6] instead of IDEAL's more general, complicated ones [21]. GLOG's important contribution lies not in its primitives, but in the way a user can put them together with logic programming methods.

Authors' address: Department of Computer Science, University of California, Santa Barbara, CA 93106.

While developing GLOG, we discovered that repetitive and infinite images are surprisingly easy to specify with certain graphical output primitives combined with Prolog's definition of unification. The following sections introduce logic programming notions, propose a scheme for displaying logical terms, and illustrate the interplay between logic programming and graphics particularly with infinite images.

2. Logic programming, unification and infinite terms

A *logic program* is a set of *procedures* each consisting of a series of Horn *clauses*[5]. Clauses are either *axioms*, such as the three axioms

```
father(zeus, athena).
father(zeus, aphrodite).
mother(aphrodite, eros).
```

or inference *rules*, such as the rule

```
parent(P,C) :- father(P,C);
               mother(P,C).
```

This rule may be read, "For all C and P, P is C's parent if P is C's father or P is C's mother." A rule thus has two parts: a *head* (before the ":-") and a *body;* an axiom is a head with no body. Axioms may be general, as in

```
X = X.
```

which may be read "For all X, X is equal to X" and defines the equality procedure. Rules may be recursive; for example,

```
ancestor(A,C) :- parent(A,C).
ancestor(A,C) :- parent(A,B),
                 ancestor(B,C).
```

defines "ancestor" recursively using "parent".

Logic programs deal with and consist of logical *terms*. A term is either a logical *variable*, or consists of an atomic *functor* and zero or more terms as *arguments*. An *atom* is a functor with zero arguments. Logical variables are capitalized and functors are in lower case, so that X is a variable and f, f(X) and f(a,X) contain zero, one and two arguments. Functors may appear in infix, prefix and suffix notation, so the terms –X+Y! and +(–(X),!(Y)) are equivalent; parenthesization overrides operator precedence in the usual way. By convention,

the term	denotes
A, B	A AND B,
A; B	A OR B,
A :– B	B IMPLIES A,
[]	the empty list,
[X|Y]	the list with first element X and tail Y, and
[A, B, C]	the list [A|[B|[C|[]]]].

[X|Y] roughly corresponds to the Lisp notation (cons X Y). For example, the following procedure concat(A,B,AB) defines the relation between two lists A, B and their concatenation AB.

```
concat([], B, B).
concat([X|A], B, [X|AB]) :-
   concat(A,B,AB).
```

These two clauses can be read, "concatenating [] to B yields B," and "concatenating [X|A] to B yields [X|AB] if concatenating A to B yields AB."

A logic program is executed by asking the system a question, or *call*, such as

```
:- concat(X, Y, [a,b]).
```

A logic programming interpreter attempts to find a consistent substitution of terms for variables so that the call logically follows from the axioms and rules. Prolog interpreters attempt to find the first clause with a head matching the call. If there is no match the call *fails;* if the match is an axiom, the call *succeeds;* if the match is a rule, its body is called in turn. Prolog interpreters use depth first search and backtrack to later clauses on failure, so that the above call can succeed three times, once for X equal to [], [a] and [a,b], respectively. More about this process, and about other logic programming issues, may be found in Kowalski's introduction [7].

Unification is the technical term for the word "matching" used above. A most general unification of two terms results from a minimal substitution of terms for variables to make the two terms equal. For example, a unification of f(X,g(Y)) and f(a,Z) is the term f(a,g(Y)) with the implied substitution of a for X and g(Y) for Z. Unification is performed often in theorem proving programs and several algorithms have been designed to handle it in linear time [10, 13].

Unfortunately, even the fastest of these algorithms must do an "occur check" to avoid unifying a variable with a term containing the variable. For speed, most logic programming systems lack the occur check, and thus can unify terms like X and f(X), even though no finite term can be substituted for X to make X and f(X) equal. Avoiding the occur check leads to significant performance improvements. For example, the first step of concatenating an n-element list [a_1, . . . , a_n] is unifying the list with the term [X|AB]; this requires $O(n)$ time with a full unification algorithm because each of a_2, . . . , a_n must be checked to be not equal to AB. Without the occur check, this example takes only $O(1)$ time.

If infinite terms are allowed, then X and f(g,X) are unifiable with X=f(g,f(g,f(g,...))). This term, though infinite, is *rational*, that is, it has a finite number of subterms, namely itself and g. Rational terms can be represented in finite memory by using circularly linked pointers to represent the repetition. Colmerauer has developed a theory of infinite trees and has shown applications in context free grammars and finite state automata [2]. However, most Prolog applications never attempt to generate rational terms, another reason Prolog interpreters lack occur checks.

GLOG uses unification as a fundamental way of dealing with images and patterns. To match an image to a pattern, one merely unifies the two; the attempt to match will succeed or fail according to the usual Prolog rules. For example, the image (box beside triangle) matches the pattern (A beside X) by substituting box for A and triangle for X. Using this primitive, it is still a research issue to attempt image analysis; the work reported here instead concentrates on image synthesis and processing. The next section describes GLOG's scheme for display of logical terms; section 4 shows how this scheme handles infinite rational terms.

3. Displaying logical terms

Traditional logic programming systems display terms only in textual format. Also, the SeeLog system [14] has added the capabilities of the Graphics Kernel System [17] to Prolog: GKS's primitives are low level graphics output primitives that resemble the ACM Core standard. We propose the following new higher level scheme for displaying logical terms. It is related to Henderson's scheme for functional graphic programming [4], but handles pictures with relation as well as functions. Instead of emphasizing primitive graphical output commands such as "draw a line from (X_0,Y_0) to (X_1,Y_1)" and writing programs to execute such commands in proper order to produce a picture, a GLOG user deals with logical terms, and uses built in graphic primitives to describe the relation between the terms. Thus a user, instead of commanding the display of a picture step by step, describes the general layout of the picture. This set of primitives is particularly useful for top down structured design of graphic programs. And unlike functional programming, relational programming lets the user describe an image in any desired order, top down, bottom up, or a mixture of the two.

