Iterated Polynomial Functions in Finite Arithmetic

Experimental Mathematics

by
Erik Winfree
and
Stephanie Wukovitz

The Subject:

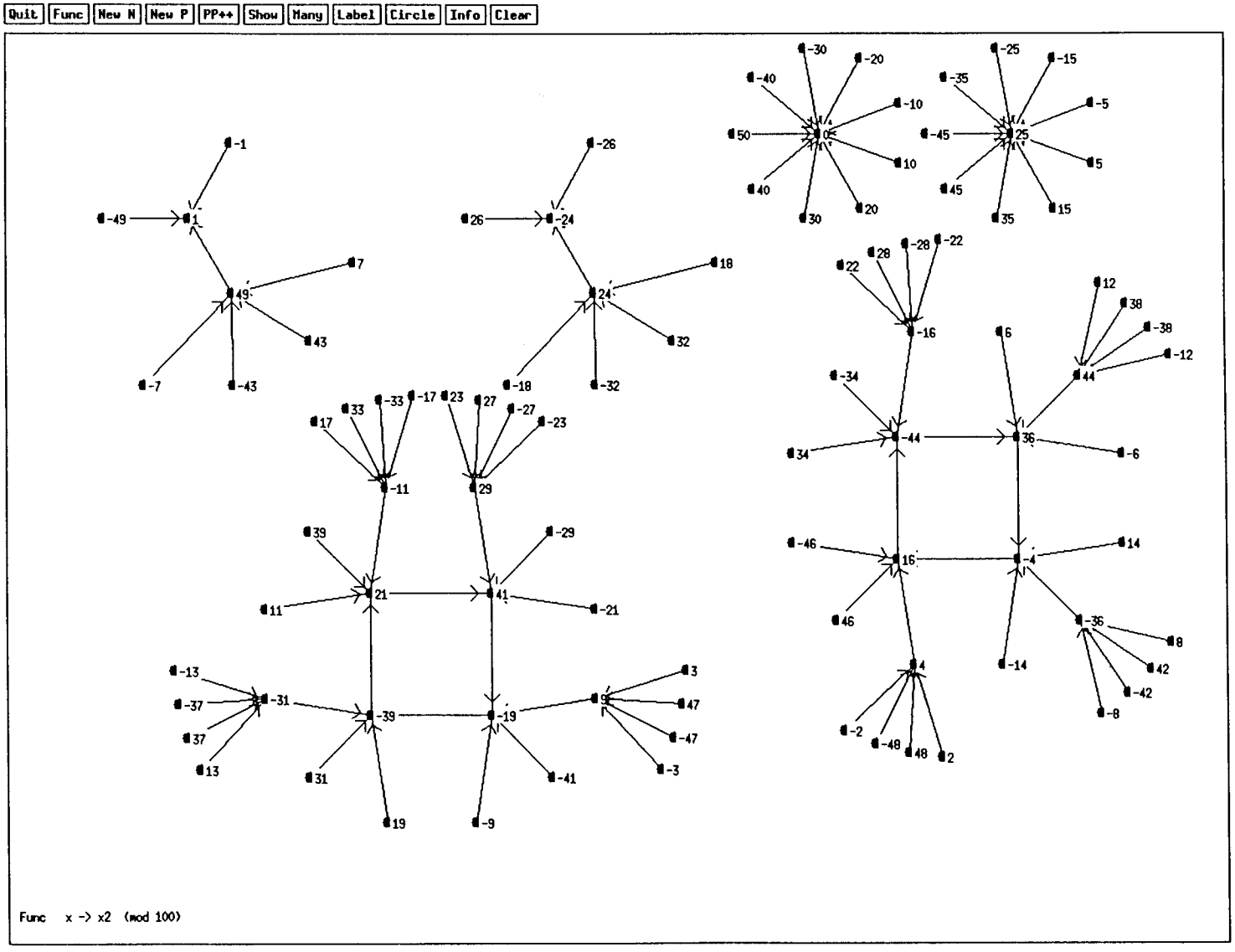
In this paper, we consider the dynamical properties of iterated polynomial functions under modular arithmetic. Given a polynomial f(x) with integral coefficients and a modulus n, we construct a directed graph with n vertices, each corrosponding to an equivalence class in $Z_{/n}$, with a directed edge from x to f(x) mod n (i.e. our edge set is { (x,f(x)) | x $Z_{/n}$ }), and we call this graph $G_n(f)$, the state transition graph of n under f. Note that this is a labeled graph -- each vertex corresponds to a specific integer mod n -- but for most purposes unless mentioned otherwise we will be considering the graphs as unlabeled; that is, we will be considering properties that are invariant under isomorphism. Also note that if we let "state transition graphs" be directed graphs with out-degree of exactly 1 for all vertices and arbitrary in-degree, then $G_n(f)$ is a state transition graph (thus defining some finite automaton); unless indicated otherwise, by "graph" we mean a (finite) state transition graph. The distictions between sets and graphs are delt with loosely, since the graph under consideration induces a graph on any subset of its points. Finally note that by polynomial f(x) we usually mean a unique polynomial in Z[x], rather than an equivalence class of polynomials in $Z_{/n}[x]$.

[example 1: $f(x)=x^2 \pmod{100}$]

The Anatomy of Graphs:

The behaviour of iterated functions is entirely described by their graph. The <u>flow</u> from a point x on a graph is the set of points y obtained by iteratively applying f(x); i.e. for some k>0 $y=f^k(x)$, so $\{y\} = \{ f(x), f^2(x), f^3(x), \ldots\}$. On finite sets, we have the simplifying property that every flow eventually cycles. Letting the <u>watershed</u> of a point x be the set of points y which

Figure 1


Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
Func x -> x2 (mod 100)

eventually lead into x, i.e. for some $k \geq 0$ $f^k(y)=x$, then the set of watersheds for points on cycles partitions the n points of the graph into c connected <u>components</u>, one for each cycle. By the <u>cycle structure</u> of a graph we mean the family of cycle lengths of the components of the graph. By definition, a <u>cycle point</u> is a point that is in it's own flow, a <u>transient point</u> is a non-cycle point, and a <u>tail point</u> is a transient with in-degree 0 (i.e. it's watershed is the empty). Note that the watershed for a transient point is a tree; we can associate a tree with a cycle point by partitioning the elements of the cycle's watershed according to which cycle point occurs first in the element's flow. Finally, we can define the <u>distance</u> of a point x to be how far it is from it's cycle, i.e. the least k such that $f^k(x)$ is a cycle point (thus the distance of any cycle point is 0).

Considering the graph as a whole, now, we can say two labeled graphs G_1 and G_2 are isomorphic if there exists a bijection $i:G_1 \to G_2$ such that $x \to y$ in G_1 iff $i(x) \to i(y)$ in G_2. All the properties discussed above of the graph an points in it are preserved under isomorphism. An automorphism of a labeled graph is an isomorphism from the graph to itself. The number of distict automorphisms of a graph, which we will call the <u>symmetry</u> of the graph, reflects in some way how interchangeable various vertices are, and works as a concrete measure of esthetics: pretty graphs have high symmetry, whereas ugly and spasmodic graphs have symmetry 1.

Since symmetry is preserved under isomorphism, we can speak of the symmetry of an unlabeled graph. A simple way to calculate the symmetry of a graph by looking at the diagram, is to compute the symmetry $\mathrm{Sym}(C) = R(C) \prod_{x \in C} \prod_{E \in E^*(x)} |E|!$ for each component, where $E^*(x)$ is the partition of the transient predecessors of x such that $y \sim z$ iff the trees associated with y and z are isomorphic, and where R(C) is the number of rotations of the cycle points which preserve the structure of the associated trees, then take the product $\mathrm{Sym}(G) = \prod_{C \in G} M(C)!$ over all distinct components C in G, where M(C) is the number of

components isomorphic to C. (This is perhaps not obvious and needs more explanation.)

[example 2: f(x)=4x3-x+1 (mod 45)]

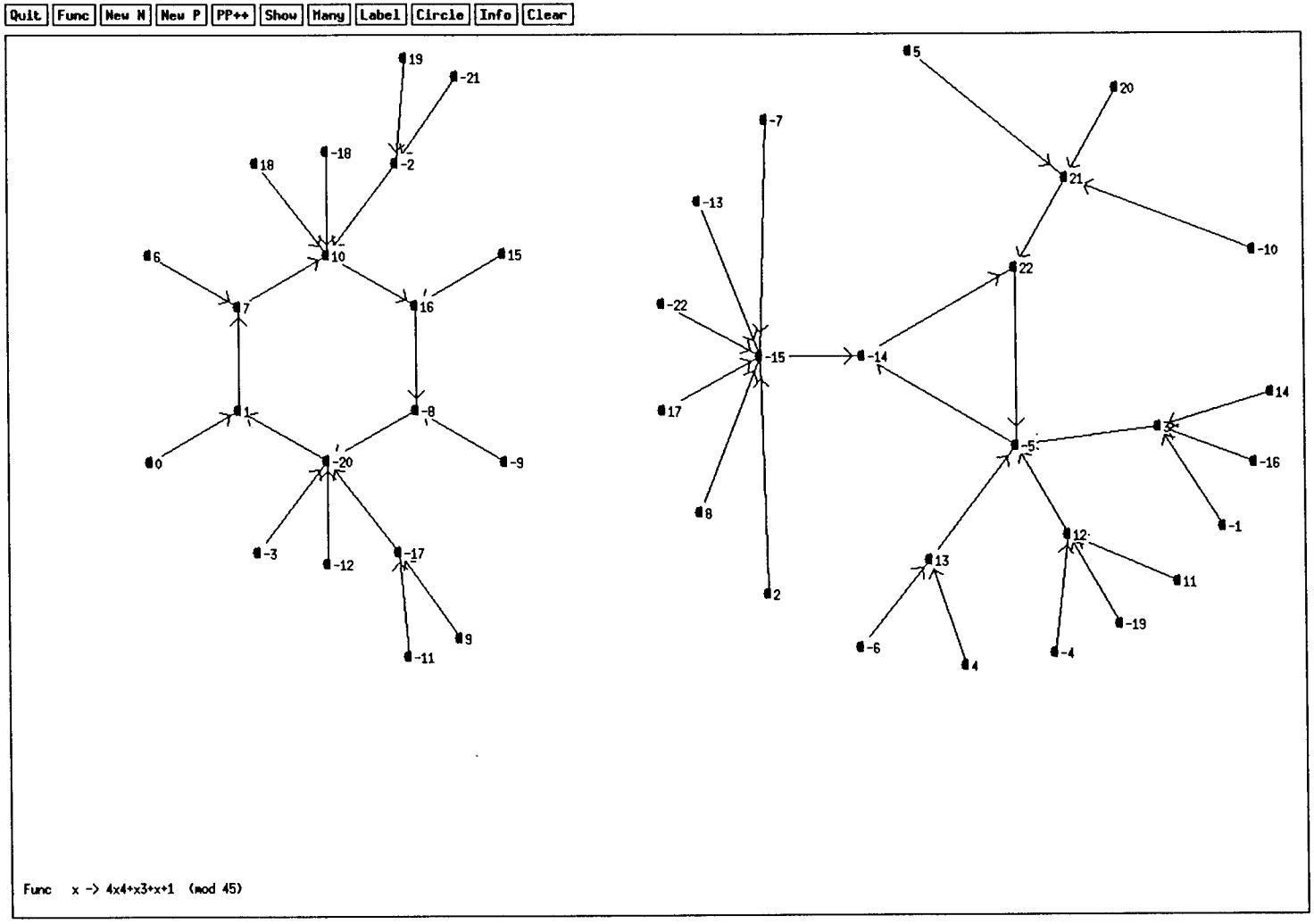
Let's compute symmetry for an example graph. For the first component, $R(C_1)=2$, $E^*(10)=\{\{18,-18\},\{-2\}\}$, and $Sym(C_1)=32$. For the second component, $R(C_2)=1$, $E^*(-5)=\{\{3,12\},\{13\}\}$, and $Sym(C_2)=2!\ 2!\ 3!\ 3!\ 3!\ 6! = 622{,}080$. Thus $Sym(G)=19{,}906{,}560$.

Factoring of Graphs:

The Cartesian product of graphs may be defined as follows: The vertices of the graph $G_1 x G_2$ are the elements of $G_1 x G_2$, and the edges are defined by the rule: if x->y in G_1 and a->b in G_2, then (x,a)->(y,b) in $G_1 x G_2$. Thus graph G factors into G_1 and G_2 if G is isomorphic to $G_1 x G_2$. Note that not all graphs are factorable into graphs of order > 1. Also note that if a graph is factorable into two graphs all of whose components have order > 1, then all the original graph's components are factorable.

We can find a simple relationship between the cycle structures of G_1, G_2, and $G_1 x G_2$. First we consider the product of cyclic graphs C^n and C^m, each having n and m points respectively, arranged in a cycle. Then, if d=gcd(n,m), $C^n x C^m$ is composed of d copies of $C^{nm/d}$. Second, we notice that (x,a) in $G_1 x G_2$ is a cycle point iff x and a are both cycle points in their respective graphs; therefore, we needn't consider transients to determine cycle structure: a component of cycle length n cross a component of cycle length m will yield d (not necessarily isomorphic) components of cycle length nm/d. Recalling that the product of two graphs is the union of all possible products of their components, we have: if the cycle structure of G_1 is the family $\{n_1,n_2,\ldots,n_r\}$ and of G_2, $\{m_1,m_2,\ldots,m_s\}$ (G_1 has r cycles, G_2 has s cycles), then $G_1 x G_2$ has $t = \sum_{i,j} \gcd(n_i,m_j)$ cycles (in particular, $t \geq rs$), and its cycle structure can be similarly computed.

Figure 2



Compare the number of cycles

What is the relationship between the symmetries of G_1, G_2, and $G_1 x G_2$? The latter is neither necessarily greater than or less than the product of the former. This question remains to be solved.

We are now ready for our first theorem, which derives from equivilantly the Chinese Remainder Theorem or the fact that the ring $Z_{/nm}$ is isomorphic to $Z_{/n} \times Z_{/m}$ when n and m are relatively prime, and yields a factorization for polynomial graphs whose order is divisible by more than one prime.

Theorem (Factorization): If gcd(n,m)=1, then $G_{nm}(f) = G_n(f) \times G_m(f)$.

For proof, we need to demonstrate an isomorphism $i:G_{nm}(f) \to G_n(f) \times G_m(f)$. We can use the canonical isomorphism from $Z_{/nm}$ to $Z_{/n} \times Z_{/m}$, independent of f. There exists r,s such that rn+sm = gcd(n,m) = 1. Let i(x,a) = smx+rna (mod nm). This is a bijection since $i^{-1}(z)$ = ((z mod n)/sm, (z mod m)/rn), which exists since gcd(sm,n)=1=gcd(rn,m). We verify that this is an isomorphism by checking that i(xy,ab) = smxy+rnab = $(sm)^2xy+smxrna+smyrnb+(rn)^2ab$ = (smx+rna)(smy+rnb) = i(x,a) i(y,b) (mod nm), since mn divides the two inner factors and $(sm)^2$ = sm(1-rn) = sm-nmrs = sm and similarly $(rn)^2$ = rn (mod nm). Since f is a polynomial, it follows that i(f(x),f(a)) = f(i(x,a)). This is exactly what is required for the graph isomorphism: (x,a) -> (f(x),f(a)) in $G_1 x G_2$ iff i(x,a) -> f(i(x,a)) in G_{nm}.

A consequence of this theorem is that we really only need to consider the graphs of f mod p^k, where p is prime.

A second perspective on this theorem is to consider contractions of the graph. A <u>contraction</u> of a general graph under a partition of its vertices creates a new graph with one vertex for each set in the partition, and with an edge from one new vertex to another if such an edge exists for any pair of the points in the one corresponding set to the other. I.e. if G has vertices V(G) and edges E(G), and P is the partition $\{P_i\}$ of V(G), then the contraction G/P has vertices $\{P_i\}$ with edges $\{(P_i,P_j)$ | there

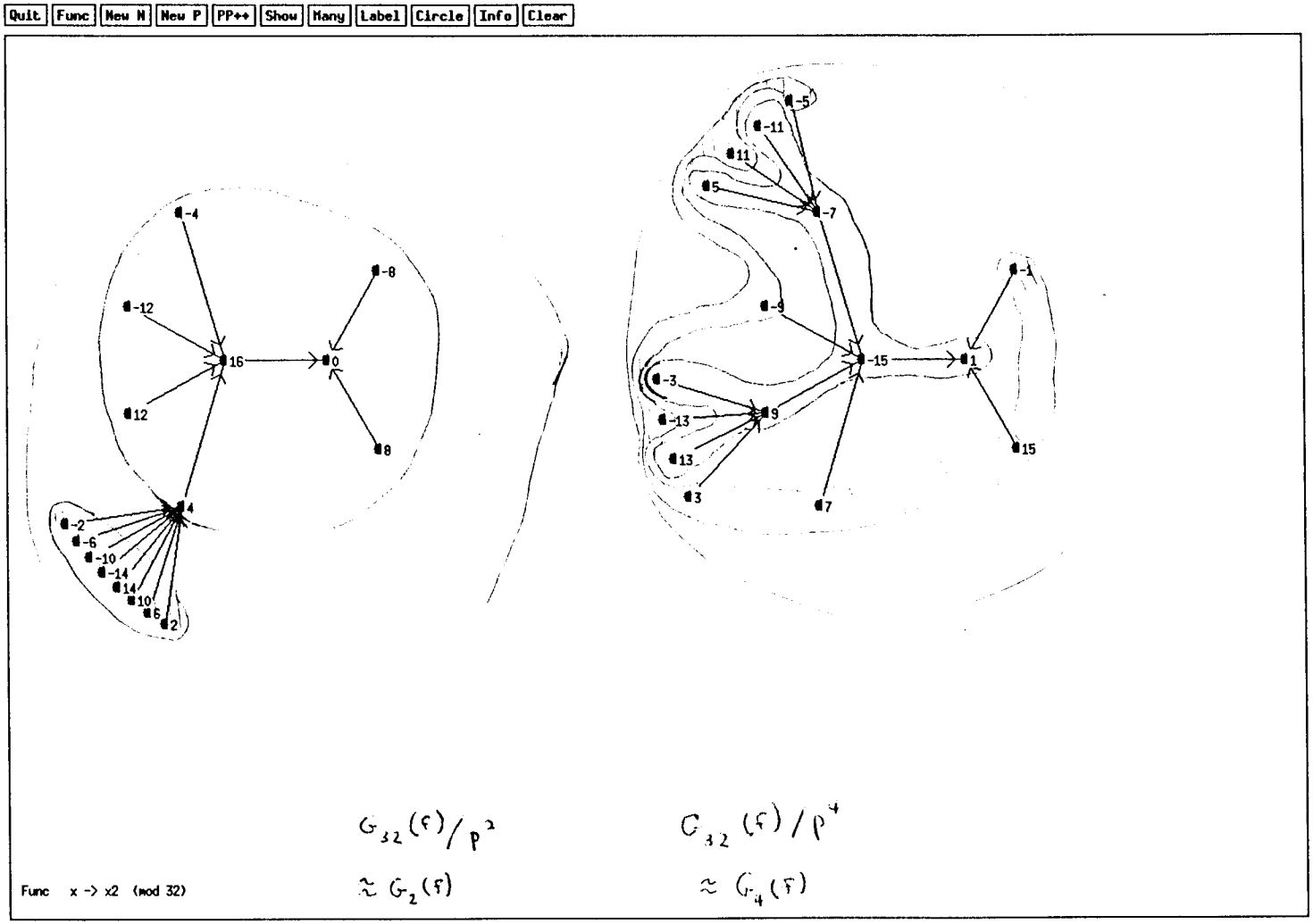
exists x in P_i and y in P_j such that (x,y) is in E(G)}. You can visualize the contraction as simply drawing a circle around

Let us define a <u>d-contraction</u> of $G_n(f)$ to be $G_n(f)/P^d$ where P^d is defined by x~y if x=y (mod d); this is only well-defined if d|n. Since polynomials are also well-defined on modulii, $G_n(f)/P^d$ is isomorphic to $G_d(f)$ (in other words, all arrows out of a given circle go to the same circle in this case - in other cases, a contraction of a state transition graph may not yield a state transition graph). If you consider n relatively prime to m, and you look at the n-contraction and the m-contraction of $G_{nm}(f)$, the Chinese Remainder Theorem tells us that each circle in the n-contraction intersects each circle in the m-contraction exactly once; this last is sufficient to define an isomorphism from $G_{nm}/P^n \times G_{nm}/P^m$ to G_{nm}, which is our factorization theorem. This is a more general approach, since any two contractions whose partitions satisfy the intersection property and which yield state transition graphs will yield a factorization of the original graph -- although the factors are not guaranteed to be polynomially generated graphs.

(I can't find any counterexamples just now, nor can I find n,m factorizations this way other that the canonical ones for n,m relatively prime. This area merits more exploration. A few easy results: If P_1 and P_2 are partitions for the factorization of a graph on n points, then $|P_1|\ |P_2| = n$ and $|P_i|$ is constant for P_i in P_1 and likewise in P_2, since the intersection property must be satisfied. If the symmetry of a factorable labeled graph is s, then there are at least s different partition pairs P_1,P_2 which produce a factorization. Unfortunately, all of these produce isomorphic factors, so we haven't found anything new here. I suspect n,m factorization is unique up to isomorphism for n,m relatively prime.)

[examples 3,4]

Figure 3



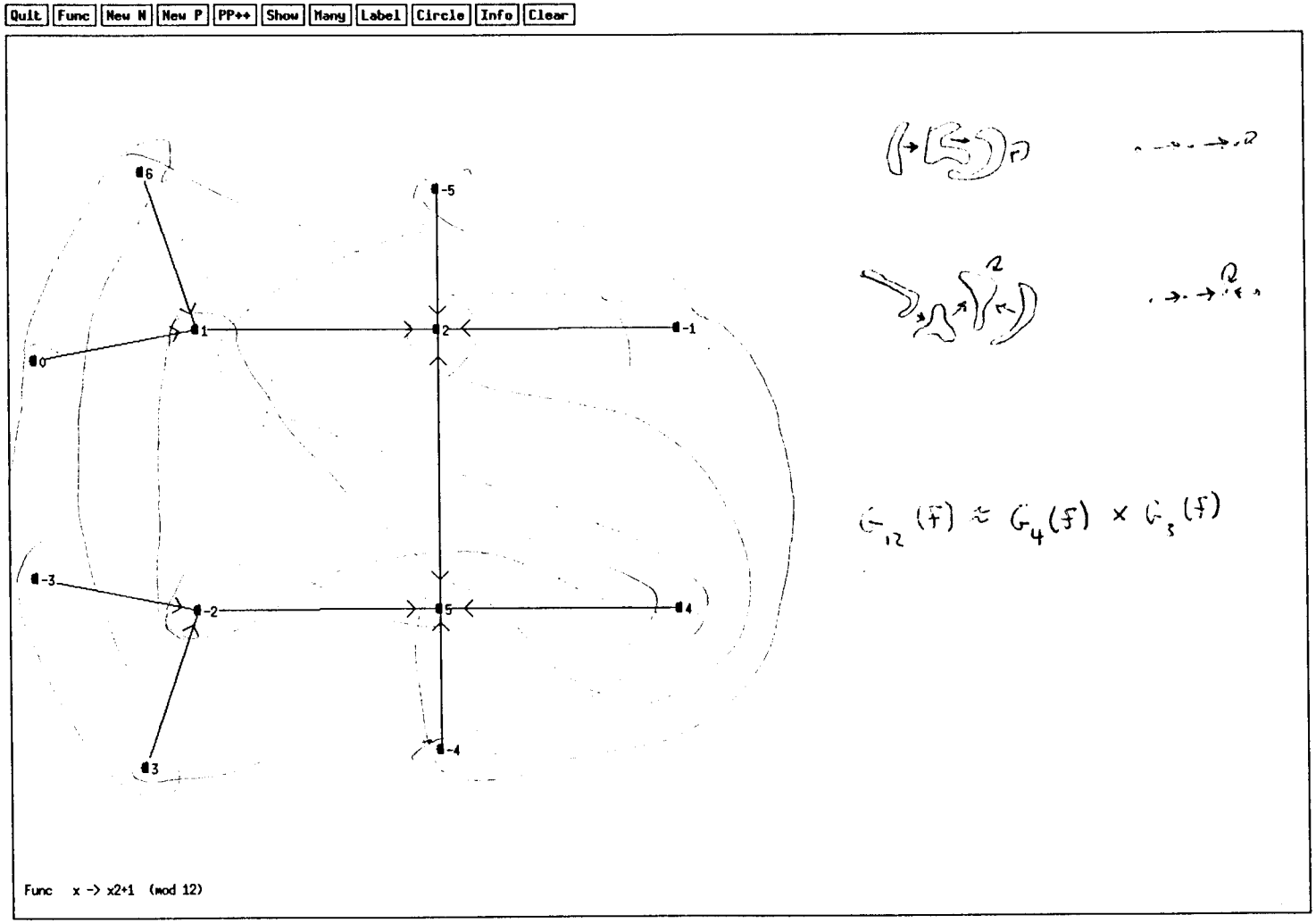
$G_{32}(f)/p^2$

$\approx G_2(f)$

$G_{32}(f)/p^4$

$\approx G_4(f)$

Figure 4



$G_{12}(f) \cong G_4(f) \times G_3(f)$

Counting Arguments:

There are a number of obvious questions to ask, few of which seem to have simple answers. One question is whether all state transition graphs on n vertices can be generated by polynomial functions, and the question can be asked directly for labeled graphs, or only up to isomorphism.

To start with the former, how many labeled state transition graphs exist mod n? n^n, since there is one for each arbirary function $Z_{/n} \to Z_{/n}$. How many distinct (equivalence classes of) polynomials are there mod n? Again, we really only need to ask this question for $n=p^k$, since our factorization theorem tells us that if there are N polynomials mod n and M polynomials mod m (where n and m are relatively prime), then there are NM polynomials mod nm. This is hopelessly insufficient, since even $n^n m^m$ is nowhere near $(nm)^{nm}$. Now for primes and prime powers: Since $Z_{/p}$ is a field, we can use LaGrange interpolation to build polynomials; i.e. let

$$L_i(x) = \frac{(x-0)(x-1)\ldots(x-i-1)(x-i+1)\ldots(x-p-1)}{(i-0)(i-1)\ldots(i-i-1)(i-i+1)\ldots(i-p-1)}.$$

Now $L_i(x) = 1$ if $x=i$ and 0 otherwise, so any arbitrary function can be fitted. Thus we have all p^p polynomials mod p. However, performance decreases as k increases; we haven't been able to find an exact formula, but we have an upper bound on the number of polynomials mod p^k: $p^{p(\frac{p^k-1}{p-1})}$. It is obtained by induction as follows: Assume the bound holds for k. A given polynomial mod p^{k+1} yields a polynomial when considered mod p^k.

How many distinct polynomials mod p^{k+1} map to a particular distinct polynomial mod p^k? Let the distinct polynomial f(x) mod p^k be represented uniquely by its p^k values on $Z_{/p^k}$; then if f'(x) is a distinct function mod p^{k+1} similarly represented such that $f'(x)=f(x)$ mod p^k, its value at x can be one of p possible values. Since we have p^{k+1} places to vary, there are exactly $p^{p^{k+1}}$ possible functions f' (some or all of which may be obtainable with polynomials). Thus the number of polynomials mod $p^{k+1} \leq p^p$ times the number of polynomials mod p^k, and by induction, the

number of polynomials mod $p^k \leq p^p+p^2+p^3+\ldots+p^k$. (This is not a strict upper bound for p=2, at least.) Clearly not every labeled graph is obtainable using polynomials mod p^k for k>1, either.

The question remains open whether all graphs up to isomorphism can be generated with polynomials. If the number of unlabeled graphs on n vertices drops off even faster than do the number of polynomials, then this remains a possibility. Unfortunately, we have a simple counterexample: this graph on 6 vertices is not factorable, and thus it could not be generated by a polynomial. Can maybe all graphs on p^k points be generated? No; add two isolated points to the example.

