COGNITIVE SYSTEMS RESEARCH PROGRAM

CORNELL UNIVERSITY ITHACA, N. Y.

REPORT NO. 8

A Computer Program for Simulation of Perceptrons and Similar Neural Networks: Users' Manual

By

TREVOR H. BARKER

14 July, 1966

Prepared Under Contract No.
NONR 401 (40) and NSF GK-250

CSRP

COGNITIVE SYSTEM RESEARCH PROGRAM

Cornell University Ithaca, New York

Report No. 8

A COMPUTER PROGRAM FOR SIMULATION OF PERCEPTRONS AND SIMILAR NEURAL NETWORKS: USERS MANUAL

by

Trevor H. Barker

14 July 1966

Prepared under Contract No. NONR 401 (40) and NSF GK-250. Reproduction, translation, use and disposal in whole or in part by or for the United State Government is permitted. Distribution of this document is unlimited.

PRECIS

Title: "A Computer Program for Simulation of Perceptrons and Similar Neural Networks: Users Manual," Trevor H. Barker, Cognitive Systems Research Program Report No. 8, Cornell University, Ithaca, New York, 14 July 1966; NONR 401 (40) and NSF GK-250.

Report Contents: The purpose of this report is to describe the capabilities of, and give instructions for the use of, a very general program designed for the simulation of perceptrons and perceptron-like networks. This program was written for the IBM 7090/7094 and was developed by the author for the Cognitive Systems Research Program at Cornell University. The program is available for use by interested workers in the field.

This report makes no attempt to describe any of the results obtained by use of the program. It is intended solely to make it possible for others to make use of the program.

This program can be used on several levels of complexity. Many experiments of a simple nature could be set up with no more than a fairly casual reading of the first few examples in Section 3. More complicated situations could be handled after reading the first portion of Section 4; while for the user who wants to take the trouble to go into the fine points, there are many special features of great power which allow almost any conceivable situation to be handled. The program is capable of simulating any of an extremely wide variety of perceptrons and has been designed to obtain very efficient timing, especially for perceptrons with γ or Γ reinforcement and/or with non-zero decay.

For Further Information: The complete report is available in the major Navy technical libraries and can be obtained from the Defense Documentation Center. A few copies are available for distribution by the Cognitive Systems Research Program. Card decks are also available from Cognitive Systems Research Program for persons wishing to make use of the program.

TABLE OF CONTENTS

1. Introduction

The purpose of this report is to describe the capabilities of, and give instructions for the use of, a very general program designed for the simulation of perceptrons and perceptron-like networks. This program was written for the IBM 7090/7094 and was developed by the author for the Cognitive Systems Research Program at Cornell University. The program is available to any interested workers in the field.

This program can be used on several levels of complexity. Many experiments of a simple nature could be set up with no more than a fairly casual reading of the first few examples in Section 3. More complicated situations could be handled after reading the first portion of Section 4; while for the user who wants to take the trouble to go into the fine points, there are many special features of great power which allow almost any conceivable situation to be handled. The program is capable of simulating any of an extremely wide variety of perceptrons and has been designed to obtain very efficient timing, especially for perceptrons with γ or Γ reinforcement and/or with non-zero decay.[1]

Knowledge of programming is not necessary for use of the program, although a person with some experience of programming languages will probably find it easier to learn the more complicated aspects. Detailed knowledge of the 7090 is not needed to use it (except for the most esoteric fine points). A brief summary of such superficial information about the machine as might be useful to the reader is given in Appendix IV.

[1] This report does not, of course, attempt to give any exposition of perceptron theory. It is assumed that the reader has some background in the field or he would not be reading this report in the first place. It would be helpful, although not absolutely necessary, if the reader were on speaking terms with the standard terminology used in reports of Cognitive Systems Research Program.

Section 2 gives a brief description of the entire program, sufficient to give a rough idea of the types of things it is capable of.

Section 3 gives a series of examples illustrating the various features of the system. These are intended to familiarize the reader with the grounds to be covered before plunging into the complete description.

Section 4 gives a complete description of the system, ordered (more or less) according to complexity.

Section 5 is a concise summary, suitable for reference once the material has been learned from Section 4.

If the reader is daunted by the sheer volume of material, let us reiterate the remark made above, that the casual user may safely ignore most of this material. He would be best advised to go quickly through Section 2 and the first couple of examples in Section 3, and then read the first few subsections of Section 4.

IMPORTANT NOTE: The reader may note in various places in this manual the following mysterious-looking symbol; Ø. This is nothing but a capital O. The slash through the O is a standard convention in hand-writing material to be keypunched in order to avoid confusion with the number zero. It mainly occurs in examples, where material is written exactly as it would be punched, and serves the same purpose. Occasionally, alas, it has snuck into text, but not very frequently, and we hope this will cause no confusion.

2. General Description

2.1 Perceptron Generation

A quite general definition of "perceptron" is used. We consider a perceptron to consist of

a) two or more "layers," each consisting of a number of units, and with an associated threshold (the same for all units in a layer[1]). Each unit has two states, on and off.

b) one or more "linkages". A linkage is a set of inputs to the units of a given layer, (the terminal layer), dependent on the condition of the units in another (perhaps the same) layer (the initial layer); it may be the current condition or the condition one or more time-steps in the past (this number is called the delay.) The actual nature of the linkage may be one of several types-- some completely determined, some subject to random choices-- some unchanging with the passage of time, some with weights which may be varied according to various reinforcement rules. (For example, the S-A connections in a simple perceptron represent one type of linkage; the A-R connections represent another-- The A-A connections in a cross-coupled perceptron are also of this latter type.) The desired perceptron is specified by giving the number of units desired in each layer, the threshold, and the various types of linkages. Full details are given in section 4.

[1] This restriction can be circumvented if one goes into the finer points of the system.

2.2 Overall Organization

The general outline of the program is perhaps best understood by the following analogy. Imagine a machine consisting of the following parts:

A. A robot, capable of building perceptrons according to prescribed specifications.

B. A small (not very efficient) general purpose computer capable of

i) Controlling the robot
ii) Communicating with and controlling the perceptron constructed by the robot
iii) Plus the usual things a general purpose computer can do.

In a simple experiment, the computer can communicate with the perceptron on a very general level, e.g., "present a stimulus, "perform error-correction reinforcement," etc. At the other extreme, communication can be in great detail, e.g., the computer can examine in detail any or all of the units in the perceptron and perform sophisticated arithmetic and/or logical processes before deciding the future course of the experiment.

IMPORTANT NOTE: From a purely logical point of view, the computer alone, (i.e., Biii) can simulate a perceptron and perform tests on it. This is NOT what we are talking about. Considerable knowledge of perceptrons is "built in" to the robot - so that a perceptron can be constructed by a very small number of simple commands. Again, considerable knowledge of perceptron testing procedure is "built in" to the computer in the form of specialized commands which allow many testing procedures to be programmed considerably more easily than with the more "normal" instructions.

This complex machine (robot and computer) is simulated by the program. The input to the program is, in effect, an input language for this hypothetical computer. (Considered as such it lies somewhere between an assembly language and a compiler.)

The following is a brief summary of the general features of the language.

(1) The usual type of arithmetic, boolean, and decision-making features. In particular, boolean operations can work on bit strings of any length (e.g. activity vectors from the perceptron in packed form) regardless of whether or not they fit into a single 7090 word (36 bits).

(2) Perceptron specifications. A small number of simple commands allow the specification of a very wide variety of perceptrons. When the various parameters have been specified, the perceptron generator automatically creates the necessary machine code to perform very efficient simulation of the internal workings of the perceptron, and sets up the necessary communication with the rest of the program.

(3) Perceptron control (training and testing). Many types of training and/or testing procedures can be performed by the use of a small number of special commands. For more complicated situations, other commands allow more detailed communication with, and control of, the perceptron. For example, it is possible to obtain the current value (that is, sum of inputs minus threshhold) of any unit or units in any layer of the perceptron. These values can, if desired, be operated on by arithmetic commands and can, for example, form the basis for decisions on the future course of the program, or be printed in the output, etc. Similarly for any or all of the variable weights. The reinforcement of each linkage can be controlled separately, subject to decisions of any desired complexity.

(4) Stimulus generation. The language contains several features specifically designed to facilitate production of stimulus sequences for use in training and/or testing the perceptron. Specific stimuli may be provided to the program by the user in one of several forms. Certain standard stimuli, e.g. horizontal and vertical bars, may be generated internally by the program, and stimuli may be subjected to various translations, rotations, dilations. A large number of stimuli may be temporarily stored and put onto tape in a sequence of any desired complexity, either

a predetermined sequence or one involving random choices from various groups of stimuli.[1]

(5) Output. The perceptron generator automatically prints out a description of the perceptron constructed, at no effort on the part of the user. Certain of the specialized perceptron testing phrases also automatically print out results. However, for more complicated experiments, the language has very flexible output provisions to allow the user to specify just what information he would like printed, and in what form.

(6) Automatic repetition. Many perceptron experiments require repetition of essentially similar situations with simply changes in certain parameters. The language provides a very simple yet powerful method for handling this automatically. One or several parameters may be varied and the entire sequence of experiments will be performed automatically. If a deck made up in this way contains so many experiments that they cannot all be run at a single sitting on the computer, the addition of a single card at the end of the deck on a subsequent run will allow the program to ignore those cases which had been run previously. Instructions for doing this are printed online at the conclusion of each experiment so that this can be handled by an operator who knows nothing about the program.

[1] Normally these stimulus generation features of the language can be considered as completely independent of the rest of the language. However, in especially complicated situations where one cannot do exactly what one would like with these features alone, it is possible to use them in conjunction with the arithmetical and other features of the program, e.g. in preparing stimuli for figure-ground experiments, one could produce a small number of figures and a small number of backgrounds separately and then combine these in various combinations, using the boolean commands.

3.1 A Quick Look at the Forest.

Before proceeding to a formal and detailed description of the program, we shall give some examples to familiarize the reader with the territory to be covered. Naturally these cannot all be explained in full detail at this stage--they are intended to be read over lightly, ignoring anything which may seem incomprehensible.

Logically, the reader could skip directly to the next section, as everything is completely described there and beyond; unfortunately there is a high probability of getting lost in details; hence these examples.

NOTE: these examples are chosen to illustrate the various features of the program, not to illustrate perceptron theory. Thus, although we have tried to maintain general contact with "reality" (that is to say, with the kind of experiments one might actually want to perform), the reader should not expect the specific experiments appearing in these examples to necessarily be "useful" ones.

3.2 Example

Generate a simple perceptron with a 20 x 20 retina and 1000 A-units; alpha reinforcement, with an increment of 1. The A-units should have a threshold of 1; 3 positive and 2 negative inputs chosen at random on the retina. The stimuli to be used are all possible 20 x 4 horizontal bars (in the positive class) and all possible 4 x 20 vertical bars (in the negative class). Train by presenting 100 randomly chosen stimuli (alternating classes) and using error correction reinforcement. Then test by presenting another sequence of 20 stimuli and print the number of correct, zero and incorrect responses.

Solution: (Note the numbers at the beginning of each line are <u>not</u> punched on the cards. They are written in here only to facilitate the comments which follow the example.)

```
1.  SSINIT 20 20, SSOTAP T1, REW T1,
2.  (60, SSHBAR 4, SSTRAN 0 $RAND 20, SSPOS, SSOUT,
3.       SSVBAR 4, SSTRAN $RAND 20 0, SSNEG, SSOUT,)
4.  RETINA 20 20,
5.  LAYER 2 1000 1, LINK 2 TO 2 FROM 1 POS 3 NEG 2,
6.  LAYER 3 1 0, LINK 3 TO 3 FROM 2 INCR 1 ALPHA,
7.  END,
8.  REW T1,
9.  EC 100 T1, TEST 20 T1,
10. END,
```

Discussion of Solution:

The first three lines deal with the preparation of the required stimuli.

SSINIT 20 20,	indicated that we are about to generate stimuli, and that they are to be 20 x 20

SSOTAP T1,	causes tape unit T1 to receive any stimuli generated. (We could have said, e.g. A5, or B3, etc., thus assigning a specific tape chosen by us. This would be necessary if the tape were to be removed from the machine and saved for later use. In this case, however, the tape is only going to by used by a later part of this job, and it is more convenient to use T1, which means that the program will decide for itself which specific unit to use.[1]
REW T1,	REWind tape T1.
(60, ...)	Everything between "(60," and ") is to be repeated 60 times.
SSHBAR 4,	Form a stimulus consisting of a Horizontal Bar of width 4 at the top of the retina.
SSTRAN h v,	Causes the stimulus currently on the retina to be translated h places in a horizontal direction and v places in a vertical direction. Points shifted off one edge (horizontal or vertical) will reappear at the opposite edge.[2]

[1] See appendix IV for a brief discussion of tape units.

[2] This is a technicality used in many simple experiments to avoid complications due to change of stimulus size as part is lost over the edge. It is possible to do translations without this effect if desired, as explained at the appropriate place in section 4.

$RAND 20	This expression represents an integer, chosen at random from 1 to 20. Each time phrase containing this expression is used, a new random choice is made.
SSPOS,	Assign the stimulus now on the retina to the positive class.
SSNEG,	Guess.
SSOUT,	Write the stimulus now on the retina onto the output tape (in this case, T1).

The next three lines (4-6) specify the perceptron which is to be generated.

RETINA 20 20,	The retina is to be 20 x 20.
LAYER 2 1000 1,	Layer 2 (A-units) is to have 1000 units, each with a threshold of 1.
LINK 2 TO 2 FROM 1 POS 3 NEG 2,	There is to be a linkage of type 2 from layer 1 (retina) to layer 2 (A-units). There are several different types of linkages, which will be discussed in detail later. For the purposes of this example, accept on faith that the type required here is 2, (and that on line 6 is 3) POS 3 indicates 3 positive (excitatory) inputs per A-unit, and NEG 2 indicates 2 negative (inhibitory) inputs per A-unit.
END,	The input is divided into <u>sections</u> by the appearance of the word END,. It is absolutely necessary that the generation of a perceptron and the training/testing of the perceptron be in separate sections, hence the appearance of END on line 7. For most situations, this simple division into two sections is all that is necessary, so we will save further discussion of this matter for more complicated examples.

The last four lines deal with the training and testing of the perceptron.

REW T1, — REWind tape unit T1 again (ready to read the stimuli which were written on it).
NOTE: This is an illustration of poor technique. In this case T1 could have been rewound in the first section, as soon as the stimuli were written on it. In order to avoid unnecessary waste of time, it is always advisable to rewind a tape as soon as it is logically possible.
The example as given will work correctly, but it will be a little slower than necessary.

EC 100 T1, — Present 100 stimuli from tape T1 performing Error-correction reinforcement of the response unit.

TEST 20 T1, — 20 stimuli will be presented from tape T1, the response to each computed, and a count kept of the number of correct, zero, and wrong responses. After all have been presented, these numbers will be printed in a line such as the following

CORRECT 15 (75.0), WRONG 3 (15.0), ZERO 2 (10.0)

(The numbers in parentheses are of course percentages.)

To perform this experiment, the input as given above would be punched on cards, and used as an input deck for the program. The format is free, i.e. there are no requirements for particular items to be punched in particular columns. Extra blanks may be inserted between items, but not in the middle of an item (e.g. it is not correct to write RET INA for RETINA). At least one blank must separate two consecutive alphabetic symbols. A separating blank is not necessary

between an alphabetic symbol and a number, nor between a special character (comma, parantheses, etc.) and any other item. Everything must be punched in columns 1-72 (that is to say, columns 73-80 are not used). Column 72 of one card and column 1 of the next are considered to be contiguous.

Finally, we should make the following clarifying remarks about the term "retina" as used above in the discussion of the generation of the stimuli. The stimulus generation features have a retina on which various stimuli are created and perhaps altered (e.g. translated) and from which they are eventually written on tape, (or used in some other fashion). This "retina" is <u>not</u> the same as the retina of the perceptron, but is the private property of the stimulus generation features.

3.3 Example

We wish to perform the above experiment on 100 different perceptrons. (Note that the construction of the perceptron involves random choices (in the selection of the S-A connections). A different set of random choices gives a different perceptron with the same specifications).

Solution: The random number generator may be primed by

RANDOM n,

(at the very beginning it is primed with 1, in case RANDOM is not used.)

So the desired series of experiments could be performed by making 100 copies of the deck given in 3.2, and inserting RANDOM n, with a different value of n, in each one; or alternatively by using the same deck 100 times and changing the card with RANDOM on it.

Both these methods are, to say the least, cumbersome: the system is designed to avoid the necessity for such nonsense. We simply add to the deck given in 3.2 in example 1 the following:

RANDOM $(A1, 1, 2, 3, 4, ..., 99, 100),

(of course this can't all be punched on a single card, but this is no matter; as noted in the last example, we simply continue from column 72 of one card to column 1 of the next).

Without further ado, the program will recognize that the deck is to be repeated 100 times, using the values 1,2,3,... etc. on successive repetitions. Note that it is not necessary for the sequence of values to follow any systematic pattern, e.g. we could have had

RANDOM $(A1,23,41,2,4,3,6,1,...),

(When the list of values follows such a simple pattern as in this example, and especially when it is so long, there is a simpler way to write this, as we shall see later.)

We must now consider exactly where to insert the RANDOM phrase. If we put it at the very beginning, then the random number generator would be primed before generating the sequence of stimuli, with the

result that each perceptron would be trained and tested with a different sequence. Suppose however that we wish to use the same sequence for each perceptron. We could put the RANDOM phrase after line 3, and then the random number generator would not be reprimed until after the generation of the stimuli.[1]

While this will work correctly, it is very inefficient, because the same sequence of stimuli is generated many times. It would be much quicker to generate the stimuli once, and then do each perceptron, using the same tape over and over. To accomplish this, we must place END, after line 3, thus dividing the deck into three sections, instead of two as before. Now only the second and third sections will be repeated: the second because it is the one containing the

$(A1,...)

expression, and the third because the program knows that the section containing a perceptron description is always followed by a section dealing with the training/testing of that perceptron. (Exact rules as to what sections would be repeated in more complicated situations are given later.)

Finally we note that the tape with the stimuli (T1) must be rewound before each new use. In accordance with the comment in the last example, the best time to do this is as soon as we finish using it on the previous repetition.

Putting all this together, we get:

```
1.  SSINIT 20 20, SSOTAP T1, REW T1,
2.  (60, SSHBAR 4, SSTRAN 0 $RAND 20, SSPOS, SSOUT,
3.       SSVBAR 4, SSTRAN $RAND 20 0, SSNEG, SSOUT,)
4.  REW T1, END,
5.  RANDOM $(A1, 1,2,3,...
6.       ....,98,99,100),
```

[1] In this case it would be necessary also to put RANDOM 1, at the beginning, because the generator is not reset at the start of each repetition, but keeps whatever seed it had at the end of the previous one.

```
 7.  RETINA 20 20,
 8.  LAYER 2 1000 1, LINK 2 TO 2 FROM 1 POS 3 NEG 2,
 9.  LAYER 3 1 0, LINK 3 TO 3 FROM 2 INCR 1 ALPHA,
10.  END,
11.  EC 100 T1, TEST 20 T1, REW T1,
12.  END,
```

3.4 Example

We wish to perform a series of experiments similar to example 3.2: we wish to use perceptrons with 100, 500 and 1000 A-units; with each number of A-units we wish to have a perceptron with 3 positive and 2 negative inputs per A-unit and threshold of 1, and also one with 6 positive and 2 negative inputs per A-unit, and threshold of 2;. This gives 6 different types of perceptron; we wish to test 10 different ones of each type, a total of 60 perceptrons. Finally, we wish to test each one 5 times, using 5 different sequences of stimuli.

First solution:

```
1.  SSINIT 20 20, SSOTAP T1, REW T1,
2.  RANDOM $(A4, 1,2,3,4,5),
3.  (60, SSHBAR 4, SSTRAN 0 $RAND 20, SSPOS, SSOUT,
4.       SSVBAR 4, SSTRAN $RAND 20 0, SSNEG, SSOUT,)
5.  REW T1, END,
6.  RANDOM $(A1,1,2,3,4,5,6,7,8,9,10),
7.  RETINA 20 20,
8.  LAYER 2 $(A2,100,500,1000) $(A3,1,2),
9.  LAYER 3 1 0,
10. LINK 2 TO 2 FROM 1 POS $(A3,3,6) NEG $(A3,2,6),
11. LINK 3 TO 3 FROM 2 INCR 1 ALPHA,
12. END,
13. EC 100 T1, TEST 20 T1, REW T1, END,
```

Discussion: We see several different expressions of the form $(An,...) with different values of n (to be specific, 1,2,3 and 4). The "$(A1" expression on line 6 will cause 10 repetitions (of the second and third sections) as in the previous example. The "$(A2" expression (line 8) will in its turn cause 3 repetitions of this whole group of 10, a total of 30; and similarly with the higher numbered ones.