Logical terms are used to describe images. The built in procedure plot(Image) displays the term defined by the term Image. Most of the effort in a GLOG program will be to create an image, which can then be displayed.

Images are defined within a unit square. The term (X,Y) stands for the point with the given coordinates in the unit square; for example, (0, 1/2) stands for the middle of the square's left side. Here "," is used as an infix operator, so that (X,Y) is equivalent to ,(X,Y). This use of the comma and its use for logical AND are distinguished from context.

The term P1−P2 represents the line segment from P1 to P2 if P1 and P2 are points. The "−" operator may be nested, so that P1−P2−P3−P4 represents the union of P1−P2, P2−P3 and P3−P4.

For example,

```
Triangle =
    (0, 1/2)−(1, 1)−(1/2, 0)−(0, 1/2),
plot(Triangle).
```

produces the image in Fig. 1. In later examples the line "plot(Image)" will be omitted though it is currently necessary in GLOG.

The empty list [] represents the empty picture, that is, the all white unit square. A list of pictures [P_1,P_2,...,P_k] represents the *k* pictures overlaid on each other. A happy result is that [A] and (A) both mean just A. More formally, the cons term [X|Y] represents X and Y overlaid; hence [A,B], which is the same as [A|[B|[]]], represents the overlay of A, B and the empty picture. For example, two overlaid sequences of line segments form the basic pattern to build Fig. 2.

```
Left_fish =
[ (1,1)-(1/8,3/5)-(1,1/8)
        -(3/4,0)-(1,0),
  (1/8,3/5)-(1,4/5)
]
```

The term (P flipped) is the reflection of P in the vertical axis that bisects P. For example, Fig. 3 is defined by

```
Right_fish = Left_fish flipped
```

The term (A beside B) is a unit square containing a copy of A with horizontal dimensions halved in its left half and a similar copy of B in its right. For example, Fig. 4 is produced by

```
Up_fish =
    Left_fish beside Right_fish
```

The term (A beside B ratio R) permits an arbitrary ratio R of A's horizontal size to B's. Using this and the empty picture [] one can introduce white space as in

```
Thin_fish =
         ([] beside Up_fish ratio 1/2)
beside [] ratio 3
```

(Fig. 5). The terms (A above B) and (A above B ratio R) are similar to those using beside.

The term (P rotated by D) represents the image P rotated anticlockwise by D degrees; (P rotated) is the same as (P rotated by 90). For example, Fig. 6 is produced by

```
Side_fish = Up_fish rotated
```

Rotations by angles that are not multiples of 90° may cause the image to violate the boundaries of the unit square; this is permitted but is not always good practice.

Other moving, scaling and titling primitives are available in GLOG, but these primitives suffice for later examples. The primitives are summarized in Table 1.

Even just these primitives are surprisingly suitable for creating images. Complex pictures can be defined clearly and compactly. Prolog creates and unifies terms efficiently; only the single plotting procedure is coded in the lower level language C. Furthermore, logic programming permits the picture to be constructed in whatever order seems best to the user. Many programs are just a series of equations that may appear in any order and have a natural interpretation in logic with infinite terms.

Before describing how infinite terms are used, an example with just finite terms seems in order.

```
matrix(LL,Mat) :- col(LL,M,Mat).

col([L], 1, P) :- row(L,N,P).
col([L|LL], M+1, P above PP ratio 1/M) :-
    row(L,N,P), col(LL,M,PP).

row([X], 1, X).
row([X|L], N+1, X beside P ratio 1/N) :-
    row(L,N,P).
```

The procedure row(L,N,P) succeeds if P is a row of the N items in the list L stacked beside each other; col(LL,M,PP) is similar for columns. For example, Fig. 7 is created by

```
U = Up_fish,    D = L rotated,
L = Side_fish,  R = D rotated,
matrix([[D,L,L],
        [D,[],U],
        [R,R,U]], Fish_cycle).
```

4. Using infinite terms

Infinite rational terms can be displayed by exhibiting the underlying directed graph implied by the terms. Waterloo Prolog [16] version 1.3, for example, displays the result of unifying X and f(a,X) as f(a,##1##). This method suits text better than images.

GLOG attempts to display the entire contents of an infinite term, given the resolution of the output device. Thus an external constraint prevents looping while attempting to display the term. In all the representation primitives given, subimages are smaller than their containing images, so the process must terminate. Some movement and scaling primitives not described here lack this property; their use is discouraged, but has not led to looping problems in practice. For greater speed, the user can tell GLOG to pretend that output resolution is coarser than the physical device's: the user defined procedure resolution(R) causes display to stop if the ratio of the current subimage's size to the original image's is less than 1/R in either the X- or the Y-dimension. Large Rs slow plotting but improve image detail.

An example of a recursively defined image is

```
Food_chain = P above P,
P = Food_chain beside Side_fish
```

(Fig. 8). A Food_chain contains four quadrants: the left quadrants are Food_chains, and the right quadrants are Side_fishes. The above definition is equivalent to

```
Food_chain =
        (Food_chain beside Side_fish)
  above (Food_chain beside Side_fish)
```

which is an equation of the form X=f(X). Food_chain is a fixed point of this functional equation.