Nevertheless, we can try to count the number of graphs up to isomorphism on n vertices. Without going through the derivations here, we claim that, where A(n) = the number of state transition graphs on n vertices, C(n) = the number of connected graphs on n vertices, T(n) = the number of rooted trees on n vertices, and r(p) = the number of ordered partitions rotationally equivalent to p,

$$A(n) = \sum_{p \text{ a partition of } n} \prod_{(v,m)\in p} \binom{m+C(v)-1}{m}$$

where m is the multiplicity of v in the partition

$$C(n) = \sum_{c=1..n} \sum_{p \text{ an ordered partition of } n-c} \frac{1}{r(p)} \prod_{(v,m)\in p} \binom{m+T(v+1)-1}{m}$$

$$T(n) = \sum_{b=1..n-1} \sum_{p \text{ a partition of } n-b} \prod_{(v,m)\in p} \binom{m+T(v)-1}{m}$$

Unfortunately, these formulas aren't much of a timesaver, nor do we know how to express them in a simpler form.

(Stephanie is going to complete this section later.)
Facts about isomorphic graphs:

i. The graphs of x->x+a mod n and x->x+b mod n are isomorphic if (a,n)=(b,n).

Proof: Assume that $y=x+a$. We want some isomorphism f such that $f(y)=f(x)+b$. We can easily obtain

$$(b/a)y = (b/a)x + b$$

So the isomorphism is $f(t)=(b/a)t$.

ii. The graphs of $x \to ax \bmod p$ and $x \to bx \bmod p$ are isomorphic if a and b are at the same depth on the $x \to x^2 \bmod p$ graph.

Proof:

iii. The graphs of $x \to ax^i \bmod n$ and $x \to bx^i \bmod n$, $i>1$, are isomorphic if

a.) $(a,n) = (b,n)$
b.) $(a/b)^k$ exists, where $k= 1/(i-1)$

Proof: Assume that $y = ax^i$. We want some isomorphism f such that $f(y)=b(f(x))^i$. After some brief algebra, we find that

$$y(a/b)^k = b(x(a/b)^k)^i$$

So the isomorphism is $f(t)=t(a/b)^k$.

iv. The graph of $x \to ax \bmod p^{i+1}$ is isomorphic to

(graph of $x \to ax \bmod p^i$) + (m cycles of length k)

for some m, k such that $mk=(p-1)p^i$.

Proof:

LINEAR FACTS

i. For some m, k such that mk=n, the function x->x+c mod n has a graph containing m cycles of length k.

Proof: The integers mod n are a group under addition. We can generate a subgroup of this group by selecting an integer and repeatedly adding a constant. It is well-known that the order (and number) of these subgroups divides the order of a group, and that all of the subgroups are isomorphic. Therefore, for some m, k such that mk=n, the graph contains m cycles, each of which contain k elements.

ii. For some m, k such that mk=p-1, the function x->ax mod p has a graph containing a1-cycle and m cycles of length k.

Proof: The nonzero integers mod p with multiplication are isomorphic to the

iii. We can describe the graph of the function x->ax mod p by examining the various graphs $x \to x^i$ mod p.

iv. The in-degree of every node in the graph of x->ax+b mod n is either 0 or k, where k=(a,n).

Proof: We will look at the node labeled c. The in-degree of c is the number of solutions to ax+b=c mod n.

ax+b = c mod n
x = (c-b)/a mod n

We can see that this yields either 0 or (a,n) solutions. In particular, this tells us that x->ax+b mod p gives a graph composed entirely of cycles.

v. If $a_1,\ldots,a_k$ is a sequence of integers and $j \leq k$, the graph of

$$x \to (a_1 a_2 \ldots a_k)x \bmod (a_1 a_2 \ldots a_j)$$

is connected.

Proof:

Other patterns:

The symmetry, or even just esthetics, of graphs over a family of functions seems to vary in a sometimes very orderly fashion.

Limit graphs:

In determining the behaviour of a particular polynomial f in Z[x] under various modulii, we note that though the graphs may be startlingly different for powers of two different primes, there is often a certain structure preserved between different powers of the same prime. Sometimes (if not always) $G_p k(f)$ is a subgraph of $G_p k'(f)$. Often they are identical but for added components, or new tail points building upon previous components. They might be though of as approaching some countably infinite state transition limit graph. (A simple example of an infinite graph is G(f) with vertices from Z and edges {(x,f(x))}.) Being able to determine the structure of such a graph (such as whether it has a finite or infinite number of components, the ratio of cycle points to transients,...) could help answer similar questions about particular graphs.

In order to make these ideas a little more mathematically concrete, we introduce the idea of a <u>limit graph</u> for a function given a sequence N of increasing modulii $n_1, n_2, n_3, \ldots$ Let $G_N(f)$ be defined as the least graph such that $G_{ni}(f)$ is a subgraph of $G_N(f)$ for all i, where we consider subgraphs up to isomorphism, and where "least" is according to the subgraph partial order.

A few quick questions: Is $G_N(f) = G(f)$? Probably not, unless f is a constant function. Is $G_N(f)$ uniquely defined? Yes, since if two graphs are both least, then they must be subgraphs of eachother, and they are isomorphic. Does $G_N(f)$ necessarily exist for any sequence N? Yes, but we need to use Zorn's Lemma to prove it. Let $M = \{ \text{graphs } G \mid G_{ni}(f) < G \text{ for all } i \}$. M is not empty since the infinite disjoint union of $G_{ni}(f)$ is in M. For M to have a least element by Zorn's Lemma, we need to show that every chain in M has a lower bound in M. So suppose M' is a chain in M, that is, M' is a subset in which all elements are comparable. Let G^* be the intersection of all G in M' (which is well-defined using the subgraph isomorphism for two graphs, since they are comparable). G^* is obviously a lower bound for M', and further, G^* is in M, since for any n_i, $G_{ni}(f)$ is a subgraph of all G in M', and thus $G_{ni}(f)$ must be a subgraph of G^*. Therefore Zorn's Lemma holds.

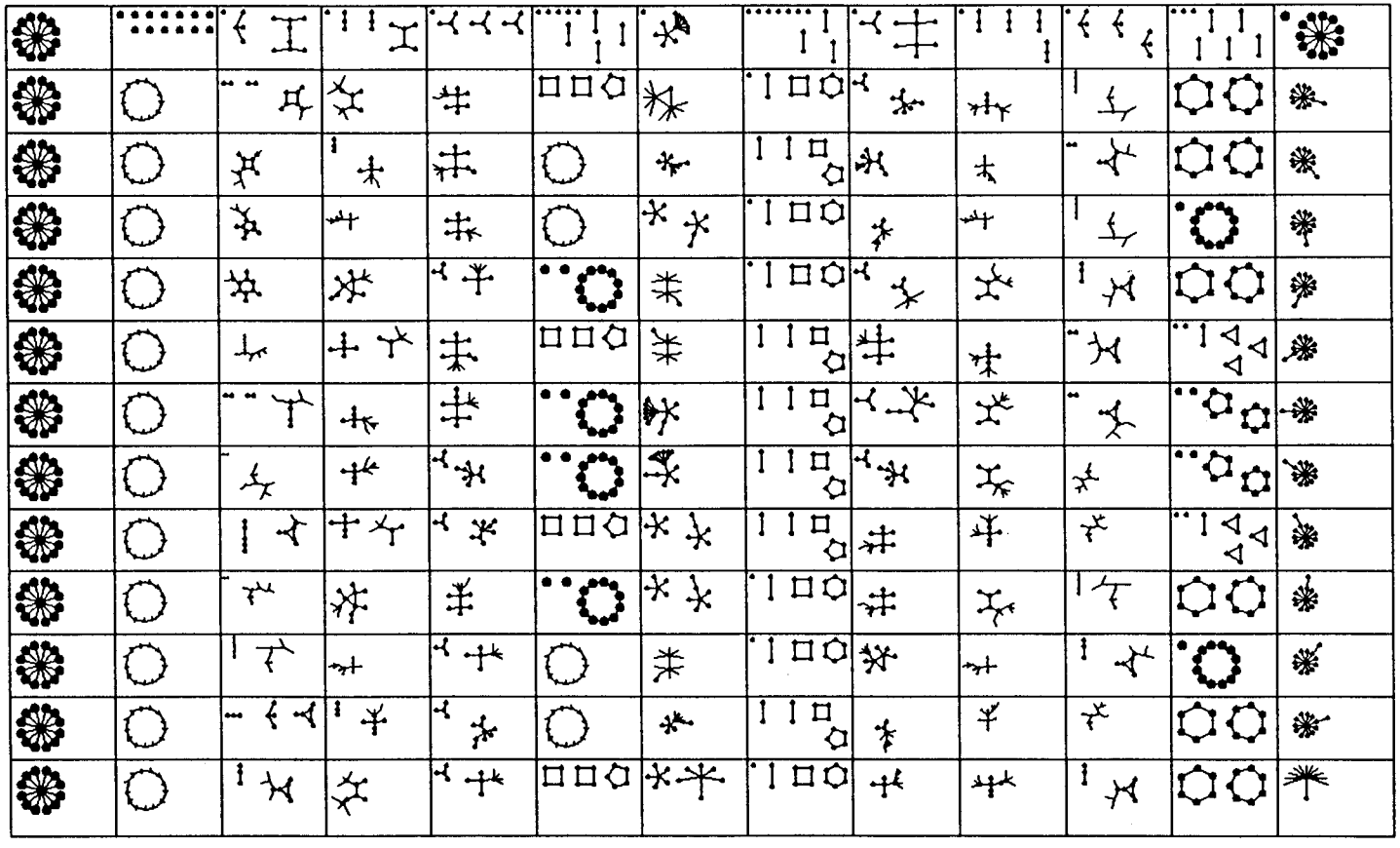
We've now defined what the limit graph is, but what do we mean by saying that the sequence of graphs is "approaching" the limit graph? This: The sequence of graphs $G_1, G_2, G_3, \ldots$ is said to <u>converge</u> to G if there exist subgraph homomorphisms $h_i: G_i \to G$ such that for all x in G, there exists an i_x such that for all $i \geq i_x$, x is in image(h_i). In other words, all points in the limit graph "crystallize" after some finite time and are in every single graph from then on.

You might be wondering what the use could be for limit graphs using sequences other than powers of the same prime. There isn't much. But for a prime p, let's consider the sequence $P = 1, p, p^2, p^3, \ldots$ We will close this paper with the following conjecture: $G_{n1}(f), G_{n2}(f), \ldots$ converges to $G_N(f)$ iff N is a subsequence of P for some prime p. Unfortunately, I can think of

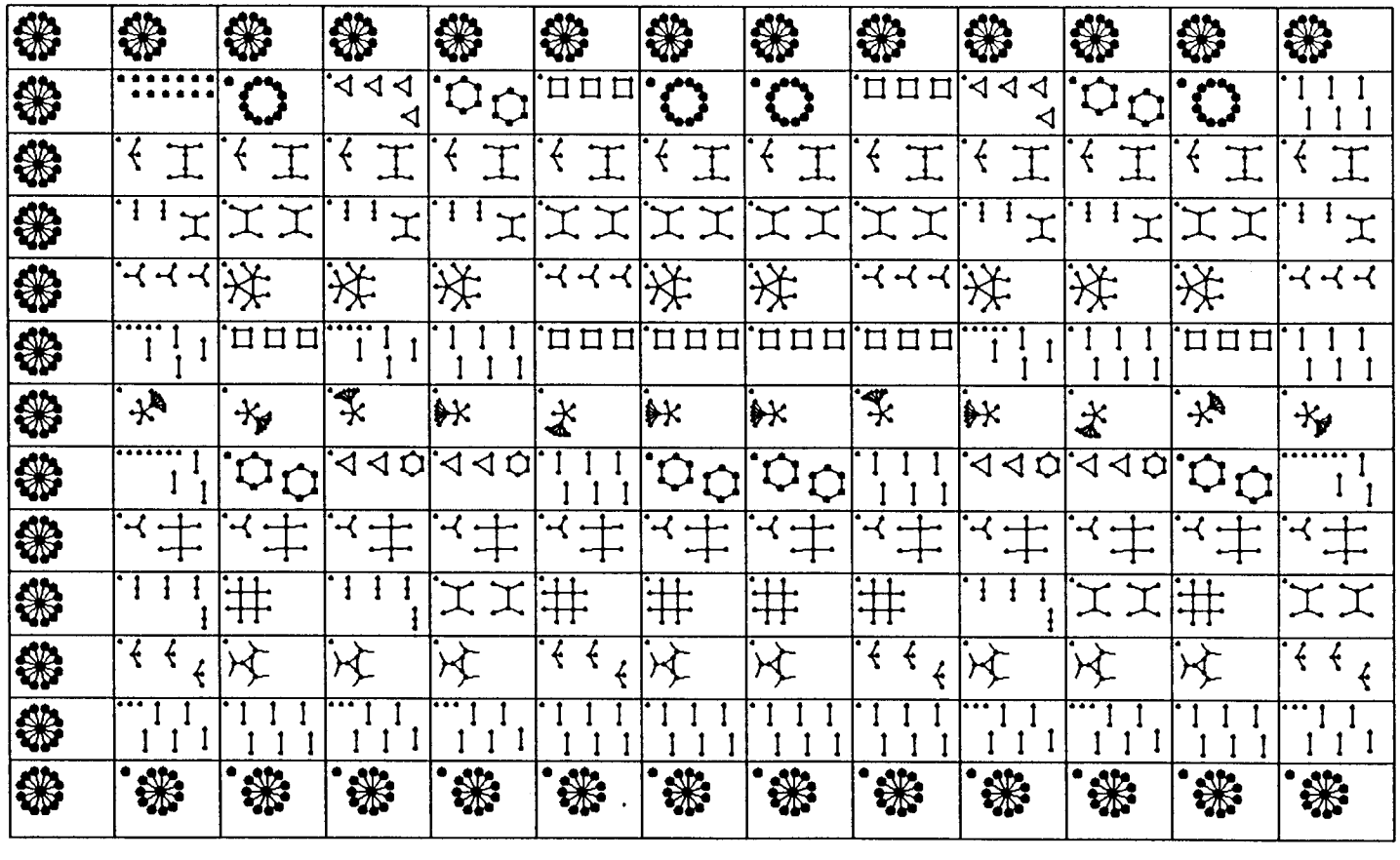
two cases for which this does not hold: constant functions, in which all sequences converge to the same limit graph, and x->x+1, which converges for no sequence at all. Which functions converge for which primes???????

Tacked on to the end here are some fun slides of radiolaria.

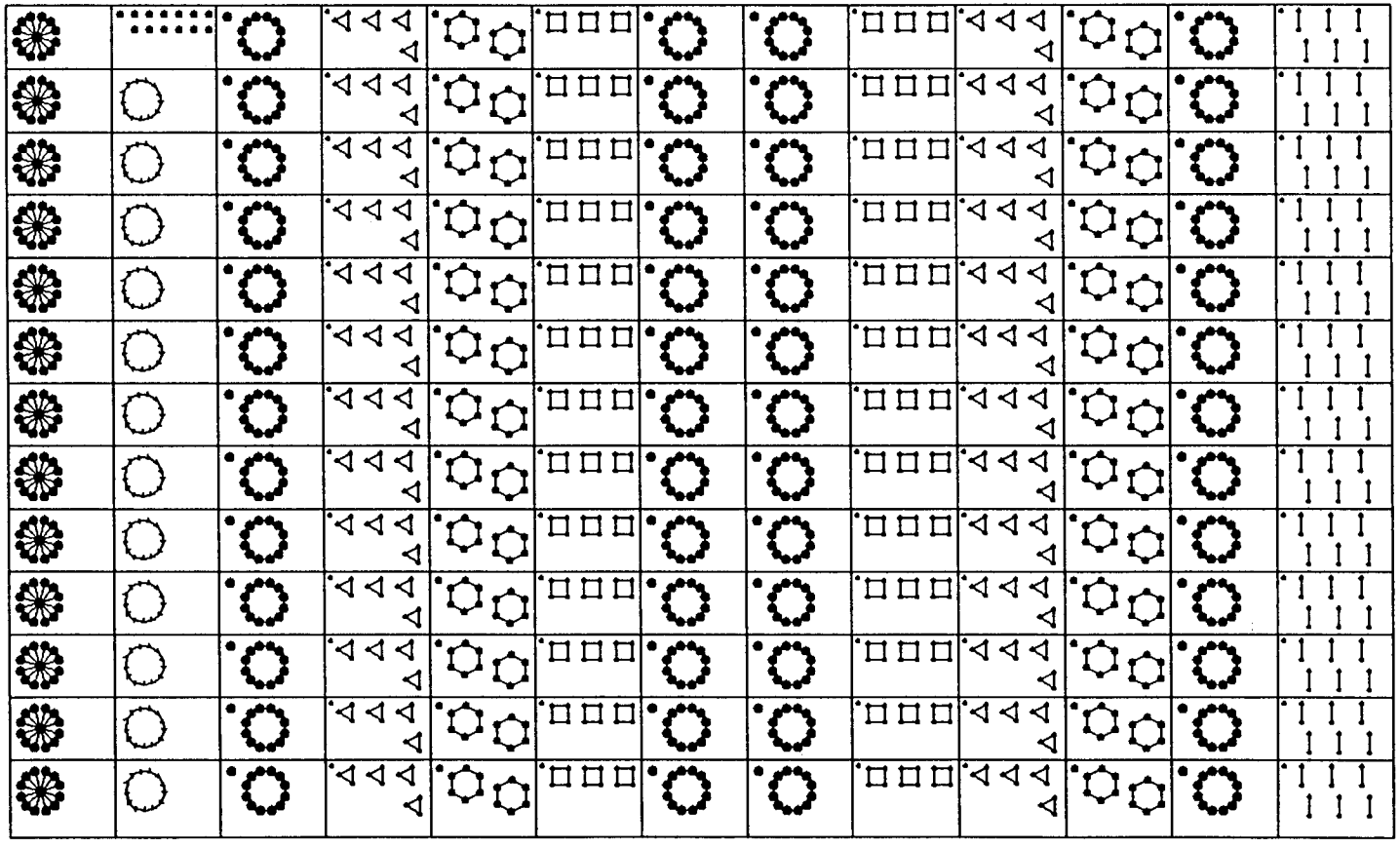
Quit | Func | New N | New P | PP++ | Show | Many | Label | Circle | Info | Clear



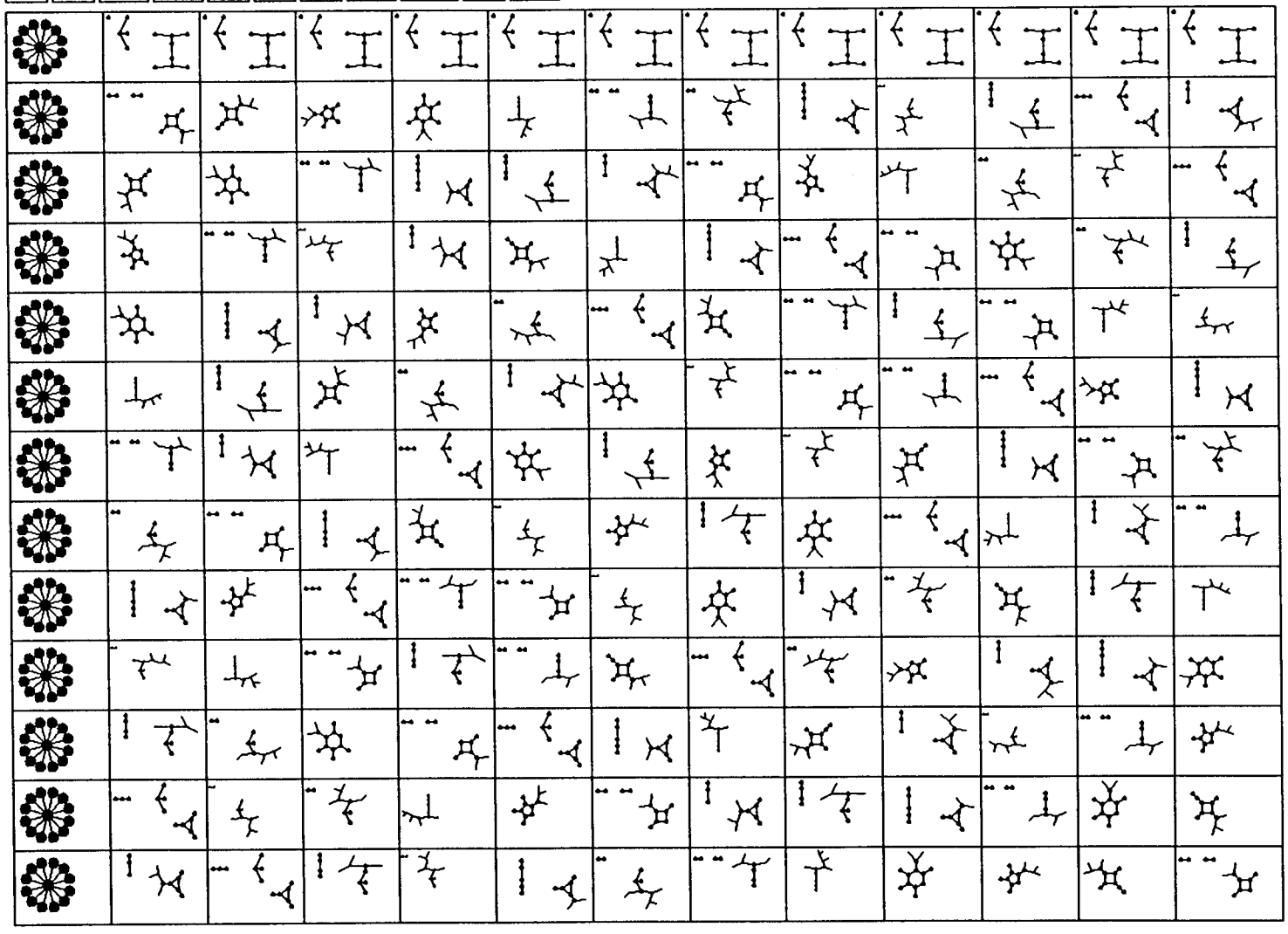
Quit Func New N New P PP++ Show Many Label Circle Info Clear



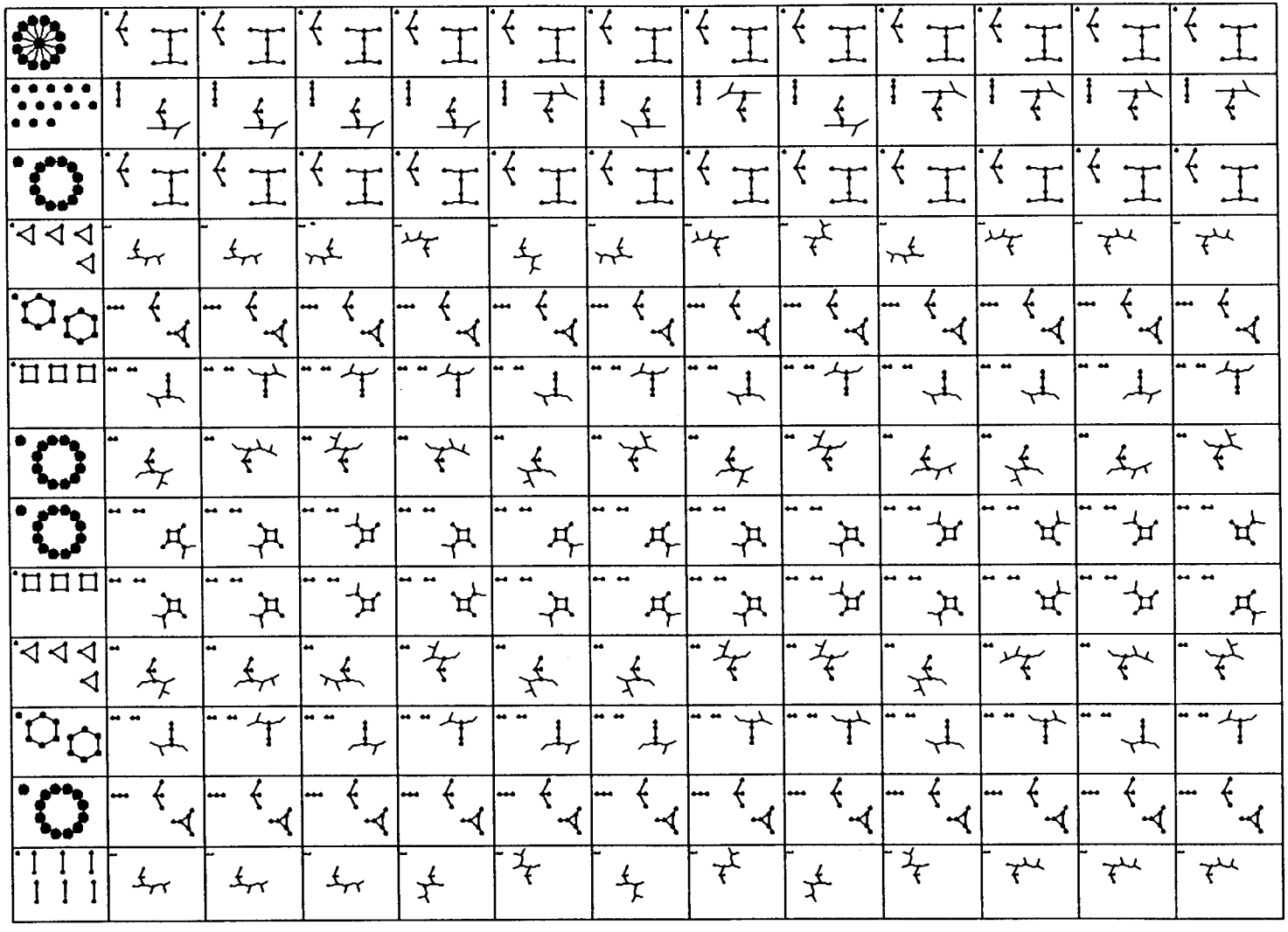
Quit Func New N New P PP++ Show Many Label Circle Info Clear



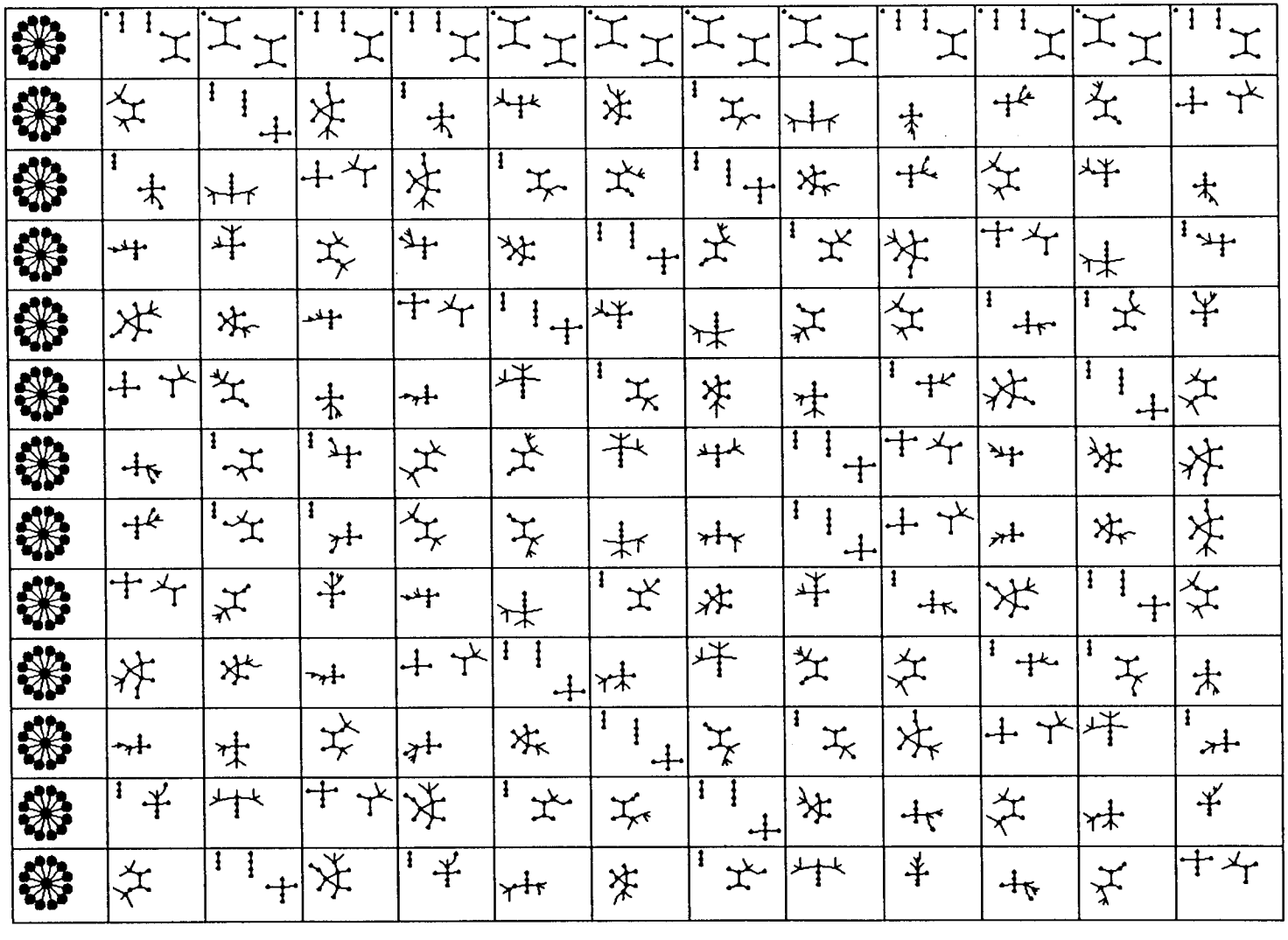
Quit Func New N New P PP++ Show Many Label Circle Info Clear