Note 1: There are several "$(A3" expressions; as far as the pattern of repetitions is concerned they have the same effect as if there were only one of them.

Note 2: There is a "$(A4" expression in the first section; hence that section is included in the repetitions caused by it. Thus the total effect is:

Write stimuli using seed 1

generate perceptron-100-3-2-1-1 (i.e. perceptron with 100 A-units, 3 positive and 2 negative inputs, threshold 1, using random seed 1) and test it.
generate perceptron-100-3-2-1-2 and test it
generate perceptron-100-3-2-1-3 and test it
.....
generate perceptron-100-3-2-1-10 and test it
generate perceptron-500-3-2-1-1 and test it
.....
generate perceptron-1000-3-2-1-1 and test it
.....
generate perceptron-100-6-6-2-1 and test it
.....
generate perceptron-500-6-6-2-1 and test it
.....
.....

Write stimuli using seed 2

generate perceptron-100-3-2-1-1 and test it
.....
etc.

Note 3: The actual numbers used for n in the $(An expressions are unimportant; only the order matters. E.g. we could have used 3,7,9 and 10 instead of 1,2,3, and 4, and the effect would have been the same. (If we had used, e.g. 3,1,2,4 the same cases would have been done, but in a different order.)

Note 4: The reader may think that it is very inefficient to generate all the perceptrons anew for each stimulus sequence. If so, he will be right; it would be much faster to put all 5 sequences on tape one after another, and then run all 5 tests on each perceptron as it is generated. The example has been done this way because it allows illustration of certain points not brought out the other way.

Now this deck contains 300 cases. Suppose it is not possible to run them all at one time (insufficient computer time, or machine

trouble part way through, etc.). This is no trouble: suppose for example that on the first run we complete the first 73 cases--we simply add at the end of the deck a card reading

SKIP 73

Next time we run it it will skip the first 73 cases and start with number 74. Now if we complete 39 more on the second run, we could either replace the SKIP 73 by

SKIP 112

or even simpler, leave it alone and add

SKIP 39

after it. ETC.

Each time a case is completed, a message is printed giving the number of cases completed to this time.

The other major problem with the example as given is that there will be many pages of output, and no way to identify what output goes with what experiment other than by counting from the beginning. (A description of the perceptron is automatically printed as it is generated, so we could identify this much, but there is no way to tell what stimulus sequence is being used.) We can solve this by inserting after line 5, e.g.

```
5a.  HEAD BCD * SERIES A.  STIMULUS SEQUENCE *
5b.      V $(A4,1,2,3,4,5) BCD *.  PERCEPTRON-*
5c.      V $(A2,100,500,1000)  (A3,-A-,-B-)
5d.      V $(A1,1,2,3,4,5,6,7,8,9,10) BLANK 3 CP,
```

The effect of this is to create a "heading line" which will be printed at the top of each page of output. As we have set it up, the line will be different for each case. This line, which may be up to 110 characters long, is built up from left to right.

BCD * ... * — Everything between the *s is inserted exactly as it is into the heading line. Between the

	asterisks may appear any desired characters except asterisk and apostrophe. (See General Rules for Control Cards--at start of Section 4.)
V	The value of the following expression is inserted.
(A3,-A-,-B-)	The appropriate group of characters, (groups being separated by commas) is inserted, according to which A3 repetition is taking place.
BLANK 3	3 blank spaces are inserted.
CP	"CASE n" is inserted, where n is the number of the current case.

A typical heading line produced by this example is

SERIES A. STIMULUS SEQUENCE 2. PERCEPTRON-100-A-5 CASE 65

Pages will be automatically numbered at the right end of the line (this is beyond the 110 characters which may be generated). Each time a new heading line is given, a new page is started, and page numbering goes back to 1.

There is still one unpleasant thing about the example as given. Several expressions, e.g. $(A1,1 ,4,5,6,7,8,9,10), are exactly duplicated in two places. The e. ess energy involved in keypunching these is a very minor consideration; however, suppose that we make a punching error in one and not in the other. We may wind up getting the correct value printed in the heading line, but the wrong value actually being used. It would be very nice to be able to feel confident that the value in the heading line is the value actually used. This can be done as follows:

1. ALWAYS N,
2. SSINIT 20 20, SSOTAP T1, REW T1,
3. SET N $(A4,1,2,3,4,5), RANDOM N,

```
4.  (60, SSHBAR 4, SSTRAN 0 $RAND 20, SSPOS, SSOUT,
5.       SSVBAR 4, SSTRAN $RAND 20 0, SSNEG, SSOUT,)
6.  REW T1, END,
7.  SET NA $(A2,100,500,1000), SET RSEED $(A1,1,2,3,4,
8.       5,6,7,8,9,10),
9.  RETINA 20 20, LAYER 2 NA $(A3,1,2), LAYER 3 1 0,
10. LINK 2 TO 2 FROM 1 POS $(A3,3,6) NEG $(A3,2,6),
11. LINK 3 TO 3 FROM 2 INCR 1 ALPHA,
12. RANDOM RSEED,
13. HEAD BCD *TEST SERIES A.  STIMULUS SEQUENCE *
14.      V N  BCD *.  PERCEPTRON-*
15.      V NA  (A3,-A-,-B-)  V RSEED BLANK 3  CP,
16. END,
17. EC 100 T1, TEST 20 T1, REW T1, END
```

Comments:

SET NA $(A2,...), (line 7) NA will be used as the name of a variable which may be given any desired value, and later used instead of a number. SET NA..., sets the value of NA to the value of the following expression. (INSTEAD of "NA" we could have used any other name of six or less letters not starting with B.) Note how NA is used on lines 9 and 15. Similarly with N and RSEED.

Normally a variable (such as N, NA and RSEED in this example) is good for one section only. (The same name may be used in two or more sections, but the value given to it in one section will no longer be available in another.) However, in this example, we want to give a value to N in the first section (line 2) and use it in the second section (line 14). To accomplish this we put

```
ALWAYS N,
```

at the very beginning of the deck. This indicates that the name "N" is to represent the same variable always, i.e. in every section. (If we needed to do this for more than one variable, we could say e.g.
ALWAYS N JOE X, .)

3.5 Example

We wish to experiment on the activity states of a fully cross coupled perceptron with 100 A-units, 10 x 20 retina; A-units to have 3 excitatory and 1 inhibitory inputs, threshold of 1. We wish to use perceptrons with a decay of .01 and others with a decay of .05 (in the cross-connections). For each value, we want several perceptrons, with increments chosen to give the quantity Na η/δ the values 200, 250, 300, 350, and 400.

The experiments to be run are as follows: We present a series of pairs of stimuli, reinforcing the cross connections; the first of each pair is a random stimulus of area 10% (i.e. 10% of the retina points, chosen at random, are on) the second is the first translated halfway across the retina (toroidally, i.e. points translated off one edge reappear at the other). After presenting 100 such pairs (200 stimuli) we present a sequence of 10 pre-specified stimuli and their transforms (no reinforcement) and form a G-matrix (normalized) from the activity vectors. Then we present 200 more random stimuli, test stimuli again, etc. until 2000 training stimuli have been presented.

Solution:

```
1.  DATA *
2.      (cards with 10 pre-specified stimuli, as described in
        comments)
3.  *,
4.  SSINIT 10 20, SSOTAP T1, REW T1,
5.  (*I 10, SSC, SSDEFS I,)
6.  (10,
7.      (100, SSRAND .1, SSOUT, SSTRAN 5, SSOUT, )
8.      (*I 10, SSN I, SSOUT, SSTRAN 5, SSOUT, )  )
9.  REW T1, END,
10. SET RATIO $(A1,200,250,300,350,400),
```

```
11.  SET D $(B35)$(A2,.01,.05),
12.  SET DD D, FLOAT(35)DD, FIX(18)DD, MPY DD RATIO E, DIV E 100,
13.  RETINA 10 20, LAYER 2 100 1,
14.  LINK 2 TO 2 FROM 1 POS 3 NEG 1,
15.  LINK 3 TO 2 FROM 2 DELAY 1 DECAY D INCR E ALPHA,
16.  HEAD BCD * SERIES B. RATIO * V RATIO
17.       BCD *, DECAY * VB 35 D * 2 BLANK 4 CP,
18.  END,
19.  BLOCK BSAVE 20 3, BLOCK BX 3,
20.  OPEN A *N 10, PRECON 200 T1,
21.       OPEN B *I 20, STIM T1, ACTIVE, GETACT 2 BSAVE(I), CLOSE B,
22.  LINE START BOTTOM 22 BCD *G-MATRIX NUMBER * V N PRINT SPACE,
23.  OPEN C *I 20, LINE START,
24.       OPEN D *J 20, AND BSAVE(I) BSAVE(J) BX,
25.       BITS BX Q, FLOAT Q, DIVF Q $(F)100,
26.       LINE F5 VF Q*2, CLOSE D,
27.       LINE PRINT, CLOSE C,
28.  CLOSE A, REW T1, END,
```

Comments:

The first problem is the "pre-specified stimuli", meaning certain stimuli given by the experimenter which cannot be conveniently be generated within the program (as the horizontal and vertical bars were). We must have some way of communicating these to the program. There are two basic ways available; we shall illustrate the one which is more convenient for small stimuli. Each stimulus is punched as a "picture" on one card, that is, each possible position for a punch represents one point on the retina, and is punched if that point is active. The stimulus starts at column 4;

and the picture is placed sideways on the card. More exactly, the leftmost column of the stimulus is punched along the top row of the card, starting at column 4; the next column along the next row, etc.[1] Note that each stimulus will fit on a single card; it is possible to do larger stimuli on several cards, but we will not discuss this here. Each of these cards must have 7 and 9 punched in column 1, in addition to the picture of the stimulus as described above. See figure 3.5.1 for an example of a stimulus punched in this way.

The 10 cards bearing the stimuli, as above, are placed in the deck at the indicated point (line 2). They are preceded by "DATA*" (line 1) and followed by "*," (line 3). This DATA phrase is a special case of a more general facility which will be discussed elsewhere. If this facility is not used for any other purpose than that illustrated here there is no need to know any more details about it.

(*I 10, ...)	This is similar to (10,...) as discussed previously; in addition the variable I will automatically be given the value 1 on the first repetition, 2 on the second, and so on, to 10 on the last.
SSC,	One of the stimuli supplied in the DATA phrase will be placed on the retina. Each time the SSC phrase is executed the next stimulus will be brought in.
SSDEFS n,	The stimulus currently on the retina will be saved internally and can later be referred to as "stimulus n".

We see that the total effect of line 5 is to save the 10 prespecified stimuli as "stimulus 1", "stimulus 2", ... "stimulus 10".

[1] The reason for this is due to certain internal complexities in reading the cards, which need not concern us.

Now lines 6 to 9 write the stimulus tape.

SSRAND .1,	makes a stimulus by activating .1 (i.e., 10%)of the retina points, at random.
SSN n,	The stimulus previously saved as "stimulus n" is placed on the retina.

Note on lines 7 and 8, that after the stimulus on the retina is written out by SSOUT, it is still available also on the retina, (so in this example it can be translated and written out again).

Lines 10 to 12 set up the desired parameters. The variable RATIO is given the desired value of $Na\eta/\delta$. (Note: Na is number of A-units, η is increment, δ is decay.) D is given the desired value of decay and E is computed to be the appropriate value of increment.

Line 11 needs a little more explanation: If we were to write the value of the decay directly into line 15 (instead of using a variable) we could have

.... DECAY $(A2,.01,.05)...

the program when interpreting the cards, knows from the context that the numbers .01, .05 are to be used internally as fractions and represented with 35 binary places, and there is no need for us to indicate this in any special way (or even to know about it). But if we write

SET D $(A2,.01,.05),

the program cannot be expected to know what we intend to do later with D; hence it would just set D to the integer part of 0.01 or 0.05 (i.e., 0 in either case). So in this case we need to know the magic number 35 (the magic number 18 applies to <u>all other</u> cases where a fraction may be used) and indicate it to the program by the prefix $(B35)

SET DD D, FLOAT(35)DD, FIX(18)DD,	put the value of decay into DD, convert to floating point, then convert back to fixed point but with <u>18</u> binary places.
MPY DD RATIO E,	multiply the value of DD by the value of RATIO and put the result in E

Normally such an operation should be done only on integers, as fractions become tricky; in this case we have an integer times an 18 place fraction, which gives an 18 place fraction.

DIV E 100, — Same as DIV E 100 E, namely divide value of E by 100, put result back in E. As above, note that E comes out an 18 place fraction, which is what we want to use on line 15.

NOTE: it is possible to do arithmatic on fractions using these phrases, with great care and understanding; but it is more convenient usually to use floating point, as illustrated later in this example, for non-integer calculations.

Lines 13 to 18 specify the perceptron and set up a heading line, as in previous examples. The only new thing worthy of comment is under HEAD, on line 17.

VB 35 D *2 — the value of D is to be inserted into the heading line, with 2 decimal places. VB 35 indicates that the value found in D is a fraction with 35 binary places.

Lines 19 to the end deal with the testing of the perceptron and the printing out of the G-matrices. There are two major new concepts involved. The first is the LINE phrase; this is similar to HEAD as discussed previously; but instead of forming a line of print to use as a heading for all following pages, it simply forms a line which can be printed out on command. The same subcodes as in HEAD (BCD, V, etc.) are applicable and work in the same way. In addition there are several others which would not make sense in HEAD. These are as follows:

START — prepare to start a new line

PRINT — print the line currently formed
NOTE 1: the line is not automatically printed out at the end of a LINE

phrase, only when PRINT is given. Thus it is possible to use several LINE phrases to form one line, as in this example (lines 23, 26, 27): and in other situations we may like to print more than one line with a single phrase, which is also possible.

NOTE 2: "print" as used through-out this discussion actually means "write on tape A3 in suitable form for printing offline"; it does not refer to printing on the online printer unless this is explicitly mentioned.

BOTTOM n — next line printed will start a new page unless there is room for at least n lines on this page. (Note how this is used in the example at line 22 to make sure each matrix is contained on one page.)

SPACE — leave one blank line

SPACES n — leave n blank lines

The other important concept is a block, which is in some ways similar to an array in Fortran. For example, at line 19 we have:

BLOCK BSAVE 20 3,

This defines BSAVE as a 20 by 3 array. (Instead of BSAVE we could use any name of one to six letters, the first of which is B.) The space for the array is assigned when the phrase is executed; so we can redefine BSAVE with different dimensions at different points in the program; also, the dimensions could be given by variables if we so choose.

BSAVE can be considered as consisting of 60 words, which can be referred to individually as e.g. BSAVE(2,1) etc. However when more convenient we can consider BSAVE as a collection of 20 3-word vectors which can be

referred to as BSAVE(1), BSAVE(2), etc. (As we shall see shortly, some operations permit multiple word operands.) For that matter, if desired, the whole 60 word block can be used as a single operand, referred to simply as BSAVE. In this particular example the fact that the vectors are 3 words long is of no real significance to us; we simply want a vector long enough to hold 100 bits (1 for each A-unit) and due to the accident that one word holds 36 bits 3 is the smallest number to use. Note that this number is not used explicitly other than in the definition of the blocks BSAVE and BX; when the vectors are used the program takes care internally of remembering the length. (If we desired, we could avoid all explicit use of this number by writing line 19 as

BLOCK BSAVE 20 $NPACK 2, BLOCK BX $NPACK 2,

The expression $NPACK 2 automatically has for its value the required number of words to hold the packed activity vector of layer 2 in the current perceptron. This way of writing it is especially useful if we expect to be changing the number of A-units in further experiments, as there will then be no need to make changes at this point.

When a program contains several sets of nested parentheses, it may become confusing to read and easy to make errors. To avoid this, we can replace any left parenthesis by

OPEN name

where name is any name of one to six letters; and replace the matching right parenthesis by

CLOSE name

using the same name. If we use a different name for each pair of parentheses, the program becomes easier to read. Furthermore, if the interpreter discovers an incorrectly matched pair, a warning message will be printed out to call our attention to it.

We now discuss in detail the last section (lines 19 to end). Line 19 defines BSAVE to hold 20 activity vectors, and BX to be used for intermediate results.

OPEN A *N 10, ... CLOSE A,

by the remarks above we see this has the same effect as

(*N 10, ...)

This loop is done once for each G-matrix.

GETACT 2 BSAVE(I),	this gets the activity vector for layer 2 and places it in packed form in BSAVE(I)
AND BSAVE(I) BSAVE(J) BX,	this forms the bit by bit logical product of BSAVE(I) and BSAVE(J) and puts the result in BX Note that this means a bit in BX will be 1 if and only if the corresponding A-unit was active for both stimulus I and stimulus J
BITS BX Q,	Q will be set to the number of 1-bits in the vector BX (This means, in this example, that Q is now the number of A-units active for both I and J)
FLOAT Q,	turn Q from an integer into a floating point number. NOTE: the program does <u>not</u> keep track of whether a given variable contains an integer or a floating point number: it is up to us to treat it appropriately.
DIVF Q $(F)100,	DIVF is like DIV, but it works with floating point numbers. $(F)100 is a floating point 100

Under LINE on line 26 we have the following two new subcodes.

F5

print the following information in a 5 column field. Normally a number is inserted in exactly the required number of columns, e.g. 10 take two columns, while 2371 take four. In this situation, where we are printing a regular array of numbers, this is ungood; hence the use of F5 forces the numbers to line up neatly.

VF Q *2

print the value of the (floating point) number in Q, with 2 decimal places.

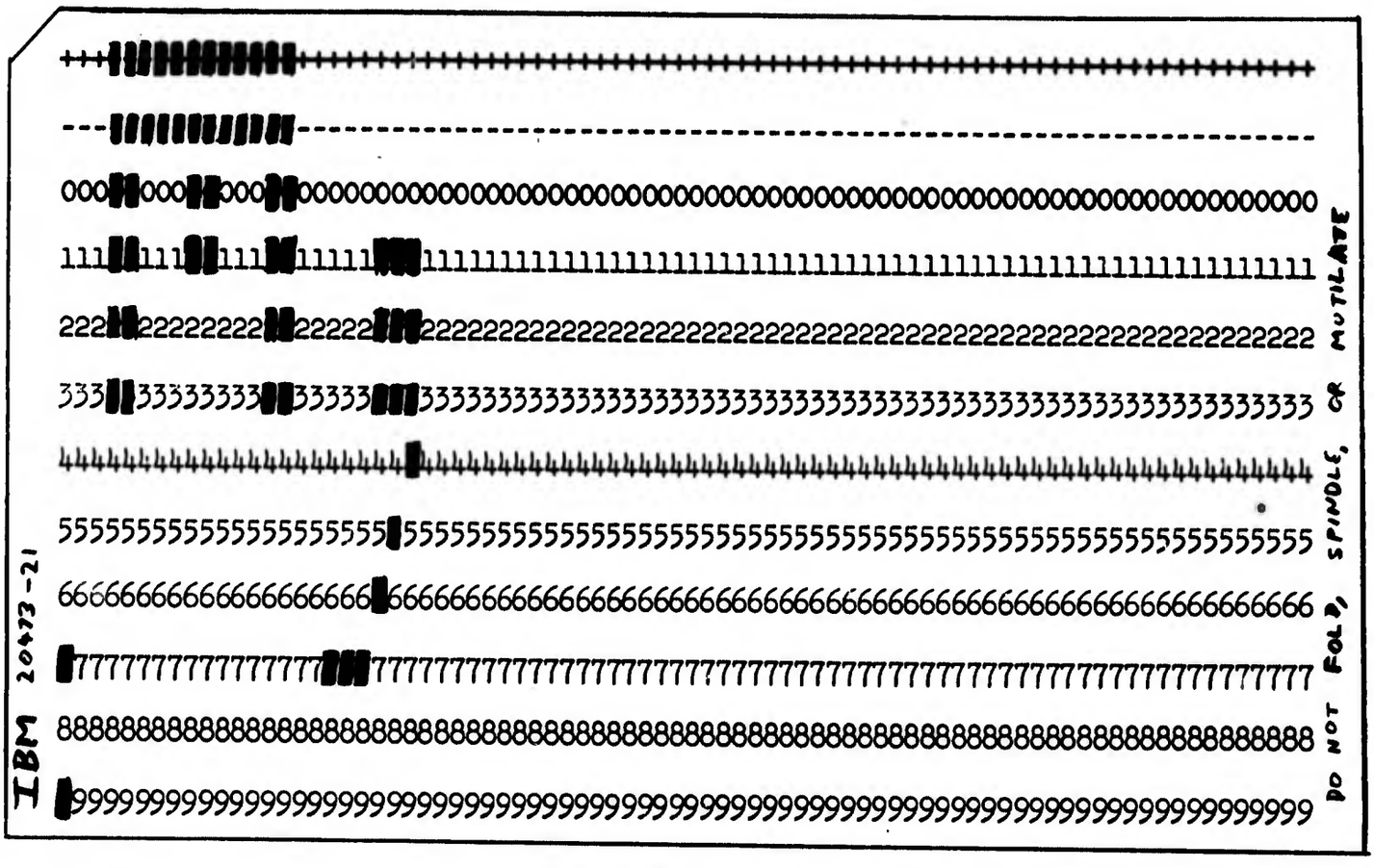


FIGURE 3.5.1

```
OOOOOOOOOOOO
O++++++     O
O++++++     O
O++         O
O++         O
O++         O
O++++       O
O++++       O
O++         O
O++         O
O++         O
O++++++     O
O++++++     O
O           O
O           O
O          +O
O          +O
O          +O
O   +++  + O
O   +++ +   O
O   ++++    O
OOOOOOOOOOOO
```

Bottom: A 10 by 20 stimulus. The 0's form a boundary to show the extent of the retina; they are not part of the stimulus. The active points are indicated by +'s.