Procedures may be used to define pictures containing the same pattern but different basic pictures. The calls htree(line,Htree_line) and htree(box,Htree_box) yield Figs. 9 and 10.

```
htree(Type,H) :-
    element(Type,Width,P),
    H = [P, S flipped beside S],
    S = [] beside H rotated
        ratio Width.

element(line, 0,
    (1/2,1/2)-(1/2,1) ).
element(box, 1/4,
    [ (2/5,0)-(2/5,9/10)
     -(3/5,9/10)-(3/5,0)-(2/5,0),
     (1/2,9/10)-(1/2,1) ]).
```

Escher's "Square Limit" (Fig. 19) is the source of the following extended example of the power of logic programming graphics using infinite terms [8]. This image was done in a functional style by Henderson [4]. Henderson's approach required specification of the image to a given depth; the following approach has the advantage that the image is specified to be infinite, and the amount of displayed detail depends only on the resolution of the output device. Henderson built the picture out of just four basic images. We use a different decomposition but start with the same basic images P, Q, R, S, which are assumed to satisfy the procedures p(P), q(Q), r(R), s(S) (Figs. 11-14). The rest of this section describes how "Square Limit" is constructed.

Two basic patterns, T and U (Figs. 15, 16), are formed from P, Q, R and S by using the following procedures

```
quartet(P,Q,R,S,
    (P beside Q) above (R beside S)).
cycle(P,U) :-
    PR = P rotated,
    PRR = PR rotated,
    PRRR = PRR rotated,
    quartet(P,PRRR,PR,PRR,U).
```

together with the calls

```
quartet(P,Q,R,S,T),
cycle(Q rotated, U) .
```

"Square Limit" has nine equal square parts, which are labeled by Ce (for Center) and the compass points No, Ea, So, We, NE, NW, SE and SW. These parts have the following relationships.

```
No = Ea rotated,   NW = NE rotated,
We = No rotated,   SW = NW rotated,
So = We rotated,   SE = SW rotated
```

Thus only three parts are different: Ce, Ea and NE. But Ce = U, and Ea and NE can be defined by

```
RT = T rotated rotated rotated,
NE = (No above U) beside (NE above Ea),
Ea = (T above RT) beside (Ea above Ea)
```

(Figs. 17, 18). By now the alert reader should be adept at following the picture's decomposition.

In summary, "Square Limit" can be defined as follows, where the procedures cycle, matrix and quartet were defined earlier.

```
square_limit(Square_limit) :-
p(P), q(Q), r(R), s(S),
quartet(P,Q,R,S,T),
cycle(Q rotated, U),
matrix([[NE,No,NW],
        [We,Ce,Ea],
        [SW,So,SE]], Square_limit),
No = Ea rotated,  NW = NE rotated,
We = No rotated,  SW = NW rotated,
So = We rotated,  SE = SW rotated,
Ce = U,
RT = T rotated rotated rotated,
Ea = (T above RT) beside (Ea above Ea),
NE = (No above U) beside (NE above Ea).
```

5. Conclusion

The important thing about these examples is not the images themselves, but the method used to describe and generate them. Logic programs can specify images cleanly and concisely; they are suited for naive users, and they seem to fill a certain procedural gap in current picture specification languages. Because pattern matching and backtracking are an integral part of logic programming systems, they also seem suited for the other part of image processing: image analysis. Any logic programming system that could both analyze and synthesize images would be a powerful tool indeed for building future knowledge based systems. The work reported here concentrated on the easy half, synthesis, and work is already under way to extend its framework to image analysis.

6. References

1. CLOCKSIN, W. F. AND MELLISH, C. S. *Programming in Prolog*. Springer-Verlag, New York (1981).
2. COLMERAUER, A. Prolog and infinite trees. *Proc. logic programming workshop*, Long Beach, CA (1981).
3. EGGERT, P. R. AND SCHORRE, D. V. Logic enhancement: a method for extending logic programming languages. *Proc. 1982 symp. on LISP and functional programming*, Pittsburgh, PA (August 1982), 74-80.
4. HENDERSON, P. Functional geometry. *Proc. 1982 ACM symposium on LISP and functional programming*, Pittsburgh, PA (August 1982), 179-187.
5. HORN, A. On sentences which are true of direct unions of algebras. *J. symbolic logic* **16** (1951), 14-21.
6. KERNIGHAN, B. W. PIC — a crude graphics language for typesetting. *SIGPLAN notices* **16**, 6 (June 1981), 92-98.
7. KOWALSKI, R. *Logic for problem solving*. Elsevier North Holland, New York (1979).
8. LOCHER, J. L. (ED.) *The world of M. C. Escher*. Harry N. Abrams, Inc., New York (1971).
9. MARKUSZ, Z. Design in logic. *Computer-aided design* **14**, 6 (November 1982), 335-343.
10. MARTELLI, A. AND MONTANARI, U. An efficient unification algorithm. *ACM trans programming languages & systems* **4**, 2 (April 1982), 258-282.
11. MCDERMOTT, D. The PROLOG phenomenon. *SIGART newsletter*, 72 (1979), 16-20.
12. O'KEEFE, R. A. Prolog compared with Lisp?. *SIGPLAN notices* **18**, 5 (May 1983), 46-56.
13. PATERSON, M. S. AND WEGMAN, M. N. Linear unification. *J. of computer and system sciences* **16** (1978), 158-167.
14. PEREIRA, F. SeeLog — a Prolog graphics interface. Report 83/04, EdCAAD, Dept. of Architecture, U. of Edinburgh (August 1982).
15. PEREIRA, F. (ED.) CProlog user's manual. Report 82/11, EdCAAD, Dept. of Architecture, U. of Edinburgh (September 1982), Version 1.1.

16. ROBERTS, G. M. *An implementation of PROLOG.* U. of Waterloo (1977), Master's thesis.

17. SCHLECHTENDAHL, E. G. Basic graphics for data representation. pp 329-344 in *Computer aided design modelling, systems engineering, CAD-systems* , ed J. Encarnacao, Springer-Verlag, New York (September 1980).