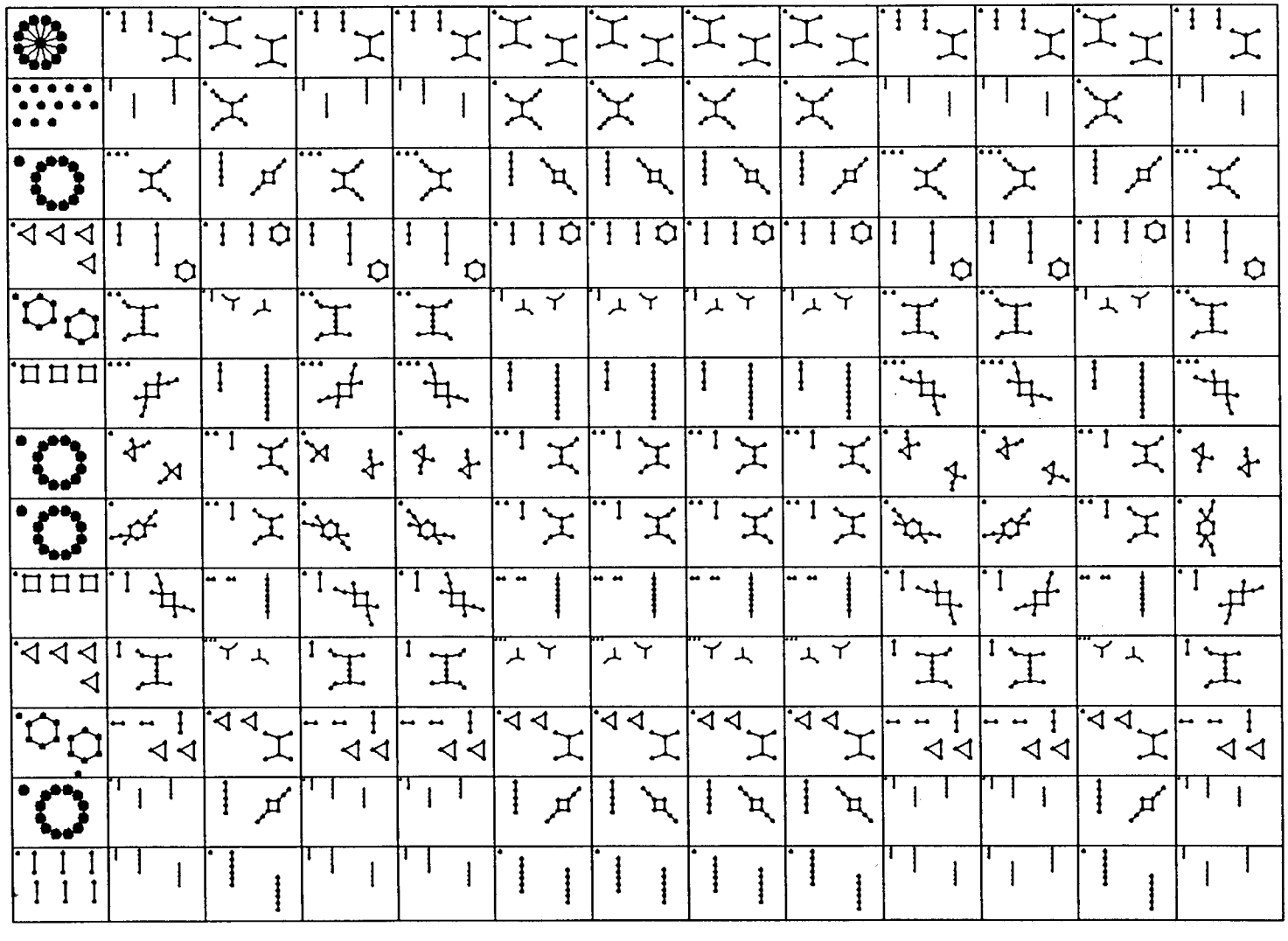
Quit Func New N New P PP++ Show Many Label Circle Info Clear



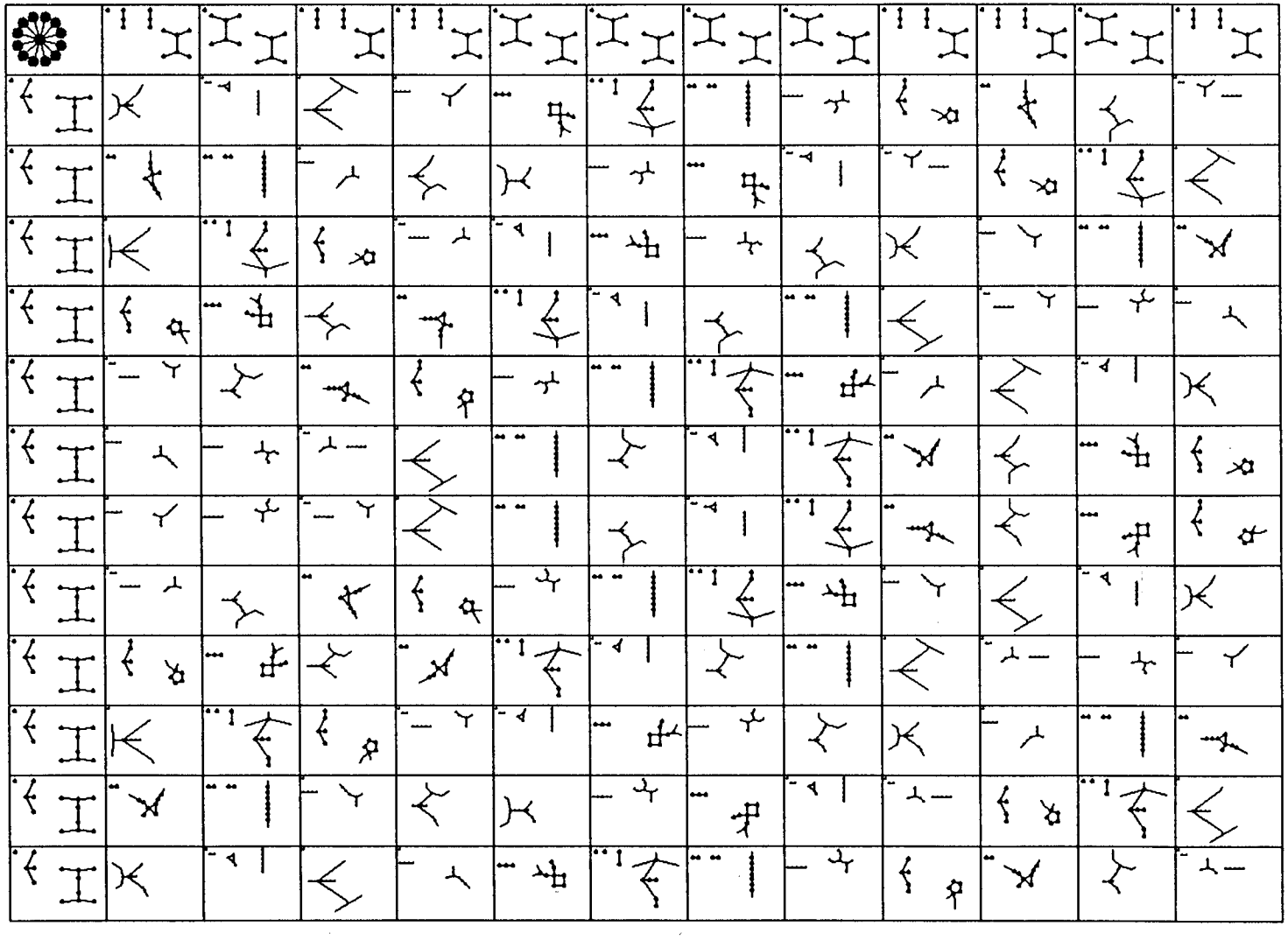
Quit Func New N New P PP++ Show Many Label Circle Info Clear



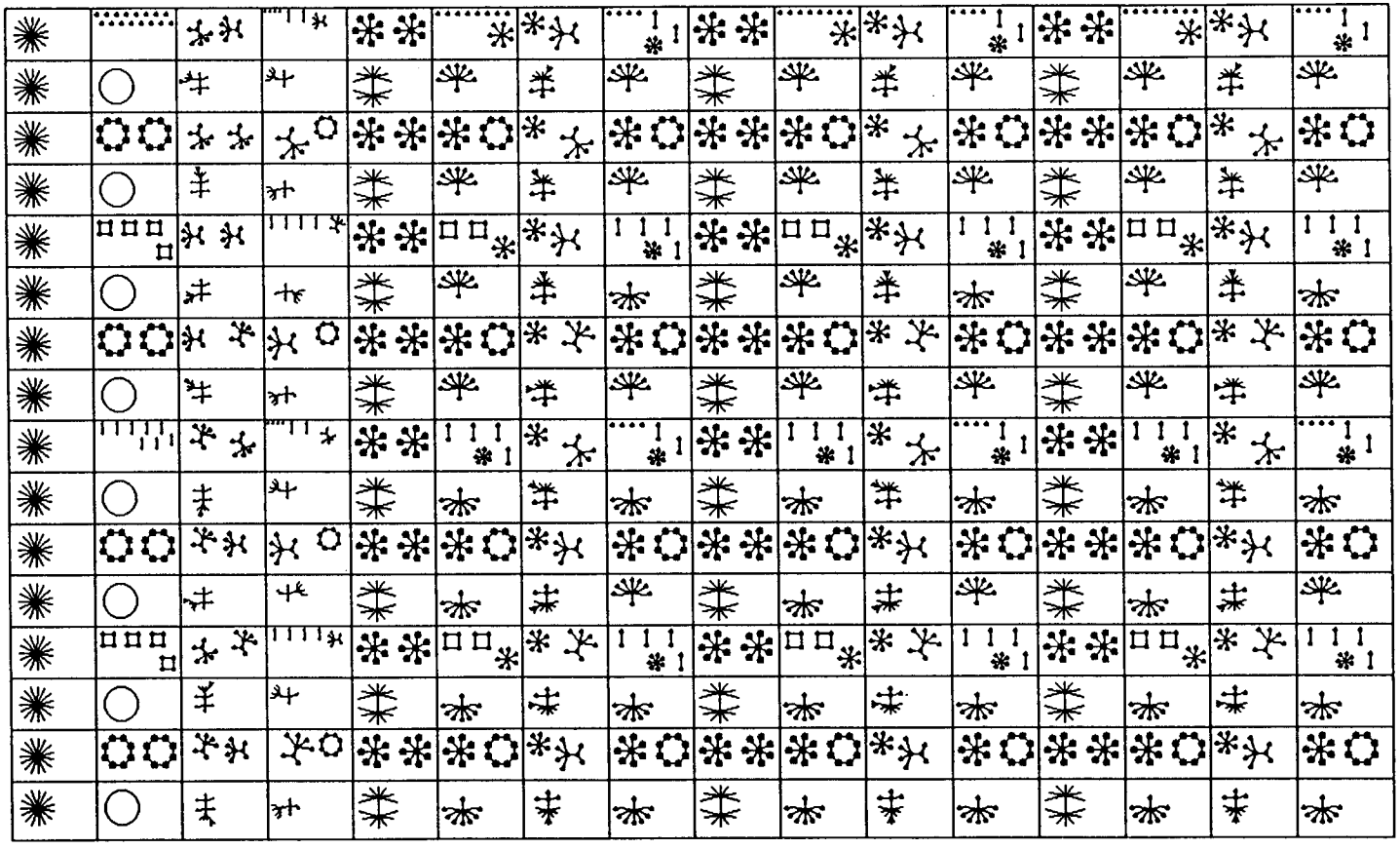
Quit Func New N New P PP++ Show Many Label Circle Info Clear



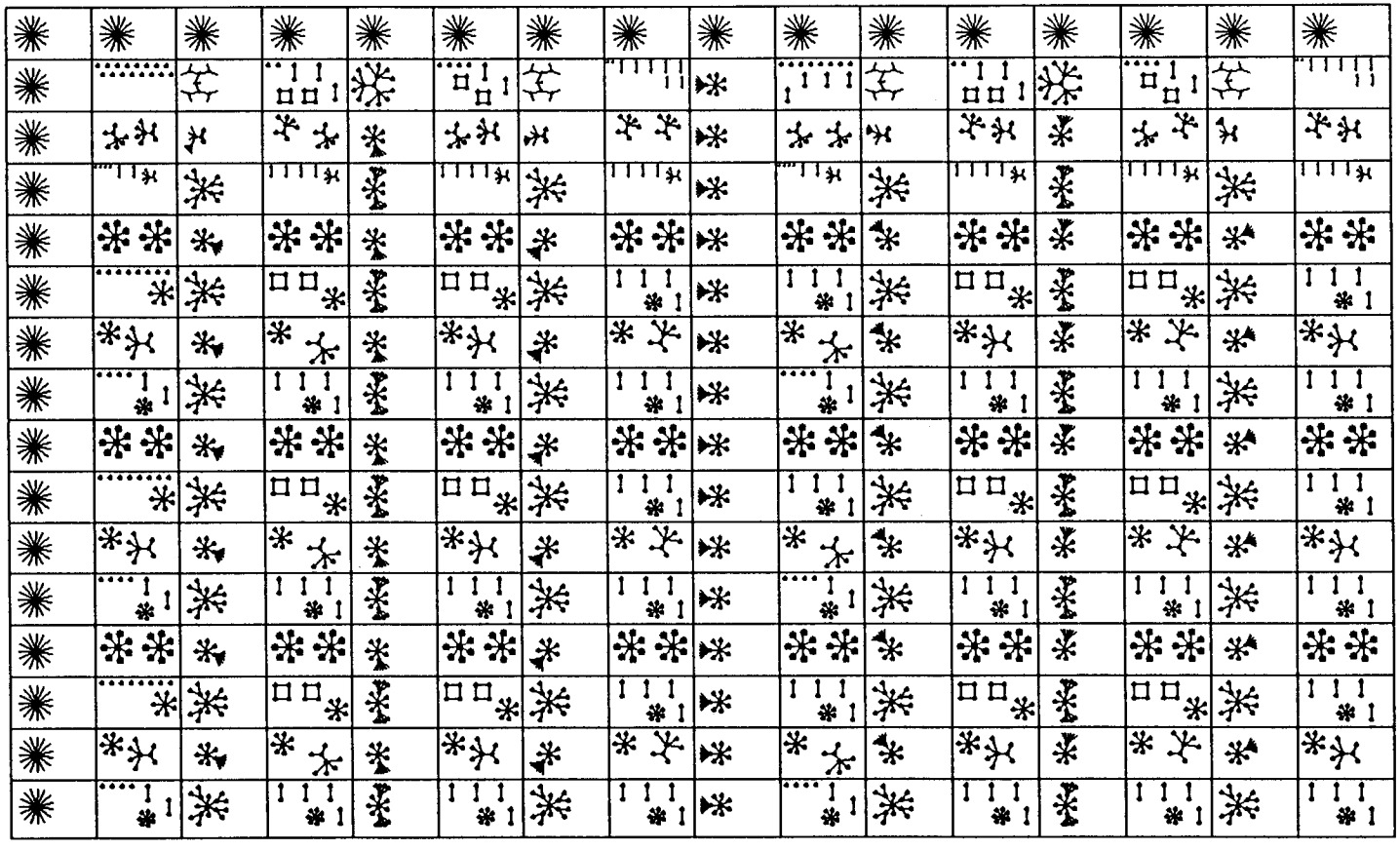
Quit Func New N New P PP++ Show Many Label Circle Info Clear



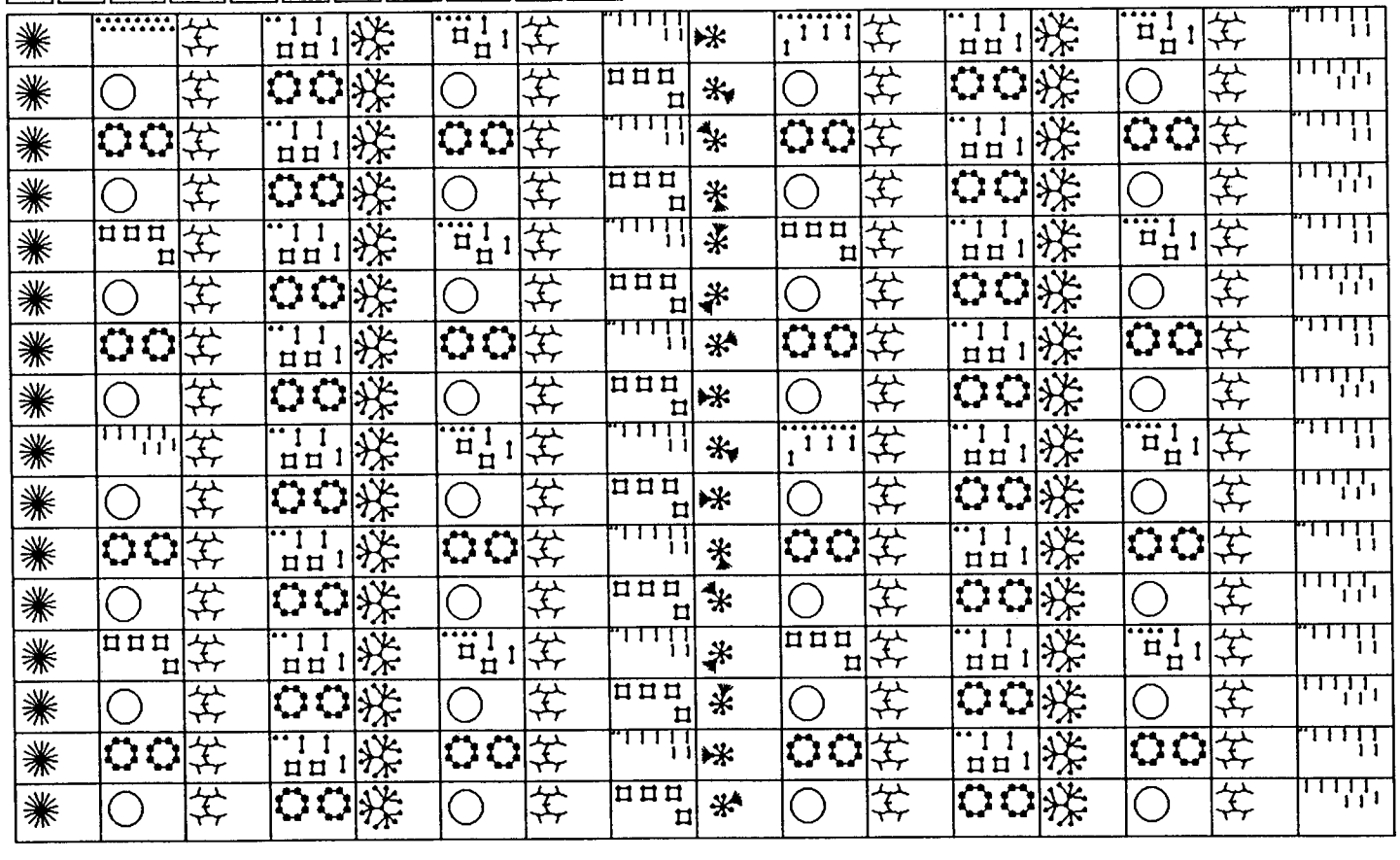
Quit Func New N New P PP++ Show Many Label Circle Info Clear



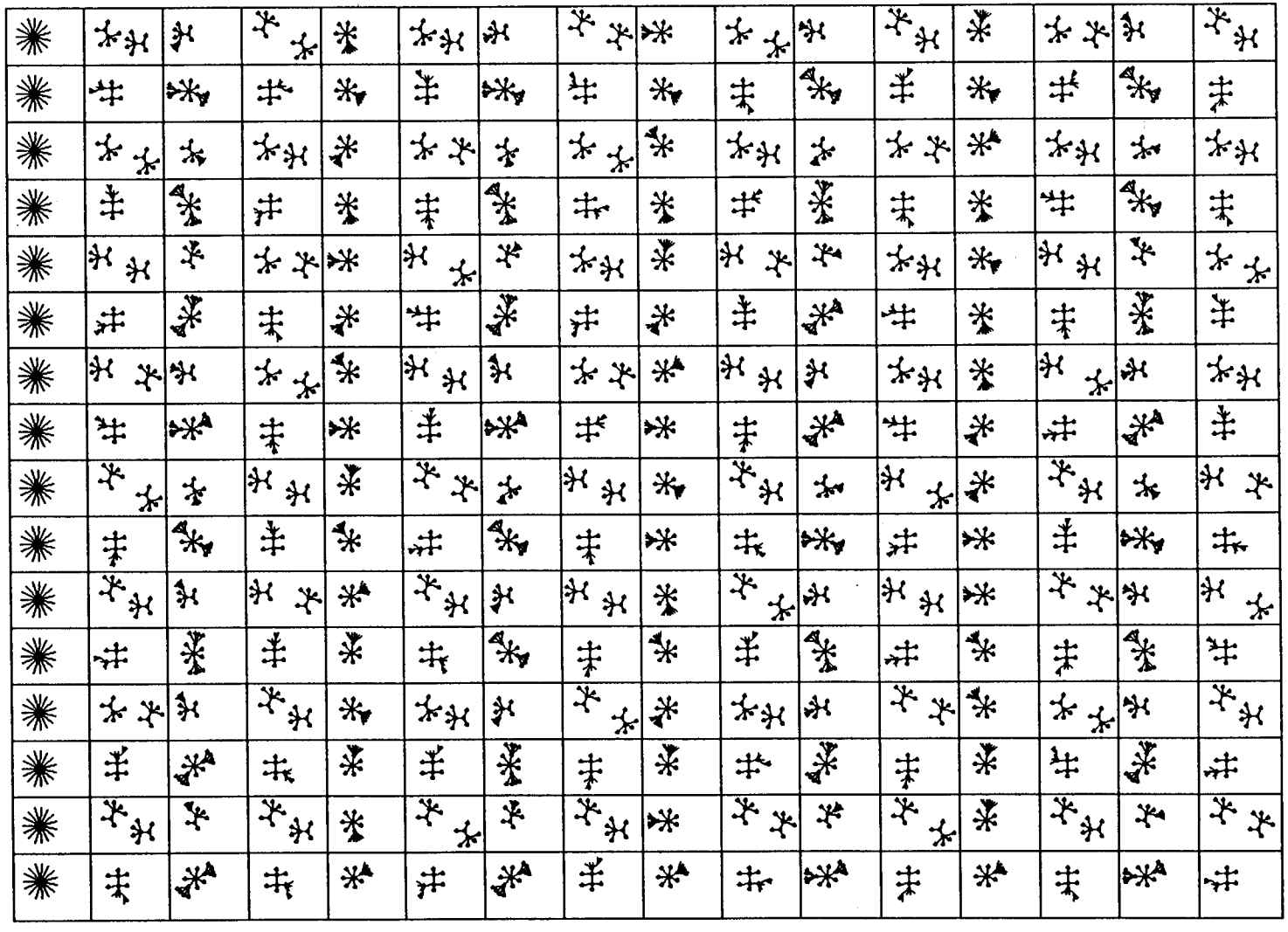
Quit Func New N New P PP++ Show Many Label Circle Info Clear



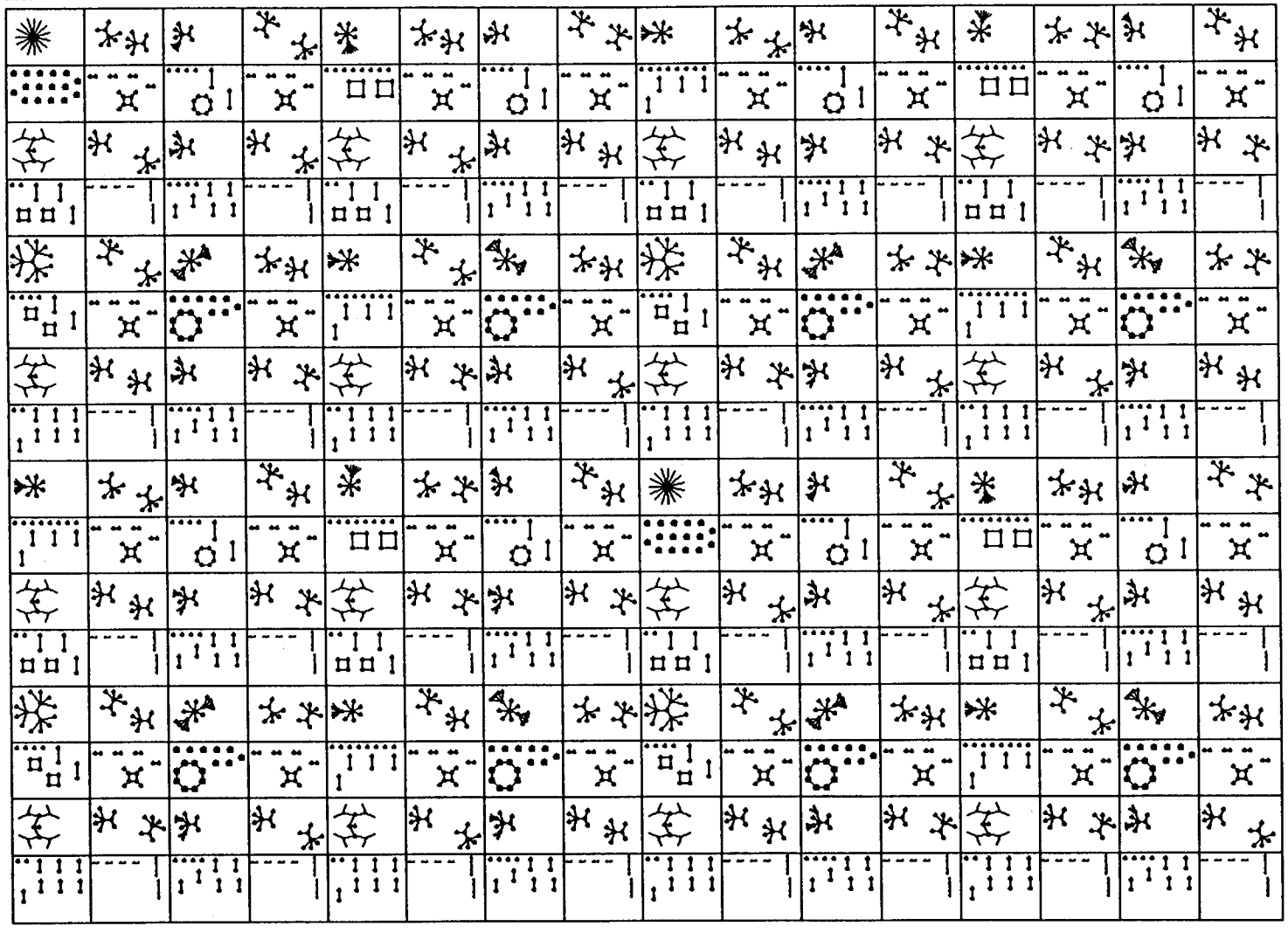
Quit Func New N New P PP++ Show Many Label Circle Info Clear