Top: The same stimulus punched on a card, as described in Example 3.5.

BLANK PAGE

4. Detailed Description

Sections 4.1 to 4.5 introduce most of the simpler concepts, and for many users these sections alone may well be sufficient. Section 4.6 gives a rough outline of the more involved concepts to be discussed in the remaining sections; and 4.7 and following go into complete details.

The later sections need not be taken in order; in particular, it may be best to omit starred sections (*4.8, etc.) on first reading.

4.1 General Description of the Input Cards

Input cards are subject to the following general rules:

a. All information is punched in columns 1 to 72 (columns 73-80 are completely ignored). Positioning on the card is completely free, i.e., there is no requirement to make any punches in specific columns. Excess blanks are simply ignored, unless the contrary is explicitly stated.

b. Cards are considered one continuous string of characters with no special significance attached to card boundaries. That is, column 72 of one card is followed by column 1 of the next, exactly as, say, column 38 is followed by column 39.

c. The character ' (apostrophe)[1] may be used to terminate a card before column 72. E.g., if ' appears in column 45, then column 44 is followed at once by column 1 of the next card. (Whatever is punched after the ' on the current card is completely ignored-- so these columns can be used for punching remarks, if desired.) NOTHING SAID ELSEWHERE SHOULD BE INTERPRETED AS PERMITTING ANY OTHER USAGE OF THE CHARACTER '; there is no exception to this rule.

d. Normally the input is considered as a sequence of "items": there are two kinds of items, "symbols" and "numbers". A number is more or less what you would expect it to be: e.g., 17 -23 14. 14.0 -14.23 1935.246. A symbol is a string of one to six letters, or else one of the characters ( ) , $ * / (these latter are referred to as special symbols.) Two consecutive numbers, or two consecutive alphabetic symbols must be separated by at least one blank; a number and a symbol do not have to be separated from any other item. However, as noted above, it is never wrong to separate any two items by one (or more) blanks. (Of course, a blank should never occur in the middle of a number or a symbol.)

The items on the control cards are grouped into "phrases;" each phrase begins with a symbol (called a primary code), and ends with a comma; what comes between may vary widely according to the particular

[1] On some keypunches this character is - (dash) or @.

primary code, and will be discussed in detail in the appropriate place below.

In general, each phrase performs a certain action; this may be one of several kinds, e.g.

i) Specify parameters of a perceptron to be generated.

ii) Perform operations on a perceptron (e.g. present stimuli, reinforce linkages, etc.).

iii) Generate stimulus sequences for use in training/testing the perceptron.

iv) Perform arithmetic and logical operations.

v) Make decisions and alter flow of control.

vi) Print out results.

4.2 Perceptron Specification

The parameters of a perceptron, which we desire to simulate, may be specified as follows.

For each layer there is a phrase

LAYER k n t,

k is an integer, the number of the layer
n is an integer, the number of units in the layer
t is the threshold. It may be an integer or a decimal fraction, or it may be omitted if zero is intended.

e.g.

LAYER 2 50 3.47,

LAYER 3 12 ,

LAYER 4 17 1,

The first layer may be specified by LAYER 1 etc., but may instead be (and usually is) done by

RETINA h v ,

which states that the first layer is to have hv units, and in fact is to be considered as an h x v rectangular grid.

The layers of the perceptron must be numbered consecutively from 1 to the highest number. However the phrases defining them do not have to be given in order (though it will usually be most convenient to do so).

Each layer (except the first) has one or more <u>linkages</u>, or sets of inputs. All the inputs in a given linkage depend on the state of the units in a specified layer (the <u>initial layer</u> of the linkage). It may be the state of the units at the current time, or at a specified number of time steps in the past. (This number is referred to as the delay.) The initial layer and the delay are specified for the linkage, they cannot be different for the various inputs in the same linkage.[1] A layer may have two or more linkages with (possibly) different initial layers or delays or both.

[1] We shall see in later sections that there are, in fact, ways to get around this restriction.

Each linkage is specified by a phrase beginning with the code LINK followed by an integer specifying the type of linkage (the various types of linkage will be discussed in detail shortly), then by one or more subcodes specifying the initial and terminal layers, the delay if any, and other information required by the specific type of linkage. The terminal layer is specified by TO followed by the number of the layer. Similarly, the initial layer is specified by FROM followed by the number of the layer. The delay may be specified by DELAY followed by an integer, however this need not be done if the delay is to be zero. Further information required will be discussed under the specific types of linkages below.

* * * * * * * * * *

Example 4.2.A A type 2 linkage to layer 3 from layer 1 with delay 2 could be specified by either of the following:

LINK 2 TO 3 FROM 1 DELAY 2...,

LINK 2 DELAY 2 FROM 1 TO 3...,

* * * * * * * * * *

Some of the linkages involve random choices and it will usually be desired to test several different perceptrons with the same organization but different random choices. The phrase RANDOM n, can be used to prime the random number generator.

> If the reader is unfamiliar with random number generators, the following comments may prove useful. The random number generator produces a series of numbers by an iterative process, i.e., each number is produced by performing certain operations on the preceding one. (Of course, such a series of numbers is not literally "random". However the nature of the process is such that the sequence will behave statistically like a series of genuine random numbers.) The current random number, which is held in the generator until a new one is formed, is sometimes called the <u>seed</u> of the generator. To <u>prime</u> the generator is to set the seed to a specific number. If an experiment is performed several times, and the random generator is primed with a different number each time, then a different set of "random" choices will be made each time. For technical reasons it is best to prime the generator with an <u>odd</u> (positive) number; hence
>
> RANDOM n, will prime it with 2n-1.

Suppose we have a perceptron with several linkages involving random choices: we may wish to run several experiments, making different choices in one linkage but keeping the same choices in another. It is not possible to do this simply by using several "RANDOM" phrases (the reason is that the various "LAYER," "LINK," "RANDOM," etc., phrases feed the parameters to the perceptron generator itself, which does not go into operation until it has received all the parameters (see "END" below). Hence, if several "RANDOM" phrases were used, only the last one would have any effect. However, it is possible also to use "RANDOM n" as a subcode in a LINK phrase. In this case, the random generator will be primed with 2n-1 when the perceptron generator starts to generate this linkage, and after, the previous seed will be restored.

* * * * * * * * * *

Example 4.2.B.

```
RETINA 10 20,                          line 1
LAYER 2 500 3,                         line 2
LAYER 3 500 3,                         line 3
LINK 2 TO 2 FROM 1...,                 line 4
LINK 2 TO 3 FROM 1 RANDOM 7...,        line 5
LINK 2 TO 3 FROM 2...,                 line 6
RANDOM 1,                              line 7
END,
```

By changing the "7" on line 5 to another number, the random choices in this linkage would be altered, but those in the other linkages would not; and vice versa if the "1" on line 7 were changed.

* * * * * * * * * *

When the perceptron description is complete, the phrase "END," will signal the perceptron generator to go to work; it will then generate the perceptron according to the accumulated parameters.

We now proceed to a discussion of the specific types of linkage. (Note on terminology: the input signal to a unit receives contributions from

each of the relevant linkages, and the total signal is the sum of these contributions. The unit is active if this is $\geq$ the threshold. By the "value of the unit," we mean the total signal minus the threshold [hence the unit is active if its value is ≥ 0]. Similarly, in the following discussion of linkage types we frequently say something such as "... is added to the unit...", meaning "... is added to the input signal to the unit...".)

Type 1. This is normally used between two layers with the same number of units. Each unit in the terminal layer has a standard value (the weight of the linkage) added to it if the corresponding unit in the initial layer is active. Weight is specified in the LINK phrase and normally may not be changed during the testing of the perceptron. It is specified by adding to the LINK phrase the subcode WEIGHT (or alternatively the subcode W) followed by the desired weight, which may be an integer or a decimal fraction. The weight may be negative if desired. If the weight is to be 1 (which will probably by the usual case), it is not necessary to specify it as this value will be understood if no other is specified.

Type 2. This is the type of linkage used in the S-A connections of a simple (binomial) perceptron. Each unit in the terminal layer receives a specified number of positive (excitatory) inputs and a specified number of negative (inhibitory) inputs from randomly chosen units in the initial layer. As in type 1 there is a specified (fixed) value known as the weight of the linkage. For each active input this value is added into the terminal unit (or of course subtracted if it is a negative input). The weight is specified in the LINK phrase by the subcode WEIGHT or W just as in type 1. The number of positive inputs may be specified by POS (or EXCITE, or X), followed by the number of inputs, and the negative connections may be specified by the subcode NEG (or INHIB, or Y), followed by the number of negative inputs.

Normally each input is chosen at random from the entire initial layer and it may happen that two or more inputs to the same unit actually come from the same initial unit. If it is desired to prohibit this possibility the subcode NONREP may be included in the LINK phrase. If this is done all the inputs to a given unit will come from different units (of course,

two different terminal units may quite possibly each receive an input from the same initial unit).

Type 3. This is a linkage with reinforceable weights (such as is used in the A-R connections of a simple perceptron or the A-A connections of a cross-coupled system). Each unit in the terminal layer receives an input from every unit in the initial layer. Every one of these inputs has its own variable weight which may be altered according to one of the reinforcement rules discussed below. Each terminal unit has added to it the sum of the weights corresponding to the active units in the initial layer. The reinforcement rule may be one of several types (α, γ, or Γ, with or without decay, as discussed in a moment). Reinforcement is also affected by the reinforcement procedure. The type of reinforcement is specified in the LINK phrase and becomes a permanent part of the structure of the perceptron. The reinforcement procedure, on the other hand, is decided upon when the perceptron is being tested, and in fact different procedures may be applied to the same linkage at different times. Specific reinforcement procedures will be discussed in later sections. Essentially, the reinforcement procedure selects certain units in the terminal layer and an (positive or negative) increment. The increment is added to each weight if its initial unit is active and its terminal unit is one of the ones selected by the procedure. If the reinforcement type is γ, all the weights to a terminal unit will then have a suitably chosen value added to them so that their sum is 0. If the reinforcement type is Γ a similar procedure is followed but it is the set of weights from an initial unit which is balanced. If the reinforcement type is α neither of these adjustments is made. Furthermore, if there is a non-zero decay, then all weights will be reduced by a constant fraction of their original value, before the increment is added (note that a decay of e.g. .05 means that each weight is reduced by .05 of its original value, that is to say, multiplied by .95). For most of the reinforcement procedures the increment is a fixed value which is specified in the LINK phrase.

The parameters are specified as follows. The increment, by the subcode INCR (or ETA) followed by the desired value. The decay, if

non-zero, by DECAY (or DELTA) followed by the desired value (which must be less than 1). If γ-reinforcement is desired one of the subcodes LG, LGAM or LGAMMA may be given. If Γ-reinforcement is desired, one of the subcodes BG, BGAM, or BGAMMA may be given. If α-reinforcement is desired one of the subcodes A or ALPHA may be given.

Restrictions:

η, the increment, may be given to an accuracy of five decimal places, and δ, the decay, to ten places. (With "error correction" type reinforcement proceedures, it may be best to use increment values which can be represented exactly in binary-- that is, integers or multiples of 1/2, 1/4, etc.; if not, the initial behavior of the perceptron may be altered due to internal round-off effects, although in most cases this should have no effect on the limiting behavior.)

If non-zero decay is used, then the following should be satisfied

$N\eta/\delta < 2^{16}$ (about 64,000)

where N is number of units in the initial layer.

If decay is 0, there is no absolute restriction on the value of η, but it should be noted that the perceptron may not function correctly after $2^{17}/N\eta$ reinforcements, so with large values of N it may be necessary to use a small value for η.[1]

Type 4. This is similar to type 2, but all the inputs to a given unit are constrained to lie close together on the retina (NOTE: although in theory this type of linkage, like any other, may have any layer for its initial layer, it is normally intended for use with the retina as initial layer, and involves the rectangular structure of the retina). All the inputs to a given unit will be chosen within a small rectangle (or sub-retina) on the retina, the dimensions of which are specified in the LINK phrase. The actual position of this rectangle on the retina is chosen at random and a different rectangle is chosen for each terminal unit. We may specify fixed dimensions for the sub-retina or we may specify certain

[1] In any normal situation one can probably get away with many more reinforcements than this, possibly even several orders of magnitude more. But there is no larger number for which we can give an absolute guarantee, without going into complicated considerations of the specific situation.

bounds within which the dimensions will be chosen at random, so that the different terminal units may have different-sized sub-retinas. The horizontal dimension is specified by the subcode HORIZ (or H) followed by the dimension. The vertical dimension is likewise specified by VERT (or V). If we desire to have the horizontal dimension varying, we may use instead of HORIZ the subcode HVAR followed by two numbers a and b, in which case the horizontal dimension of the sub-retina is chosen at random in an interval of width b centered at a. Similarly, if we desire the vertical dimension to vary we may use VVAR in a similar fashion (NOTE: it is permissable to use HVAR and VERT, or to use HVAR and VVAR, but it is not permissible to use HORIZ and VVAR). If we wish the rectangle to be square we may use the subcode SQUARE, in which case all the subrectangles will be chosen square (even if HVAR is used). Another alternative for specifying vertical dimension is SQVAR or SVAR followed by an integer, which means that the vertical dimension will be chosen in an interval of this width about the value actually used for the horizontal dimension of this particular rectangle. The number of positive and negative inputs and the weight of the inputs is specified as in type 2.

Type 5. Currently there is no type 5.

Type 6. Type 6 is another variation of type 2. The first terminal unit will receive a pattern of inputs chosen at random on the retina, as in type 2, but then one or more of the following units will receive translations of this same pattern (the translations being chosen randomly for each unit). After this the next set of terminal units will receive translations of a new pattern and so on. The LINK phrase should contain the subcode SAME followed by the number of units which are to receive similar input patterns. The number of positive and negative inputs per unit are specified as in type 2.

Type 7. This is a generalization of type 1. Suppose, for example, the initial layer has three times as many units as the terminal layer, then the first unit in the terminal layer will have the weight added to it if any of the first three units in the initial layer are active, and similarly, the second unit in the terminal layer for the next three units

in the initial layer and so on. All that need be specified in the LINK phrase is the weight, specified just as in type 1. Type 7 is normally used in conjunction with type 6, with the terminal layer for a type 6 being the initial layer for a type 7.

* * * * * * * * * *

Example 4.2.C.

A simple perceptron with 100 A-units, 20 x 20 retina, 3+ and 1- inputs per A-unit, threshold 2, and γ-reinforcement with increment .1 and no decay, could be specified thus:

```
RETINA    20 20,
LAYER     2 100 2,
LINK      2 TO 2 FROM 1 POS 3 NEG 1,
LAYER     3 1,
LINK      3 TO 3 FROM 2 INCR .1 LGAMMA,
END,
```

Example 4.2.D.

A five layer perceptron with the following structure: layer 1, 2, 3 with a similarity-constrained inputs structure (layer 1, 10 x 30, layer 2, 500 units, layer 3, 100 units). Layer 3 serves as the "retina" of an open loop cross-coupled system (layers 3, 4, 5).[1]

```
RETINA    10 30,
LAYER     2 500 2,
LAYER     3 100 1,
LAYER     4 80 1,
LAYER     5 1,

    LINK    6 TO 2 FROM 1 X 5 Y 2 SAME 5,
    LINK    7 TO 3 FROM 2,
    LINK    2 TO 4 FROM 3 X 3 Y 1,
```

[1] This is not, of course, a complete description of the perceptron. Details not directly relevant to the example-- e.g., value of increment-- are glossed over, and filled in without comment in the solution. This will be done in many other examples too.

```
        LINK    3 TO 4 FROM 4 DELAY 1 ETA .025 DECAY .05 BGAM,
        LINK    3 TO 5 FROM 4 ETA .1 ALPHA,
    END
```

* * * * * * * * * *

Type 8. Type 8 linkages will be discussed at a later time, along with more esoteric details relating to the previous material. (Section 4.20)

All the material relating to perceptron specifications is summarized in 5.8.

4.3 Perceptron Testing

Having specified the details of the perceptron, it is now necessary to be able to conduct whatever tests we desire on it. In this section we shall introduce some special phrases which may be used to perform the most commonly used types of testing procedures. Later we shall show how less standardized types of tests can be performed.

In all the following we shall assume that the stimuli to be used in the training and/or testing have been already supplied on a tape (or tapes). (In the next section we will see how stimulus tapes can be generated in the system.)

When any of the following phrases require a tape designator, this may be one of the following forms:

An (n an integer)	tape unit n on channel A
Bn (n an integer)	tape unit n on channel B

(Cn, Dn etc., if the 7090 being used has channels B, C, D, etc.)

Tn (n an integer)	nth tape chosen by the system

Normally, a specific tape will be used (An or Bn) only if the tape was prepared separately (by another program, or during a previous use of the system). If the tape is prepared and used by the system during the same run, it is better to use T1, T2, etc.-- as this allows the system to choose any tape which happens to be available.

The phrase

TEST n τ, where n is an integer and τ is a tape designator, will cause the next n stimuli on tape τ to be presented to the perceptron, and the responses to be tabulated. (Each stimulus on tape is classified as positive or negative; each response will be classified as Correct, Zero, or Wrong, according as the response unit has the same sign as the stimulus [and is non-zero] or is zero, or has the opposite sign [and is non-zero]. Note that it is assumed that the highest numbered layer has a single unit, which is referred to as the response

unit.) The number of Correct, Zero, and Wrong responses are added into cumulative counts. When all the stimuli have been presented, the counts are printed and reset to zero. However, if n is a negative integer, then |n| stimuli will be presented, but counts will not be printed, nor reset to zero. Thus, if desired, two or more phrases may contribute to the counts.

The phrase

TESTP,

will cause the counts accumulated by previous TEST phrases to be printed. (Both actual counts and percentages will be printed.) Then the counts will be reset to zero, ready for further use.

The usual types of training (i.e. Error correction reinforcement, quantized or non-quantized, and S-controlled reinforcements) may be performed by the following phrases:

EC n τ, present the next n stimuli from tape τ, and after each one apply error correction reinforcement. (The linkages to the terminal layer are reinforced if the response is Zero or Wrong; the reinforcement is positive or negative according as the desired response is positive or negative.)[1]

EN n τ, present the next n stimuli from tape τ, and after each one apply error correction reinforcement, non-quantized. (Like the previous, but the weights are incremented by a sufficient amount to correct the error, plus the basic increment, [specified in the LINK phrase, see last section], instead of just by the basic increment, as in the quantized case.)

ES n τ, present the next n stimuli from tape τ, and after each one apply S-controlled reinforcement to the response unit. (The linkages to the response layer are reinforced positively or negatively, according

[1] If EC, EN, ES, PRECON, or PRESNT is used, one must take note of Appendix II.

as the desired response is positive or negative; no account is taken of the actual response.)

NOTE: In a given linkage, the choice of α, γ or Γ-reinforcement is considered part of the perceptron design, and is specified in the LINK phrase (as described in the last section). However, the choice between quantized and non-quantized, the sign of the reinforcement, and whether or not to reinforce at all after a given stimulus, are considered part of the training/testing procedure.