18. SHAPIRO, E. Y. Algorithmic program debugging. Research Report 237, Yale U., Dept. of Computer Science (May 1982).

19. SWINSON, P. S. G. Logic programming: a computing tool for the architect of the future. *Computer-aided design* **14**, 2 (March 1982), 97-104.

20. TRELEAVEN, P. AND LIMA, I. Japan's fifth-generation computer systems. *Computer* **15**, 8 (August 1982), 79-88.

21. VAN WYK, C. J. A high-level language for specifying pictures. *ACM trans. on graphics* **1**, 2 (April 1982), 163-182.

22. WARREN, D. H. D., PEREIRA, L. M., AND PEREIRA, F. PROLOG -- the language and its implementation compared with LISP. *SIGPLAN notices* **12**, 8 (August 1977), 109-115, also *SIGART Newsletter*, 64.

23. WARREN, D. H. D. Logic programming and compiler writing. *Software—practice & experience* **10**, 2 (February 1980), 97-125.

24. WARREN, D. H. D. AND PEREIRA, F. C. N. An efficient easily adaptable system for interpreting natural language queries. Research Paper 155, Dept. of Artificial Intelligence, U. of Edinburgh (1981).

25. WARREN, D. H. D. A view of the fifth generation and its impact. *AI magazine* (Fall 1982), 34-39.

7. Appendix: implementation details

GLOG is written in C and runs under Berkeley Unix. It can run on a VAX or a Sun work station, and can generate device independent output suitable for several different plotting devices.

Two procedures control the output besides the procedures and operators defined in Table 1. The user defined procedure term(T) specifies the output terminal type T: term(sun) causes the output to be suitable for a Sun work station. Other terminal types, for example, 4014 (Tektronix 4014), ver (Versatec printer/plotter), direct output to a file F defined by the procedure outfile(F). The graph can be obtained by the standard Unix plotting program *plot*(1).

Two low level output terms are not described in Table 1. The term move(X,Y,P) yields a copy of P, with (X,Y) added to all P's coordinates. The term scale(X,Y,P) similarly multiplies P's coordinates by (X,Y). These terms may generate a picture that exceeds the unit square boundaries, but no checking or clipping is performed.

GLOG contains two independent plotting subsystems. One is specially for the Sun work station display, the other is for everything else. Each subsystem is about 600 lines of C code. A small interpreter of thirty lines of Prolog handles terminal types, resolutions and output control.

GLOG's performance is satisfactory. Fig. 10 needed only seven seconds of CPU time on a VAX-11/780 at resolution(80); Fig. 19 needed a minute at resolution(120). Increasing resolution increases the cost. GLOG requires about 12K extra bytes of memory over CProlog, including the extra program text for both plotting subsystems. Little stack space is needed at run time by Prolog standards, because GLOG does not create extra terms on the Prolog user stack. Instead, it recursively descends the infinite tree and stops at the resolution limit.

Table 1. GLOG output primitives.

procedure	*interpretation*
A = B	Images A and B unify
plot(A)	Plot the image A
resolution(R)	Ignore subimages smaller than 1/R
term	*represents*
(X,Y)	Point with given X- and Y-coordinates
$P_1 - P_2$	Line segment from P_1 to P_2
$P_1 - \cdots - P_n$	Connected line segments from P_1 to P_n
[]	the empty picture
[A|B]	A and B overlaid
$[A_1, \ldots, A_n]$	$A_1, \ldots, A_n$ overlaid
A above B ratio R	A above B; R is size ratio of A to B
A above B	A above B ratio 1
A beside B ratio R	A beside B; R is size ratio of A to B
A beside B	A beside B ratio 1
A rotated by D	A rotated anticlockwise by D degrees
A rotated	A rotated by 90