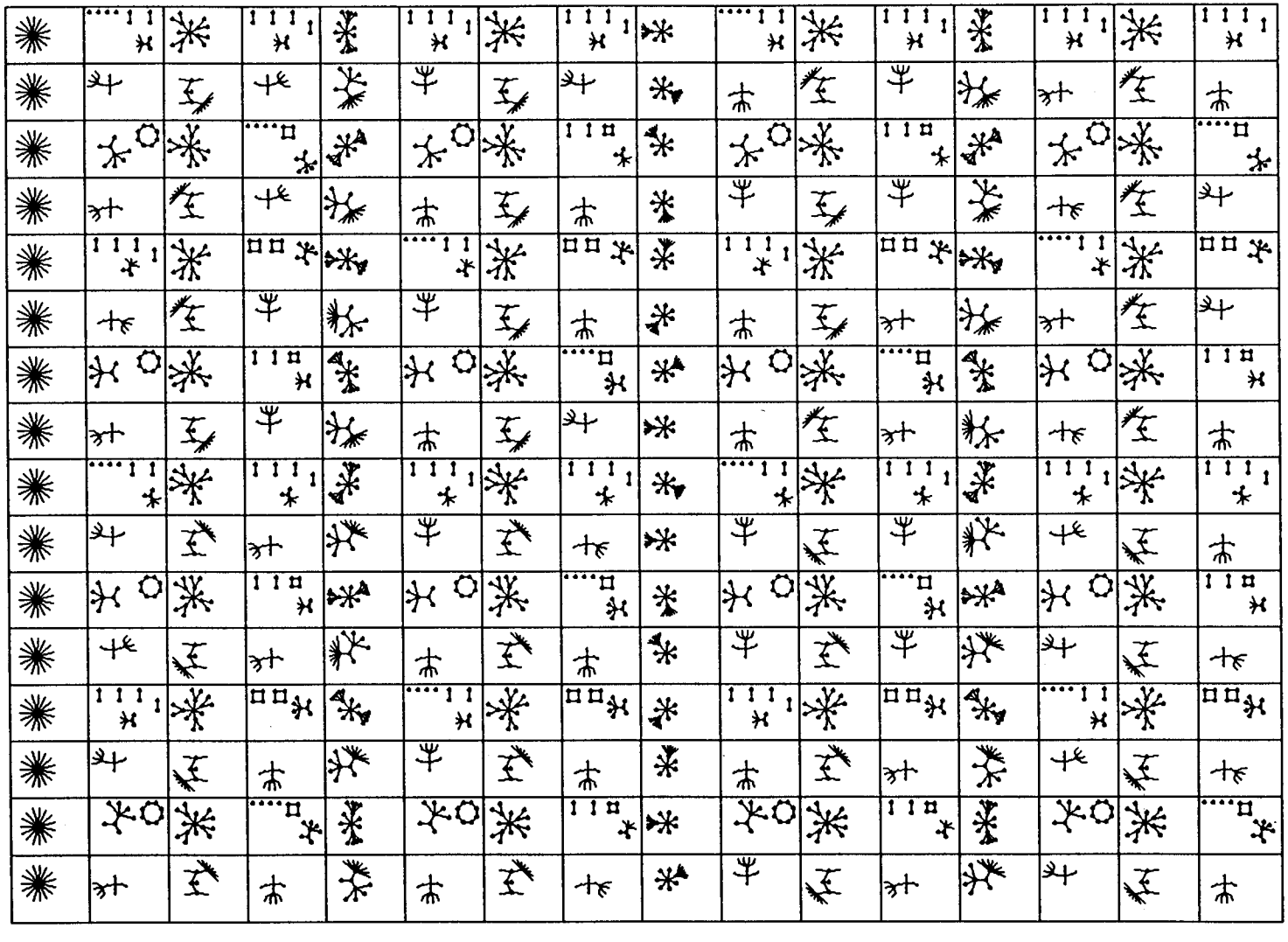
Quit Func New N New P PP++ Show Many Label Circle Info Clear



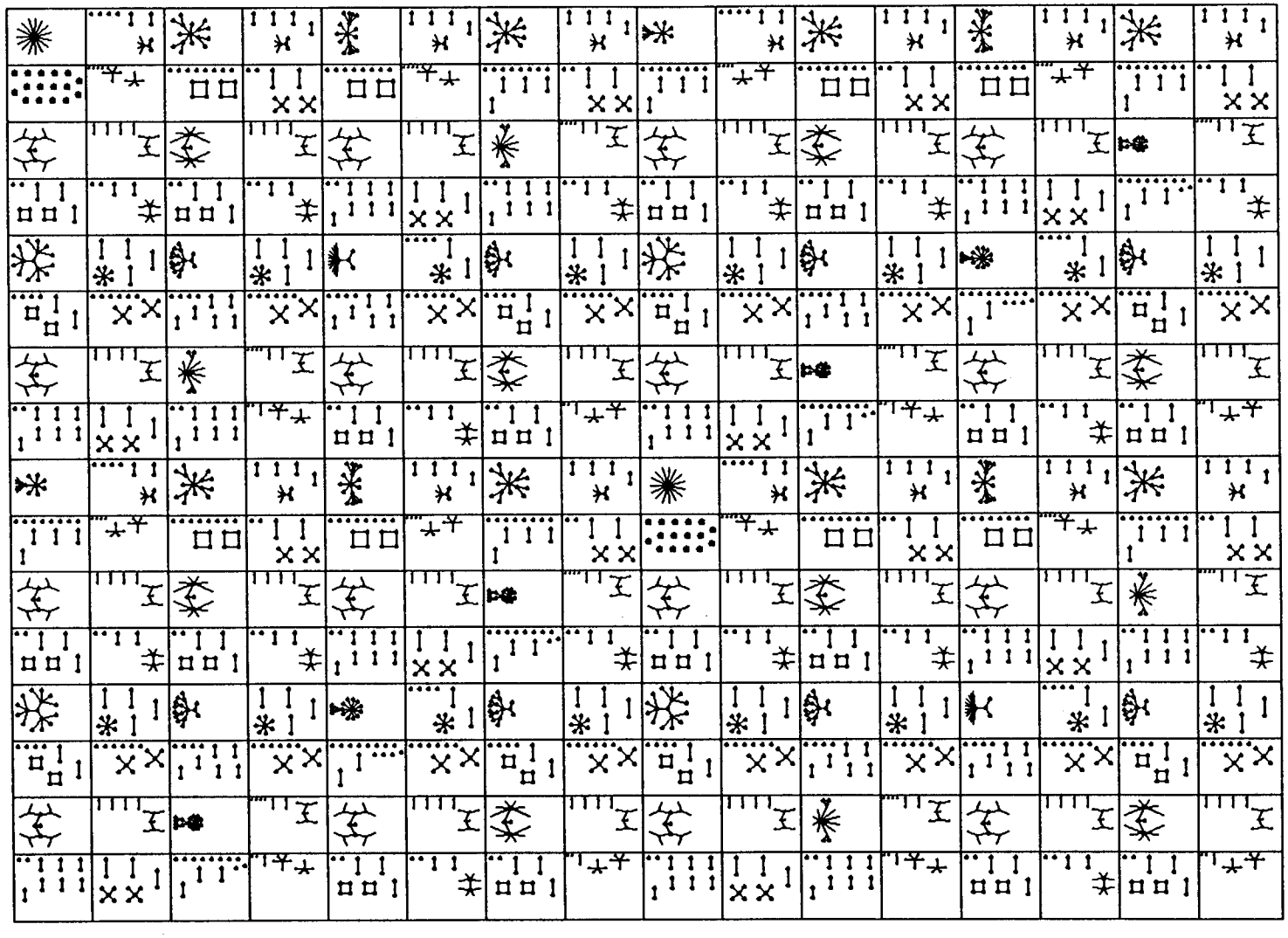
Quit Func New N New P PP++ Show Many Label Circle Info Clear



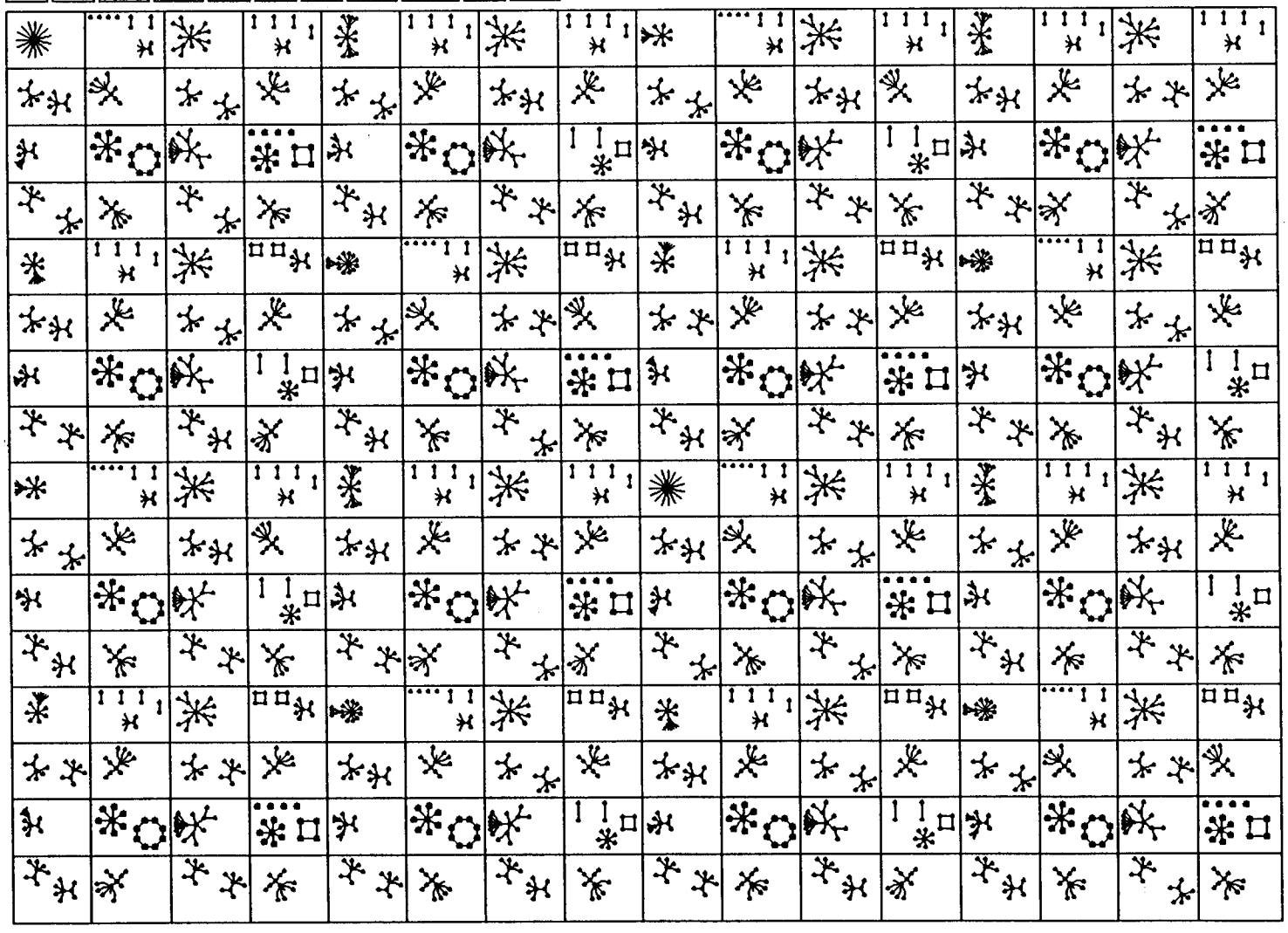
Quit Func New N New P PP++ Show Many Label Circle Info Clear



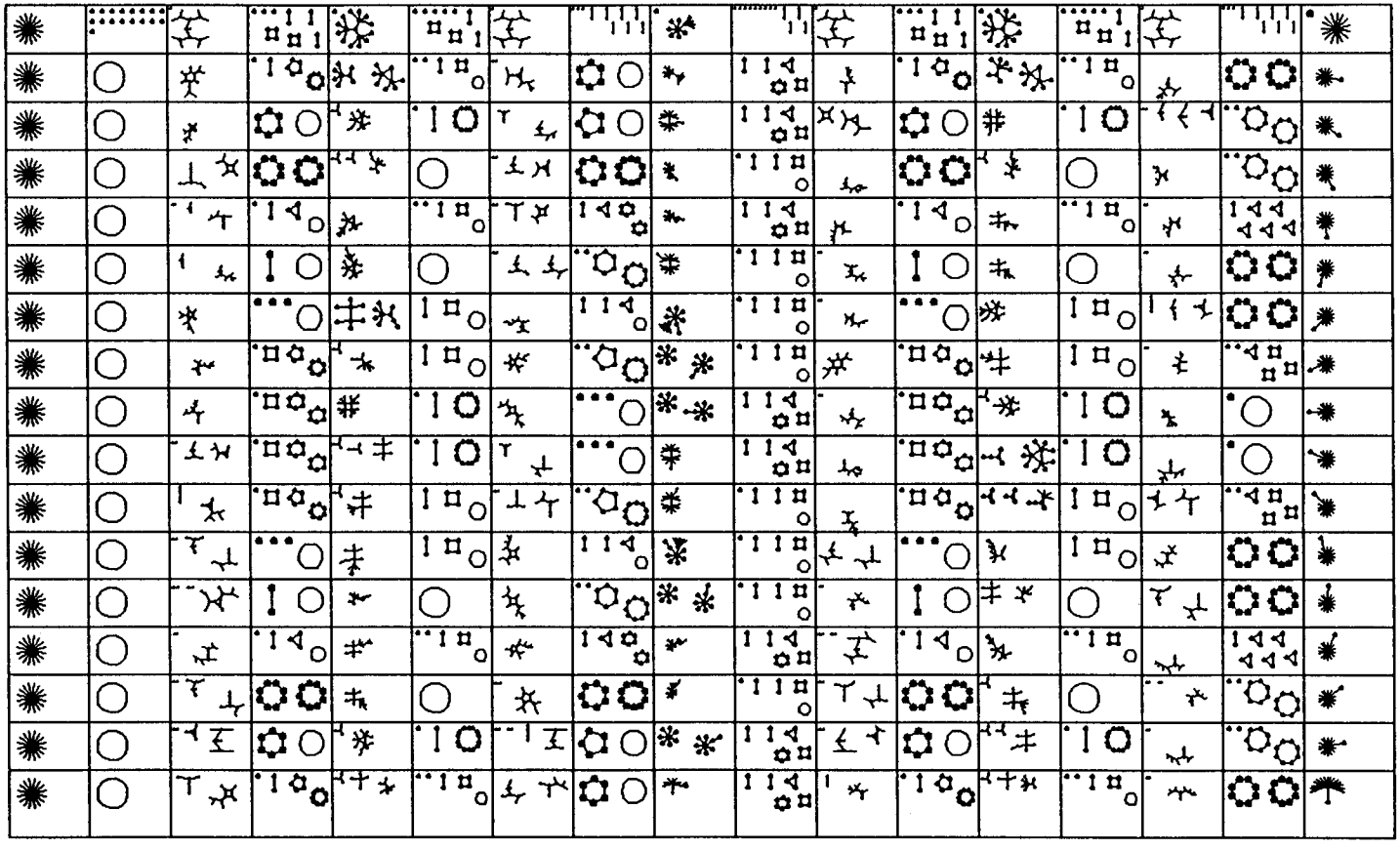
Quit Func New N New P PP++ Show Many Label Circle Info Clear



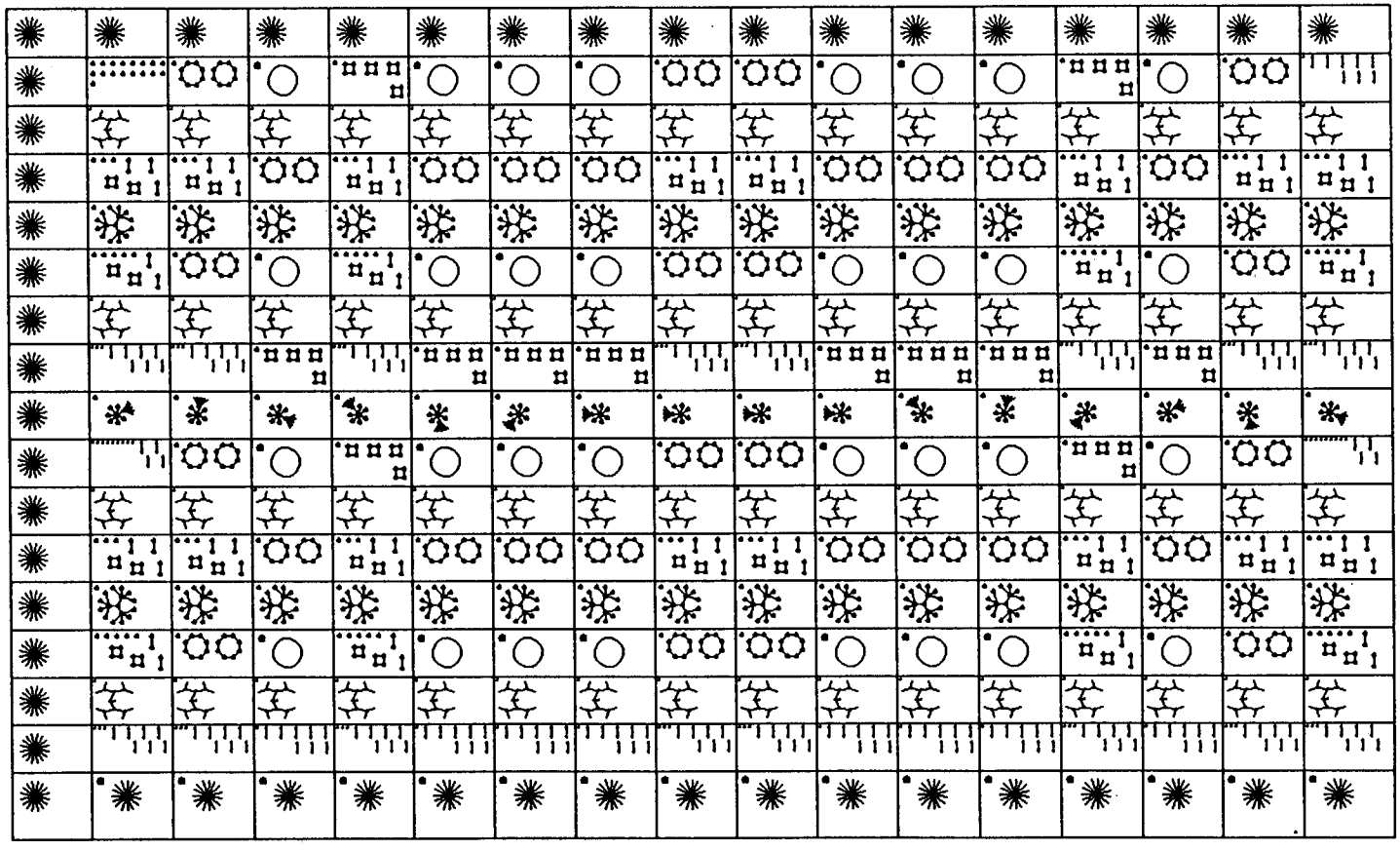
Quit Func New N New P PP++ Show Many Label Circle Info Clear



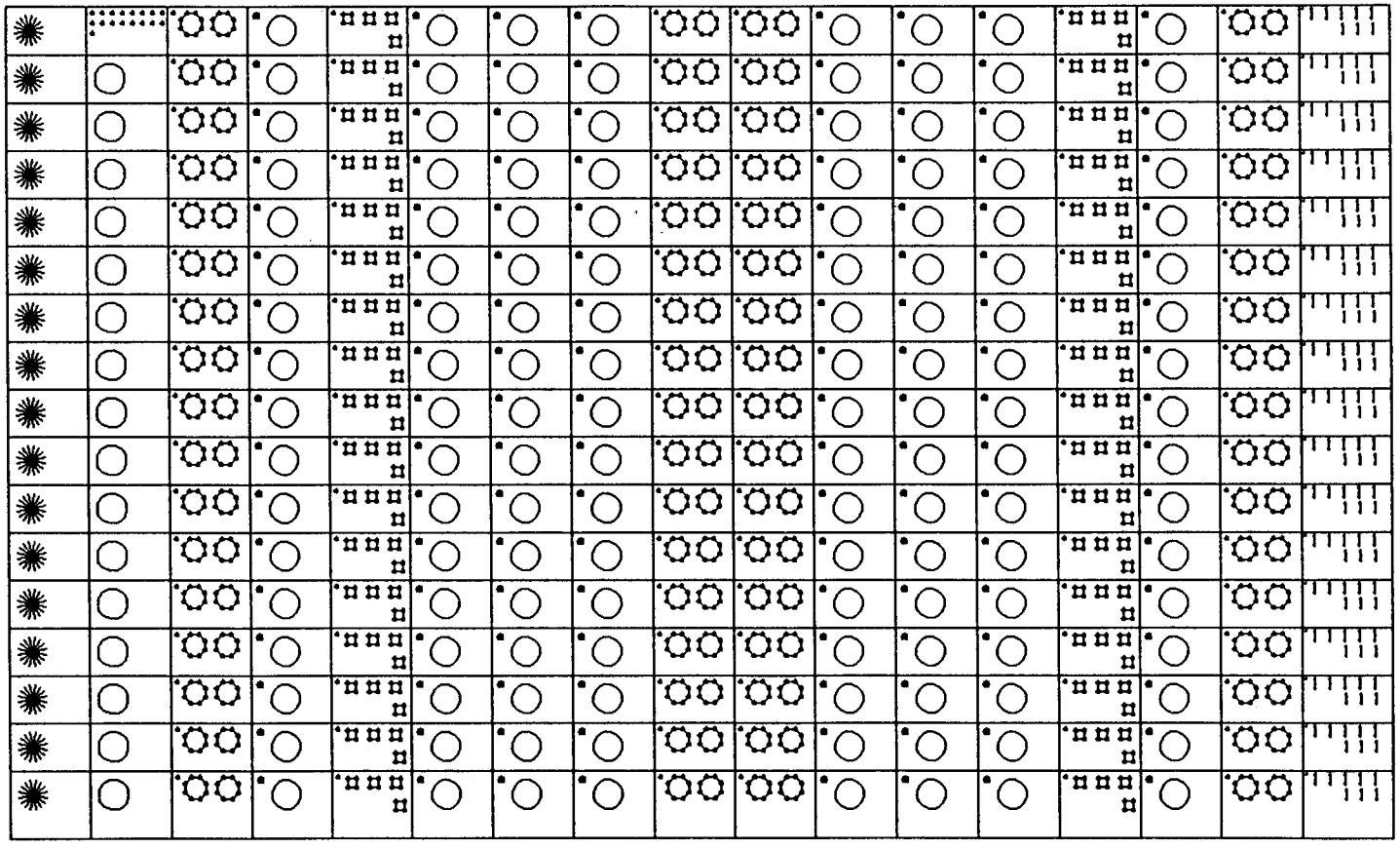
Quit Func New N New P PP↔ Show Many Label Circle Info Clear



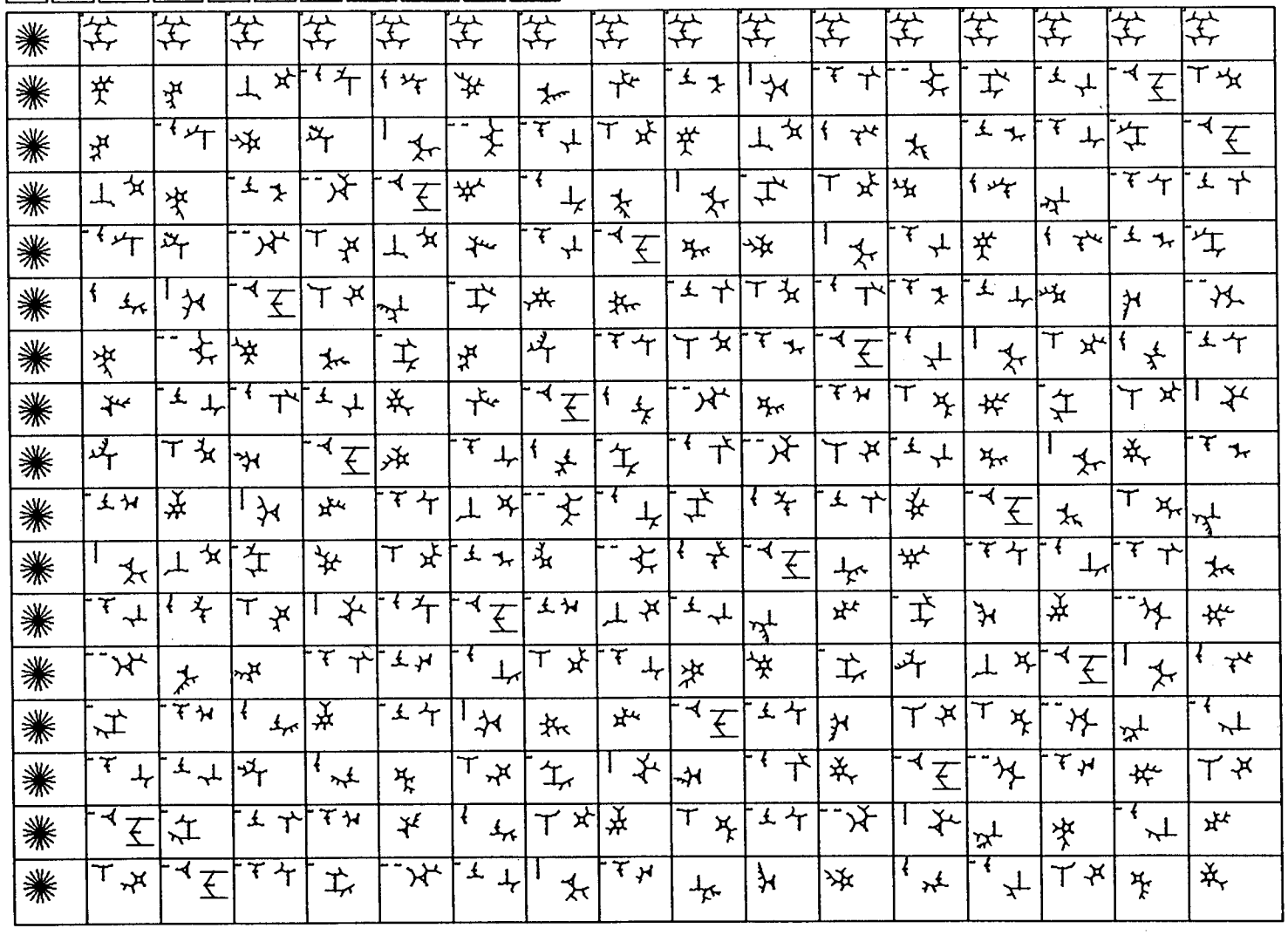
Quit Func New N New P PP++ Show Many Label Circle Info Clear