The normal type of preconditioning, as used in, e.g. a cross-coupled or a 4-layer perceptron, may be accomplished by

PRECON n τ, the next n stimuli from tape τ will be presented, and all linkages to all layers but the last, will be reinforced. (That is to say, each weight will be incremented positively if both its initial and terminal unit is active.)[1]

The stimulus tape can be rewound (i.e. moved back to the start of the sequence) by

REW τ,

* * * * * * * * * *

Example 4.3.A. We have specified a cross-coupled perceptron. We have prepared on tape T1 2000 stimuli and on tape T2 45 stimuli.

We wish to precondition the cross connections with the 2000 stimuli on tape T1, train the response layer (error correction reinforcement) with the first 20 test stimuli on T2, and count responses to the last 25.

```
REW T1, REW T2,
PRECON 2000 T1,
     EC 20 T2, TEST 25 T2,
```

Example 4.3.B. Suppose in the preceding example we wished to perform the tests after every 200 stimuli. One way to do this would be to put the

[1] See footnote on page 4.3.2.

same as the preceding example, (except of course use 200 instead of 2000 in the PRECON phrase) and then repeat this set of cards identically 10 times. However, this annoying repetition can be avoided as follows:

```
REW T1,
(10, REW T2, PRECON 200 T1,
     EC 20 T2, TEST 25 T2,)
```

The notation (10, ...) indicates that everything between the parentheses is to be repeated 10 times.

Note that since the same test stimuli are used 10 times, we must rewind tape T2 each time they are used, so the"REW T2", is within the ( ) while the"REW T1", is outside. Note that it would have been equally correct to put the REW T2, after the PRECON phrase, but somewhat less efficient, as the machine would have to wait for the tape to rewind before it could read it in the EC phrase. By putting it before the PRECON phrase, we allow the tape to rewind while the preconditioning stimuli are being presented, and when we come to the EC phrase the tape should be ready.

* * * * * * * * * *

The activity vector for layer a may be printed (as a series of 0's and 1's, 1 for active units), by

```
PACT a,
```

To make reading of large vectors less difficult we may use

```
PACT a k,
```

in which case k units will be printed on each line, making a nice rectangular array if k is suitably chosen.

To present a single stimulus and compute the activity states, but perform no reinforcement (e.g., to prepare for printing activities by PACT), we may use [1]

```
PRESNT  τ,
```

If we wish to reset activity vectors, or weights, or both, to zero, we may use the following phrases;

[1] See footnote on page 4.3.2.

CLEAR LAYER α,

set all units for layer α off; if α is 0 or omitted, do this for <u>all</u> layers.

CLEAR WEIGHT α,

set all weights to zero, for all (type 3) linkages to layer α; if α is 0 or omitted, do this for all layers.

With the phrases discussed to this point, it is possible to conduct many tests. Of course, many possibilities have not been covered. To name a few, it may be desired; i) to examine results in order to decide when to terminate the experiment, or when to terminate one part of the experiment, ii) to print selected weights, or examine the weights to determine how to continue the test, iii) to reinforce selected linkages only, at a given time, iv) to form G-matrices for one or more layers, v) to reinforce linkages and then recompute the activity states for the same stimulus, vi) to use multiple response units, vii) to use continuous value response units, and many others. To try to make codes as specialized as those above to cover <u>all</u> possible situations would be impossible; instead, means are provided for working directly with the component parts of the perceptron, e.g., examining individual units or weights, individual control over the reinforcement of different linkages, etc. Thus training/testing procedures not covered already can be built up from these more fundamental units. Detailed discussion of these matters will be postponed until a later section (4.21).

All the material covered in this section (and more) is summarized in compact form in section 5.9.

4.4 Generation of Stimuli

The system has facilities to aid in preparing a set of stimuli for use in training and/or testing a perceptron. Stimuli may be read in from cards or certain standard stimuli may be generated internally; also random stimuli of various types may be generated internally. The stimuli may be subjected to various transformations, in particular translations, rotations, dilations. Also, some or all of the stimuli generated may be printed out if desired. A large number of stimuli may be held in storage so that more or less complicated sequences of them can be produced. Before we give specific details, let us give a brief outline of how these facilities work.

When we specify that we are about to generate stimuli, the program sets aside a space which will hold a single stimulus, and which we shall refer to as the retina. (IMPORTANT NOTE: This is not to be confused with the retina of the perceptron. In normal usage stimuli will be generated and put onto tape before the perceptron is ever generated, and later used from tape, as illustrated in the previous section. In special applications it is possible to generate stimuli after the perceptron has been constructed and present them directly to the perceptron without writing on tape. In such a situation there would be two retinas, the perceptron's retina and the generation retina, which should not be confused.) Certain phrases discussed below will obtain a stimulus from various sources and place it on the retina. Other phrases will perform various transformations upon the stimulus currently on the retina. Others will provide for saving the stimulus now on the retina or writing it out on tape or printing it. We now proceed to a discussion of some of the specific phrases.

SSINIT

Before any of the following phrases may be used we must warn the system that stimulus generation is going to take place by using the phrase SSINIT followed by two numbers representing the horizontal and vertical dimensions of the stimuli. At any time we may change the dimensions by using a new SSINIT phrase.

The following phrases may be used to get a stimulus on the retina.

SSN n , — Place on the retina the stimulus which has been saved as S n (see below).

SSR n , — Choose k randomly from 1 to n and place S k on the retina.

SSZ, — Set the retina to zero, that is, all points inactive.

SSPT* ..., — "..." is a series of integers representing the retina points which should be turned on in this stimulus (the points are numbered 1, 2, 3, etc., reading from left to right and top to bottom, e.g.,

1	2	3	4
5	6	7	8
9	10	11	12

(Note that points already on will be left on; so normally SSPT* should be preceded by SSZ,)

SST τ, — Take the next stimulus on tape τ and put it on the retina (this refers to stimuli previously written on the tape).

SSC , — Get the next stimulus from the cards on the data tape (see below) and place it on the retina.[1]

SSHBAR n , — Place on the retina a stimulus consisting of a horizontal bar of width n at the top of the retina (if n is omitted, 4 will be used).

SSVBAR n , — Place on the retina a stimulus consisting of a vertical bar of width n at the left of the retina (if n is omitted, 4 will be used).

[1] If SST or SSC is to be used, one must take note of Appendix II.

SSRAND n , — Form a random stimulus and place on the retina. n may be an integer representing the number of points in the stimulus or a decimal fraction representing the fraction of the retina to be covered. (There is also provision for generating stimuli by random walk methods. These will be discussed later.)

When the user wishes to provide specific stimuli himself he may do it by the phrase SSPT* discussed above (which is convenient for stimuli with a small number of active points), or he may use the phrase SSC (which is more convenient for those with a larger number). In this case he must insert at some point the phrase DATA$ followed by cards bearing the stimuli he wishes to use. These may be in one of two forms:

(1) Binary form. Each card must have in column 1 a 7 and 9 punched. The stimulus is punched starting in column 4 with each place on the card representing one unit. The first column of the stimulus runs along the top row of the card, the next column along the next row, etc. If the horizontal dimension of the stimulus is more than 12 then additional cards must be used. The vertical dimension may not exceed 69. (See figure 3.5.1 for an illustration.)

(2) Integer form. In this form the stimulus is specified by a series of integers, (as in the SSPT* phrase). Do not use any commas or other separating marks. The end of a stimulus should be indicated by a zero.

It is perfectly permissible to put some stimuli in integer form and some in binary form; there is no need to specify in the SSC phrase in which form they are provided. (NOTE: When a stimulus is provided in binary form the card [or cards] on which it is punched belong to it alone. No part of any previous or following stimuli may be punched on one of these cards.) After all the stimuli have been given we must finish with $ and a comma, either on the last stimulus card (if it was not a binary card) or on a card by itself.

(NOTE: This DATA phrase is a special case of a more general facility which is discussed in detail later. If it is not used for any other purpose then the description given here will be sufficient.)

Having placed a stimulus on the retina (and possible having transformed it by means of one or more of the phrases to be discussed below) we may now do one of the following things with it.

SSDEFS n ,	Will cause it to be saved as S n (so that it can be later used by an SSN phrase).
SSOUT ,	Will cause it to be written out on the output tape.

If stimuli are to be written on an output tape, this tape must be specified before the SSOUT phrase is ever used. It is specified by the phrase SSOTAP τ, . The output tape may be changed at any time, that is, it is possible to write two or more output tapes if so desired.

* * * * * * * * * *

Example 4.4.A. We wish to put in the following five stimuli (dimensions 10 x 10): 1) a diagonal line from top left to bottom right; 2) a diagonal line the other way; 3) a 3 x 3 square in the top left corner; 4) a 3 x 3 square in the top right corner; 5) a line from the top left corner to the center and back up to the top right. We wish to write on tape T1 a series of 200 stimuli chosen at random from these five. We also wish to write on tape T2 100 random dot stimuli, each covering ten percent of the retina.

```
SSINIT 10 10,
SSC, SSDEFS 1, SSC, SSDEFS 2, SSC, SSDEFS 3,
SSC, SSDEFS 4, SSC, SSDEFS 5, SSOTAP T1, REW T1,
(200, SSR 5, SSOUT,)
SSOTAP T2, REW T2,
(100, SSR .1, SSOUT,) REW T1, REW T2,
DATA $ 1 12 23 34 45 56 67 78 89 100 0 10 19
     28  37  46  55  64  73  82  91   0   1   2   3  11  12
     13  21  22  23   0   8   9  10  18  19  20  28  29  30  0
      1  12  23  34  45  55  64  73  82  91   0  $,
```

NOTE: That the DATA phrase may be placed anywhere as the information contained on it is written on a special tape when the control cards are first read, whereas the other phrases are not executed until after all control cards have been read.

* * * * * * * * * *

The following transformations may be performed on the stimulus:

SSTRAN h v, the stimulus will be translated h places to the right and v places down. h may be negative for a translation to the left and similarly v may be negative for an upward translation. The translations will be toroidal, that is points shifted off one edge will come back on the opposite edge. If non-toroidal transformations are desired, we may use SSTRAN (NT) h v, .

SSROT r, the stimulus will be rotated r degrees. r may be a fraction (e.g., 32.314).

SSDIL h v, the stimulus will be dilated by a ratio of h in the horizontal direction and v in the vertical direction (h and v may be fractional). If v is omitted it is assumed to have the same value as h.

The coordinates of the center point about which the rotations and dilations take place may be specified by

SSCENT p q,

where p, a are 18 place operands. Note that the positive directions are right and down, and the top left corner is (1, 1).

* * * * * * * * * *

Example 4.4.B. We have two stimuli (dimensions 10 x 20) punched on binary cards, call them A and B. We wish to put on tape T1 a series of 100 stimuli, the odd ones being A translated at random to various positions on the retina and the even ones being B likewise.

(NOTE: This example is cheating slightly because it uses a feature which is not explained until a later section. The expression $RAND n may be used instead of an integer and represents a number chosen at random from one to n.)

```
SSINIT 10 20,
SSC, SSDEFS 1, SSC, SSDEFS 2,
DATA $
     (binary cards with stimuli)
     $,
SSOTAP T1, REW T1,
(50, SSN1, SSTRAN $RAND 10  $RAND 20, SSOUT,
     SSN2, SSTRAN $RAND 10  $RAND 20, SSOUT,)
REW T1,
```

Example 4.4.C. We have on binary cards a 20 x 20 stimulus; we wish to put on tape T1 a sequence of 100 stimuli, each of which consists of the basic stimulus rotated by a random multiple of 30°, and then translated to a random position on the retina. We also wish alternate stimuli to be compressed by a ratio of 1/2 (in both directions).

```
SSINIT 20 20, SSCENT 10.5  10.5,
SSC, SSDEFS 1, SSROT 30, SSDEFS 2, SSROT 30, SSDEFS 3,
     SSROT 30, SSDEFS 4, SSROT 30, SSDEFS 5,
     SSROT 30, SSDEFS 6, SSROT 30, SSDEFS 7,
     SSROT 30, SSDEFS 8, SSROT 30, SSDEFS 9,
     SSROT 30, SSDEFS 10, SSROT 30, SSDEFS 11,
     SSROT 30, SSDEFS 12,
DATA $
     (binary stimulus card)
$, SSOTAP T1, REW T1,
(50, SSR 12, SSTRAN $RAND 20 $RAND 20, SSOUT,
SSR 12, SSDIL .5 .5, SSTRAN $RAND 20 $RAND 20,
SSOUT,) REW T1,
```

Example 4.4.D. In the previous example, there is an improvement which can be made. When the stimulus is rotated 30°, a certain distortion is

inevitable, (due to the discrete nature of the retina). Successive rotations will tend to accumulate the distortions. This can be avoided by the following alternate method.

SSC, SSDEFS 1, SSROT 30, SSDEFS 2,
SSN1, SSROT 60, SSDEFS 3, SSN 1, SSROT 90,
SSDEFS 4, etc.

* * * * * * * * * *

Each stimulus on tape has associated with it its class. For simple applications stimuli are divided into two classes which we may call positive and negative. (Later we shall see that for more complicated situations it is possible to have as many as 18 bits to indicate the class.) When a stimulus is defined by SSPT, SSHBAR, SSVBAR, SSZ, or SSR it is a positive stimulus. Stimuli obtained by SST will have the same class that they had when they were written on tape. Stimuli obtained by SSN or SSR will have the class they had when they were saved by an SSDEFS phrase. Stimuli obtained from data tape by SSC will be positive unless otherwise specified in the DATA phrase, as follows:

(1) Binary form. A negative stimulus can be indicated by a twelve-punch in column 2 (of the first card of the stimulus if it has more than one).

(2) Integer form. The word NEG inserted before the numbers defining the stimulus will cause the stimulus to be negative.

The class of the stimulus on the retina may be changed by one of the following phrases:

SSPOS ,	Will set the stimulus positive.
SSNEG ,	Will set it negative.

* * * * * * * * * *

Example 4.4.E. We wish to put out a series of 200 horizontal and vertical bars, alternating; we wish the width of the bar to be chosen at random from 1 to 5, and we wish the bars to be placed at random positions on the retina. We wish the horizontal bars to be positive and the vertical bars to be negative.

```
SSINIT 20 20,
SSHBAR 1, SSDEFS 1, SSHBAR 2, SSDEFS 2,
SSHBAR 3, SSDEFS 3, SSHBAR 4, SSDEFS 4,
SSHBAR 5, SSDEFS 5,
SSOTAP T1, REW T1,
(100, SSR 5, SSPOS, SSTRAN $RAND 20, SSOUT,
    SSR 5, SSNEG, SSTRAN $RAND 20, SSROT -90, SSOUT,)
REW T1,
```

* * * * * * * * * *

In a later section (4.22) we shall consider more esoteric details, such as (1) naming stimuli (so that the name may later be used in the output, to make it easier to read); (2) generating stimuli by random walk processes; (3) using more general features described later, in conjunction with stimulus generation features, to handle more complex problems; and others. The features already discussed should suffice to handle many situations, especially the simpler ones.

The above material, and more, is summarized in 5.14.

4.5 Automatic Repetition and Skipping

Many times it is necessary to run a series of experiments which differ from one another only in having different values for one or two parameters. See, for instance, Examples 3.3 and 3.4 in Section 3. Automatic repetition is caused by replacing one or more parameters by an expression of the following form

$$\$(An, a_1, a_2, \ldots, a_k)$$

This is referred to as a counter-list and its appearance causes the automatic creation within the program of a mechanism referred to as an auto-counter, or simply a counter. n is an integer which serves to identify this counter. Two or more such expressions with the same value of n will all relate to the same counter, referred to as counter An. a_1, a_2 etc. are the values which we wish to be given to the parameter which was replaced by the counter-list. The length of the counter-list is the number of values provided, e.g., k in the above example. All lists relating to the same counter should have the same length. Lists relating to different counters may have the same or different lengths. The length of the list(s) relating to a counter is called the range of the counter.

If a single counter exists, the system will automatically repeat the deck a number of times equal to the range of the counter. The counter is given a value of 1 on the first repetition, 2 on the second, and so on. On any repetition the counter list $\$(An, a_1, a_2, \ldots, a_k)$ will behave as if it had been replaced by a_i, where i is the current value of counter An. If there is more than one counter, the program will be repeated a number of times equal to the product of their ranges. First of all, the lowest numbered counter (that is, the one with the lowest value of n) will take successive values as above, while the others will all have the value 1. Then the next lowest will be given the value 2 and the lowest will again take all values from 1 up to its range, and so on. When the second has received its maximum value, the third (if there is one) will be given the value 2 and the first two will start again at 1, and so on. Note that this implies that if two counter-

lists are used, with different ID numbers, then every possible pair of values will be used eventually; whereas if they have the same ID number, corresponding values in the two lists will be used simultaneously, but other combinations will not be used.

In the preceding discussion we stated simply that automatic repetition causes the entire deck to be repeated, and this needs some qualification. The input deck is divided into sections, each section being terminated by the code END. As mentioned in Section 4.2, END must follow the phrases specifying perceptron parameters. Thus the deck contains a minimum of two sections, the first specifying the perceptron and the second testing it. Usually this simple division into two sections is all that is necessary. However, in some cases it is desirable to make three or more sections. One such situation was illustrated in Example 3.4 where we wished to do certain operations (in this case, creating a stimulus tape) only once for several perceptrons. In these cases, automatic repetition does not necessarily include the entire deck. The sections which are repeated due to the existence of a certain counter are referred to as the scope of the counter. The scope always includes any section in which the counter is explicitly used (that is, in which a counter-list is written relating to this counter, or one of the other expressions involving counters, which will be discussed later, is written); it also always includes the section specifying the perceptron and the following section which tests the perceptron. For example, note in Example 3.4 how, as explained in the discussion there, counters A1, A2 and A3 cause repetition of the second and third sections only while A4 causes repetition of all three sections, because it appears in the first section. If there had been an A5, its scope would also have included all three sections, even if it did not appear in the first section, because its scope must include the scope of any lower numbered counter.

A deck containing several counters may have a very large total number of repetitions, e.g., the one given in Example 3.4 has a total of 300. It is usually not feasible to run such a deck completely at a single turn on the computer. Therefore, we need some mechanism which will allow us to rerun the deck and skip over those cases which have already been completed.

This is provided by SKIP cards. A card added at the very end of the deck reading, e.g.,

SKIP 13,

will cause the first 13 cases to be skipped, and the program will begin on the 14th case. SKIP cards are accumulative, that is to say, if on the second run we complete 107 more cases, we can simply add another SKIP card reading

SKIP 107,

Note in particular the following point. Suppose the deck given in Example 3.4 is run with a card reading "SKIP 97," this means we begin with the ninety-eighth case. Now the first section would not be performed as part of case 98 if the deck were being run without SKIP cards. It would have been performed at the beginning of case 61 and would have written the desired stimuli to be used by cases 61 to 120. Now, however, it will be run before we commence the second and third sections for case 98. In other words, this first section should not be thought of as somehow becoming part of cases 1, 61, 121 and so on, but rather it should be thought of as being run once for cases 1 to 60, once for cases 61 to 120, and so on; and if certain cases are skipped it will nevertheless be run before the first case in each group (with the appropriate value for counter A4 of course). This is, of course, the way we would like it to happen in most situations, e.g., in the above example the stimuli written on tape on a previous run will no longer be available when we come back to the computer on another day and the appropriate set must be rewritten before the first perceptron of the new series can be tested.

While the deck is running a message will automatically be printed (on the online printer) giving the number of the case just completed, the number of cases completed on this run, and the amount of time used. The message will also give instructions for punching the appropriate SKIP card to restart the job from this point. Thus the running of the job can be entrusted to a person who has no knowledge of the program.

Another kind of situation where we might like to have more than two sections is the following. Suppose we wish to perform certain tests on

a series of perceptrons using certain specified sets of parameters; we wish to test ten different perceptrons for each combination of parameters, to accumulate the data obtained from all ten, and to print out some statistical results, e.g., averages and standard deviations for this data. To accomplish this we would have an extra section following the one which performs the testing procedure, e.g., the deck might look as follows:

```
(phrases creating stimulus tape)
END
(phrases specifying perceptron)
RANDOM $(A1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
END
(phrases performing statistical analysis and printing out
     results for ten perceptrons)
END
```

If A1 were the only counter involved, this would mean that sections two and three, dealing with the creation and testing of the perceptrons, would be repeated ten times, and then the fourth section would perform once to do the required statistical analysis and print out results. (The methods by which the results can be saved and made available to the last section will be discussed later. These matters are not necessary to an understanding of the points being illustrated here.) Now if other counters are also involved, e.g., sets of perceptrons with different values for certain parameters are to be tested, then these counters must include the fourth section in their scope. However, they will probably not appear explicitly in the fourth section. We may accomplish the desired result by including the phrase

```
EXTEND A2,
```

in the <u>third</u> section. This will cause the scope of A2 to be extended one section beyond the section containing the **EXTEND** phrase. This phrase may be given for each of the counters we wish to extend; however, it is sufficient to give it for the lowest numbered one only as all higher numbered ones will be automatically extended.