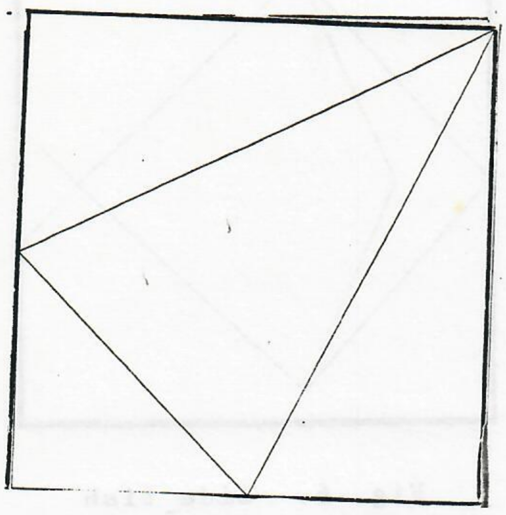

Fig. 1. Triangle

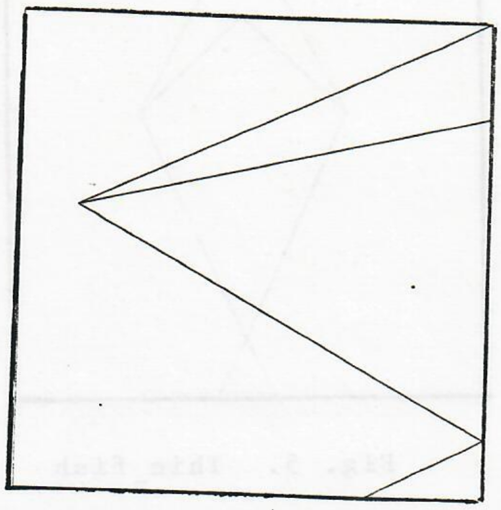

Fig. 2. Left_fish

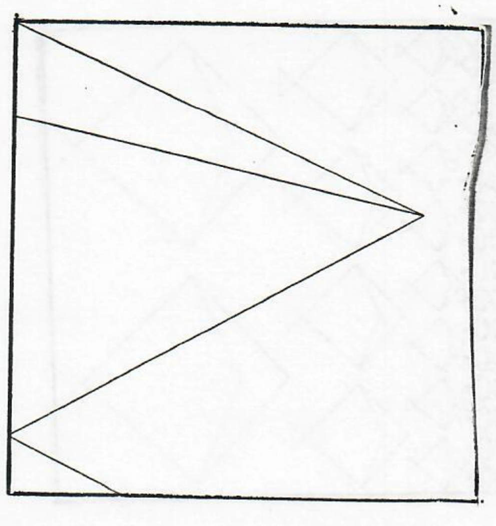

Fig. 3. Right_fish

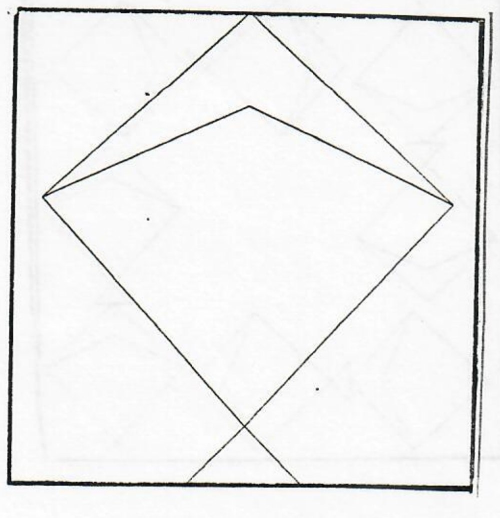

Fig. 4. Up_fish

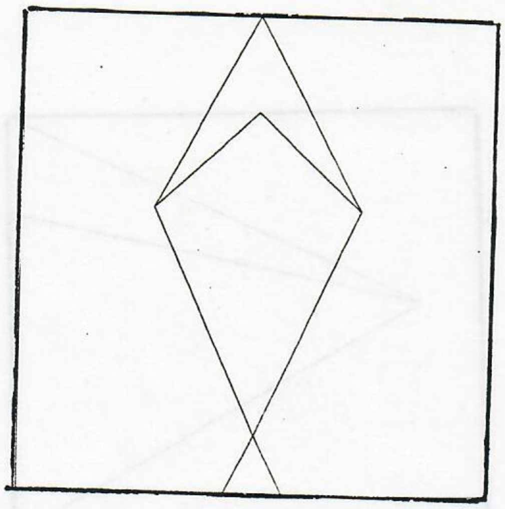
Fig. 5. Thin_fish

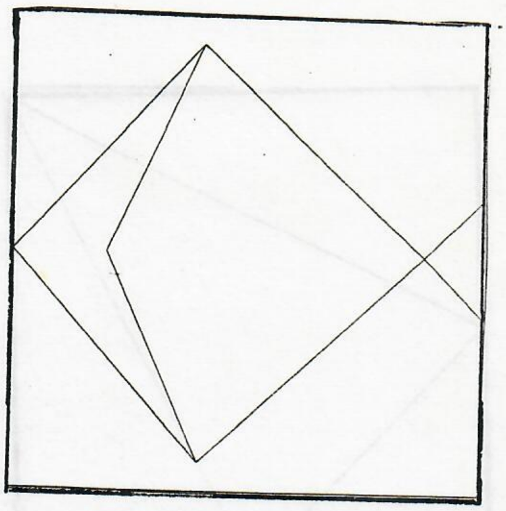
Fig. 6. Side_fish