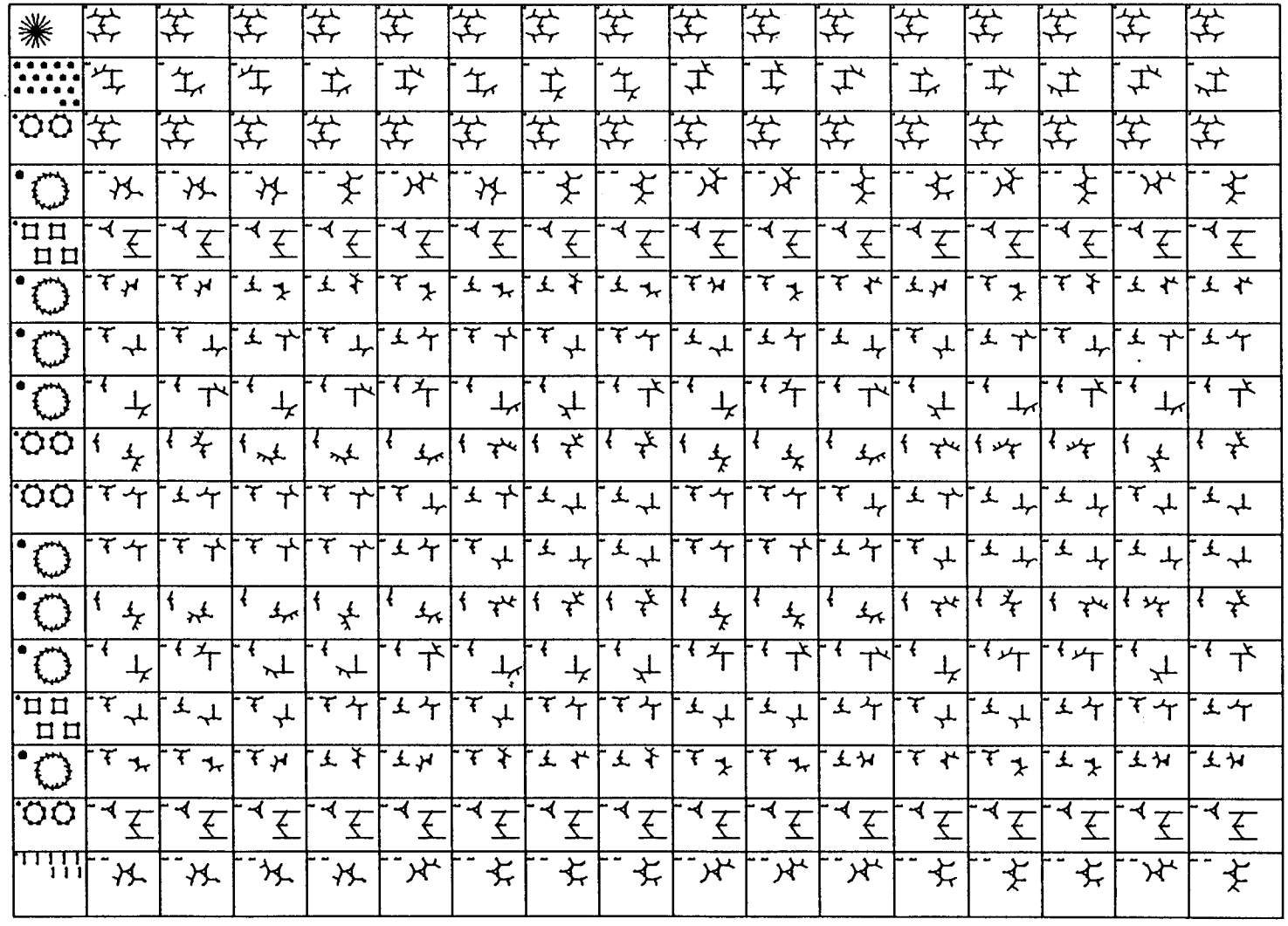
Quit Func New N New P PP++ Show Many Label Circle Info Clear



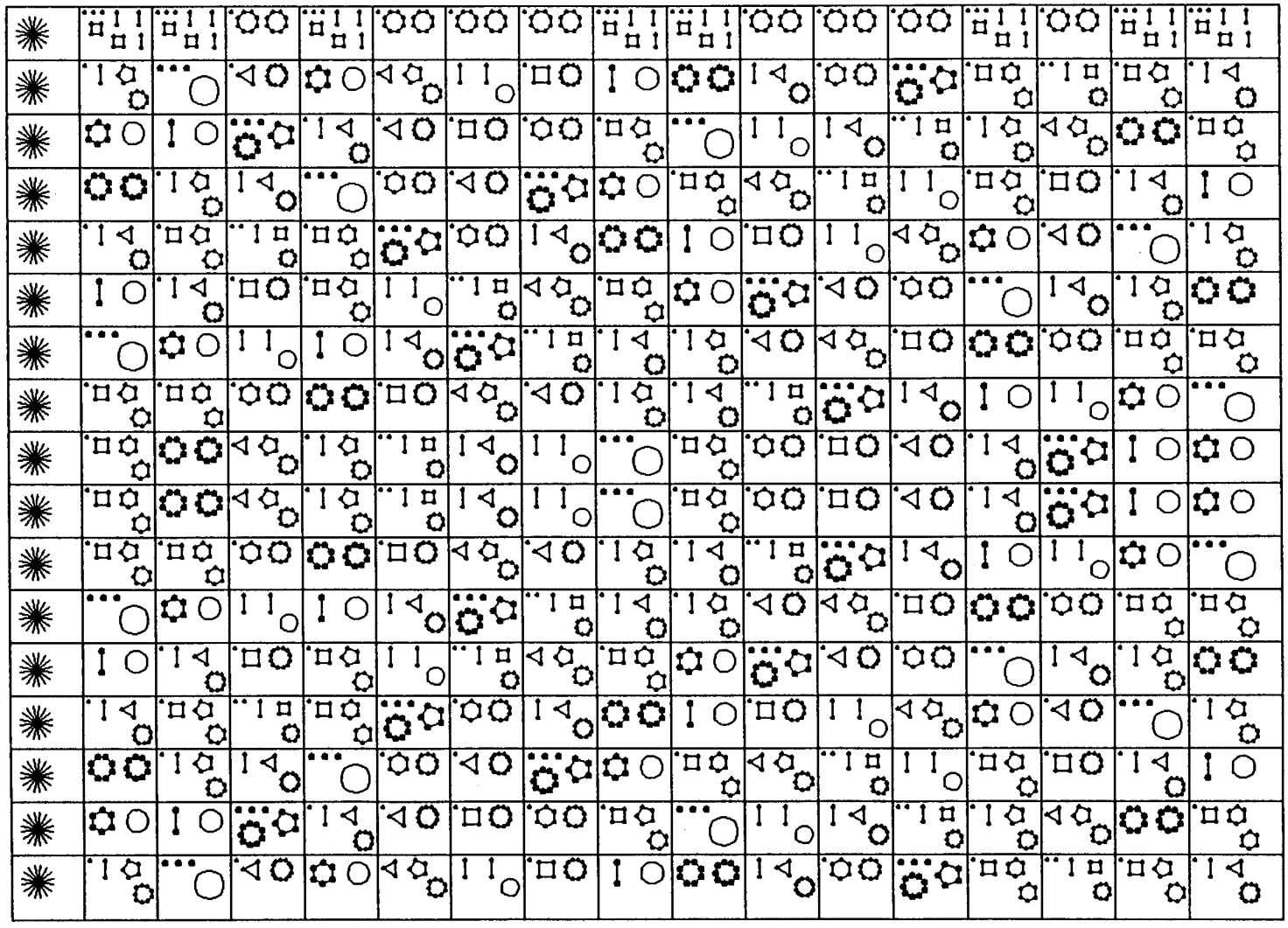
Quit Func New N New P PP++ Show Many Label Circle Info Clear



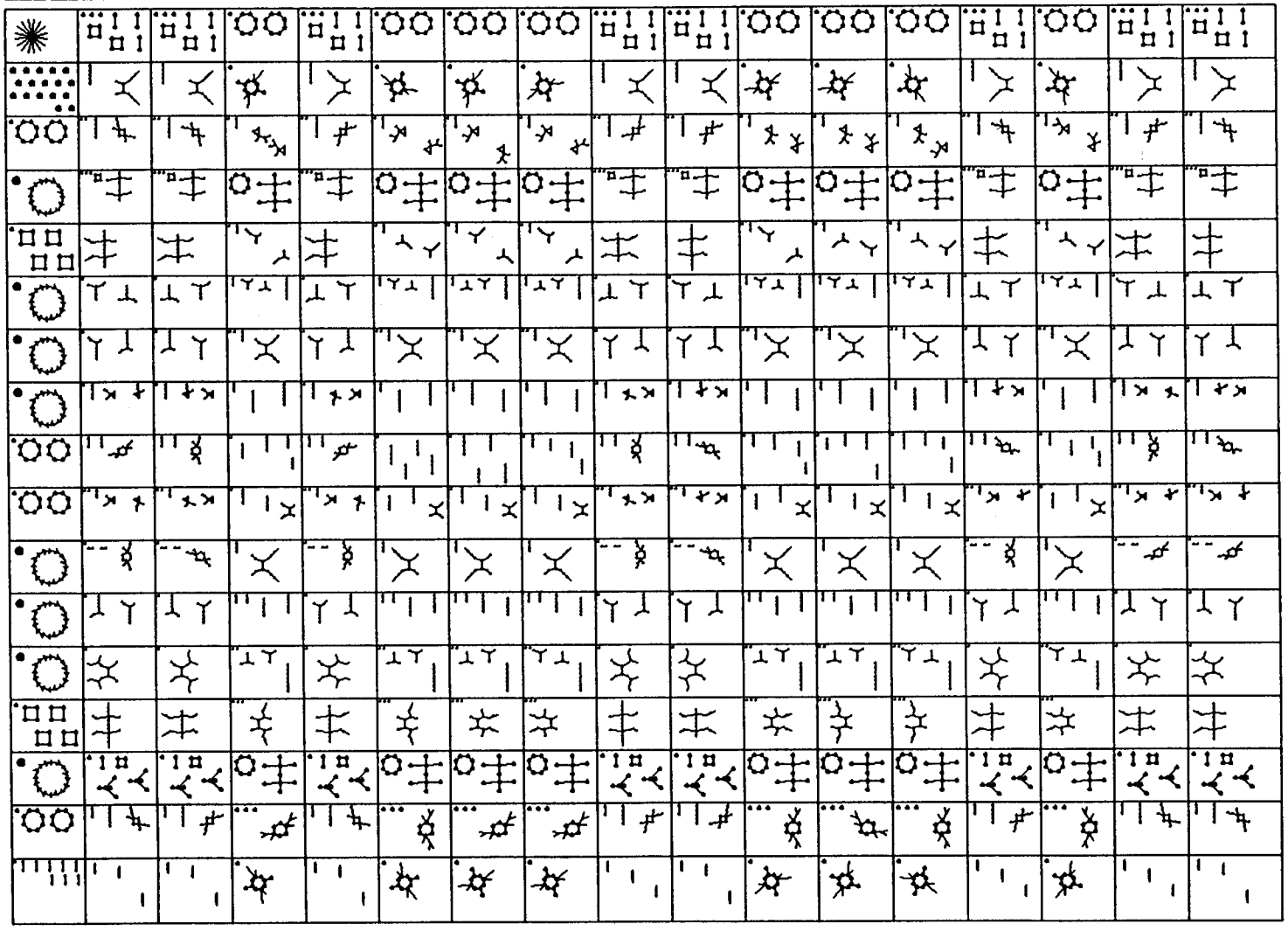
Quit | Func | New N | New P | PP++ | Show | Many | Label | Circle | Info | Clear



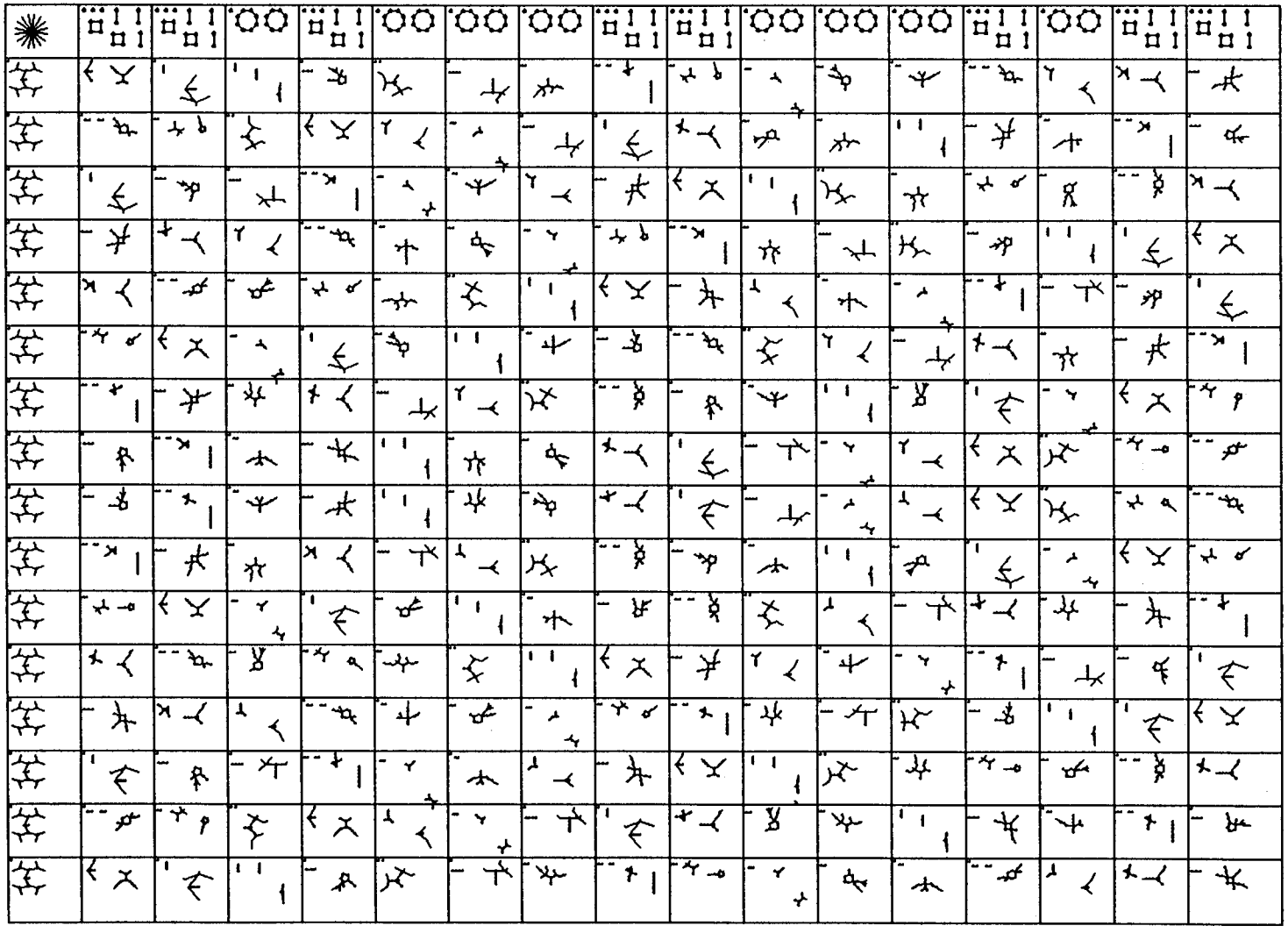
Quit Func New N New P PP++ Show Many Label Circle Info Clear



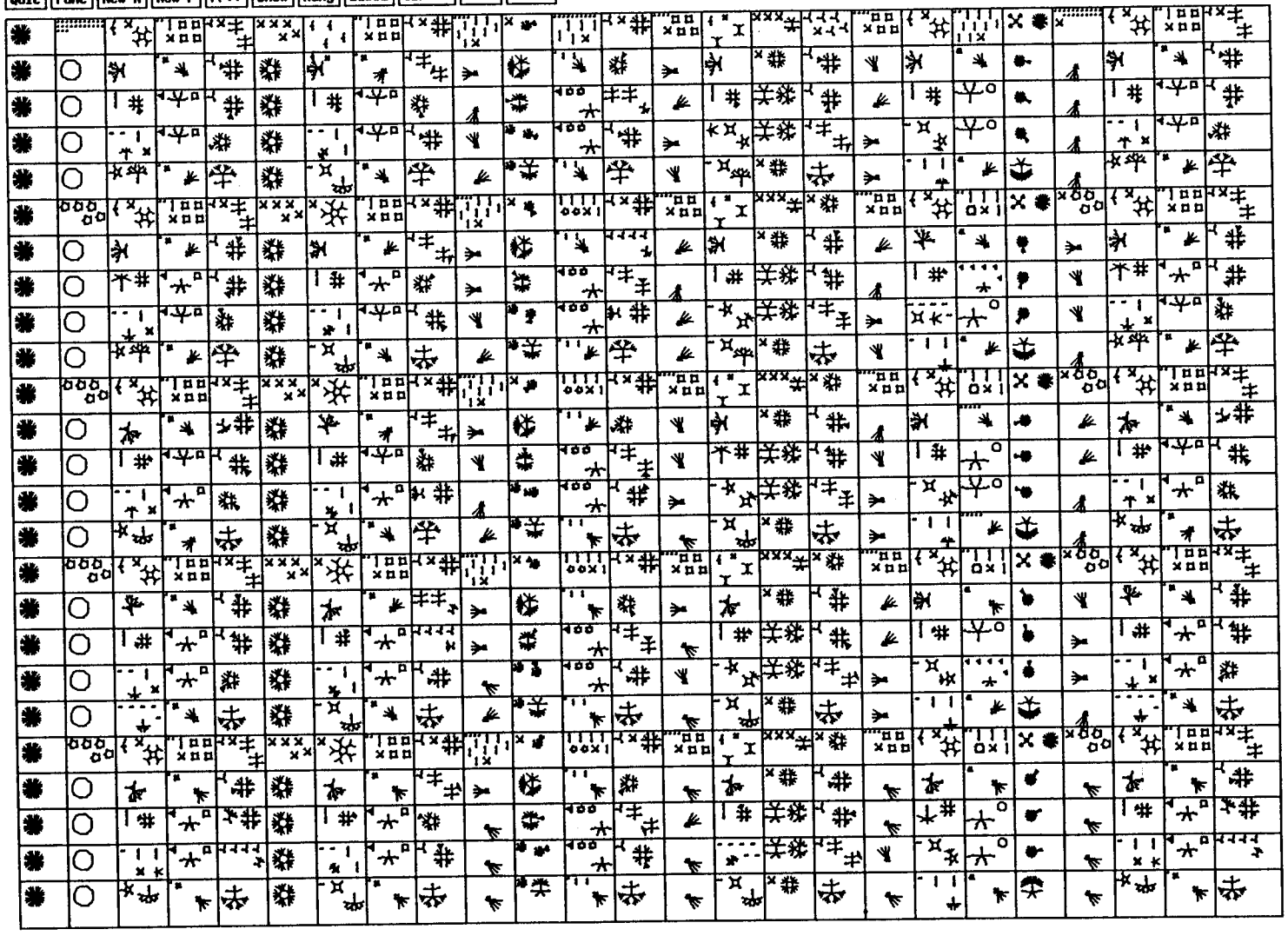
Quit Func New N New P PP++ Show Many Label Circle Info Clear



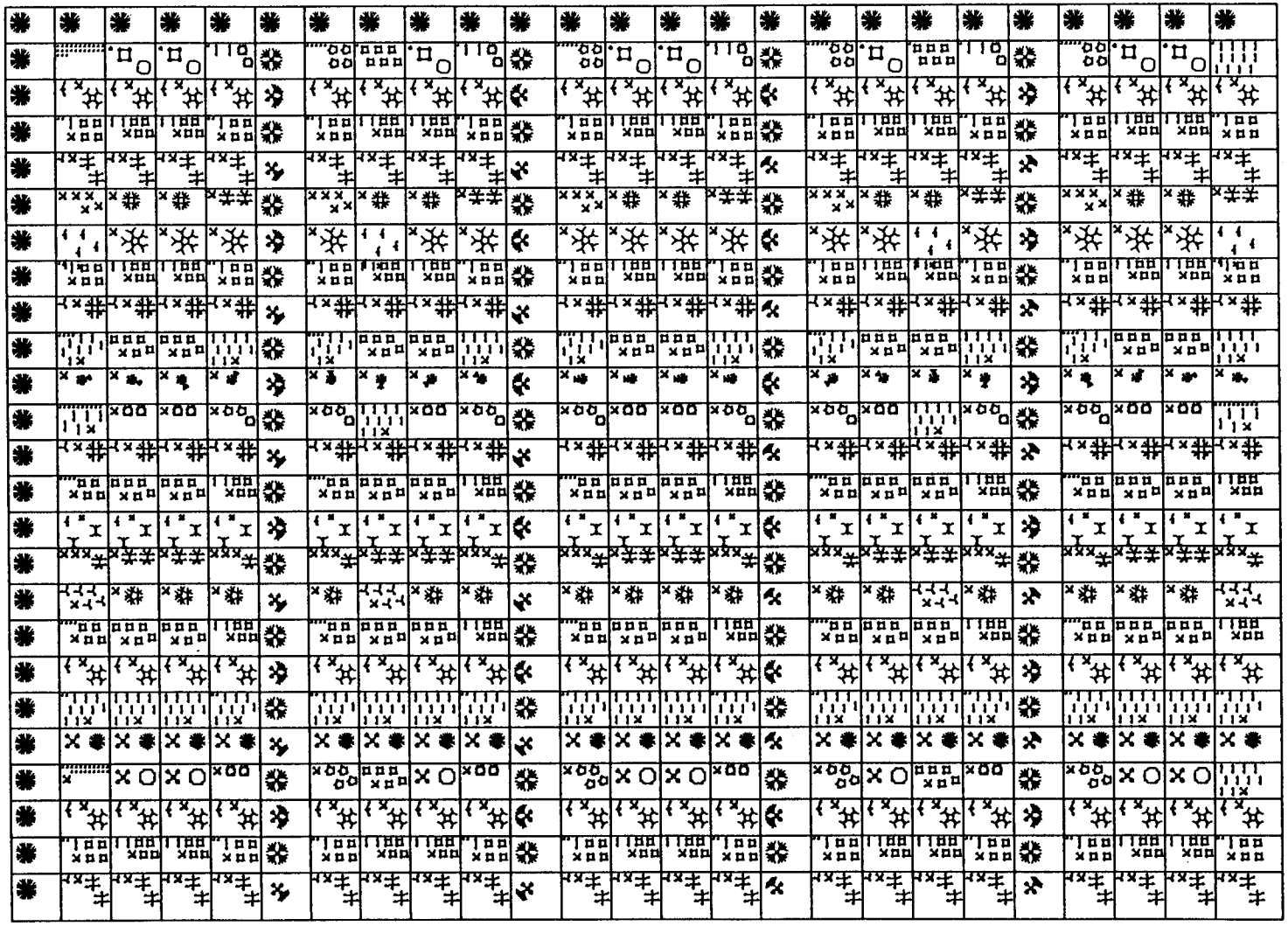
Quit Func New N New P PP++ Show Many Label Circle Info Clear



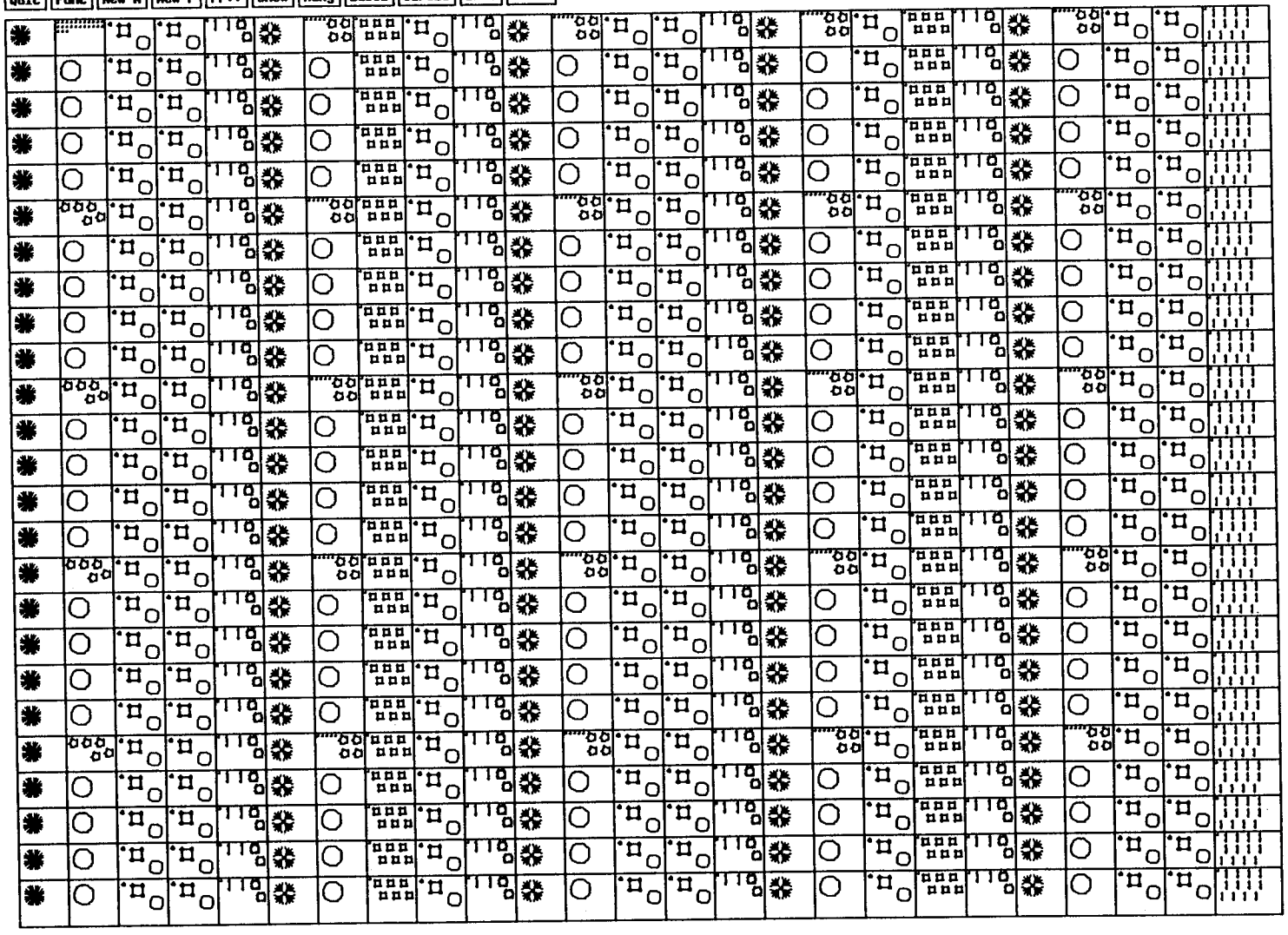
Quit Func New N New P PP++ Show Many Label Circle Info Clear



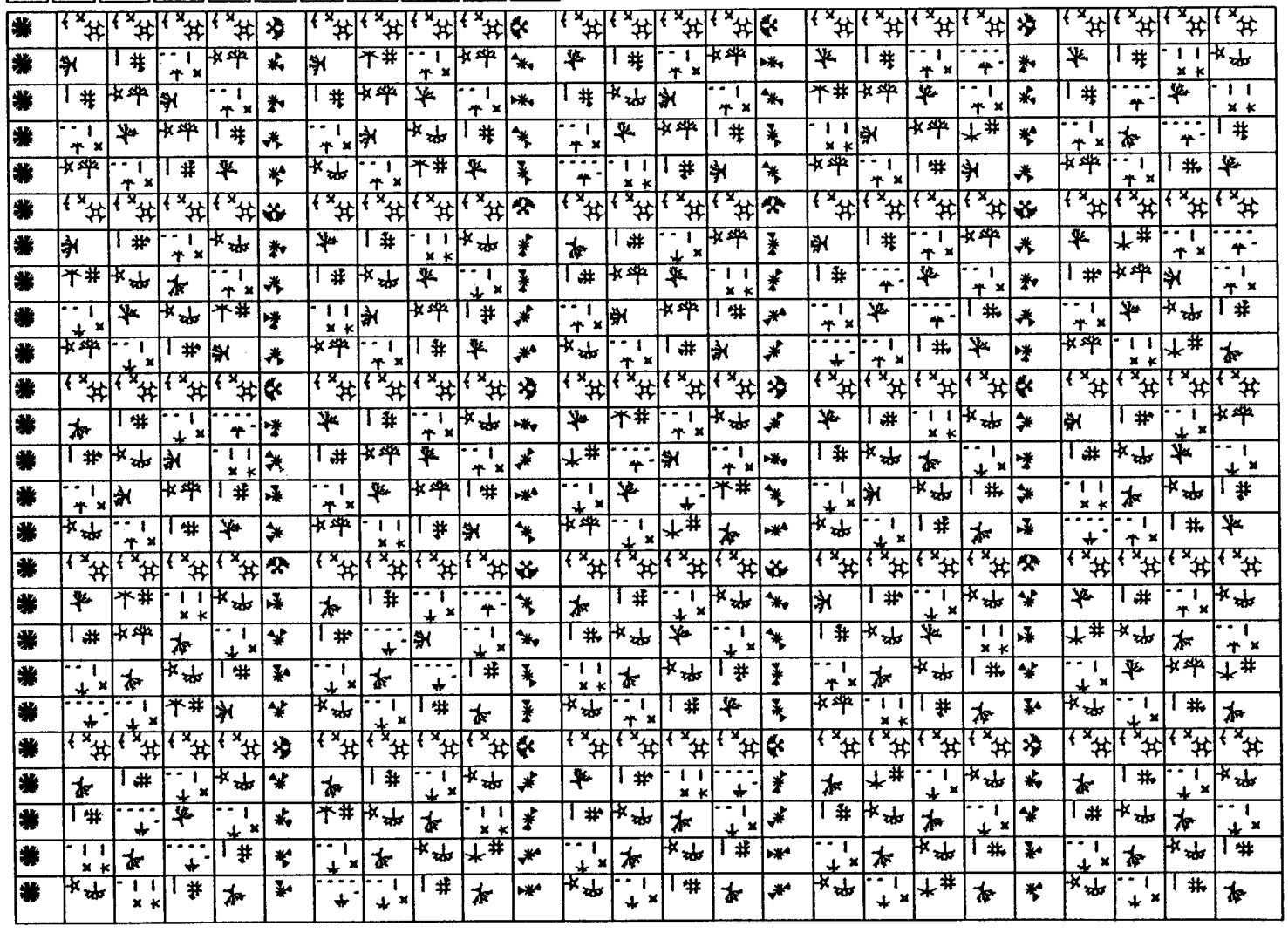
Quit Func New N New P PP++ Show Many Label Circle Info Clear