Now there is one problem which will come up if the number of cases is so large that we have to make use of SKIP cards. Each perceptron will be considered a separate case and an "end of case" message will be printed after each one. However, if the job were interrupted after, say, case 17 and restarted later at case 18 the results which are saved from perceptrons 11 to 17 will of course no longer be available. Hence it would be necessary to insist on running multiples of ten cases at a time.[1] We can make things nicer by including somewhere in the scope of counter A1 (that is to say, the second or third section) the phrase

BIND A1,

This tells the program in effect that we do not wish to consider each repetition due to counter A1 as a separate case. Thus, if the deck is run with this addition, case 1 will consist of the first ten perceptrons and the following statistical analysis; case 2 the next ten perceptrons and their statistical analysis; etc.

The general rule is as follows. A counter is said to be bound if it is mentioned in a BIND phrase within its scope. If any counter is bound, then all lower numbered counters, if any, are automatically bound also. A case consists of one repetition of the lowest numbered unbound counter; the case end message will be printed at the end of the last section in the scope of this counter.

Now suppose that we have a certain deck involving a total of 100 cases, and suppose that we have run the first 31 of these but for some reason we wish to run again cases 5, 6, and 7 (for example, the results for these cases may have been damaged or mislaid). We could place at the end of the deck either of the following cards:

SKIP 4 -3 24

or

SKIP 4, DO 3, SKIP 24,

[1] Even this is not quite enough, because the end case message for case 10 will be printed as soon as the last section in the scope of counter A1 [that is, the third section] is completed.

The effect of either of these is the same; it causes the first 4 cases to be skipped, the next 3 to be done as usual, and the next 24 to be skipped.

As another illustration, consider the following:

* * * * * * * * * *

Example 4.5.A

Suppose we wish to perform experiments on perceptrons with the following sets of inputs to A-units: 2 positive and 1 negative, threshold 1; 3 positive, 1 negative, threshold 1; 6 positive, 6 negative, thresholds 1, 2 and 3; 8 positive, 8 negative, thresholds 1, 3 and 5. We wish to use perceptrons with 100, 200, 500, and 1,000 A-units. However, we are not interested in the 500 unit perceptron for the 8,8 cases nor in the 1,000 unit one for the 6,6 or 8,8 cases. This can be conveniently set up as follows. Use $(A4,100,200,500,1000) for number of A-units; $(A3,2,3,6,8) and $(A3,1,1,6,8,) for number of positive and negative connections respectively; and $(A2,1,2,3,5) for threshold. This will make a total of 64 cases, of which 23 are the ones we want and the others are of no interest to us. Now we simply supply a SKIP phrase to pick out the ones we are interested in, e.g.

SKIP -1 3 -1 3 -3 1 -1 1 -2 -1 3 -3 1 -1 1 -2 -1 3 -1 3
-3 5 -1 3 -1 11,

One further point may be illustrated by this example. We would like to be able to consider the deck, complete with SKIP card as described above, as a normal deck with 23 cases. In particular, if we perform several cases, e.g., 5, on the first run we would like to be able to restart by adding a SKIP 5 card at the end of the deck as usual. However, this will not work correctly as the SKIP 5 will simply be added to the end of the list already given. We could, of course, simply change the previous SKIP card but this is a cumbersome process (especially if we want to entrust the running of the deck to someone who doesn't understand the program). This problem can be resolved as follows. The word REST is added at the end of the SKIP phrase as given above (or alternatively "SKIP REST," can be added after that phrase). Now the deck will behave like a normal deck with 23 cases and we can

completely forget about the fact that it contains these SKIP cards. A SKIP 5 added at the end will cause skipping of the first 5 not already skipped by the earlier SKIP phrase; that is to say, the first 5 of the cases we are actually interested in.

* * * * * * * * * *

Example 4.5.B

Suppose in the previous example we wish to test ten perceptrons of each type. We could take one of two approaches. If we wish to test all ten of one type before proceeding to the next, we can use counter A1 for the random seed, and change the SKIP phrase to read SKIP -10 30 -10 ... etc. On the other hand, if we wish to test one perceptron of each type, then a second of each type and so on up to ten, we could use counter A5 for the random seed. In this case the SKIP phrase would have to be changed to repeat the series of numbers given above ten times; or equivalently, repeat the entire SKIP phrase as given ten times. We can simplify this, however, by writing instead

(10, SKIP -1 3 -1 ...)

The effect of the (10,...) is to cause everything contained within to be repeated ten times. (This is a general facility pertaining to all phrases, not just specifically SKIP phrases; it will be discussed in more detail in Section 4.10. We mention it here to illustrate the fact that when necessary SKIP phrases can be combined with any of the other features to be discussed later, although normally this will not be necessary.)

* * * * * * * * * *

One final point about counters. From what we have said so far the value of a counter is not directly accessible, but is only used indirectly in picking out one of a list of values given explicitly. Sometimes it would be more convenient to be able to use the value itself. For example

$(A3,1,2,3,...,99,100,)

seems an awfully cumbersome and round-about way of saying that when the value of the counter is n we wish to use the value n; and in general in any situation where we have a counter-list whose values follow a simple pattern it may be move convenient to be able to compute them from the value of the counter rather than writing them all out.

The expression

$CTV An

may be used to represent the current value of counter An. However, if we use this form instead of a counter-list then we must specify the range of the counter and this is done by using the phrase

CTOP An m,

where m is the desired value for the range of Counter An.

* * * * * * * * * *

Example 4.5.C.

Suppose we wish to perform tests on certain perceptrons using respectively 100, 200, 300, ... 2000 A-units. We could do this as follows:

```
CTOP A2 20,
MPY 100 $CNV A2 NA,
LAYER 2 NA 1,
```

* * * * * * * * * *

4.6 Shape of Things to Come

We have seen in the previous sections that many phrases consist of a primary code followed by one or more numbers. In certain circumstances we replaced one or more of these numbers by an expression such as $(A1,100,200,500). In one or two places we made use of an expression which we referred to as a tape designator. These are three examples of what we shall in general refer to as operands. In some situations it will prove convenient to be able to use several other forms of operand. For example, suppose we are performing some form of test on a perceptron (for the purpose of this example we need not specify just what this test might be), and we wish to count the number of times this test is successful. We can do this by using a variable. We make up any name which suits our fancy provided that it consists of six letters or less and the first letter is not a B. In this particular situation we might decide on COUNT. This may be used as an operand in many phrases. For example, we could start by

SET COUNT 0,

which would set the value of the variable named COUNT to 0. Then at some suitably chosen point in the testing procedure we could put

ADD COUNT 1 COUNT,

which would add the current value of the variable COUNT to the current value of the number 1 (that is to say, 1) and put the answer into the variable COUNT; or to express things in simpler terms, COUNT is increased by 1.[1] Another situation where a variable might be of use might be the following. Suppose we have a fairly complicated testing procedure in which a certain parameter is used in many different places. Suppose that we expect to have to change this parameter in different experiments. If we write it as an explicit number all through the deck, each time we wish to change the value we must change it in all these many places, a situation

[1] SET, ADD, and other similar phrases brought into this and subsequent examples in this ad hoc fashion will be discussed systematically and in more detail in their proper place in a later section. They are simply brought in here as needed for purposes of the examples.

which easily leads to errors, as one only has to overlook one of the many occurrences or punch an incorrect value in one of them to ruin the whole experiment. Therefore, it would be much more convenient to give this parameter a name, use a SET phrase at the beginning of the deck to give it the desired value and then use this name wherever the parameter is needed. Now when we need to change the value we simply have to change it at one place. Another advantage, if the deck is fairly complicated, is that if we choose a name which has some mnemonic significance for ourselves, the deck becomes much easier to read.

These and several other types of operands will be discussed in the following sections. Section 4.7 will cover the simpler ones while 4.8 will go into more complicated ones, which may be ignored unless one is trying to cover all the finer points.

Sometimes it is necessary to handle information which cannot be simply expressed as a single number. For example, in some situations it is convenient to process the states of the units in one of the layers of the perceptron as an "activity vector," that is, a string of ones and zeros indicating the active and inactive units. If such a string is short enough (that is, if the number of units is small enough), it could be fitted into a single machine word and treated, as far as the 7090 is concerned, as a single number. However, if we took this approach it would mean that longer strings would have to be broken up among several words and we would have to go to extra trouble in processing them, being continually aware of the number of words involved and processing each word individually. To avoid this, there exist operands (known as extended operands) which may consist of more than one machine word, and some phrases are capable of processing such operands without the user's needing to be continually aware of exactly how many words are involved. Another aspect of extended operands is that a whole group of related pieces of information, which may be numbers, activity vectors, or other things, may be grouped together in a single (extended) operand. It is possible (using a subscript type notation) to operate with individual items in the group, while in other circumstances some phrases will allow operations to be conducted simultaneously on all members of the group. These and related matters are discussed in some detail in Section 4.9, which, like 4.8, is optional at first reading.

In many cases it is necessary to repeat a phrase or series of phrases many times and it is awkward to have to write then out over and over, sometimes even impossible (for example, if the number of repetitions depends upon variable parameters).[1] Often it is desirable to have an "indexing variable" which takes a series of different values on successive repetition. Methods of accomplishing this are discussed in Section 4.10.

Sections 4.11 to 4.13 discuss the basic data manipulating features; arithmetic operations, logical operations (particularly useful in dealing with activity vectors), and testing operations by means of which the future course of the program may be determined by results of previous operations.

In some more complicated situations it often happens that a certain group of phrases occurs at several different points in the program, sometimes identically in each case, sometimes with slight variations. In such a situation the program can be simplified considerably by using "subsequences" as discussed in Sections 4.14 and 4.15.

In very simple experiments, the user may be content to rely for his final results upon information automatically printed out by certain of the perceptron testing phrases, as discussed in Section 4.3. However, in more complicated situations he will probably wish to print out other information and to have some control over the format in which it is printed in order to make it as readable and convenient to use as possible. Very flexible means exist for doing this; the basic usages are discussed in 4.16, while more complicated aspects are gone over in 4.17.

Sections 4.18 and 4.19 discuss special features for allowing the user to make use of the magnetic tape units, but these are not likely to

[1] What we are referring to here is not, of course, the same thing as was discussed in Section 4.5, which dealt with repetition en masse of entire sections of the program, each repetition usually representing a complete new experiment. Here we are referring to the repetition of some simple operation taking place in the course of a single experiment, e.g., we may wish to perform some operation or test separately upon each of the A-units.

be of use except in quite complicated situations.[1] (As an example of a situation where they might be useful, suppose that we are dealing with a very large perceptron and we wish to save the activity vectors from a large number of stimuli and perform some operation [e.g., forming a G-matrix] which cannot simply be performed on one activity vector at a time. It may be that there would be insufficient room in the machine to save all these activity vectors while the perceptron is using up most of the space. In such a case we could use the features described in 4.18 to save each activity vector on tape as it is formed and then when we are finished with the perceptron and have more room available, could bring them all back in at once and do what we want with them.)

Sections 4.20, 4.21, and 4.22 go into more advanced details of perceptron specification and control and stimulus generation than could be considered before.

[1] Of course the magnetic tape units are made use of implicitly in several ways, e.g., stimuli are put on tape by the stimulus generation phrases and read from it by many of the perceptron testing phrases. Knowledge of the material in 4.18 and 4.19 is not necessary for implicit uses such as this, but only when the user desires to explicitly make other use of the tape units, e.g., as in the example given in the text.

4.7 Operands (Basic)

There are three important classes of operands, which we will call simple operands, standard operands, and extended operands. Each of these classes contains the preceding; that is to say, any simple operand is a standard operand, but some standard operands are not simple, and similarly, any standard operand is an extended operand but not all extended operands are standard. Most phrases will accept any standard operands and throughout the rest of Section 4 we shall assume this wherever the contrary is not explicitly stated. Most of the exceptions will turn out to be either (1) situations where only a simple operand is allowed but any simple operand may be used; or (2) situations where not merely any standard operand but any extended operand may be used: or (3) situations where a tape designator is required. Only in very rare cases do we come upon a more exceptional exception.

One form of a simple operand is a number, i.e., an integer of up to 10 digits, which may be optionally preceded by a minus sign.[1] Another is a variable, which is a name of 1 to 6 letters, the first of which is not a B. (Names beginning with B are used for a special purpose -- see Section 4.9.) It represents a value which may be changed by various phrases during the course of the program (as in the example in the last section). A variable preceded by a minus sign (e.g., -JOE) is also permitted as a simple operand.

An expression of the form

v+a

or

v+-a

(where v is any variable and a is any simple operand) is not eligible as a simple operand, but is one form of standard operand.

[1] Of course, this latter depends on context-- e.g., we would obviously not use RETINA 5 -3, because a retina -3 units high does not make sense. The reader may object that in 4.3 we allowed such a thing as "TEST -20 T1"; however, note that a meaning was explicitly given for such a situation. We are not trying to present -20 stimuli; we are presenting +20 stimuli, and the - simply indicates an optional way of treating them which would not be used if we said "TEST 20 T1".

* * * * * * * * * *

Example 4.7.A. Suppose JOE, PETE and UGH are variables, and suppose they have the following values

JOE 3
PETE 2
UGH -5

The expression is a	variable?	simple operand?	standard operand?	with value
JOE	Yes	Yes	Yes	3
-PETE	No	Yes	Yes	-2
-UGH	No	Yes	Yes	5
7	No	Yes	Yes	7
JOE+3	No	No	Yes	6
JOE+-4	No	No	Yes	-1
JOE+PETE	No	No	Yes	5
JOE+-PETE	No	No	Yes	1
JOE+-UGH	No	No	Yes	8
JOE-PETE	No	No	No!	
-JOE+2	No	No	No!	
3+JOE	No	No	No!	
2+3	No	No	No!	
JOE+PETE+2	No	No	No	

* * * * * * * * * *

In a later section we shall see that the phrase

ADD α β γ,

where α, β, γ may be any standard operands, has the following effect. The current value of α is added to that of β and the result is put into γ. E.g.,

ADD 2 3 JOE,

would make the value of JOE to be 5, while

ADD JOE -1 JOE,

would reduce the value of JOE by 1. If the value of A is 3 and the value of UGH is 5, then

ADD A+4 -UGH SAM,

would make the new value of SAM, 2.

However,

ADD 3 2 6,

would not make sense, because 6 has a fixed value. Hence, when we said above that "γ may be any standard operand", we did not really mean it. This illustrates an important point. All operands (simple, standard, or what-have-you) are classified as changeable or unchangeable. (Thus a variable is changeable but a number is unchangeable.) To be perfectly correct we should say (in reference to "ADD α β γ,") that "α and β may be any standard operands and γ may be any changeable standard operand." However, the fact that γ must be changeable is so immediately obvious that we do not clutter things up by mentioning it. The statement that a certain operand may be "any... operand" is always subject to the qualification "changeable" when the context implies this.

Although, as stated above, it is always obvious whether or not a given context requires a changeable operand, it is not always so obvious whether a given kind of operand is changeable or unchangeable. This will always be carefully stated when a new kind of operand is described. For example, one could easily imagine a reasonable interpretation for a phrase such as

ADD JOE 3 -A,

or

ADD JOE 3 A+2,

However, the interpretation does seem a little forced. But what really counts is not how natural or unnatural an interpretation one could imagine; what counts is whether the program can do it, and in these cases it cannot. Therefore, both these types of operand are unchangeable.

Of the types of operand we have discussed so far, variables are changeable and all others are unchangeable.

One other important point must be made about variables. A variable is good for only one section; that is to say, the same name may be used in more than one section but values will not be preserved from one section to another.[1] (The value of any variable will be zero until it is made otherwise by the program.)

In addition to the operands already discussed, there is a whole class of operands referred to as special operands or $-operands. These fulfill various special functions and each particular one must be discussed separately. A few of the simpler ones will be discussed in a moment and others will be brought up in later sections wherever they fit most appropriately. (For example, many $-operands are for the purpose of aiding communication with the perceptron and these will mostly be discussed in Section 4.21; while others which relate to very special features of the stimulus generation setup are discussed in 4.22.) The general form of a $-operand is as follows: $ followed by an alphabetic name identifying the particular operand, followed possibly by further information, depending upon the specific $-operand involved. Some $-operands are admissible as simple operands, others are standard but not simple, and others are extended but not standard. Likewise, some are changeable, some are unchangeable. This will be specified as each is discussed, but for convenient reference Section 5.2 contains a complete list of all $-operands with a brief reminder of their function and an explicit indication as to which are simple, standard, or extended, changeable or unchangeable.

One kind of $-operand was used in Section 4.4, example 4.4.B. This is

$RAND a

where a may be any simple operand. The value of a should be a positive integer n and the value of the expression will be an integer chosen at

[1] Normally this restriction should not be troublesome; there is a way around it which will be discussed in 4.8.

random from 1 to n. Each time the phrase using this expression is executed, a new random choice will be made.

For another example, consider the units in a given layer of the perceptron. These are nominally two-state devices which are "on" if the sum of their inputs is greater than or equal to the threshold, otherwise "off". If we call the sum of the inputs minus the threshold the value of the unit, then we can say a unit is "on" if its value is greater than or equal to zero, otherwise "off". In some circumstances, it is useful to be able to have access to the values of the various units. The expression

$UNIT a (c)

has for its value the value of unit number c in layer a. This is a changeable standard operand (a and c may be any simple operands).

* * * * * * * * * *

Example 4.7.B. Suppose that in a certain experiment we have a 3-layer perceptron with 3 response units and we wish these to have different thresholds, say, 1, 2, and 3 respectively. In the normal course of events, all units in the same layer must have the same threshold, but now we see a way around this restriction. We could, for example, specify a threshold of 1 in the perceptron specifications and then in the testing procedure immediately after the phrase which causes computation of the activity states insert the following

SUB $UNIT 3 (2) 1, SUB $UNIT 3 (3) 2,

To understand this, we must note first that "SUB a b c," as you might expect, subtracts the value of b from the value of and puts the result in c. We must also note that if we wish to put the result back into a, instead of writing "SUB a b a," it is sufficient to use the abbreviated form "SUB a b,". This is particularly convenient when, as in this example, a is a fairly lengthy expression. Now we see that the effect of the above phrases is to reduce the value of the second unit by one (giving the same effect that a threshold one larger, that is to say, a threshold of 2, would have had) and similarly for the third unit.

Example 4.7.C. Suppose we wish to have a perceptron with a threshold of 1 for the A-units; however, if for any given stimulus more than half the units become active, we wish to increase the threshold to 2. To do this we could specify threshold 1 in the perceptron specifications. After the phrase which causes computation of the A-unit activity we insert some test of the number of active units. (Later we shall see there are several ways of doing this. For purposes of this example, we don't care how it is done.) If more than half are active, we do the following:

```
SUB $UNIT 2 1,
```

Note that the expression "$UNIT 2" is not the same as the expression given above; specifically, only the layer number, 2, is indicated but no unit number. This expression represents not merely the value of a single unit but the entire set of values of all the units in layer 2. It is an example of what is called an extended operand. It so happens that the phrase SUB is capable of accepting extended operands. Various possibilities will be discussed in a later section, but in this particular case what will happen is that 1 will be subtracted from each of the values in the first operand; that is to say, the value of each unit is reduced by 1 as is desired.

* * * * * * * * * * *

The above example has introduced one kind of extended operand, which is in effect a collection of several standard operands. Another kind of extended operand is one which holds some type of information which will not fit into a single operand, as mentioned briefly in 4.6. A few $-operands are extended operands, but the most common type of extended operand is the block, a concept which will be discussed in detail in Section 4.9. Note that any standard operand may be considered as an extended operand consisting of just one word and hence may be used in any context where an extended operand is permitted.

In previous sections we have seen several places where a fraction, rather than merely an integer, is permitted (e.g., in specifying thresholds). One should beware of using a variable in such a context. The

casual user should probably have no need to do so anyway, and if he resolves not to, then he may safely skip the remainder of this section.

For the reader who is still with us, we expound further. The internal representation of a number may use any number of bits from 0 to 35 for a fractional part. Numbers used in a context where only an integer is permitted are translated into an internal representation with 0 fractional bits. However, numbers written in a context where a fraction is permitted will be translated into an internal form with 18 fractional bits.[1] Therefore, if we use a variable in a context (such as a threshold) where a fraction is permitted, the variable must have the expected 18 fractional bits. For example, if we have a program containing

```
. . . . .

SET TH 2,

. . . . .

LAYER 2 100 TH,

. . . . .
```

then this will <u>not</u> work correctly because the 2 will be interpreted with 0 binary places rather than the desired 18.[2] To get around this we would have to use the following notation

```
SET TH $(B18)2,
```

The notation "$(B18)" is an indication to the program that the following number is to be interpreted with 18 binary places, <u>even though this is not what is normally expected in this context</u>. (Note that when we give a numerical value for the threshold, it would be correct to write, e.g.,

```
LAYER 3 50 $(B18)3.5
```

[1]There is just one exception to this and that is the decay rate in a type 3 linkage which has 35 fractional bits.