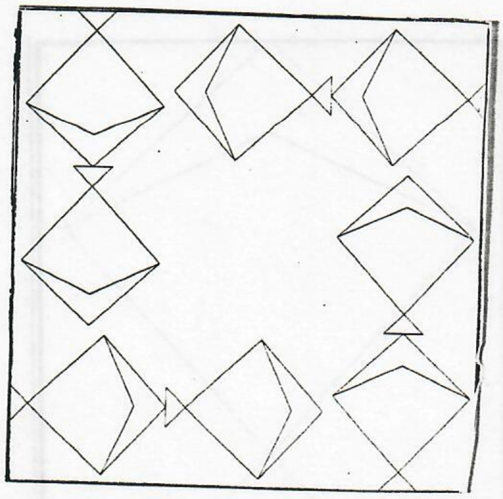
Fig. 7. Fish_cycle

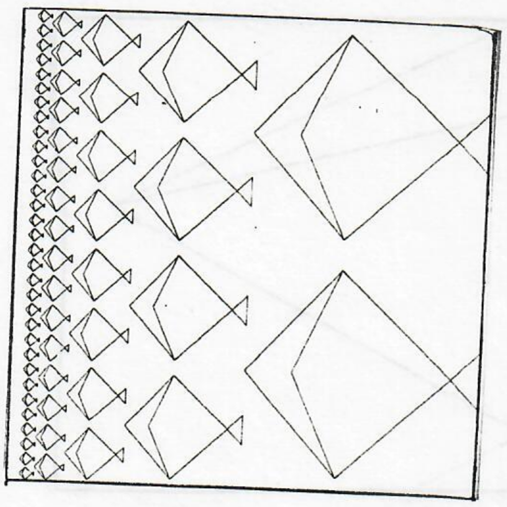
Fig. 8. Food_chain

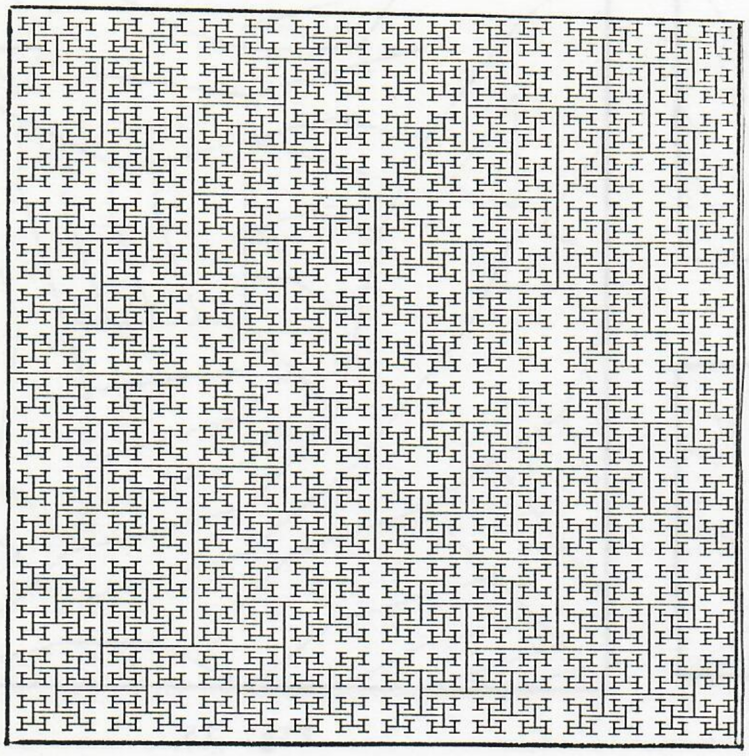

Fig. 9.
Htree_line

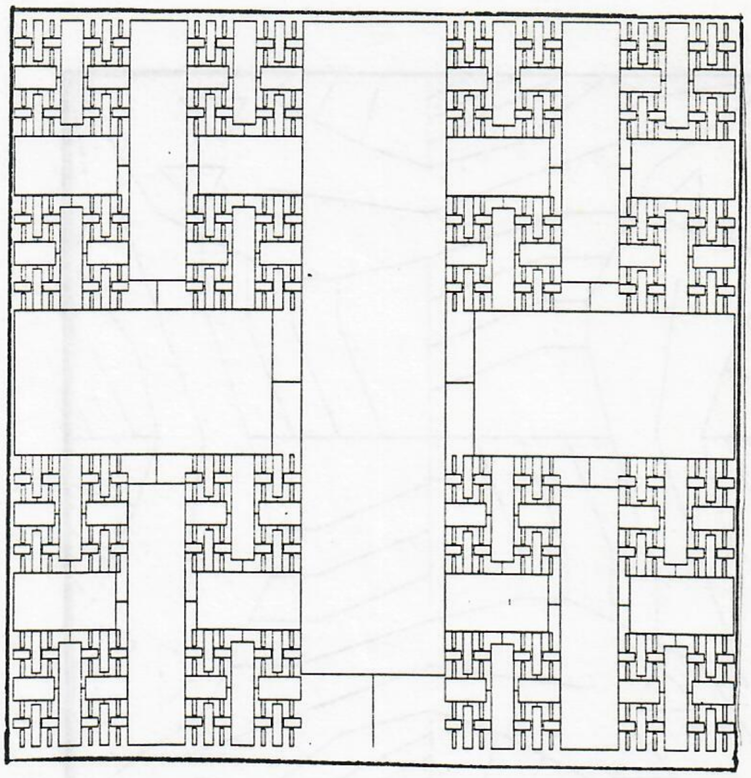

Fig. 10.
Htree_box

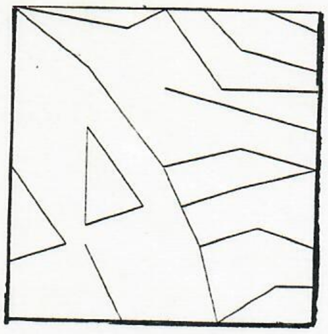
Fig. 11. P

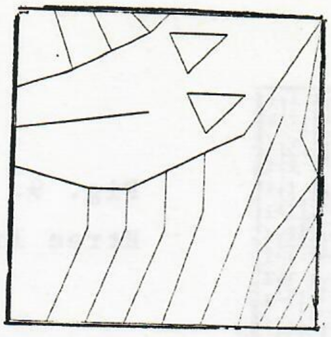
Fig. 12. Q

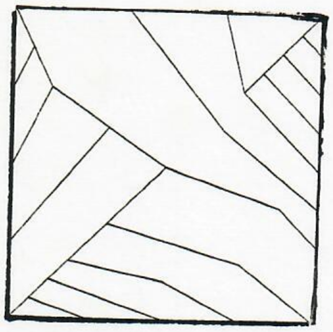