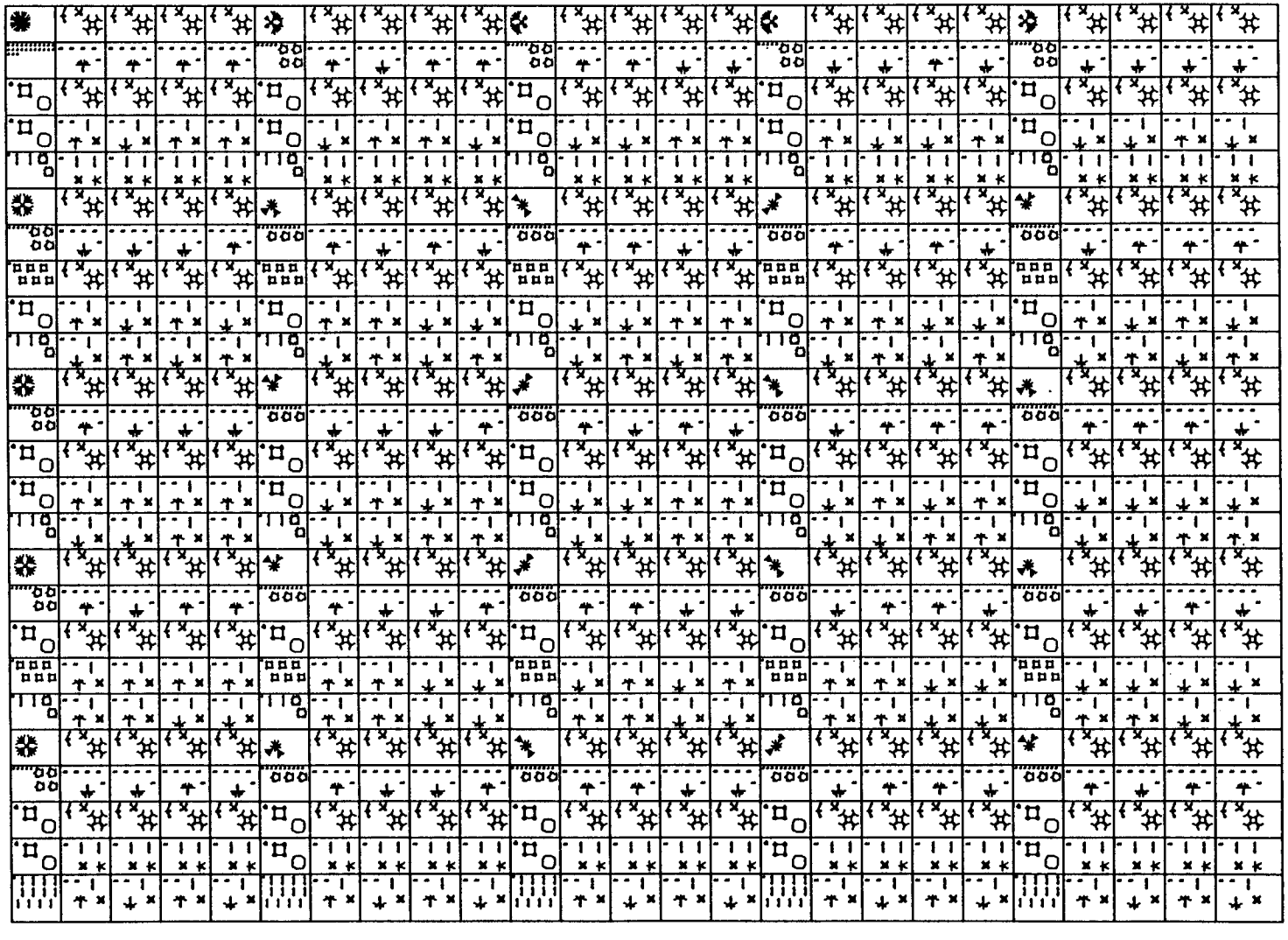
Quit Func New N New P PP++ Show Many Label Circle Info Clear



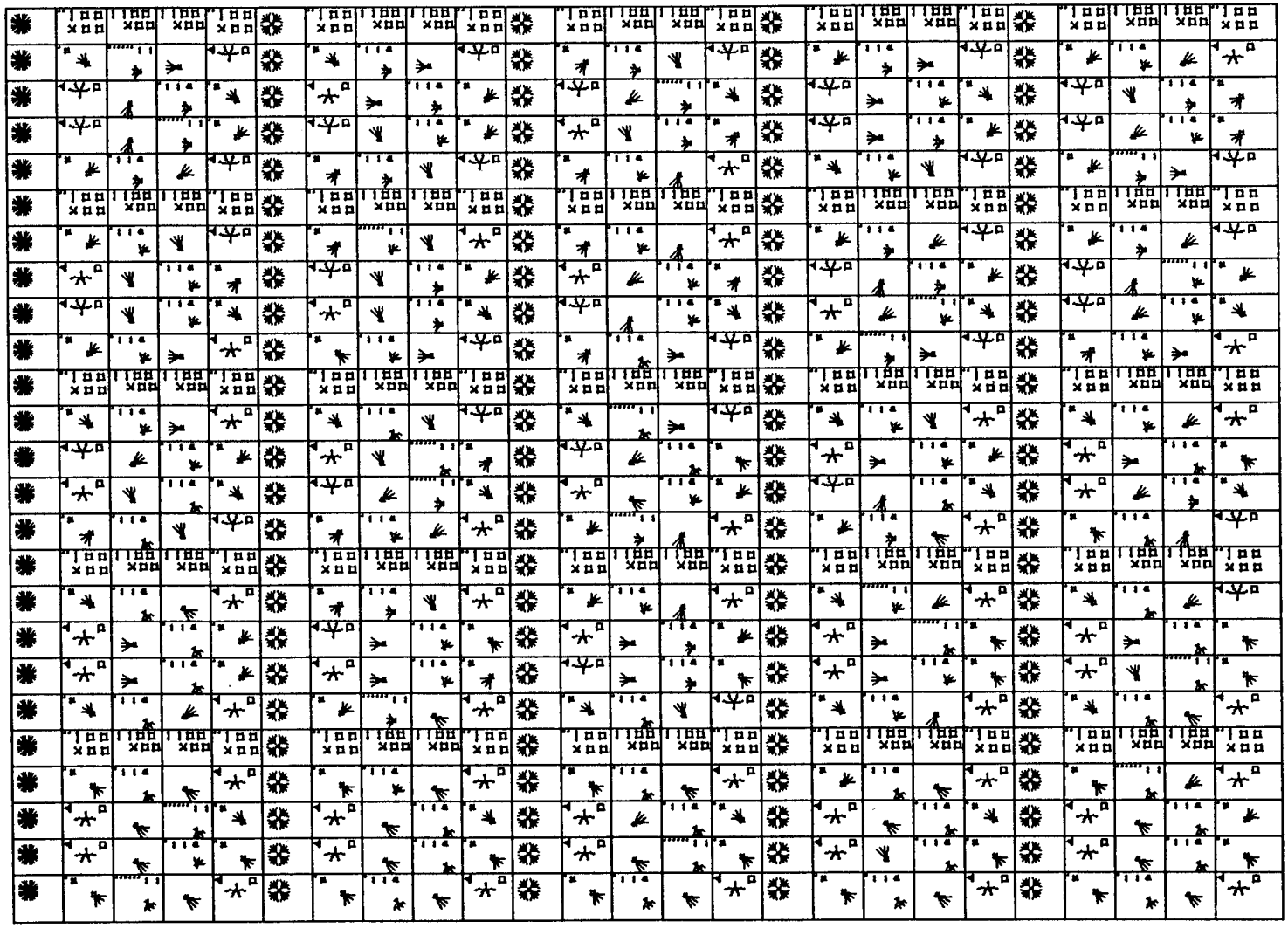
Quit Func New N New P PP++ Show Many Label Circle Info Clear



Quit Func New N New P PP++ Show Many Label Circle Info Clear

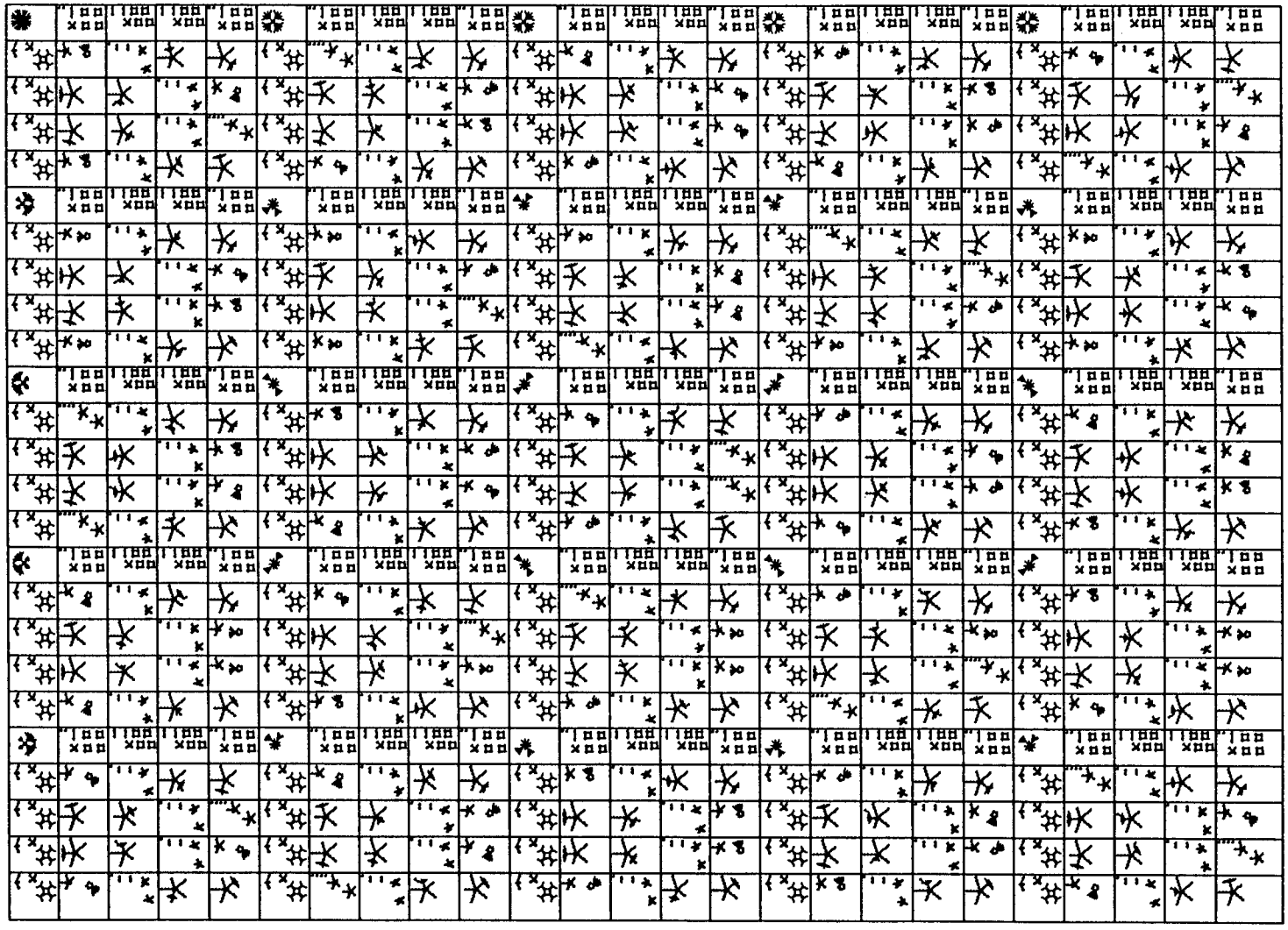


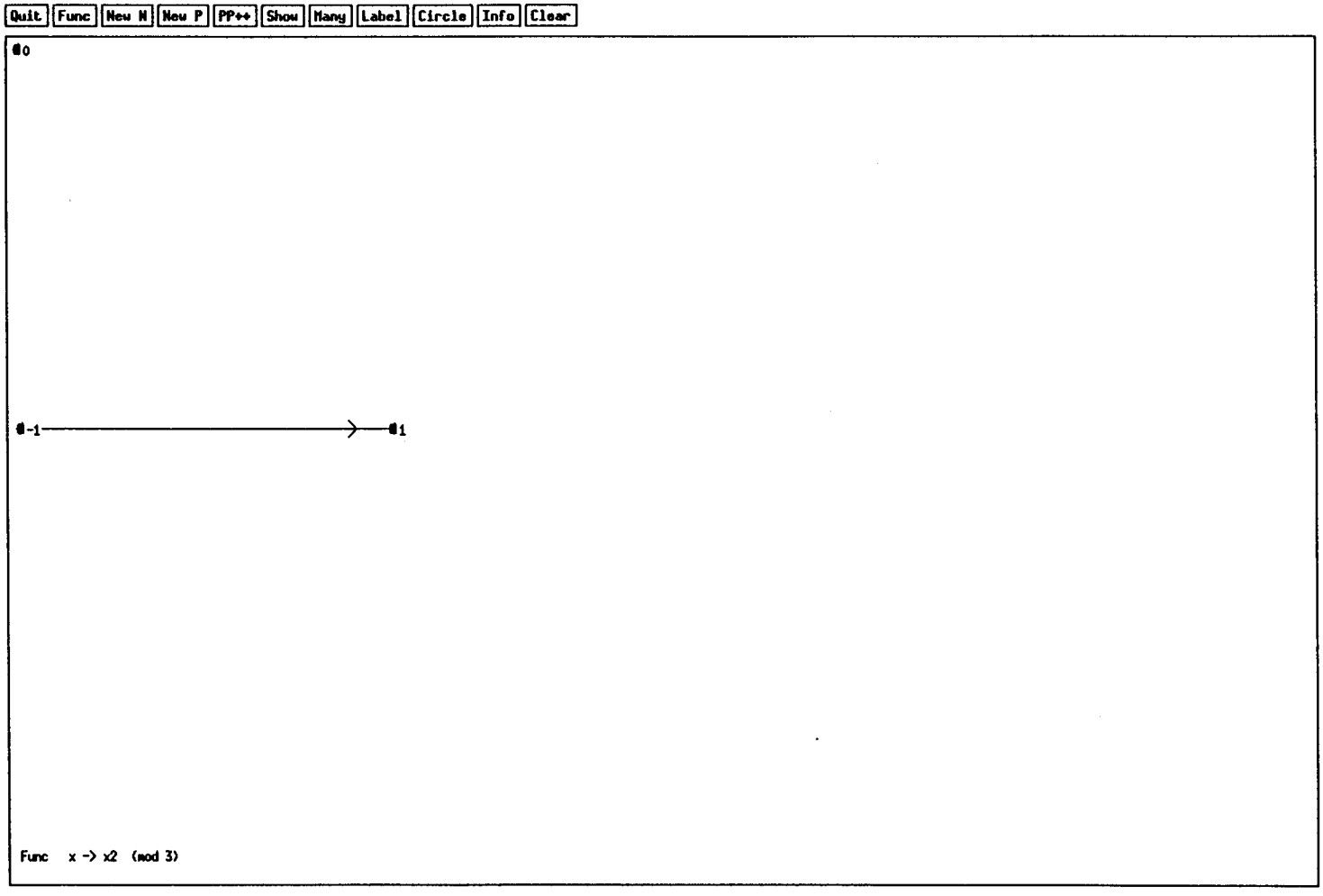
Quit Func New N New P PP++ Show Many Label Circle Info Clear

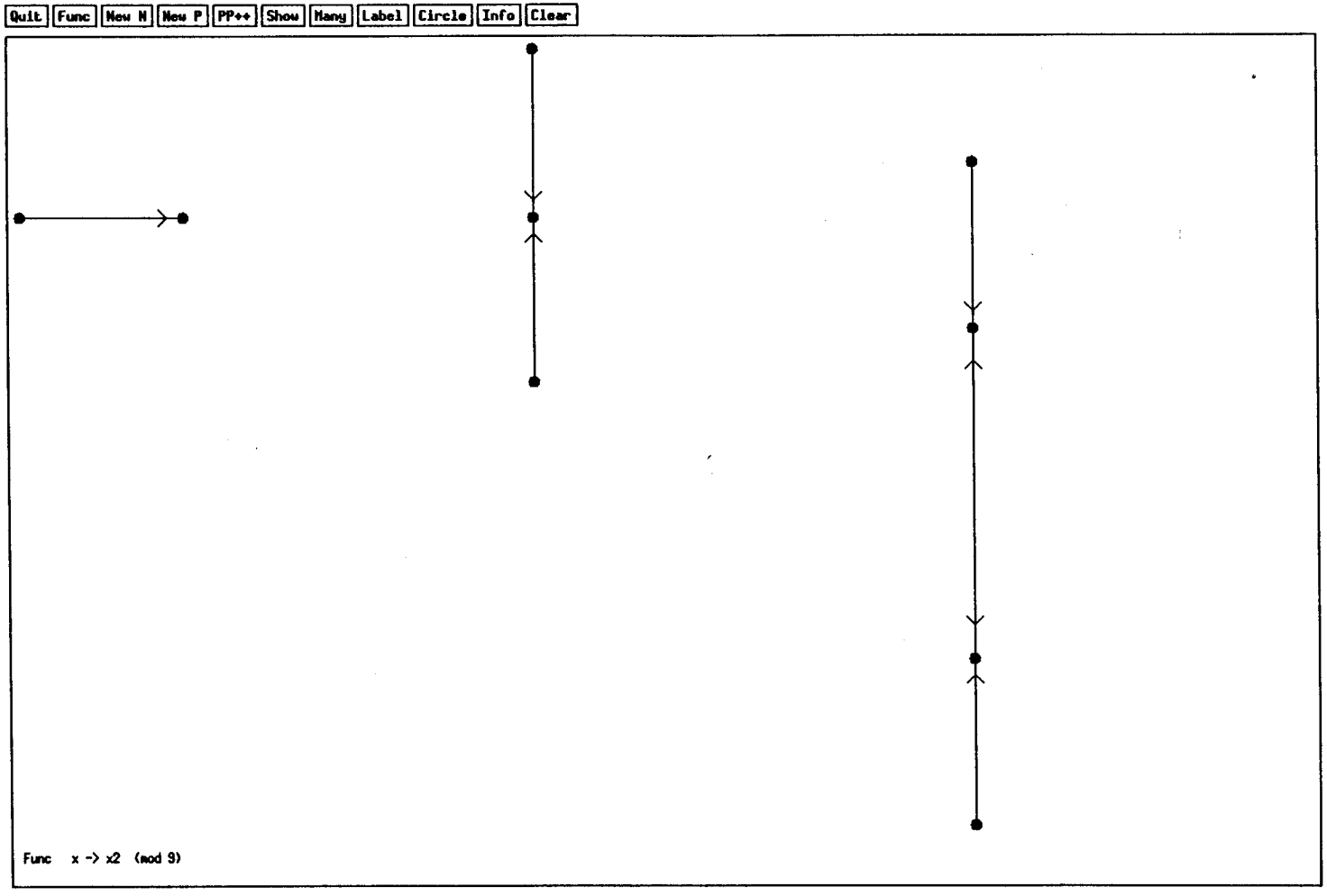


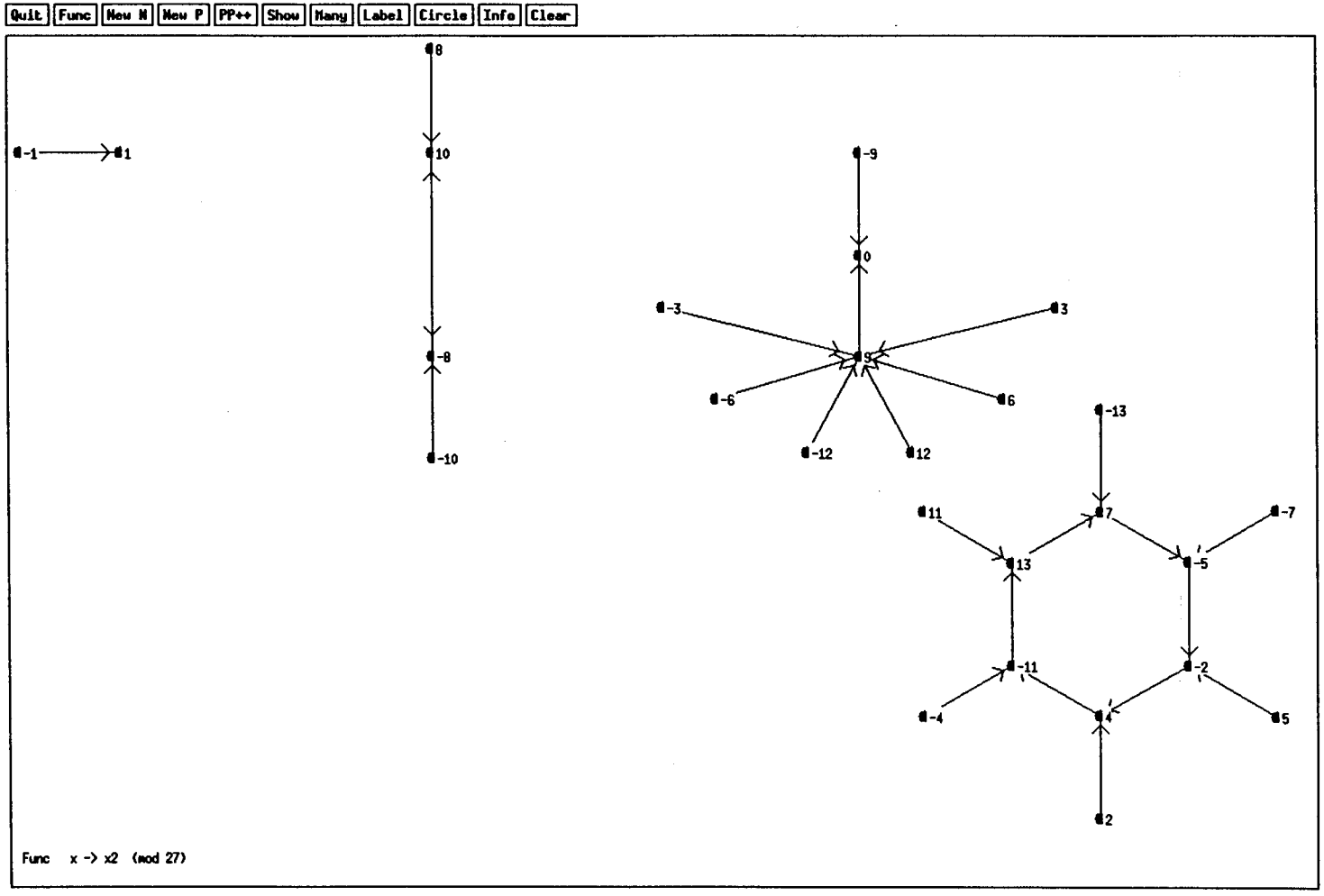
Quit Func New N New P PP++ Show Many Label Circle Info Clear

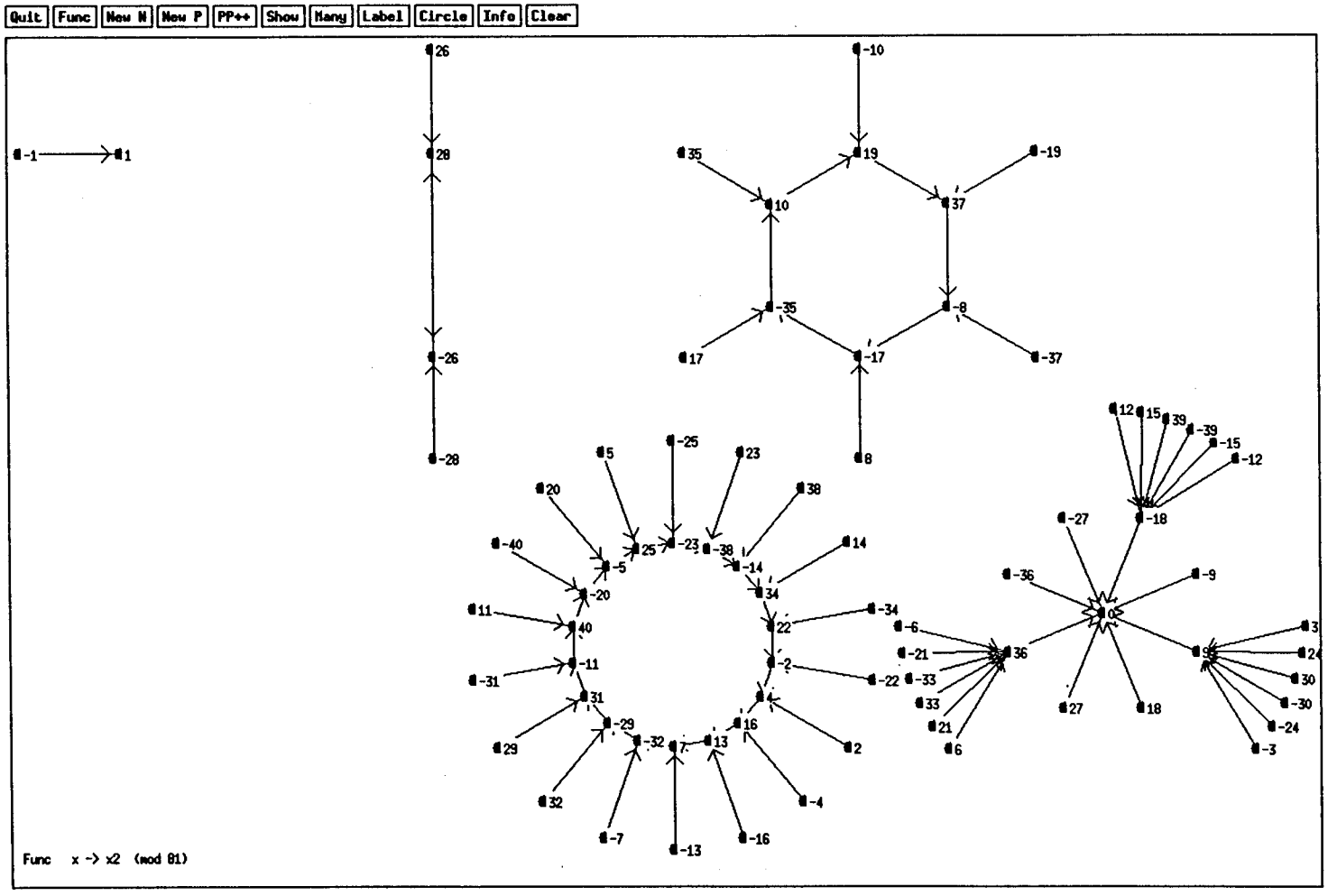
Quit Func New N New P PP++ Show Many Label Circle Info Clear