[2]Note that it would not help to write 2.0 because the program determines the number of binary places by context, not by the way the number is written.

However, the "$(B18)" is redundant because in this case it is implied by context.) Warning: One should normally avoid doing arithmetic with values containing fractional bits. It is quite safe to add or subtract two fractions with the same number of fractional bits or to multiply or divide a fraction by an integer. Any other situations, however, should be treated with great care and only if absolutely necessary; the relevant parts of 4.8 should be read before doing such things.

We shall use the terminology n-place operand to mean any standard operand used in a context where a representation with n fractional bits is expected, e.g., a complete specification of "LAYER n α p," would state that "n is a positive integer, α may be a standard operand, and p may be an 18-place operand".[1]

[1] There is a reason for the motley assortment of characters (n, α and p). In section 5 the convention is employed of using certain groups of characters for certain types of operands (e.g., α, β, γ, δ for standard operands, k, l, m, n for integers, etc.). This convention allows the elimination of vast amounts of wordage from the concise description given in Section 5. Although these conventions are not depended upon in Section 4 we shall in general try to abide by them.

*4.8 Operands (Exotic)

This section gives further information about operands. This material can be ignored by the casual user, and in any case should probably be omitted on first reading.

A constant is a piece of information which does not change during the course of the program, and which will fit into a single machine word. The most obvious example is a number; another we have already disucssed is a tape designator. In the last section we also discussed the form "$(Bk)n" which is used when we wish n to be interpreted as having k fractional bits, but the program cannot tell this itself from context. (E.g., "SET TH $(B18)2," when TH is to be used as a threshold.)

In some circumstances it is useful to have an operand whose value is a particular pattern of bits. This could be achieved by using the integer whose representation happens to be the desired pattern, but this is not always convenient. It is easy enough to see that, e.g., 1101 is 13, but what is 111111000111? We may write this as "$(2)111111000111"; the 2 indicates that the following number is in base 2 (binary) rather than the usual base 10 (decimal). We could also, if we choose, utilize the fact that it is quite easy to translate binary to octal (base 8) and write this as "$(8)7707".[1] This may also be written "$(Ø)7707" (Ø for octal).[2]

Certain other forms of constant with specialized applications will be discussed later in connection with those applications.

Variables and simple operands are as discussed in the previous section.

[1] In general, we may write a constant in similar form, using any base from 2 to 9; but there will probably be no reason for ever using values other than 2 or 8.

[2] Ø is a capital o. The / is sometimes used to avoid mistaking it for the number 0.

An I-operand is either a simple operand or else an expression of the form

v+a

or

v+-a

where v is the name of a variable and a is any simple operand.[1] The value of the I-operand is the value of v plus (or minus as the case may be) the value of a.

A standard operand may be any I-operand (hence any simple operand is a standard operand), any block slice or list slice of size 1 (as discussed in the next section), or any $-operand which is classified as standard.

An extended operand may be any standard operand, any block slice or list slice (as discussed in the next section) or any $-operand.

By a tape designator we mean a letter (A,B,C,...H, or T) followed by an integer from 1 to 8. If the letter is A, B,..., or H, it designates the actual machine data channel to be used, e.g., B5 would mean tape unit 5 on channel B. (For the reader unfamiliar with the 7090: there are several sets of magnetic tape units attached to the machine-- each set is called a "channel" and is denoted by a letter, A, B, etc. Most installations have only channels A and B, though some may have up to H. On each channel, the individual units are numbered 1 to 8 [at a given installation, there may not actually be 8 tapes on a channel; 8 is the maximum number.]) These designators are rarely used. Normally only Tn will be used. This means that the actual choice of tape unit is taken care of by the system-- e.g., T1 would always mean the first tape chosen by the system-- T2 the second, etc.[2]

[1] v-a is not a legitimate form for an I-operand.

[2] Normally tapes are only used for intermediate data. It is not necessary for the user to worry about details such as what tape is actually chosen.

In some situations it may be desirable to put a tape designator into a variable, and later use this variable in a phrase which requires a tape unit to be specified. It would not be correct to write, e.g., "SET V A3," or "SET TAPE T1,"; the SET phrase does not expect to find a tape designator at this point and would try to interpret the A3 or T1 as a standard operand. For such a situation the following notation is permitted:

SET V $(T) A3,

In other words, the prefix $(T) indicates that what follows is to be interpreted as a tape designator, no matter what would normally be expected in this context.

In using a phrase such as those discussed in Section 4.3, instead of an explicit tape designator we may substitute an operand whose value is a tape designator. The user must take great care if doing this to make sure that the value of the operand is indeed a proper tape designator. Otherwise strange errors will occur.

If we say in a certain context that a simple tape designator may be used, this means we may use either an explicit tape designator, or any simple operand whose value has been previously set to a tape designator. If we say a standard tape designator may be used, this means we may use an explicit tape designator, or any standard operand whose value has previously been set to a tape designator. These must, however, be qualified by the proviso that such a simple or standard operand must not be one which could be confused with an explicit tape designator; in other words it must not contain a name consisting of a single letter A, B, C, D, E, F, G, H, or T.

The expression

$(L a, ib_1, ib_2,..., ib_k)

will take the value ib_1 if the value of a is 1, the value ib_2 if the value of a is 2, etc. a may be any simple operand, ib_1, ib_2 may be any I-operands. If any one of ib_1, ib_2, etc. are unchangeable, the

whole operand is considered unchangeable; otherwise it is changeable. Incidentally, this last remark also applies to the "$(An,...)" operands discussed in Section 4.5.

We saw in Section 4.5 how a program is divided into sections, and it was mentioned that the values placed in a variable are lost when we go from one section to another. Sometimes, however, it is necessary to carry certain values across from one section to another. To do this we may designate certain chosen variables as being permanent. We do this by putting at the very beginning of the program a phrase ALWAYS followed by the names of one or more variables and terminated by a comma. All the variables so named will be made permanent; that is, whatever values we place in them in one section will not be lost at the end of the section but will be available in any succeeding sections.

Two or more consecutive standard or extended operands may (optionally) be separated by slashes (/). This may help to make the program easier to read if complicated operands are frequent. (E.g., ADD $UNIT $LAYERS / $RAND $2 / JOE,)

*4.9 Blocks and Lists

We now take up some concepts which have been hinted at several times earlier. Let us start by considering the following examples.

* * * * * * * * * * *

Example 4.9.A

In a certain situation we wish to know the number of active units in a certain layer (let us say, layer 2) of the perceptron. This could of course be done by testing each A-unit in turn and keeping count of the number of active ones. However, it can be done more neatly and quickly as follows:

```
GETACT 2 A,
BITS A N,
```

The first phrase forms the activity vector for layer 2 and puts it in A; that is, each bit of A represents one unit and is one if the unit is active, zero if not. The second phrase counts the number of one-bits in A and puts this number into N. The only problem with this is that a machine word holds a maximum of 36 bits. Suppose a perceptron has, e.g., 100 units in layer 2. To use the above technique we would have to replace A by an operand capable of holding that many bits. We could write as follows:

```
BLOCK BA 3, GETACT 2 BA, BITS BA N,
```

The first phrase defines BA to be an extended operand 3 words long, which is sufficient to hold the required 100 bits. Now the remainder of the example proceeds as before. Notice that it is not necessary for the user to include the number 3 in the GETACT and BITS phrases as they are able to determine for themselves the actual length of the operand. The first phrase could also have been written

```
BLOCK BA $NPACK 2,
```

The expression "$NPACK 2" automatically takes for its value the number of words required to hold the packed activity vector for layer 2. This form is more convenient for two reasons:

1) If the number of A-units is changed in a later experiment or if we run several cases with different numbers of A-units, there is no need to make further changes at this point.

2) It frees the user of the need for remembering what should be an irrelevant detail about the 7090, namely, that its word size is 36 bits.

* * * * * * * * * *

In the above example BA was defined to be a block of 3 words. Instead of BA any name of one to six letters beginning with B could have been used. In this example the 3 words were used to hold an activity vector, which we wish to treat simply as a single entity. Hence there was no need for us to refer to these 3 words separately, but only as a unit. However, in a different situation we might wish to use a block of several words to hold, not a single piece of information covering several words, but several different pieces of information. It is possible to refer to each word separately by writing, e.g., "BA (2)" which would refer to the second word of BA. In place of the number 2 we could have used any simple operand, or in fact any I-operand, as described in the previous section.

The blocks described so far are one-dimensional blocks. There may be occasions in which we would like to contain information in a higher dimensional block. For one example we might wish to hold a matrix of numbers, or for another we might wish to work with several activity vectors. For either of these situations we might define a block by e.g., "BLOCK BB 5 3,". In one case we would consider BB as a 5 by 3 array of single words whereas in the other we would think of it as consisting of five three-word operands. We could refer to any individual word with a notation such as "BB (4,2)" or we could refer to one of the three-word operands as e.g., "BB(2)"; or in fact we could refer to the whole thing as a single 15-word operand simply "BB". Both of these uses are illustrated in Example 3.5

The following points should be noted concerning BLOCK.

1) The BLOCK phrase is executed just as any other phrase.[1] Thus the size (or sizes in case of a multi-dimensional block) may be given as any standard operands, not necessarily as an explicit number; and it may be changed at different parts of the program.[2]

2) If the reader is familiar with FORTRAN he should note that the indexes here are in the opposite order; that is to say, if B is defined to be 5 by 3 its words are arranged in the order B(1,1), B(1,2), B(1,3), B(2,1), B(2,2), etc.

See example 3.5 for further illustration for the use of blocks.

Sometimes we may have a situation where we wish to save certain information and we do not know in advance how much of it there will be. For example, we may wish to form and save activity vectors until a certain condition is met. We may do this by means of a list. In effect, a list is a series of blocks, all with the same dimensions. The number of blocks is not specified in advance and we may add new blocks at the end whenever we wish. A list is designated by the same kind of name as a block (that is, one to six letters, the first of which is a B), and is defined by a LIST phrase, which is just like a BLOCK phrase except it begins with the code LIST. The nth block on list BL may be referred to as BL*n. Subscript notation, as discussed above for blocks, may be used, e.g., "BL*n (2,3)".

It is possible, if desirable, to define a list consisting of a single word per item. (This could be done for a block also, but in that case it would be rather pointless as it would be no different from any variable.) If we wish to add a new block to a list (e.g., BL), we may use an expression such as "BL**NEW," "BL**NEW(2,3)", etc. Note carefully the following. Suppose we have a list, BA, with three-word blocks and we wish to add a new block to the list and put the numbers 5, 7 and 9 into it. It would not be correct to write

MOVE 5 BA**NEW (1), MOVE 7 BA**NEW (2), MOVE 9 BA**NEW (3),

[1] Unlike, for example, the DIMENSION statement in FORTRAN.

[2] If the dimensions of a block are changed any information previously placed in the block will be lost.

This would have the effect of adding three new blocks to the list containing, respectively, 5 0 0, 0 7 0, and 0 0 9. (A new block is automatically set to all 0 when first added.) To do what we wish to do, we could write

MOVE 5 BA**NEW (1), MOVE 7 BA**LAST (2), MOVE 9 BA**LAST (3),

"BA**LAST" will always refer to the last block currently on the list.

It may be desirable, having a number of blocks on a list, to perform certain operations on each one in turn. This could be done by using "BA*I" and giving I successive values from 1 up to the maximum. An alternate way is to use the expression "BA**NEXT" which, as one may guess, always refers to the next item on the list after the one most recently referred to. If it is necessary to make several references to the same block, further ones can be made using "BA**SAME". References to the list using **NEW or **LAST do not have any effect on which block will be referred to by a future use of **NEXT or **SAME. The phrase "LLSTRT BA," may be used to initialize list BA for this process; that is to say, after executing this phrase the next use of "BA**NEXT" will refer to the first item on the list. If the last item on the list is referred to (by number, or by **SAME or **NEXT), then a further use of **SAME will again refer to the last item on the list. If we have a list of unknown length and wish to process all the items in it, we have a choice of two facilities. The $-operand "$LENGTH BA" has for its value the number of items currently on list BA. Alternatively, we may make use of the phrase "IF LAST BA," which will skip the next phrase unless the block most recently referred to on list BA (other than by **NEW or **LAST) was the last on the list.[1]

* * * * * * * * * *

Example 4.9.B

Suppose list BV has an unknown number of blocks, each containing an activity vector formed during certain testing procedures. We wish

[1] See Section 4.13 for a general discussion of IF phrases and other conditional phrases.

to form a new list, let us say BN, of one-word items, each of which is the number of active units in the corresponding activity vector. (This example and several of the others in the rest of this section use several simple features which are more completely explained in the appropriate places in later sections.)

```
LIST BN 1, LLSTRT BV,
(10000, BITS BV**NEXT BN**NEW, IF LAST BV,) NOP,
```

The phrase BITS, as will be explained in Section 4.12, sets the value of its second operand to the number of "one" bits in its first operand. The above could also have been done in the following way:

```
LIST BN 1,
(* I $LENGTH BV, BITS BV*I BN**NEW,)
```

The second version has an advantage over the first in that it avoids the use of the "IF LAST" phrase and thus saves some time. On the other hand, it has a disadvantage in that referring to items by number is less efficient than referring to them by **NEXT, since in the former case the program must search all the way along the list from the beginning for each new reference. If the list is quite long, this can be very time-wasting. The following combines the better points of both previous versions:

```
LIST BN 1, LLSTRT BV, SET N $LENGTH BV,
(N, BITS BV**NEXT BN**NEW,)
```

(Note that we could not use "$LENGTH BV" directly as the iteration number for the loop as it is not a simple operand.)

* * * * * * * * * *

The use of LLSTRT and **NEXT may perhaps be made clearer by the following: imagine that with each list is a "pointer" which indicates a particular item on the list. Then

**NEXT	moves the pointer one place, then refers to the item now pointed at
**SAME	refers to the item pointed at, but does not move to pointer
*n	refers to item n, and moves the pointer to this item
LLSTRT	moves the pointer before the first item
**LAST **NEW	do not affect the pointer

The phrases LLSAVE and LLUNS may be used to save and restore the position of the pointer; e.g., in the following sequence

....

LLSAVE B,

....

....

LLUNS B,

....

After "LLUNS B," the pointer for list B will be at the same position it was before the "LLSAVE B", even if it was moved elsewhere by the intervening phrases.

* * * * * * * * * *

Example 4.9.C

BL has been defined as a list of one-word items and has had placed in it a sequence of numbers, b_1, b_2, ... etc. Suppose for the sake of the example that "X α β γ δ," is a phrase which performs some computation on the values of α, β and γ and puts the result in δ. We wish to find the three numbers in the list which give the biggest result. (Assume the result is always positive.)

```
ZERO V,
LLSTRT BL,
SET N $LENGTH BL,
(N, SET AA BL**NEXT,
     LLSAVE BL, LLSTRT BL,
     (N, SET AB BL**NEXT,
          LLSAVE BL, LLSTRT BL,
          (N, SET AC BL**NEXT,
               X AA AB AC D,
               IF MORE D V,
               (, SET V D, SET VA AA,
                    SET VB AB, SET VC AC,)
          ) LLUNS BL,
     ) LLUNS BL,
)
```

Note that "LLSAVE BL," is used twice; each LLUNS restores the pointer to the position at the most recent unmatched LLSAVE. (Three or more LLSAVE-LLUNS pairs may also be nested in this fashion.)

* * * * * * * * * *

When a block (or list) is no longer needed, the phrase

LLCLR bv,

where bv is the name of the block (or list) will release the space which it was using and make it available for other purposes.

If the reader is acquainted with "list processing" languages such as IPL-V, SLIP, etc., he should note that there is no pretense at complex list processing instructions or provisions for handling involved list-structures. A few further phrases will be discussed, providing very primitive means for moving blocks from one list to another, but it will probably be a very rare application where even these are needed.

In the following, let bv and bw be names of two lists. (Just the name, e.g., "BA," not "BA*n", etc.)

LLMOV bv bw,

will remove the first item (if there is one) from list bv and add it at the end of list bw.

LLMOV bv - bw,

will remove the first item (if there is one) from list bv and add it at the beginning of list bw.

LLMOV bv *,

has the same effect as LLMOV bv bv,

LLMOV bv,

will remove the first item from list bv and release the space it occupied for use elsewhere.

LLMOV (c) bv bw,

will have the same effect as c repetitions of

LLMOV bv bw,

and similarly for the other forms of LLMOV

LLADD bv bw,

will add the whole list bv at the end of bw (thus leaving bv with zero items on it).

LLADD bv -bw,

will add the whole list bv at the beginn of bw.

LLSYN bv bw,

The name bv will become synonymous with bw. That is to say, any list or block operation except another LLSYN (e.g., BLOCK, LIST, LLSTRT, etc., as well as references to $LENGTH bv, bv (i,j), etc.) which uses the name bv will act as if it used the name bw.

* * * * * * * * * *

Example 4.9.D

```
LLSYN BA BB, BLOCK BA 2 3,
LLSYN BA BC, LIST BA 1,
MOVE 1 BA**NEW, MOVE 7 BA**NEW,
```

will define BB as a 2 x 3 block, BC as a list of 1-word items, and add 2 new items to list BC, putting values 1 and 7 into them.

Example 4.9.E

```
BLOCK BZ 8, BLOCK BY 8,
LLSYN BA BB, LLSYN BC BZ,
LLSYN BB BC,
MOVE 1 BA(1), MOVE 2 BB(2),
MOVE 3 BC(3), MOVE 4 BZ(4),
LLSYN BB BY,
MOVE 5 BA(5), MOVE 6 BB(6),
MOVE 7 BC(7), MOVE 8 BZ(8),
```

will leave BY and BZ as follows:

BY(1) = 0	BZ(1) = 1
BY(2) = 0	BZ(2) = 2
BY(3) = 0	BZ(3) = 3
BY(4) = 0	BZ(4) = 4
BY(5) = 5	BZ(5) = 0
BY(6) = 6	BZ(6) = 0
BY(7) = 0	BZ(7) = 7
BY(8) = 0	BZ(8) = 8

* * * * * * * * * *

By using this "indirect" feature, any extended operand (of sufficient size) can play the part of a block or list name. In this way we can get the effect of an indexed set of lists, or a list of lists.

To make this clearer it is necessary to say a brief word about the internal functioning. Each list or block name has a group of words set aside for it, called its locator. This is assigned when the input cards are first interpreted by the program. It has n + 3 words, where n is the largest number of subscripts ever given to this name. The locator is used to hold maximum subscript sizes and other information. It is not directly accessible to the user, and in

particular it does not contain the block itself, (or the list items themselves). Space for the block is assigned when the BLOCK phrase is executed, or when new list items are added.

LLSYN bv *eα,

will allow eα to become the "locator" of a block or list, which can be referred to as bv. Of course, the user should make no changes in the contents of eα when it is being used for such a purpose. eα should contain all zeros before the above is done. (eα must have at least n + 3 words, where n is the number of subscripts which will be used.)

IMPORTANT NOTE: the phrase

LLSYN bv bw, (or LLSYN bv *eα,)

does not itself do anything to bw (or eα)

E.g., if BB has not been previously used,

```
LIST BA 1,
LLSYN BA BB,
MOVE 7 BA**NEW,
```

is incorrect. By virtue of the LLSYN, the last phrase is actually referring to BB, which has never been defined as a list. We must do

```
LLSYN BA BB,
LIST BA 1,
MOVE 7 BA**NEW,
```

or

```
LIST BB 1,
LLSYN BA BB,
MOVE 7 BA**NEW,
```

* * * * * * * * * *

Example 4.9.F

Suppose for the sake of the example that A eα β, is a phrase which takes a stimulus from the input tape and presents it to the perceptron;

puts the activity vector in $e\alpha$, and classifies the stimulus into one of 10 classes, setting β to a number from 1 to 10. We wish to perform this on 500 stimuli and form a separate list of the activity vectors for each class of stimuli.

```
BLOCK BA 10 4,
(*I 10, LLSYN B *BA(I),
    LIST B $NPACK 2, )
BLOCK BB $NPACK2,
(500, A BB N, LLSYN B *BA(N),
    MOVE BB B**NEW,)
```

Note:

The phrases between "(*I 10," and the matching ")" are repeated 10 times with I taking values 1,2,3,...,10 on successive repetitions. The phrases between "(500," and the matching ")" are repeated 500 times.