Fig. 13. R

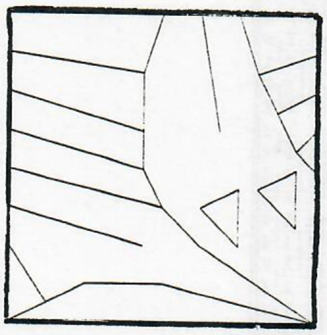
Fig. 14. S

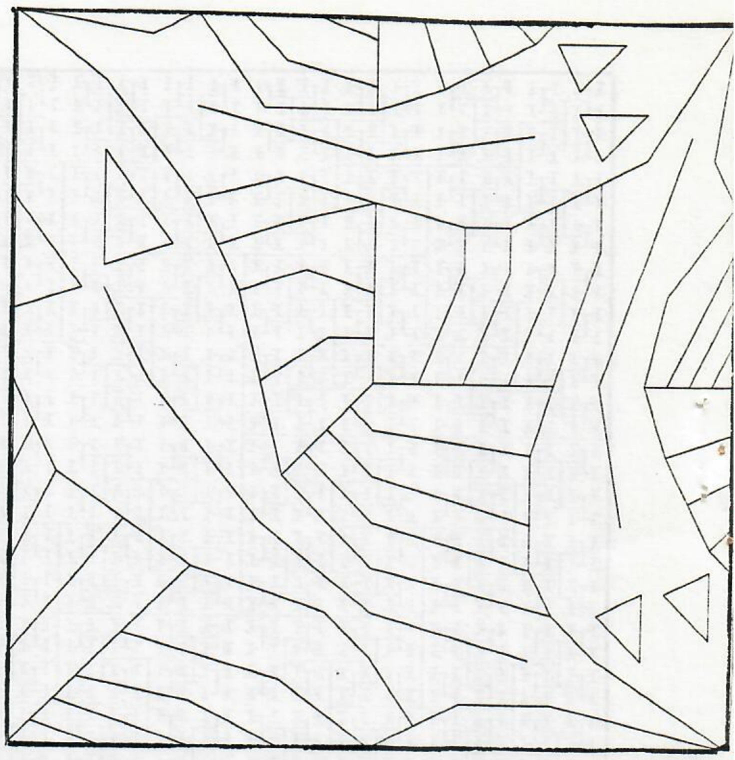
Fig. 15.

T = PQ
    RS

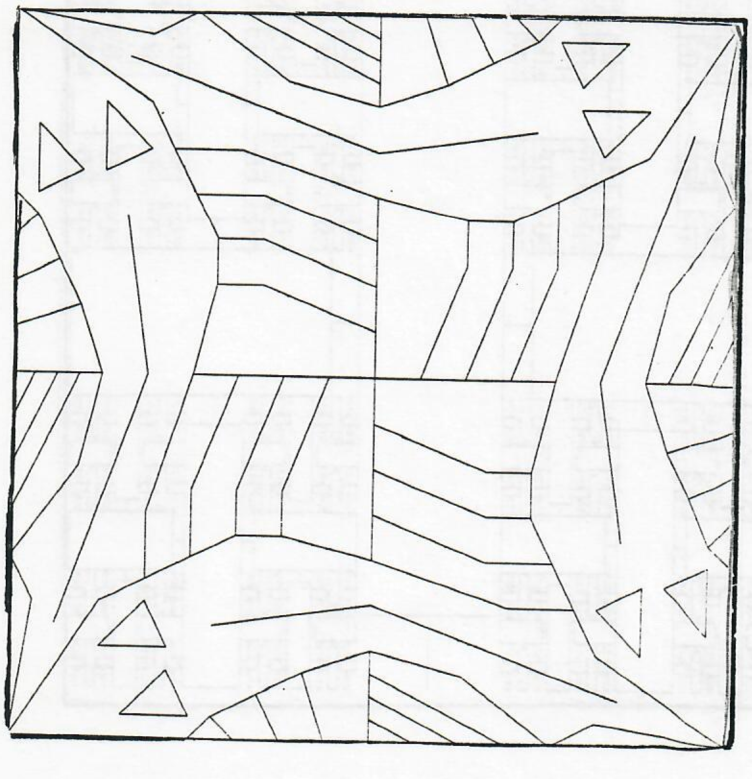
Fig. 16.

U = ϭ Q
    ∂ ø

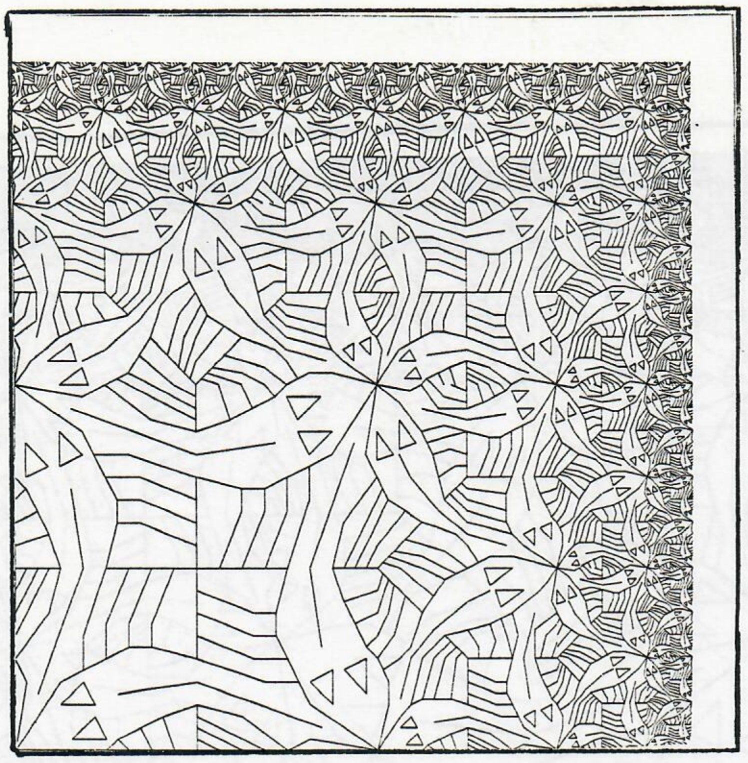

Fig. 17.

$$\mathrm{NE} = \begin{array}{ll} \mathrm{Ea} & \mathrm{NE} \\ \mathrm{U} & \mathrm{Ea} \end{array}$$