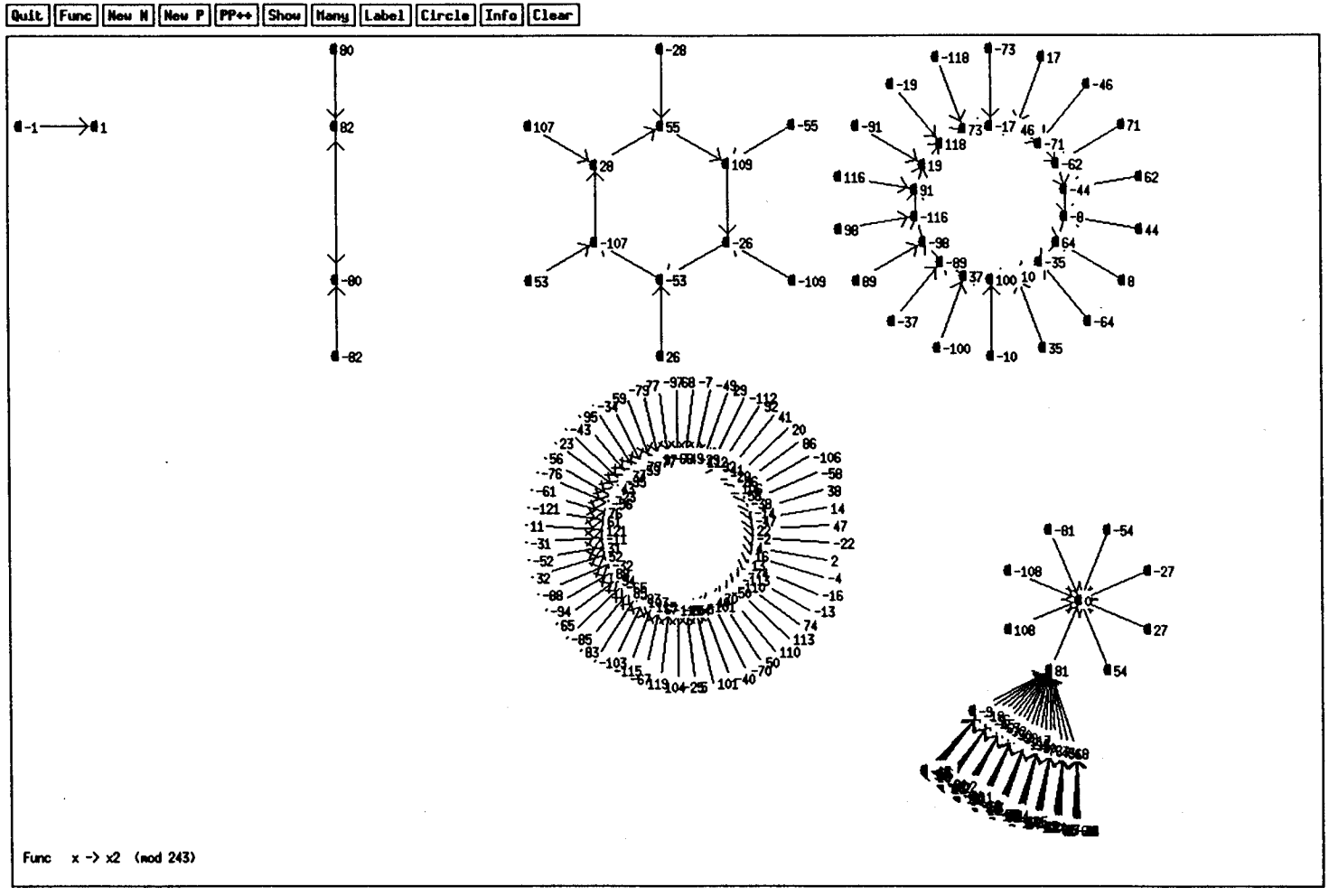


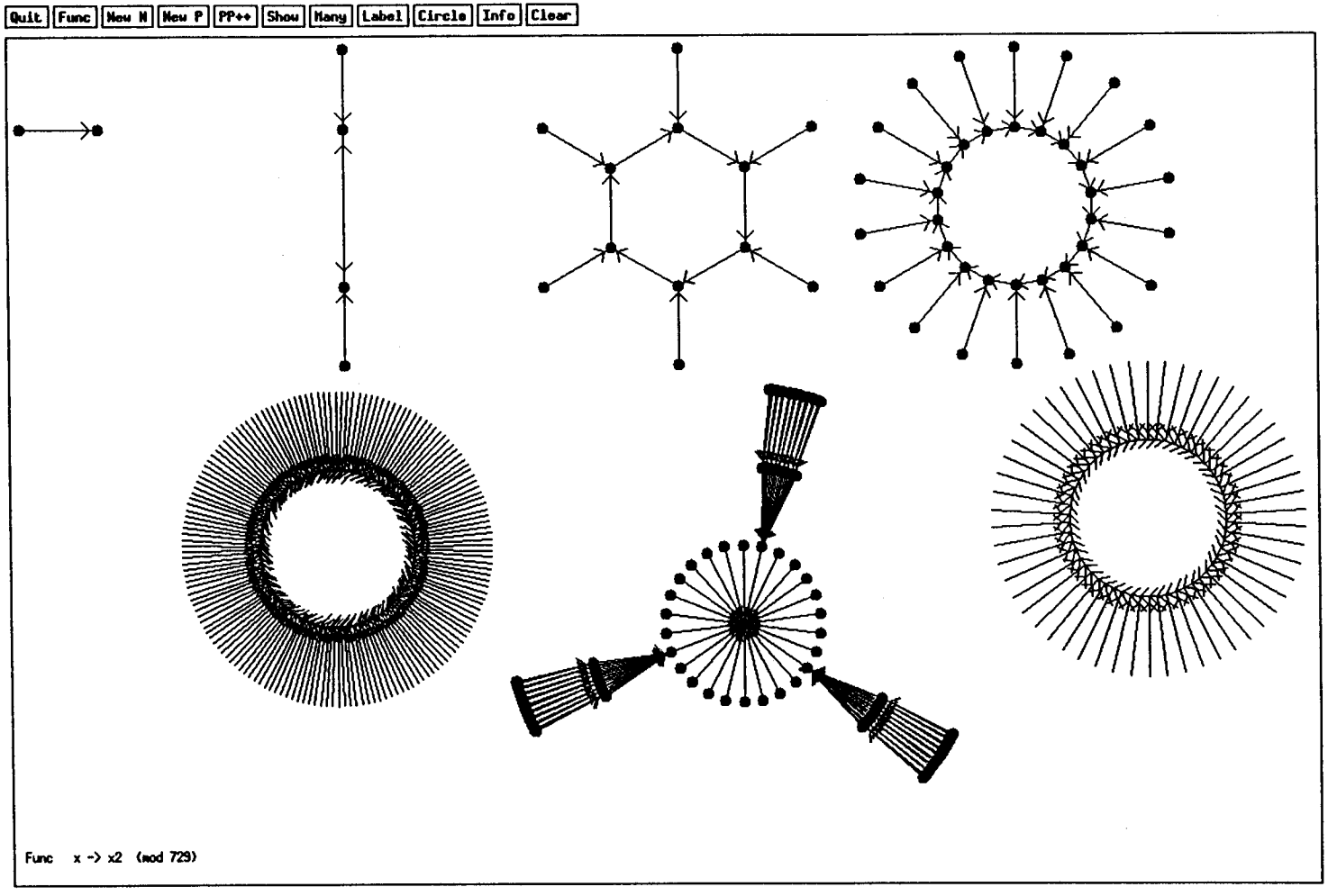

Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
0
-1
1
Func x -> x2 (mod 3)

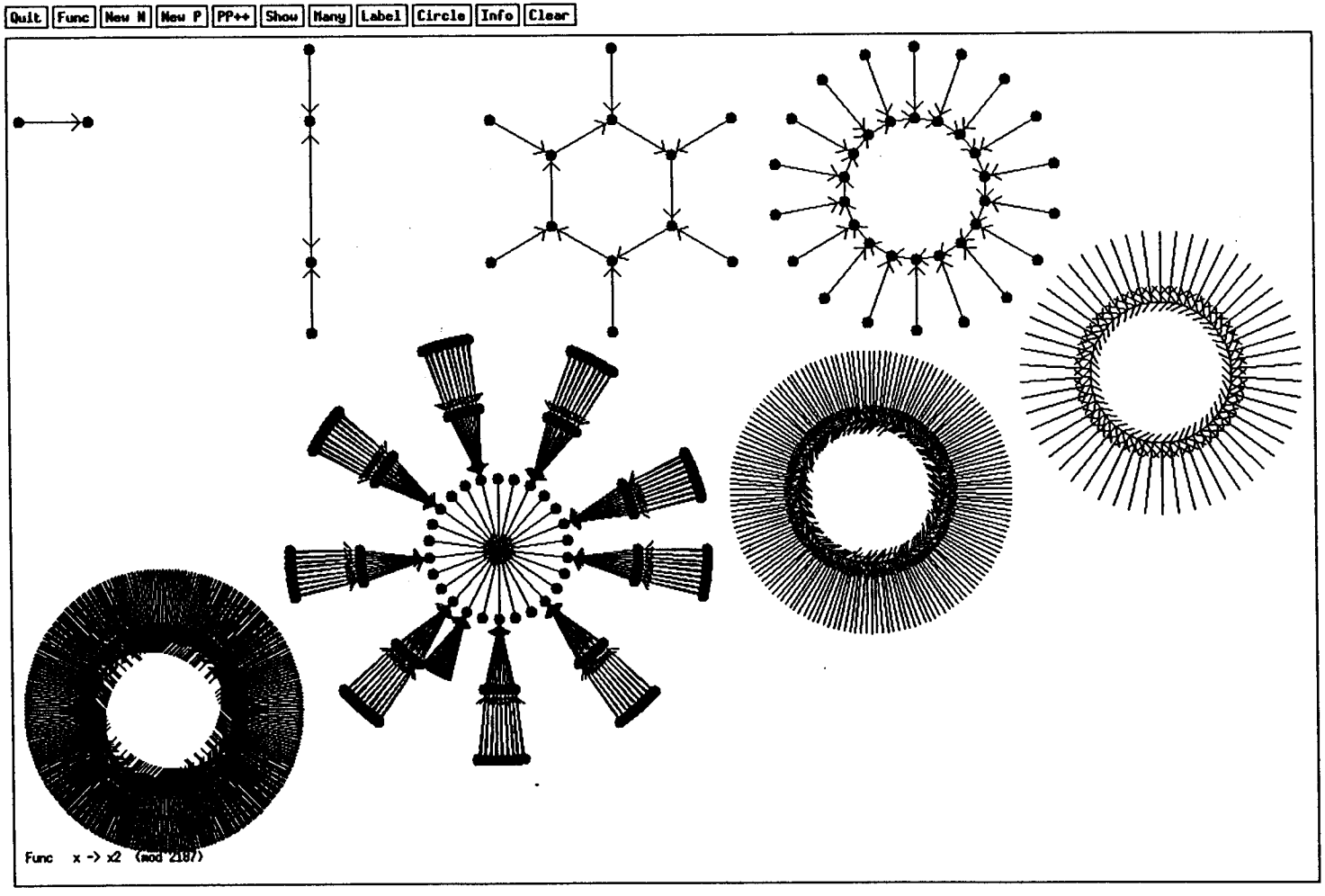

Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
Func x -> x2 (mod 9)


Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
-1
1
8
10
-8
-10
-9
0
-3
3
9
-6
6
-12
12
-13
11
7
-7
13
-5
-11
-2
-4
4
5
2
Func x -> x2 (mod 27)

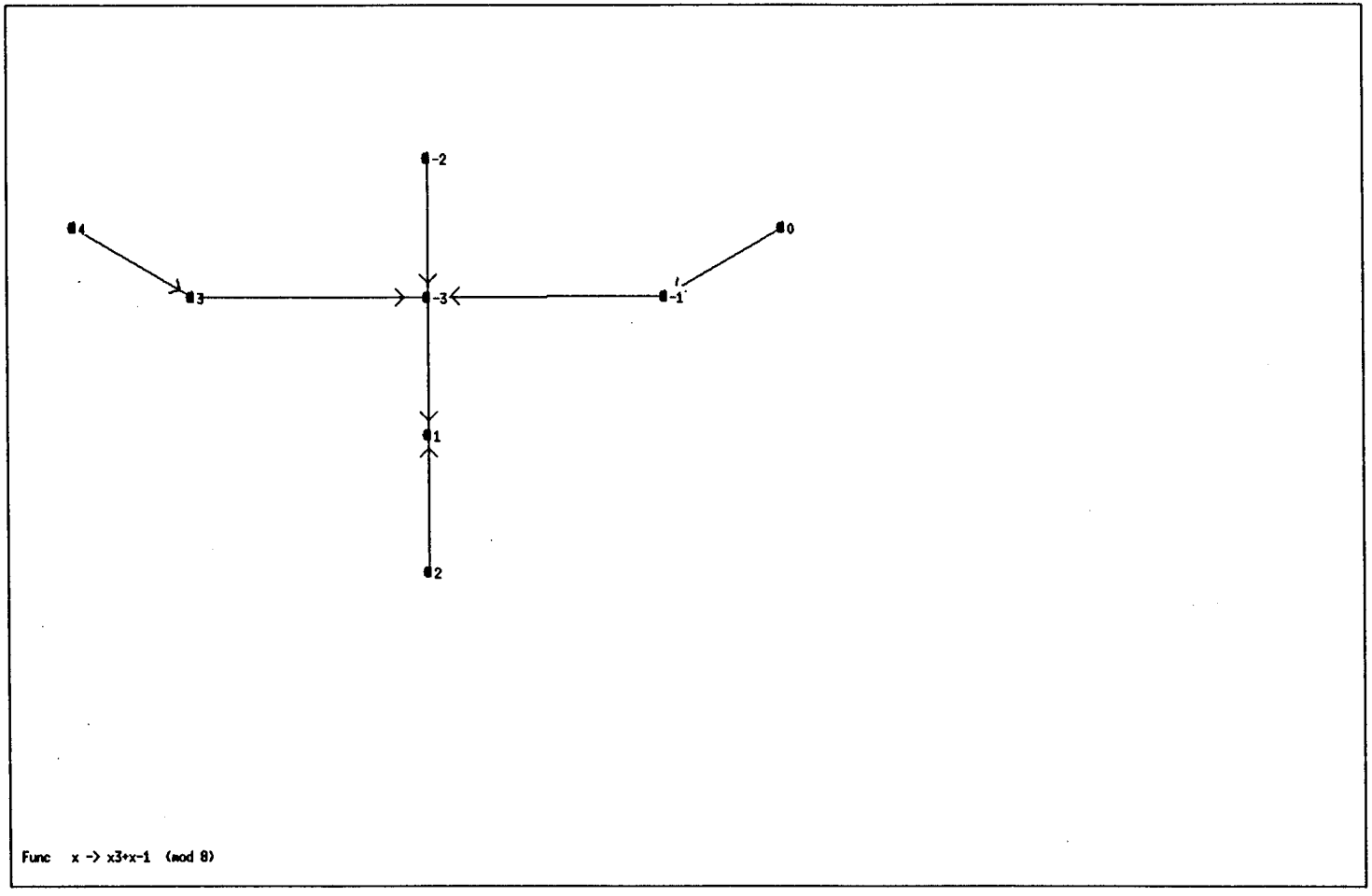

Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
Func x -> x2 (mod 81)

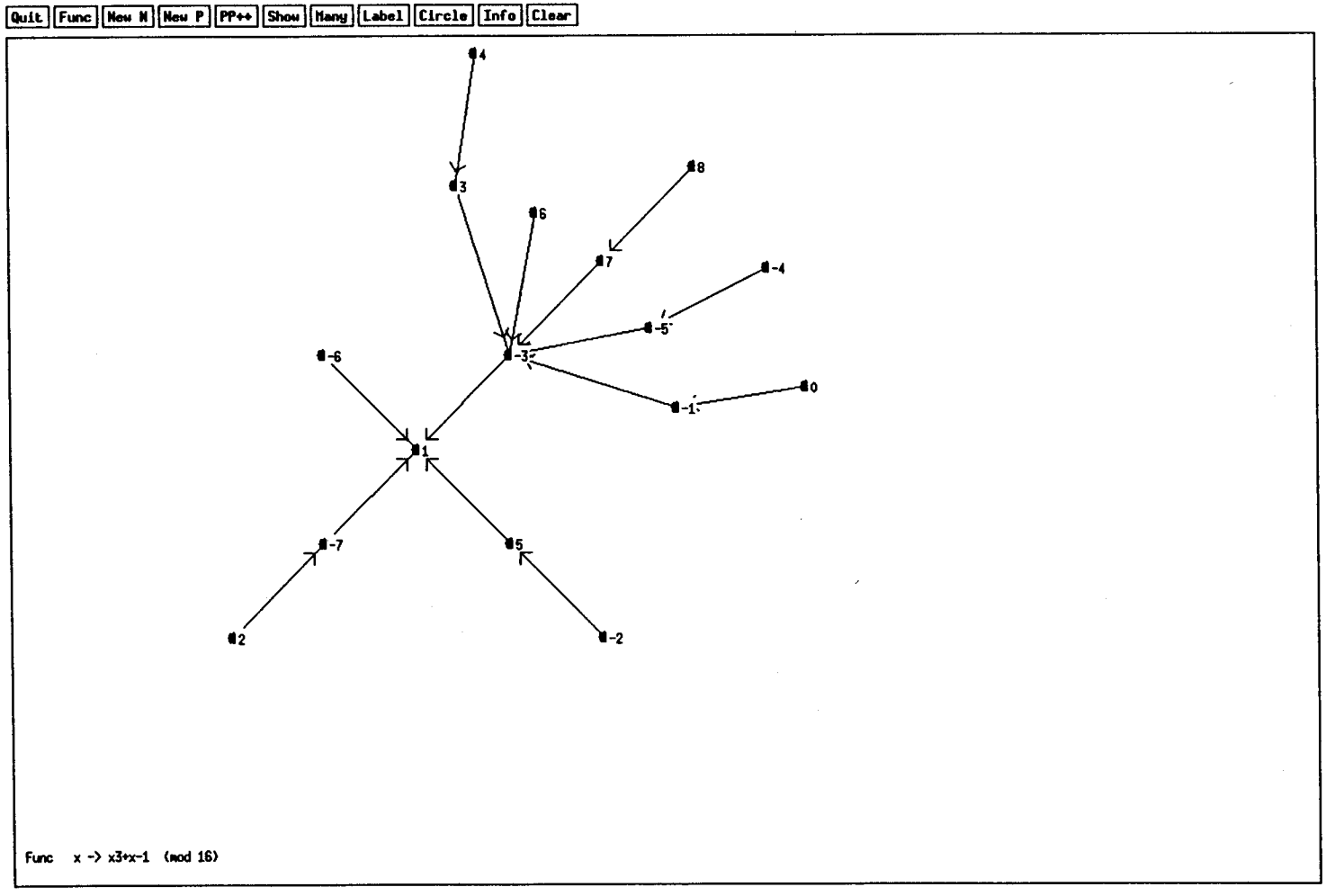

Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
Func x -> x2 (mod 243)

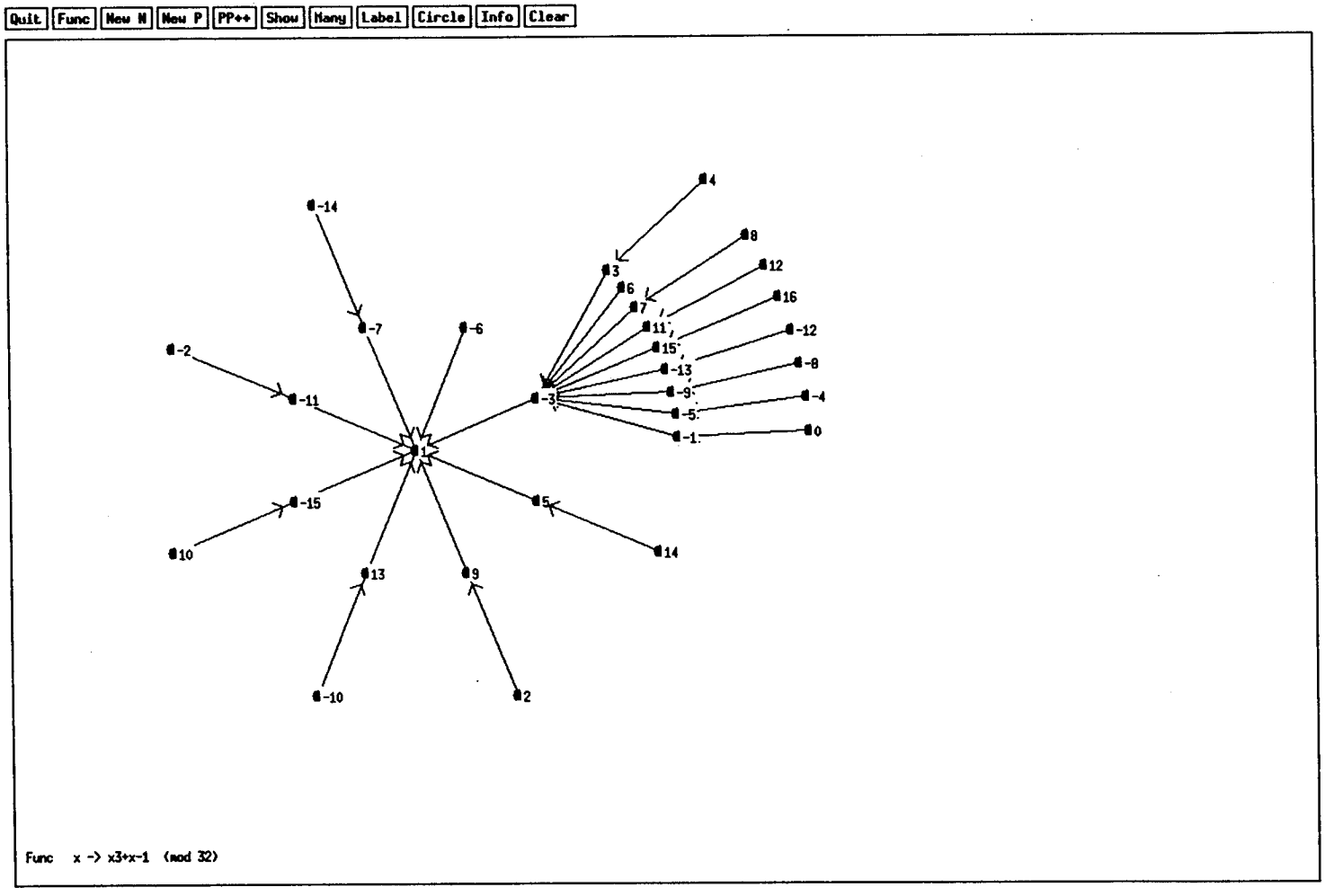

Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
Func x -> x2 (mod 729)


Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
Func x -> x2 (mod 2187)

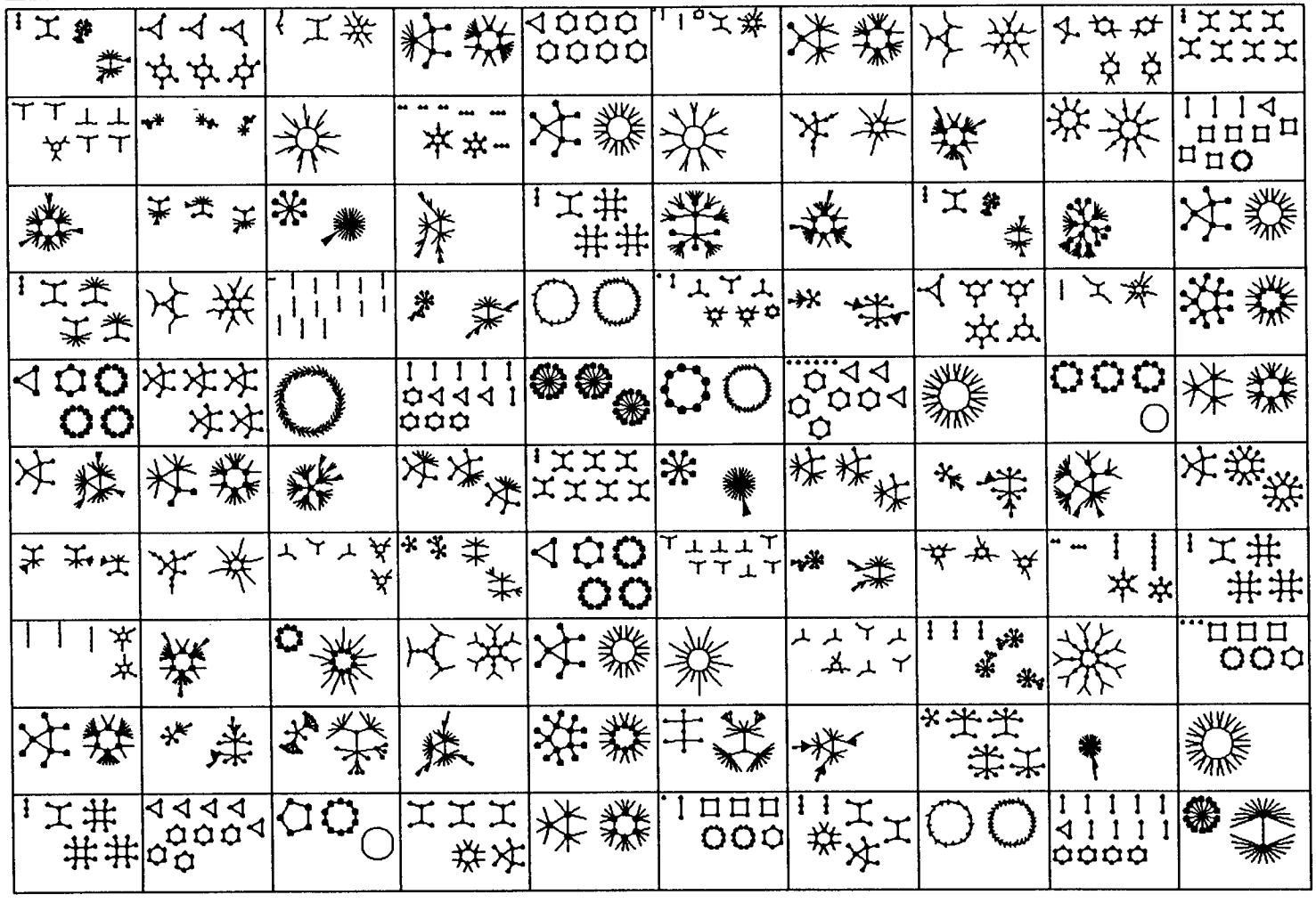
Quit Func New N New P PP++ Show Many Label Circle Info Clear




Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
4
3
6
8
7
-4
-5
-6
-3
0
-1
1
-7
5
2
-2
Func x -> x3+x-1 (mod 16)


Quit
Func
New N
New P
PP++
Show
Many
Label
Circle
Info
Clear
4
-14
8
3
12
6
16
7
-7
-6
11
-12
-2
15
-8
-13
-11
-3
-9
-4
-5
0
-1
1
-15
5
10
14
13
9
-10
2
Func x -> x3+x-1 (mod 32)