BA is defined as 10 4-word blocks; each one is the right size to be used as a locator for a (one-subscript) list. They are all so defined. Then as each stimulus is processed, the activity vector is placed in BB and the class number in N; the appropriate list is selected and the activity vector is moved to it.

One improvement in speed may be obtained by changing the last part to

```
LIST BB $NPACK2,
(500, A BB**NEW N, LLSYN B BA(N),
    LLMOV BB B,)
```

(The improvement is due to the fact that MOVE actually moves all the information from one operand to the other, but LLMOV only changes the linking words in the list without actually moving the information physically.)

The above job could be done without use of lists, by using one 3-subscripted block; we give an illustration of this, to show how detail-work is avoided by using lists, as above.

```
BLOCK BA 10 500 $NPACK 2,

BLOCK BB $NPACK2, BLOCK BC 10,
(500, A BB N, ADD 1 BC(N) M,
     MOVE M BC(N), MOVE BB BA(N,M),)
```

Note that it is necessary to use BC to keep count of the number of vectors in each class, which was unnecessary before. However, this drawback is minor. The real problem with this method is that, since we have no advance information on the number of stimuli which will fall in each class, it is necessary to allow for the possibility that there may be as many as 500 in any class. Hence we had to define BA as 10 x 500 x $NPACK2; that is, we had to reserve space for 5000 (not 500) activity vectors. If the perceptron is very large this will be too much.[1] Most of this space, of course, remains unused, but there is no way of reclaiming it until the entire block can be released. Using lists, however, only the space which is actually needed is used.

Note 1:

The following is <u>not</u> correct. (Compare with first solution, above.)

```
BLOCK BA 10 4,

LIST B $NPACK 2

BLOCK BB $NPACK 2,
(500, A BB N, LLSYN B *BA(N),

     MOVE BB B**NEW,)
```

[1]Appendix VI discusses considerations governing amount of space available.

The word in BA(N) are never made into a list locator. (See "important note" before this example.)

Example 4.9.G

This example relies quite heavily on material from later sections, and should be skipped at first reading.

Suppose that "A eα β," is similar to the last example, but the value of β, instead of ranging from 1 to 10 may be an arbitrary number; suppose also, we do not know in advance how many different values of β there will be. We wish to form a single list, BL, of all activity vectors, with all vectors corresponding to stimuli of a given class being together in the list. We do not care about any particular order for the classes; but we do wish to form another list, BLA, of 2-word items, where the first words are the classes, in the order they occur on BL, and the second words are the number of stimuli in the corresponding classes.

```
BLOCK BB $NPACK 2,
BLOCK BA 37 5,
(*I 37, LLSYN B *BA(I), LIST B 4,)
(500, A BB N, BITS N M,
     LLSYN B *BA(M+1), LLSTRT B,
     IF ZERO $LENGTH B, GO NONE,
     *AGAIN IF EQUAL B**NEXT(5) N, GO OK,
     IFN LAST B, GO AGAIN,
     *NONE MOVE N B**NEW(5), GO AGAIN,
     *OK LLSYN EX *B**SAME,
          MOVE BB EX**NEW, )
LIST BL $NPACK 2, LIST BLA 2,
(*I 37, LLSYN B *BA(I), LLSTRT B,
     SET N $LENGTH B,
     IFN ZERO N,
          (N, MOVE B**NEXT(5) BLA**NEW(1),
               LLSYN EX * B**SAME,
               MOVE $LENGTH EX BLA**SAME(2),
               LLADD EX BL, ) )
```

The first part forms a separate list for each class number. The basic idea is to form a list containing the different class numbers; each item contains one class number, and the locator for the list of activity vectors corresponding to that class number. In the example as given, we have made things slightly more complicated to gain a little speed. We have set up 37 lists of class numbers; the n^{th} one contains those class numbers with (n-1) 1-bits. Thus instead of having to look for each class number in one long list, we can quickly select the correct one of several short lists and search down that only.

The second part combines the several lists of activity vectors onto one list, BL, as required. (Note that, as mentioned previously, LLADD does not cause any actual movement of the bulk of the information on the lists.)

* * * * * * * * * *

If necessary a block may be made available to all sections by means of ALWAYS. (See page 4.8.4) In this case the name of the block must be followed by its dimensions, which must be integers. E.g.,

ALWAYS B 3 27 BA 4,

A block name used in this way must never appear in a BLOCK or LIST phrase.

It is not possible to do this for a list.

4.10 Loops, or Compound Phrases

If one or more phrases are surrounded by parentheses, with a number or variable (i.e., a simple operand) following the opening parenthesis, then these will be repeated n times, where n is the value of the simple operand when the loop is first entered. E.g.,

(5,...,)

or

(JOE,...,)

Note that if the value of JOE is changed within the loop, this will not affect the number of repetitions. If for some reason it is desired to terminate the loop before the full number of repetitions, this can be done by means of the conditional phrases, as will be discussed in 4.13.

The whole expression, from "(" to ")" may be considered as a "compound phrase". In particular, any of the phrases within one set of parenthesis may be a compound phrase.

E.g.,

(5,... (JOE,... (3,...,)...,)... (2,...,) ...,)

Later we shall see there are reasons for wanting to group several phrases into a compound phrase-- even though we do not want to repeat them. We could of course use

(1,...,) or we can omit the "1" entirely: (,...,)[1]

Frequently it is desired to have a variable take different values on successive iterations of a loop; this could be done in the above format, e.g., by

SET I 0, (10, ADD I 1,...,)

An abbreviated notation for this is permitted, namely

(*I 10,...,)

[1]The comma preceeding the closing parenthesis is redundant and may be omitted.

More generally, we could write, e.g.,

(*I 10 5,...,)

This would give 6 iterations, with I taking values 5, 6, 7, 8, 9 and 10. Or in the opposite direction, e.g.,

(*J 1 7,...,)

would give 7 iterations with J taking values 7, 6, 5, 4, 3, 2 and 1

More generally still, we could have, e.g.,

(*SAM 10 3 2,...,)

where SAM would take values 3, 5 (i.e., 3 + 2), 7 and 9
or

(*SAM -3 12 -3,...,)

where SAM takes values 12, 9, 6, 3, 0, -3

We do not have to use numbers, but may use any standard operands. Thus:

(*v α β γ,...,)

v is a variable, α, β, γ may be any standard operands. If a, b, c are the values of α, β and γ when the loop is entered, then v will receive the value b at the start of the first iteration and will have c added to its value at the start of each new iteration. If the value of v is not changed within the loop this will mean it takes values b, b+c, b+2c,... (up to, but not including, the first value >a, if c>0, or <a if c<0).[1]

If it happens that b is itself >a (when c>0) or <a (when c<0) then there will still be one iteration.

If γ is omitted, it will be taken as 1 (if b$\geq$a) or -1 (if b<a). If β is omitted it will be taken as 1. If the value of α, β and/or γ is

[1] If the reader is familiar with FORTRAN he will note the similarity of this notation to the DO-statement. However, he should note carefully that "(*I a b c,...,)" corresponds not to "DO n I = a, b, c" but to "DO n I = b, a, c". This is done so that we can use the abbreviated notation "(*I a, ...,)" in most uses.

changed within the loop, this will not affect either the number of repetitions, nor the values taken by v. If the value of v is changed by the phrases within the loop, this will affect all future values of v, but will not affect the number of repetitions. E.g.,

(*I 4, MPY I 2, PRINT I,)

would print the numbers 2, 6, 14, 30

(6 = (2+1) x 2, 14 = (6+1) x 2, 30 = (14+1) x 2)

Needless to say, this kind of shenangians will rarely if ever be used--it is simply an accidental by-product of the way the program works. The normal procedure will be not to monkey with v, but simply use the normal sequence of values given to it.

In a complicated situation with several nested levels of parentheses, it may become confusing to read the program. This can be alleviated by the following facility; any left parenthesis may be replaced by

OPEN name

and the corresponding right parenthesis by

CLOSE name

where "name" is any group of 1 to 6 letters.[1] If a different name is used for each pair of parentheses, the program becomes much easier to read. Also the system can now check for correct matching and print a warning message if there is a discrepancy. NOTE particularly that the whole expression "OPEN name" substitutes for "(" in any of the forms described above. E.g.,

(2, ... (JOE, ... (*I 3, ...) ... (*J 4 12 -2, ...)) ...)

[1] The user will probably choose, for his own peace of mind, to use names for this purpose which do not duplicate names of variables, etc., however, this is not mandatory.

might be rewritten

```
OPEN ONE 2, ..., OPEN TWO JOE, ...,
     OPEN THREE *I 3, ..., CLOSE THREE, ...,
     OPEN FOUR *J 4 12 -2, ..., CLOSE FOUR,
          CLOSE TWO, ..., CLOSE ONE,
```

IMPORTANT NOTE: the number of iterations of a compound phrase may not exceed 32,767. $(= 2^{15} - 1)$.

The material in this section is summarized in 5.3.

4.11 Arithmetic Phrases

Basic arithmetic processes may be performed by ADD, SUB, MPY, DIV, e.g.,

"ADD α β γ,"

Will add the values of α andβ and put the result in γ: α, β, γ may be any standard operands.

"SUB α β γ,

will subtract the value of β from the value of α and put the result in γ.

MPY α β γ,	will multiply the values of α andβ and put the result in γ.
DIV α β γ,	will divide the value of α by the value of β and put the result in γ.

If we wish to add the values of α and β and put this new value back in α, instead of saying

"ADD α β α,"

it is sufficient to say

"ADD α β,"

In the case of DIV, we may if we wish add a fourth operand, e.g.,

"DIV α β γ δ,"

δ will receive the value of the remainder.

We stated above that the operands of ADD, SUB, etc., may be any standard operands. Actually, they may be any extended operands. Consider ADD for example.

If we say

ADD BA BB BC,

where BA, BB, BC are extended operands, then the first word of BA will be added to the first word of BB and the result put in the first

word of BC; then the same for the second words, etc. If BA has less words then BC, then the last word of BA will be used over again; similarly for BB. If BA and/or BB has more words than BC, the last ones will be ignored. Note that this implies that if we say, e.g.,

ADD BA 3 BC,

then 3 will be added to each value in BA and the result placed in the corresponding value in BC.

Note: In the case of DIV, when a fourth operard is used for the remainders(s), the third and fourth operands must each have the same number of words.

It is assumed that all the quantities being operanted on have integer values (in approximately the range -10^{10} to $+10^{10}$). This will be sufficient for most uses.

When fractions are desired, one can use one of three methods.

a) deal with integers which are a certain power of 10 times the actual numbers we are interested in

b) deal with integers which are a certain power of 2 times the actual numbers (using the shifting instructions discussed later)

c) use floating point numbers, as described in the next paragraph.

There is in the 7090 provision for handling numbers in a floating point format. These numbers may have an accuracy of 8 decimal places and may fall in the range 10^{-35} to 10^{35} (or -10^{35} to -10^{-35}) (or may, of course, be exactly 0).

An expression of the form

$(Fn)d

may be used as a simple operand. n is a (positive or negative) integer: d a (positive or negative) integer or decimal fraction. The value of the operand is d x 10^n in floating point form. (n may be left out if 0 is intended.)

Suppose we used the phrase

ADD $(F-2) 3.75 $(F) .023 JØE

The floating point representations of 0.0375 and 0.023 will be added together as if they were integers, and the result will be complete gibberish. We must instead use the phrase

ADDF $(F-2) 3.75 $(F) .023 JØE,

which will correctly set JØE to the floating point representation of 0.0605.

Similarly we have SUBF, MPYF, DIVF.

If either DIV or DIVF attempts to cause a division by zero, then the special operand $QDIVCK will be set to a nonzero value. (This may be tested and reset to zero at any time.) (In the case of DIVF this will also happen if the magnitude of the quotient is greater than about 10^{35}.)

If a variable, e.g., SAM, has a value in integer form, and we wish to use it in floating point form, we may use

"FLOAT SAM,"

which will convert it as desired.

"FLOAT" may be followed by more than 1 operand; they may be any (changeable) standard, or even extended, operands. If we use an extended operand which is a collection of several variables each one will be converted independently. "FIX" has exactly the opposite effect to FLOAT.

(NOTE: If we use FIX on a variable which is already in integer form, or FLOAT on one which is already in floating form, the result will be gibberish.)

* * * * * * * * * *

Example 4.11.A

Suppose we have generated a perceptron, and we have on tape A4 1000 stimuli with which we wish to perform the following experiment.

We wish to present the stimuli, with error correction reinforcement, and to note the number of correct responses in each group of ten. We want as our final answer the average number of correct responses (per group of 10) and also the variance (σ^2) of these quantities.

Since some of the codes we need have not yet been discussed, we will assume for the sake of the example that the phrase "A α," will do the following: present 1 stimulus, apply error correction reinforcement, and add one to the value of α if the response was correct. Then our job can be done by the following:

```
ZERO MEAN SIG,
(100, ZERO N, (10, A N,)
ADD MEAN N,
MPY N N NN, ADD SIG NN, )
FLOAT MEAN SIG,
DIVF MEAN $(F)100,
DIVF SIG $(F)100, MPYF MEAN MEAN M, SUBF SIG M,
PRINTF MEAN SIG,
```

The first line, as we might guess, sets the values of MEAN and SIG to Zero. Next we accumulate the sum of the numbers of correct responses, and the sum of their squares, in MEAN and SIG respectively. Finally we convert these to floating point form and compute the mean ($= \frac{1}{100} \sum x_i$) and variance ($= \frac{1}{100} \sum x_i 2$ - mean ^{2}). Note the use of the expression $(F)100 to give 100 in floating point form. PRINTF MEAN SIG, prints the desired values.

* * * * * * * * * *

MAX α β_1 β_2 ..., will set the value of α to the maximum of the values of β_1, β_2, etc. The values may be all integers, or all floating point, but not mixed.

MAX α β, will set α to the maximum of itself and β.
MIN is similar to MAX, except the minimum is used.

ADK a β, — the value of a is added to the value of β and the result is put in β

SBK a β, — the value of a is subtracted from the value of β and the result is put in β.

RSBK a β, — the value of β is subtracted from the value of a and the result is put in β.

In all of these (ADK, SBK, RSBK) a may be any simple operand, β any standard operand.

ASHIFT a α β,

where a is any simple operand, α, β any standard operands, will cause the contents of α to be shifted a places (positive number means a right shift, negative means a left) and put in β. In other words, the value of α is multiplied by 2^{-a} and put in β.

ASHIFT a α, is the same as
ASHIFT a α α,

(The mnemonic significance of the A, is Arithmetic SHIFT, to distinguish from the logical shift discussed in the next section.)

SET v α,

where v is any variable, α any standard operand. v will be set to the value of α.

ZERO, followed by one or more operands, will cause each one to be set to zero. Any of the operands may be extended, in which case each of its words will be set to zero. (NOTE: Unlike other numbers, the machine represents 0 exactly the same way in floating point or integer form. Hence this phrase may be used indiscriminately on operands which are to be either.)

BRAKET eα eβ eγ δ,

eα, eβ, eγ may be any extended operands, but they must all have the same number of words. δ may be any standard operand. Each word in eα is set equal to the corresponding word in eβ, if it was less than it; or equal to the corresponding word in eγ, if it was more

than it; or else is left unchanged. δ is set to a nonzero value if any changes were made in $e\alpha$; otherwise δ is set to zero.

The material covered in this section is summarized in 5.5.

4.12 Logical Operations

The previous section dealt with phrases for performing operations on numbers. This section will deal with phrases for performing operations on other kinds of information (e.g., activity vectors, alphabetic information).

AND eα eβ eγ, will set eγ to the "logical AND" of eα and eβ; i.e., each bit of eγ will be set to 1 if the corresponding bits in eα and eβ are both 1; otherwise it is set to 0. eα, eβ, eγ may be any extended operands. (The possibility arises that the operands may be of different lengths; if eα and/or eβ is shorter than eγ, the last word (36 bits) will be repeated. If either is longer than eγ the last words will be ignored. [as in ADD, SUB, etc. "AND eα eβ" has the effect of "AND eα eβ eα"])

* * * * * * * * * *

Example 4.12.A

If eα is a single word containing (in octal) 000000123456 and eβ is three words 000000111111 000000222222 000000333333, and if eγ is two words long, eγ would become 000000101010 000000022002

* * * * * * * * * *

OR eα eβ eγ, will set eγ to "logical OR" of eα and eβ, i.e., a bit of eγ is set to 1 if either or both of the corresponding bits of eα or eβ is 1.

EØR eα eβ eγ, will set eγ to the "exclusive OR" of eα and eβ; i.e., each bit of e is set to 1 if exactly one of the corresponding bits in eα, eβ is 1.

COM eα eβ, will set eβ to the complement of eα; i.e., each bit will be set to 1 or 0 according as the corresponding bit was 0 or 1. ("COM eα," will have the effect of "COM eα eα,")

AND, OR, EOR, and COM are the most commonly used logical combinations, and any other can be built up from these.

* * * * * * * * * *

Example 4.12.B

BA and BB are extended operands of the same lengths. We wish to complement those bits of BA which correspond to 0 bits in BB, but leave the others unchanged.

EOR BA BB, COM BA,

Example 4.12.C

BA, BB as above, but we wish to set to 0 those bits of BA corresponding to 1 bits in BB, leave the others unchanged.

COM BB BC, AND BA BC,

* * * * * * * * * *

Alternatively, we may use the phrase

LOGIC wxyz eα eβ eγ,

wxzy is a 4-digit code (each digit either 0 or 1) which may be used to designate any one of the 16 possible logical combinations.

If a bit in eα is	and the corresponding bit in eβ is	then the corresponding bit in eγ is set to
0	0	w
0	1	x
1	0	y
1	1	x

* * * * * * * * * *

Example 4.12.D

LOGIC 0001 eα eβ eγ, is exactly equivalent to AND eα eβ eγ,

LOGIC 0110 eα eβ eγ, is equivalent to EOR eα eβ eγ,

To perform the operation specified in Example 4.12.C, we need

first op.	second op.	result
0	0	0
0	1	0
1	0	1
1	1	0

The required code is 0010

LOGIC 0010 BA BB,

Another way of looking at this is to replace the first operand by 0011 and the second by 0101, and perform the desired operation. The result will be the necessary code.

Thus for the above we want

$BA \wedge \overline{BB}$, which gives $0011 \wedge \overline{0101} = 0011 \wedge 1010 = 0010$

while for example 4.12.B, we want

$(BA \wedge BB) \vee (\overline{BA} \wedge \overline{BB})$, which gives
$(0011 \wedge 0101) \vee (\overline{0011} \wedge \overline{0101}) = (0001) \vee (1100 \wedge 1010) = 0001 \vee 1000 = 1001$
LOGIC 1001 BA BB,

* * * * * * * * * * *

Sometimes it may be required to shift the bit pattern in a operand. The ASHIFT phrase discussed in 4.11 is not suitable for two reasons: 1) ASHIFT only works on a single word, not any extended operand; 2) ASHIFT operates on numbers-- hence one bit in the word (the leftmost), which represents the sign of the number, is not shifted like the others. This is undesirable in dealing with logical information where this bit has no special significance. So we have the following phrase.

SHIFT a eβ eγ,

a may be any simple operand, eβ and eγ any extended operands. eβ will be shifted a bits (a positive number means a right shift, negative a left shift) and the result put in eγ. If necessary, zero bits will be added at either end of eβ in order to fill out eγ.

To be more specific, imagine eβ and eγ lined up so their beginnings (left ends) match; then shift eβ the desired amount and put it into eγ, setting bits of eγ to zero if the shifted eβ does not affect that bit. The entire operand is shifted with no special consideration to the number of words involved.

If we wish to shift eβ and put the result back in eβ we may use

SHIFT a eβ,

It is frequently of interest to count the number of 1-bits in an extended operand. This may be done by

BITS eα β,

where eα is any extended operand, β any standard operand. The value of β will be set to the number of 1-bits in eα.

Similarly,

BITSZ eα β, or BITZ eα β, will count the number of zero bits in eα.