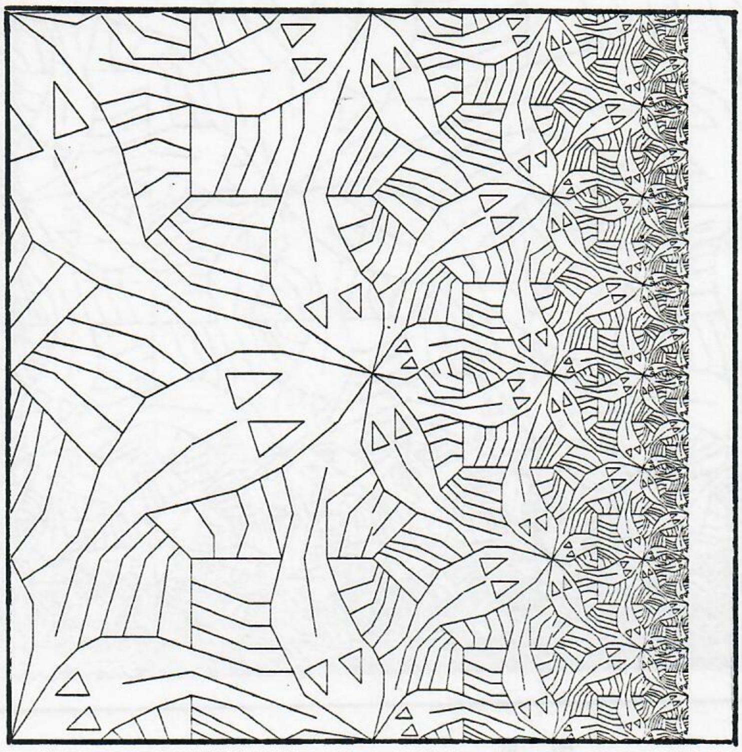

Fig. 18.

$$\mathrm{Ea} = \begin{array}{ll} \mathrm{T} & \mathrm{Ea} \\ \dashv & \mathrm{Ea} \end{array}$$

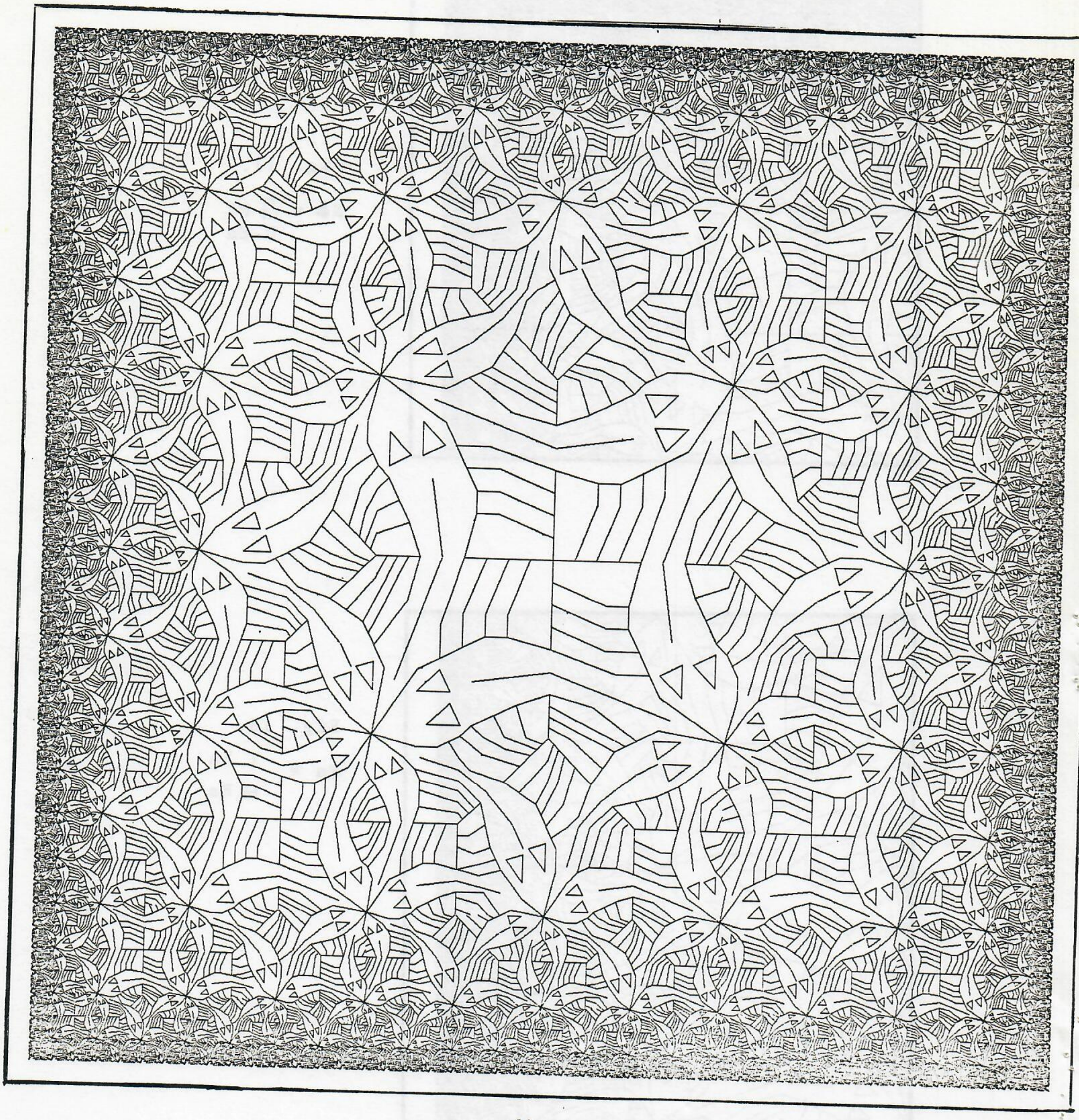

Fig. 19. Square_limit = $\begin{array}{ccc} \mathrm{NE} & \mathrm{Ea} & \mathrm{NE} \\ \mathrm{Ea} & \mathrm{U} & \mathrm{Ea} \\ \mathrm{NE} & \mathrm{Ea} & \mathrm{NE} \end{array}$