Quit Func New N New P PP++ Show Many Label Circle Info Clear



```
/*
 xpst.c
 for calculation and display of polynomial function state transtition
  digraphs mod n.
 to be used e.g. with an X Windows front-end
 Erik Winfree
*/

#include <math.h>
#include <stdio.h>

#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<(b)?(a):(b))
#define PI 3.1415926

#define GETCH() getc(stdin)

/*************************************************************************/
/* graphics primitives - modifiable per computer system (now X) */
/* functions implemented elsewhere (e.g. display.c) */
extern int depth;
#define BLACK 1
#define WHITE 0
#define BLUE    (depth==1?WHITE:7)
#define MAGENTA (depth==1?WHITE:15)
#define YELLOW  (depth==1?WHITE:23)
#define GREEN   (depth==1?WHITE:31)
#define RED     (depth==1?WHITE:39)
#define BLANCO  (depth==1?WHITE:47)
#define ORANGE  (depth==1?WHITE:55)
#define PURPLE  (depth==1?WHITE:63)

extern int gr_xmax,gr_ymax;

extern void gr_init(int i,int j);
extern void gr_clear();
extern void gr_use_window(int i, int j);
extern void gr_close();
extern void gr_circle(int x, int y, int r, int fill);
extern void gr_rectangle (int x1, int y1, int x2, int y2, int fill);
extern void gr_line(int x1, int y1, int x2, int y2);
extern void gr_color(int c);
extern void gr_fillcolor(int c);
extern void gr_textxy(int i, int j);
extern void gr_outtextxy(int x, int y, char *t);

/*************************************************************************/
/* digraph data structure */

typedef struct {
   char *func;
   int n,ng,mmd,mcy,lnk,pr;
   int *next,*group,*indeg,*dist,*x,*y;
   int *gn,*cyc,*mdist,*gx,*gy;
} digraph;

#define NEW(n) (int *) calloc(n,sizeof(int))
#define OLD(var) if (var) free(var)

#define ST (2*d->pr)
#define RAD(i) (d->mdist[i]+(d->cyc[i]>1))
#define PRAD(i) (d->lnk*RAD(i)+ST)
#define SRAD(i) (d->dist[i]+(d->cyc[d->group[i]]>1))
#define SPRAD(i) (d->lnk*SRAD(i))

/*************************************************************************/
```

```
/* graphics routines for displaying digraphs  */

clearg(digraph *d) { gr_clear(); }

drawg(digraph *d)
{  int i,ax,ay,px,py,al; char t[80];
   int n=d->n,*next=d->next,*group=d->group,*x=d->x,*y=d->y;
   int arl=min(d->lnk/5,10); if (!d) return;
   gr_color(WHITE);
   for(i=0;i<n;i++) if (i!=next[i]) {
     gr_line(x[i],y[i],x[next[i]],y[next[i]]);
     if (arl>2) {
      /* put on an arrow head */
      px=(9*x[next[i]]+x[i])/10; py=(9*y[next[i]]+y[i])/10;
      ax=x[i]+y[i]-x[next[i]]-y[next[i]];
      ay=-x[i]+y[i]+x[next[i]]-y[next[i]];
      al=sqrt(1.0*ax*ax+1.0*ay*ay);
      if (al!=0) {
       gr_line(px,py,px+arl*ax/al,py+arl*ay/al);
       gr_line(px,py,px-arl*ay/al,py+arl*ax/al);
      }
     }
   }
   for(i=0;i<n;i++)
     if (d->lnk*2*PI/d->cyc[group[i]]/ipow(2,d->dist[next[i]]) >
         2.0*d->pr*(d->dist[i]==0?1:d->indeg[next[i]]) || i==next[i]) {
      gr_fillcolor(group[i]+2);
      gr_circle(x[i],y[i],d->pr,1);
     }
   sprintf(t,"Func   x -> %s  (mod %d)",d->func,n);
   if (gr_ymax>150 && gr_xmax>8*strlen(t)+10) {
      gr_color(RED);
      gr_outtextxy(10,gr_ymax-20,t);
   }
}

circleg(digraph *d)
{  int i;
   if (!d) return;
   gr_color(BLUE);
   for (i=0;i<d->ng;i++) gr_circle(d->gx[i],d->gy[i],PRAD(i),0);
}

labelg(digraph *d)
{  int i; char t[80]; int n=d->n;
   if (!d) return;
   gr_fillcolor(BLACK);
   for (i=0;i<n;i++) {
      sprintf(t,"%d",i>n/2?i-n:i);
      gr_rectangle(d->x[i]+d->pr-3,d->y[i]+d->pr-16+3,
                   d->x[i]+d->pr+8*strlen(t)-3,d->y[i]+d->pr+3,1);
   }
   gr_color(RED);
   for (i=0;i<n;i++) {
      sprintf(t,"%d",i>n/2?i-n:i);
      gr_outtextxy(d->x[i]+d->pr,d->y[i]+d->pr,t);
   }
}

/*************************************************************************/
/* function evaluation routines */

int ipown(int x, int p, int n)
{  int i,r; for (r=1,i=0;i<p; i++) r=(r*x)%n; return r; }

int ipow(int x, int p)
```

```
{  int i,r; for (r=1,i=0;i<p; i++) r=r*x; return r; }

#define DIGIT(c) ((c)>='0' && (c)<='9')
int apply_func(char *func,int x, int n)
{  int i=0,m,p,pp,s,c,f=0;
   do { s=1;m=0;p=0;c=0;pp=0;
      while (func[i]!='-' && func[i]!='x' && !DIGIT(func[i]) && func[i]) i++;
      if (func[i]=='-') { s=-s; i++; }
      if (DIGIT(func[i]))
         while (DIGIT(func[i])) { m=m*10+func[i]-'0'; i++; }
      else m=1;
      while (func[i]==' ' || func[i]=='x' || func[i]=='^')
         { if (func[i]=='x') c=1; i++; }
      if (DIGIT(func[i]))
         while (DIGIT(func[i])) { pp=1; p=p*10+func[i]-'0'; i++; }
      f=(f+s*((m*ipown(x,pp?p:c,n))%n)+n)%n;
   } while (func[i]);
   return f;
}

/*************************************************************************/
/* functions for filling in digraph information */

int compute_func(digraph *d) /* make a state transition diagram */
{  int i;
   for (i=0;i<d->n;i++) d->next[i]=apply_func(d->func,i,d->n);
   return 1;
}

int find_groups(digraph *d) /* break a state transition diagram into groups */
{  int i,j,k,m,c,cc; int *gcount=NULL,*gord=NULL;
   int ng,n=d->n,*next=d->next,*group=d->group;
  if ((gcount=NEW(n)) && (gord=NEW(n))) {
   /* group[i] initially computed as min {states in same component} */
   for (i=0;i<n;i++) { group[i]=i; gcount[i]=0; }
   do {
      for (c=0,i=0;i<n;i++)
         if (group[i]!=group[next[i]]) {
            m=min(group[i],group[next[i]]);
            group[i]=group[next[i]]=m; c++;
         }
   } while (c!=0);
   for (i=0;i<n;i++) gcount[group[i]]++;
   for (ng=i=0;i<n;i++) if (gcount[i]) ng++;
   /* order the groups from smallest to largest */
   for (c=1,j=0;j<ng;j++)  {
      for (cc=n+1,i=0;i<n;i++)
         if (gcount[i]>=c && gcount[i]<cc) { cc=gcount[i]; k=i; }
      c=cc; gcount[k]=0; gord[k]=j;
   }
   for (i=0;i<n;i++) group[i]=gord[group[i]];
   d->ng=ng;
   /* note: we must make another useless count to create gn[j] later */
  } else printf("Daemonic dearth of DRAMs, Batman!\n");
  OLD(gord); OLD(gcount);
  return (gord && gcount);
}

int compute_stats(digraph *d) /* fill in standard digraph statistics */
{  int i,j,k,c,cc;
   int n=d->n,*next=d->next,*group=d->group,*dist=d->dist,*indeg=d->indeg;
   int ng=d->ng,*gn=d->gn,*cyc=d->cyc,*mdist=d->mdist;
   for (i=0;i<n;i++) { dist[i]=indeg[i]=0; gn[group[i]]++; }
   for (j=0;j<ng;j++) { cyc[j]=mdist[j]=0; }
   /* compute in-degree (of non-cycle), find cycles */
   for (i=0;i<n;i++) {
```

```
      indeg[next[i]]++;
      for (j=next[i],c=1;c<n && j!=i;c++,j=next[j]);
      if (j==i) { cyc[group[i]]=c; indeg[i]--; } else dist[i]=-1;
   }
   /* compute dist from cycles, crystallizing as depth increases */
   do for (c=i=0;i<n;i++)
      if (dist[i]==-1 && dist[next[i]]!=-1) { j=next[i]; c++;
         dist[i]=dist[j]+1; mdist[group[i]]=max(mdist[group[i]],dist[i]);
      }
   while (c!=0);
   for (j=0;j<ng;j++) d->mmd=max(d->mmd,mdist[j]);
   for (j=0;j<ng;j++) d->mcy=max(d->mcy,cyc[j]);
   return 1;
}

/*********************************************************************/
/* final functions for choosing exact display format i.e. x,y positions */

int isq(int x) { return x*x; }
double sq(double x) { return x*x; }

#define GDIST(i,j) \
  (sq(1.0*d->gx[i]-d->gx[j])+sq(1.0*d->gy[i]-d->gy[j]))
#define GSPACE(i,j) \
  (sq(1.0*PRAD(i)+PRAD(j)))
#define CLOSER(i,x,y) (abs((i)-(x))<abs((i)-(y))?(x):(y))

#define overlap(i) overlapp(d,i)
int overlapp(digraph *d,int i)
{ int j; for (j=0;j<i;j++) if (GDIST(i,j)<GSPACE(i,j)) return 1; return 0; }
#define conflict(i,tx,ty) conflictp(d,i,tx,ty)
int conflictp(digraph *d,int i,int tx, int ty)
{ d->gx[i]=tx; d->gy[i]=ty; return overlap(i)||
   ty<PRAD(i)||tx<PRAD(i)||tx>gr_xmax-PRAD(i); }

/* stack pretty "circle"s on display  (x,y,gx,gy) */
place_groups(digraph *d, float *deglow, float *deghi)
{  int i,tx,ty,placed,dx;
   int n=d->n,*group=d->group,*x=d->x,*y=d->y;
   int ng=d->ng,*gx=d->gx,*gy=d->gy;
   int dl=1+(gr_xmax+gr_ymax)/500, ml=2+(gr_xmax+gr_ymax)/200,
       dp=1+(gr_xmax+gr_ymax)/700;
   /* now assign screen coordinates -- first to groups, then to nodes */
   do {
    placed=1;dx=1;tx=ty=0;
    for (i=0;i<ng;i++) {
      ty=max(ty,PRAD(i)); tx+=dx*PRAD(i);
      if (tx<PRAD(i) || tx>gr_xmax-PRAD(i))
         { dx=-dx; tx=CLOSER(tx,PRAD(i),gr_xmax-PRAD(i)); }
      if (2*PRAD(i)>gr_xmax || 2*PRAD(i)>gr_ymax)
         { placed=0; gx[i]=tx; gy[i]=ty; continue; }
      if (conflict(i,tx,ty)) while(conflict(i,tx,ty)) ty+=dp;
      else { while(!conflict(i,tx,ty)) ty-=dp; ty+=dp; }
      while (!conflict(i,tx,ty)) tx-=dx*dp; tx+=dx*dp;
      gx[i]=tx; gy[i]=ty; tx+=dx*PRAD(i);
      if (ty>gr_ymax-PRAD(i)) placed=0;
    }
    if (!placed) { d->lnk-=dl; d->pr=min(1+d->lnk/5,5); }
   } while (!placed && d->lnk>ml);
   if (!placed) printf ("CAN'T FIT SHAPES!!\n");
   /* satisfactory arrangement; make it fast */
   for (i=0;i<n;i++) {
      x[i]=cos((deglow[i]+deghi[i])/2)*SPRAD(i)+gx[group[i]];
      y[i]=sin((deglow[i]+deghi[i])/2)*SPRAD(i)+gy[group[i]];
   }
}
```

```
int arrange_groups(digraph *d) /* compute angle radiation for each group */
{  int i,j,k,c,cc;
   int n=d->n,*group=d->group,*next=d->next,*dist=d->dist,*indeg=d->indeg;
   int *cyc=d->cyc;
   int *incry=NULL; float *deghi=NULL,*deglow=NULL;
  incry=NEW(n);
  deglow=(float *)malloc(n*sizeof(float));
  deghi =(float *)malloc(n*sizeof(float));
  if (incry && deglow && deghi) {
   for (i=0;i<n;i++) { deglow[i]=deghi[i]=0.0; incry[i]=0; }
   /* find cycles and assign (rotational) degree order (redundantly) */
   for (i=0;i<n;i++)
      if (dist[i]==0)
         for (c=cyc[group[i]],cc=0,j=i;cc<c;cc++,j=next[j])
            { deglow[j]=cc*2*PI/c; deghi[j]=(cc+1)*2*PI/c; }
   /* crystallize deghilow as depth increases */
   cc=0;
   do for (cc++,c=i=0;i<n;i++)
      if (dist[i]==cc) { j=next[i]; c++;
         deglow[i]=deglow[j]+incry[j]*(deghi[j]-deglow[j])/indeg[j];
         deghi[i]=deglow[j]+(incry[j]+1)*(deghi[j]-deglow[j])/indeg[j];
         incry[j]++;
      }
   while (c!=0);
   place_groups(d,deglow,deghi);
  }
  OLD(incry); OLD(deglow); OLD(deghi);
  return (incry && deglow && deghi);
}

/*********************************************************************/
/* create and destroy -- the high level digraph manipulators */

#define DEL(var) OLD(d->var)
digraph *clobber_digraph(digraph *d)
{
  if (d) {
   DEL(next); DEL(group); DEL(indeg); DEL(dist); DEL(x); DEL(y);
   DEL(gn); DEL(cyc); DEL(mdist); DEL(gx); DEL(gy);
   free(d);
  } return NULL;
}

#define ADD(var) ok=ok&&(d->var?d->var:d->var=NEW(n))
digraph *create_digraph(char *f,int n)
{  digraph *d; int ok=1;
   if (d=(digraph *)calloc(1,sizeof(digraph))) {
      d->func=f; d->n=n;
      ADD(next); ADD(group); ADD(indeg); ADD(dist);
      if (!ok) return clobber_digraph(d);
      ok=ok&&compute_func(d); ok=ok&&find_groups(d); n=d->ng;
      ADD(gn); ADD(cyc); ADD(mdist);
      if (!ok) return clobber_digraph(d);
      ok=ok&&compute_stats(d);
      if (!ok) return clobber_digraph(d); else return d;
   } else return 0;
}

int screen_digraph(digraph *d)
{  int n,ok=1;
   if (d) {
      d->lnk=sqrt(gr_xmax*gr_ymax/4.0/d->ng)+5; d->pr=min(1+d->lnk/5,5);
      n=d->n; ADD(x); ADD(y);
      n=d->ng; ADD(gx); ADD(gy);
      if (!ok) { DEL(x); DEL(y); DEL(gx); DEL(gy); return 0; }
```

```
        ok=ok&&arrange_groups(d); return ok;
    }
}

#define GFX(d) (d->x!=NULL)

/*****************************************************************************/

digraph *d=NULL;
char f[40]="x2"; int n=17,p=2,pp=1;

void do_show()
{
    gr_init(1,1); gr_clear();
    clobber_digraph(d); d=create_digraph(f,n);
    if (screen_digraph(d)) drawg(d);
    gr_close();
}

void do_many()
{   char ftemp[80]; int a,b,ac,bc,i,j; char af[40],bf[40];
    printf("Enter function with %%d for the two variable coefficients.\n");
    printf("(mod n ) x -> "); scanf("%s",&ftemp);
    printf("Give number (- rel. prime(UI)) 0..i-1, one-variable function.\n");
    printf("First  variable num. points, function: "); scanf("%d,%s",&a,af);
    printf("Second variable num. points, function: "); scanf("%d,%s",&b,bf);
    gr_init(abs(a),abs(b)); gr_clear();
    for(i=0;i<abs(a);i++) for(j=0;j<abs(b);j++) {
       gr_use_window(i,j); ac=apply_func(af,i,n); bc=apply_func(bf,j,n);
       sprintf(f,ftemp,ac,bc);
printf("Making graph x -> %s (mod %d)\n",f,n);
       clobber_digraph(d); d=create_digraph(f,n);
       if (screen_digraph(d)) drawg(d);
    }
    d=clobber_digraph(d);
    gr_close();
}

void do_func()
{
    printf("(mod n ) x -> "); scanf("%s",&f);
    do_show();
}
void do_n()
{
    printf("N for squares mod n graph? "); scanf("%d",&n);
    do_show();
}
void do_p()
{
    printf("P for squares mod p^n graph? "); scanf("%d",&p);
    pp=1; n=ipow(p,pp);
    do_show();
}
void do_pp()
{
    pp++; n=ipow(p,pp);
    if (n>8000) { char t[10];
       printf("n=%d    Really?? "); scanf("%s",t); if (t[0] != 'y') return; }
    do_show();
}
void do_label()
{   if (d) { gr_init(1,1); labelg(d); gr_close(); } }
void do_circle()
{   if (d) { gr_init(1,1); circleg(d); gr_close(); } }
```

```
#define PSP 16
void do_info()
{   int i,j,k,l; char t[10];
  if (d) for (i=0;i<d->ng;i++) {
     printf("Group %d (%d points): mdist %d, cyc %d \n",
        i,d->gn[i],d->mdist[i],d->cyc[i]);
  }
  /* printf("Mass computation?"); scanf("%s",t); */ t[0]='n';
  if (t[0]=='y') {
     int nf,nn; int *modnp; char *polyf;
     printf("Number of polynomials, number of modulii?");
     scanf("%d,%d",&nf,&nn);
     modnp=(int *)calloc(nn*2,sizeof(int)); polyf=(char *)malloc(nf*PSP);
     if (modnp==NULL || polyf==NULL) { printf("Duh, sorry, I forgot.\n"); }
     else {
         FILE *fl;
         for (i=0;i<nn;i++) {
            printf("n^1...n^m for  mod, max power:");
            scanf("%d,%d",&modnp[2*i],&modnp[2*i+1]);
         }
         for (i=0;i<nf;i++) {
            printf("Enter polynomial function x-> ");
            scanf("%s",&polyf[PSP*i]);
         }
         fl=fopen("xst.out","a");
         if (fl!=NULL) {
             printf("computing");
           switch(t[1]) {
            case 'A':
                fprintf(fl,"Stats: #components, #cycle points, #tail points, max cy
                fprintf(fl,"modulus |");
                for (i=0;i<nf;i++) {
                  fprintf(fl," x->%16s |",&polyf[PSP*i]);
                }
                fprintf(fl,"\n");
             for (j=0;j<nn;j++) for (k=1;k<=modnp[2*j+1];k++) {
                fprintf(fl,"%4d^%2d |",modnp[2*j],k);
                for (i=0;i<nf;i++) {
                   int cp=0,tp=0,cm=1,mgn=1;
                   d=create_digraph(&polyf[PSP*i],ipow(modnp[2*j],k));
                   if (d) {
                      for (l=0;l<d->n;l++)
                         { if (d->dist[l]==0) cp++; if (d->indeg[l]==0) tp++; }
                      for (l=0;l<d->ng;l++)
                         { cm=max(cm,d->cyc[l]); mgn=max(mgn,d->gn[l]); }
                      fprintf(fl," %3d %4d %3d %3d %4d |",d->ng,cp,tp,cm,mgn);
                   } else fprintf(fl," -0%18d- |",0);
                }
                fprintf(fl,"\n");
             } break;
            case 'C':
             for (i=0;i<nf;i++) {
                fprintf(fl,"x->%16s ",&polyf[PSP*i]);
                for (j=0;j<nn;j++) for (k=1;k<=modnp[2*j+1];k++) {
                   d=create_digraph(&polyf[PSP*i],ipow(modnp[2*j],k));
                   if (d) {
                      fprintf(fl,"#%2d",d->ng);
                   } else fprintf(fl,"#--");
                }
                fprintf(fl,"\n");
             } break;
            }
             fclose(fl);
             printf("\nStatistics generated into file xst.out\n");
         } else printf ("Fucking FileServer's Fried, Freddy!\n");
     }
```

```
    if (modnp) free(modnp); if (polyf) free(polyf);
    d=clobber_digraph(d);
  }
}

/*************************************************************************/
```

```
/* display.c     Erik Winfree
a simplified set of X commands.
so just this source code can be modified to get most of the program
running under other systems, such as Amiga or PCs
Also main.c must be rewritten.
*/

#include <stdio.h>
#include <X11/Intrinsic.h>

extern Widget drawform;

extern int depth;
long mwidth;
long bias = 3;                                  /* bias into color table */
long wwidth = 0;
long wheight = 0;

int xparts,yparts,xwin,ywin,x0,y0;
int gr_xmax,gr_ymax;

#define max(a,b)  ((a)>(b)?(a):(b))
#define min(a,b)  ((a)<(b)?(a):(b))

/*************************************************************************/
/* graphics primitives - modifiable per computer system (now X) */

#define BLACK 1
#define WHITE 0

    Window draw_win;
    GC draw_gc;
    Screen *draw_Screen;
    Display *draw_d;
    Visual *draw_v;

void SizeCheck() {
    XWindowAttributes draw_wattr;
    XGetWindowAttributes(draw_d, draw_win, &draw_wattr);
    if (wwidth != draw_wattr.width) {
        wwidth = draw_wattr.width;
        mwidth = (depth > 1)? wwidth: (1 + wwidth/8);
        wheight = draw_wattr.height;
    }
    gr_xmax=wwidth/xparts-1; gr_ymax=wheight/yparts-1;
    x0=xwin*(gr_xmax+1); y0=ywin*(gr_ymax+1);
  }

void gr_init(int i, int j) {
    draw_d = XtDisplay(drawform);
    draw_win = XtWindow(drawform);
    draw_Screen = XtScreen(drawform);
    draw_gc = draw_Screen->default_gc;
    draw_v = draw_Screen->root_visual;
    {
        XSetWindowAttributes attr;
        attr.backing_store = Always;
        XChangeWindowAttributes (draw_d, draw_win, CWBackingStore, &attr);
    }
    xparts=max(i,1); yparts=max(j,1); xwin=ywin=0;
    SizeCheck();
  }
void gr_clear() { int k;
    XSetForeground(draw_d,draw_gc,BLACK);
    XFillRectangle(draw_d,draw_win,draw_gc,0,0,wwidth,wheight);
    XSetForeground(draw_d,draw_gc,WHITE);
```

```
    for (k=1;k<xparts;k++) XDrawLine(draw_d,draw_win,draw_gc,
       k*(gr_xmax+1)-1,0,k*(gr_xmax+1)-1,wheight);
    for (k=1;k<yparts;k++) XDrawLine(draw_d,draw_win,draw_gc,
       0,k*(gr_ymax+1)-1,wwidth,k*(gr_ymax+1)-1);
  }
void gr_use_window(int i, int j) {
    xwin=i%xparts; ywin=j%yparts; SizeCheck();
  }
void gr_close() { }
void gr_circle(int x, int y, int r, int fill)
 { if (fill) XFillArc(draw_d,draw_win,draw_gc,x0+x-r,y0+y-r,2*r,2*r,0,64*360);
   else XDrawArc(draw_d,draw_win,draw_gc,x0+x-r,y0+y-r,2*r,2*r,0,64*360);  }
void gr_rectangle (int x1, int y1, int x2, int y2, int fill)
 { if (fill) XFillRectangle(draw_d,draw_win,draw_gc,x0+x1,y0+y1,x2-x1,y2-y1);
   else XDrawRectangle(draw_d,draw_win,draw_gc,x0+x1,y0+y1,x2-x1,y2-y1); }
void gr_line(int x1, int y1, int x2, int y2)
 { XDrawLine(draw_d,draw_win,draw_gc,x0+x1,y0+y1,x0+x2,y0+y2); }
#define cfun(c) (depth==1?(c==BLACK?BLACK:WHITE):c<2?c:(c%((1<<depth)-2))+2)
void gr_color(int c)
 { XSetForeground(draw_d,draw_gc,cfun(c)); }
void gr_fillcolor(int c)
 { XSetForeground(draw_d,draw_gc,cfun(c)); }
void gr_textxy(int i, int j) { }
void gr_outtextxy(int x, int y, char *t)
 { XDrawString(draw_d,draw_win,draw_gc,x0+x,y0+y,t,strlen(t)); }

/*****************************************************************************/
```

```
/*
 * Xmandel - written by John Freeman at Cray Research
 *
 * This file contains the main routine for driving the mandelbrot generator
 */

#include <X11/X.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Cardinals.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Form.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Paned.h>
#include <stdio.h>
Widget drawform;
Colormap c_map;
int depth;
extern bias;


#define DEFAULT_DRAWING_WIDTH 150
#define DEFAULT_DRAWING_HEIGHT 150

/*
 * setup the internal action routines to be used
 */

static void do_clear ()
{
    XClearWindow (XtDisplay(drawform), XtWindow(drawform));
}

static void do_quit (w, event, params, num_params)
    Widget w;
    XEvent *event;
    String *params;
    Cardinal *num_params;
{
    XFreeColormap(XtDisplay(drawform),c_map);
    exit (0);
}

extern void do_func(),do_n(),do_p(),do_pp(),do_show(),
            do_many(),do_info(),do_label(),do_circle();

static XtActionsRec xmandel_actions[] = {
  { "Quit", do_quit },
  { "Func", do_func },
  { "New_N", do_n },
  { "New_P", do_p },
  { "PP_Inc", do_pp },
  { "Show", do_show },
  { "Many", do_many },
  { "Label", do_label },
  { "Circle", do_circle },
  { "Info", do_info },
  { "Clear", do_clear },
};

static char *button_list[] = {
  "quit", "func", "new_n", "new_p", "pp_inc", "show", "many",
  "label", "circle", "info", "clear", NULL};

void make_cmap(w,base)
      Widget w;
```

```
        int base;
 {
        Display *draw_d;
        Window   draw_win;
        XColor c,colors[9];
        int i,j;
        unsigned long pixels[100];

        for(i=0;i<8;i++){
          colors[i].red   = 0;
          colors[i].blue  = 0;
          colors[i].green = 0;
        }
        colors[0].blue  = 65535;      /* rgb.txt - deep sky blue */
        colors[0].green = 48705;

        colors[1].red   = 65535;      /* rgb.txt - magenta */
        colors[1].blue  = 65535;

        colors[2].red   = 65535;      /* rgb.txt - yellow */
        colors[2].green = 65535;

        colors[3].blue  = 32385;      /* rgb.rxr - spring green */
        colors[3].green = 65535;

        colors[4].red   = 65535;      /* rgb.txt - red */

        colors[5].red   = 65535;      /* rgb.txt - white */
        colors[5].blue  = 65535;
        colors[5].green = 65535;

        colors[6].red   = 65025;      /* rgb.tst - orange */
        colors[6].green = 42075;

        colors[7].blue  = 61200;      /* rgb.txt - purple */
        colors[7].red   = 40800;
        colors[7].green =  8160;

        draw_d   = XtDisplay(w);

        c_map = XDefaultColormap( draw_d, DefaultScreen(draw_d));

        if(!XAllocColorCells( draw_d, c_map, True, NULL, 0, pixels, 64)){
          printf("Warning: Couldn't Allocate Color Cells\n");
          exit(0);
        }
        bias = pixels[0];
        c.flags = DoRed | DoBlue | DoGreen;
        for(i=0;i<8;i++){
          for(j=0;j<8;j++){
            c.pixel = pixels[i*8+j];
            c.red   = (colors[i].red  *(j+12))/19;
            c.blue  = (colors[i].blue *(j+12))/19;
            c.green = (colors[i].green*(j+12))/19;
            XStoreColor( draw_d, c_map, &c);
          }
        }
 }

main (argc, argv)
    int argc;
    char **argv;
 {
    Widget toplevel, pane, box, form;
    Arg av[10];
    Cardinal i;
```

```
    char **cpp;
    int j;


    j = 1;
    toplevel = XtInitialize (NULL, "Xpst", NULL, 0, &j, argv);
    XtAppAddActions (XtWidgetToApplicationContext (toplevel),
                     xmandel_actions, XtNumber (xmandel_actions));

    depth = DefaultDepthOfScreen (XtScreen(toplevel));   /* XXX crock */

    pane = XtCreateManagedWidget ("pane", panedWidgetClass, toplevel,
                                  NULL, ZERO);

    box = XtCreateManagedWidget ("box", boxWidgetClass, pane, NULL, ZERO);

    for (cpp = button_list; *cpp; cpp++) {
        (void) XtCreateManagedWidget (*cpp, commandWidgetClass, box,
                                      NULL, ZERO);
    }

    form = XtCreateManagedWidget ("form", formWidgetClass, pane,
                                  NULL, ZERO);

    i = 0;
    XtSetArg (av[i], XtNtop, XtChainTop); i++;
    XtSetArg (av[i], XtNbottom, XtChainBottom); i++;
    XtSetArg (av[i], XtNleft, XtChainLeft); i++;
    XtSetArg (av[i], XtNright, XtChainRight); i++;
    XtSetArg (av[i], XtNwidth, DEFAULT_DRAWING_WIDTH); i++;
    XtSetArg (av[i], XtNheight, DEFAULT_DRAWING_HEIGHT); i++;
    drawform = XtCreateManagedWidget ("window", widgetClass, form, av, i);
    if(depth > 1)
      make_cmap(toplevel);

    XtRealizeWidget (toplevel);
    XtMainLoop ();
}
```