* * * * * * * * * *

Example 4.12.E

BA holds an activity vector; i.e., each bit represents one A-unit, and is 1 if the unit is active, 0 if inactive. NA is the number of A-units, which is assumed to be a multiple of 3. We want to set KA to the number of active units in the first group of NA/3, KB the number in the second, and KC the number in the third. Let us assume BB, BC are other operands of the same size as BA, which we can use as we wish.

```
DIV NA 3 NB, MPY NB 2 NC,
LOGIC 1111 BB,
SHIFT NB BB,
LOGIC 0010 BA BB BC, BITS BC KA,
SHIFT -NB BA BC,
LOGIC 0010 BC BB, BITS BC KB,
SHIFT -NC BA BC,
LOGIC 0010 BC BB, BITS BC KC,
```

Discussion:

"LOGIC 1111 BB," sets BB to a

logical combination of BB and 0 (since the second operand is omitted). However, this particular combination (1111) sets each bit of the result to 1 in any case. In other words, this phrase sets every bit of BB to 1. Then by shifting it to the right, we get 0 in the first NA/3 places, and 1's in the rest.

"LOGIC 0010 BA BB BC,"

sets BC to 0 where BB is 1, and copies BA in the rest; i.e., BC will hold the first NA/3 bits of BA, so we count the 1-bits as required. Then we shift the next group of NA/3 into position, mask out the others, and count, as before; and finally we do the same for the last.

* * * * * * * * * *

The contents of one operand can be copied into another by

MOVE eα eβ,

If eα is shorter than eβ, it will be placed at the left end and filled with zeros to the right. If eα is longer than eβ, the right end will be ignored.

PACK eα eβ,

Each bit of eβ, starting from the left, will be set to 1 if the sign of the corresponding word in eα is plus, or to 0 if it is minus. If eβ has more bits than eα has words, the rest are set to 0. If eα has more words, the rest are ignored.

UNPACK eα eβ,

Is the reverse; that is, the signs of the words in eα are set to the bits in eβ. If eα has more words than eβ has bits, the rest are made plus. If eβ has more bits, the rest are ignored. The magnitudes of the numbers in eα are not changed.

NOTE: In both PACK and UNPACK, the first operand holds the unpacked form, and the second holds the packed form.

PACKZ eα eβ,

Each bit of eβ starting from the left, is set to 1 if the corresponding word in eα is nonzero; or is set to zero, if the word is 0. Excess bits are set zero, or excess words are ignored.

FILL eα n_1 n_2... n_k,

The values of n_1, n_2,... n_k are placed in successive words in eα. eα must be big enough for them all. If it is too big, the remaining words will be set to zero. If we wish to have the values in eα in floating point form, we may write

FILL* eα n_1 n_2... n_n,

If we wish them to have m binary places we may write

FILL m eα n_1 n_2... n_n,

* * * * * * * * * * *

Example 4.12.F

B is an extended operand of 8 words.

then

FILL B 3 1 2 0 -4 6,

would set the words of B as follows:

1st word = 3
2nd word = 1
3rd word = 2
4th word = 0
5th word = -4
6th word = 6
7th word = 0
8th word = 0

Example 4.12.G

(Skip this example unless you have read section 4.8)

Suppose B is a block of 6 words.

```
FILL B 2 3 $(F)5.2 $(B18)4.2 $(B18)-3.6 $(F)6,
FILL * B $(B0)2 $(B0)3 5.2 $(B18)4.2 $(B18)-3.6 6
FILL 18 B $(B0)2 $(B0)3 $(F)5.2 4.2 -3.6 $(F)6,
```

would all have the same effect. They would set

B(1) = 2	integer
B(2) = 3	integer
B(3) = 5.2	floating
B(4) = 4.2	18 places
B(5) = -3.6	18 places
B(6) = 6	floating

* * * * * * * * * *

This form of FILL is probably the most useful one; however, if desired, one may write

FILL $e\alpha$ $e\beta_1$ $e\beta_2 \ldots e\beta_n$,

Where the $e\beta_1$ may be any extended operands. Successive words from each $e\beta_1$ will be put into $e\alpha$; if the variant "FILL*" or "FILL m" is used, no change will be made in any of the $e\beta$, except those written as explicit numbers.

The material in this section is summarized in 5.6.

The remainder of section 4.12 should be skipped unless one has read *4.9 (Blocks and lists).

FILLB bv $\alpha_1 \ldots \alpha_n$,

bv must be a block name. It will be defined as a (one-dimensional) block of size n, and the values of $\alpha_1 \ldots \alpha_n$ will be placed in it. In other words, the above has the same effect as

BLOCK bv k,
FILL bv $\alpha_1\ \alpha_2 \ldots \alpha_k$,

"FILLB*" or FILLB m" may be used, analogously to "FILL*", "FILL m"

LFILL bv $e\beta_1 \ldots e\beta_k$,

bv must be the name of a list (which has already been defined as a list). A new block will be added to the list, and the values from $e\beta_1$, $e\beta_2$, etc. placed in it. ($e\beta_i$ may be any extended operands.) If the item is filled before all the $e\beta_i$ are used, then a new item is added, and we continue putting values in it. Also, any $e\beta_i$ may be preceded by an *, meaning a new item must be started at that point.

* * * * * * * * * *

Example 4.12.H

Suppose B and BA are blocks of 3 and 5 words respectively, containing

B(1) = 12, B(2) = 7, B(3) = -4
BA(1) = 6, BA(2) = 4, BA(3) = 5, BA(4) = 0, BA(5) = -1

and suppose BL is a list with blocks of size 4.

LFILL BL B 28 104 116 * 201 BA 300,

will add 4 new blocks to BL, which will contain the following values

1st new block	12	7	-4	28
2nd new block	104	116	0	0
3rd new block	201	6	4	5
4th new block	0	-1	300	0

* * * * * * * * * *

"LFILL*" and "LFILL m" may be used in similar fashion to "FILL *" and "FILL m".

4.13 Control Phrases

In many situations it is desirable to decide whether or not to perform certain actions, depending on the outcome of various other actions. There are several phrases intended for this purpose. A typical one is

IF EQUAL α β,

α, β may be any standard operands.

If the values of α and β are equal, then the next phrase will be performed as usual: bit if not, the next phrase will be skipped. If the next phrase is a compound phrase, the whole thing will be skipped. If the IF phrase is the last phrase in a compound phrase, the next phrase after the compound phrase will be skipped. This is true even if there would normally have been further repetitions of the compound phrase. This provides a means of terminating a compound phrase earlier than usual.

IF LESS α β, is similar to IF EQUAL, but the next phrase is performed if α is less than β

IF MORE α β, has the obvious meaning

IF ZERO α, is the same as IF EQUAL α 0,

IF NEG α, is the same as IF LESS α 0,

IF POS α, is the same as IF MORE α 0,

IFN EQUAL α β, is the reverse of IF EQUAL α β,
(IFN is for IF Not) i.e., the next phrase will be performed if α is not equal to β, or skipped if it is. Similarly for IFN LESS etc.

* * * * * * * * * *

Example 4.13.A

For the sake of this example, suppose "A α," is a phrase which performs a certain training procedure, followed by a certain test, and sets α to the number of errors made during the test. We wish

to repeat this test and training up to a maximum of 30 times. However, we wish to stop when we achieve perfection (no errors).

```
(30, A N, IFN ZERO N,) NOP,
```

Each time we do A, we then test N; if it is not zero, we continue normally; otherwise we skip the next phrase, which means that we stop repeating the loop, even though we have not made the total of 30 repetitions. NOP, is a phrase which does nothing; it is there so that we do not skip the following phrase when we skip out of the loop.

Example 4.13.B

As in the previous example, but we will only ask for perfection during the first 10 repetitions; during the next 10 we will be satisfied if there are not more than 5 errors, and during the last 10 if there are not more than 10. Also, we wish to print out the number of repetitions actually made, and the number of errors the last time. If the full 30 repetitions are done without a satisfactory outcome, we wish to perform phrase B, instead of printing the numbers as above. One way of doing this might be

```
(*I 30,
     SET M 0,
     IF MORE I 10, SET M 5,
     IF MORE I 20, SET M 10,
     A N, IF MORE N M,)
(, B, IF ZERO 1,) PRINT I N,
```

First we set M to the maximum number of errors we will tolerate on this repetition, then we perform A, and compare the number of errors with M. If on some repetition the number of errors is satisfactorily small, the "IF MORE..." phrase causes a skip to the PRINT phrase, which prints the number of the current repetition (I), and the number of errors (N). On the other hand if we complete all 30 repetitions without a satisfactory result, the compound phrase "(, B,...)" will be executed-- i.e., "B," will be executed as required, and "IF ZERO 1," will cause the PRINT phrase to be skipped.

* * * * * * * * * *

Sometimes it is desirable to skip several phrases. One way to do this is by making them into a single compound phrase, as "(, B, IF ZERO 1,)" in the last example. Another way is to use the following:

COMP(kmn) α β,

k, m, n may be any numbers from 0 to 7.

If $\alpha > \beta$ the next k phrases will be skipped.
If $\alpha = \beta$ the next m phrases will be skipped.
If $\alpha < \beta$ the next n phrases will be skipped.

For example,

IF EQUAL α β, is the same as COMP (1 0 1) α β,
IFN MORE α β, is the same as COMP (1 0 0) α β,

In particular, we may write

COMP α β,

instead of

COMP (0 1 2) α β,

Sometimes we want to check if a specific bit in an operand is 1 or 0. A convenient way to do this is to shift the bit to the leftmost position, which represents the sign of the word (1 for minus, 0 for plus) and check whether the word is positive or negative. A small problem arises; if all the other bits are 0, the word is a number 0, regardless of the sign; therefore "IF POS" would not perform a correct check. We can use instead

IF PLUS α,
which skips the next phrase unless the sign of α is plus (0).

IF MINUS α,
which skips the next phrase unless the sign of α is minus (1).

IFN PLUS, IFN MINUS are the reverse, as usual.

IF LEQUAL eα eβ,

will test for Logical equality of eα and eβ. This differs from IF EQUAL in 2 ways. 1) It checks for equality of the sign bit, even if the other bits are all zero. 2) eα, eβ may be any extended operands.

The next phrase will be skipped unless eα, eβ agree in every bit position. (If the lengths are different, the shorter will be imagined to have zeros added at the right to make it the same length as the other.)

IF LZERO eα,

(Test for Logical ZERO) Skip the next phrase unless every bit of eα is 0.

IFN LEQUAL, IFN LZERO, may be used as usual.

JUMP α, will cause the next α phrases to be skipped. (α may be any standard operand.) E.g., in the previous example, we could have used "JUMP 1," instead of "IF ZERO 1,".

SBSK v, will reduce the value of v (which may be any variable) by 1, and if the value becomes 0, the next phrase will be skipped.

HALT, will stop the computer

EXIT, will cause the job to terminate immediately. Either HALT or EXIT may be followed by a message. This may be any string of characters other than a comma or apostrophe. This message will be printed (offline, and in the case of HALT, online also). If the message includes the character slash (/) this character will not print, but will mark the start of a new line.

* * * * * * * * * *

Example 4.13.C

Suppose the value of PAR is calculated from certain values specified by the user. Suppose a value greater than 50 can only arise by an error on the part of the user. Further, suppose such an error will cause a costly waste of time if we allow the program to proceed. After calculating PAR we may include

```
IF MORE PAR 50,
EXIT ERROR IN PARAMETERS/  PROGRAM CANT PROCEED,
```

If the user makes such an error, the message

```
ERROR IN PARAMETERS
  PROGRAM CANT PROCEED
```

will be printed, and the job will finish at once.

* * * * * * * * * *

Any phrase (simple or compound) may be given a name by preceding it by *name, where "name" may be any alphabetic name of one to six letters.

The phrase

```
GO name,
```

where "name" is the name of some other phrase, will cause that phrase to be done next (and then the phrases after it, in normal fashion).

It is <u>not</u> permissable to Go to a phrase inside a compound phrase unless the Go-phrase is also inside it. However, it <u>is</u> permissable to have the Go-phrase inside and the phrase gone to outside. In other words, one can only get <u>into</u> a compound phrase in the normal way, but one can get <u>out</u> of it by a Go (or an IF or JUMP).

The material in this section is summarized in 5.7.

4.14 Subsequences (basic)

If a certain series of phrases is to be used in several different places, we may for convenience give this a name, and simply refer to it by this name in the various places without writing it out in full each time. The general form is

DEFINE name, desired series of phrases, END

"name" may be any combination of from 1 to 6 letters which is not a primary code. (To simplify things for the user, no one letter names and no names beginning with Z are used as primary codes[1] -- hence these are all available as subsequence names.) Henceforth, this name may be used like a primary code, and the effect will be to perform the specified series of phrases. We call such a series of phrases a subsequence.

In effect, this provides a way of adding new phrases to the language, tailored to the users special needs.[2]

A subsequence may contain uses of previously defined subsequences, and it may contain compound phrases. It may not contain part of a compound phrase. (E.g.,

DEFINE name,..., (5,...END

is incorrect.)

* * * * * * * * * *

Example 4.14.A

```
DEFINE SS, ADD JOE 1, (5, EC 3 T2,) END
SET JOE 3, (10, SS, TEST 40 T1, SS, TEST 20 T1,) END
```

would have exactly the same effect as

```
SET JOE 3, (10, ADD JOE 1, (5, EC 3 T2,)
     TEST 40 T1, ADD JOE 1, (5, EC 3 T2,)
     TEST 20 T1,) END
```

[1] Except for ZERO.

[2] If the reader is an experienced programmer, familiar with macro-instructions, he should be warned that, although the early part of this section may lead him to look upon subsequences as akin to macro-instructions, it would be less misleading to think of them in terms of subroutines.

Example 4.14.B

```
DEFINE A, ADD JOE SAM PETE, END
DEFINE B, PRINT JOE SAM PETE X Y, END
DEFINE C, ADD JOE 2, SUB SAM 1, A, MPY X 3 Y, MPY X 2, B, END
SET JOE 1, SET SAM 3, SET X 5,
C, C, SET X 3, C, SET SAM -3, A, B, END
```

would print the following results

```
3   2   5   10  15
5   1   6   20  30
7   0   7   6   9
7   -3  4   6   9
```

* * * * * * * * * * *

The definition of a subsequence must be placed at the very beginning of a section. Once defined, the subsequence may be used in later sections without need to define it again.

Closely related to subsequences are expressions of the following form

$$name $n

where "name" is the name of a subsequence and n is an integer. Such an expression may be used as a name of variable. For example, if we have a subsequence named HELP, we may have a variable named "$$HELP$2". This is a rather cumbersome expression to write. However, within the subsequence HELP itself, we may use the abrevated form "$2".

Similarly an expression such as "$$HELP$B3" may be used as a name for a block or list, and may be abrevated to "$B3" within the subsequence HELP.

There are two reasons why these expressions or subsequence variables as they are sometimes called are useful. The first is illustrated by the following:

```
DEFINE A, ADD $1 1 JOE, END
DEFINE B, ADD $1 2 JOE, END
```

The use of "JOE" in each of these sequences refers to the same variable, of course. However, the use of "$1" in the sequence A is an abreviation for "$$A$1", while in B it is an abreviation for "$$B$1", which is a different variable. Thus, these subsequence variables give the effect of "private" variables belonging to a particular subsequence. (They can be referred to from outside the subsequence, but only by a deliberate, conscious use of the unabreviated form.) This is particularly useful if we wish to make up a subsequence not merely for a specific application, but to be used repeatedly in different applications, as it eliminates the need for remembering each time we use it in a new program what variable names are used in the sequence and hence must not be used for other purposes.

The second use is of considerably greater significance. Suppose, for example, we have a subsequence XXX, which uses $1, $2 and $3. When we use it, instead of writing simply "XXX," we may, if we choose, write such a thing as

XXX 3 1 27,

The effect of this is to set the value of $1 (that is $$XXX$1) to 3, the value of $2 to 1 and the value of $3 to 27; and then perform the phrases in the subsequence. If we wrote

XXX 4,

This would set $1 to 4, and leave $2 and $3 with whatever value they already have. Thus we have a means of supplying parameters to a subsequence. Instead of writing explicit numbers after XXX we could have used any standard operands.

* * * * * * * * * *

Example 4.14.C

Suppose A is an already defined subsequence which presents a stimulus from tape T1 and performs a certain training procedure. We wish to do this to 10 stimuli, then TEST with 20stimuli on tape T2. We wish to repeat this (using new training stimuli and the same test

stimuli) until 100 training stimuli are used; then continue, but only test after every 20 training stimuli till 200, then after every 50 to 500, then after every 100 to 2000.
This could be done by

```
(10, (10, A,) TEST 20 T2, REW T2,)
(5, (20, A,) TEST 20 T2, REW T2,)
(6, (50, A,) TEST 20 T2, REW T2,)
(15, (100, A,) TEST 20 T2, REW T2,)
```

Note all the repetition. By defining a subsequence we can simplify this greatly.

```
DEFINE B, ($1, ($2, A,) TEST 20 T2, REW T2,) END
...
B 10 10, B 5 20, B 6 50, B 15 100,
```

Example 4.14.D

```
DEFINE T, PRINT $1 $2 $3, END
T 1 2 4, T 5, T 2 3, T -7 6 8,
```

would produce the following output

```
 1   2   4
 5   2   4
 2   3   4
-7   6   8
```

* * * * * * * * * *

Consider the following subsequence:

```
DEFINE A, IF ZERO JOE, END
```

If we use this subsequence, and the value of JOE is not zero, then "the phrase after the IF-phrase" will be skipped. In this case, the IF-phrase is the last phrase in the subsequence definition; the "next phrase" is understood to be the phrase after the use of the subsequence.
In other words

```
..., A, ..., ..., ..., A, ..., ...,
```

would be exactly equivalent in effect to

..., IF ZERO JOE, ..., ..., ..., IF ZERO JOE, ..., ...,

* * * * * * * * * * *

Example 4.14.E

In a certain application we frequently want to check if the absolute value of an operand lies in a certain range; so we would like to define a subsequence with the following property:

INT α β γ, will cause the next phrase to be skipped unless $\alpha \leq |\beta| \leq \gamma$. We can do this as follows

```
DEFINE  INT, MAX $4 $2 -$2,
IF LESS $4 $1, SET $4 $3+1,
     IFN MORE $4 $3, END
```

since $2 holds the value of β, $4 is set to max ($\beta$, $-\beta$); i.e., $|\beta|$. The next two phrases set $4>$3 (γ) if $|\beta| < \alpha$ so that in any case, when the last phrase is executed, 4\leq$3 if $\alpha \leq |\beta| \leq \gamma$, $4>$3 if $|\beta| < \alpha$ or if $|\beta| > \gamma$. Hence "IFN MORE $4 $3," causes the required skip.

Example 4.14.F

As a slightly more complicated example, we would like

INT α β γ δ,

to check for $\alpha \leq |\beta| \leq \gamma$ as in the preceding example,

if $\delta = 0$

or to check for

$\alpha < \|\beta\| \leq \gamma$	if $\delta = 1$
$\alpha \leq \|\beta\| < \gamma$	if $\delta = 2$
$\alpha < \|\beta\| < \gamma$	if $\delta = 3$

```
DEFINE INT, MAX $5 $2 -$2,
     DIV $4 2 $7 $6,
     ZERO $8,
     COMP(2 0 1) $5 $1, IFN ZERO $6, SET $8 1,
     COMP(1 0 2) $5 $3, IFN ZERO $7, SET $8 1,
     JUMP $8, END,
```

The first two phrases set $5 = $|\beta|$

$6 = 0 if we are to check for $\alpha \leq |\beta|$,
= 1 if we are to check for $\alpha < |\beta|$,
$7 = 0 if we are to check for $|\beta| \leq \gamma$,
= 1 if we are to check for $|\beta| < \gamma$

now consider

"COMP(2 0 1) $3 $1, IFN ZERO $6, SET $8 1,"

If $5>$1 (i.e., $|\beta| > \alpha$) then either $|\beta| > \alpha$ or $|\beta| \geq \alpha$ is true and we skip 2 phrases, leaving value of $8 unchanged.

If $5<$1 (i.e., $|\beta| < \alpha$) then neither $|\beta| > \alpha$ nor $|\beta| \geq \alpha$ is true and we skip 1 phrase, coming to "SET $8 1,"

If $5 = $1 (i.e., $|\beta| = \alpha$) we test the value of $6 to see whether to set $8 to 1.

Similarly the other end of the interval is tested. The upshot is that $8 is set to 1 if the required condition is violated or to 0 if not. Then "JUMP $8," performs the required skip if necessary.

* * * * * * * * * *

It may on occasion be desirable to define a subsequence which can return a value it has computed to the using program; it may also be desirable to communicate other parameters, such as extended operands, tape designators, etc. We have discussed so far the simplest case, where a single value is given to the subsequence for each parameter. In more complex situations it is necessary to specify in the definition the types of parameters expected. This is best illustrated by an example

DEFINE XYZ (S SR T S), ..., END

When XYZ is used the first parameter may be any standard operand; its value will be put in $1. This is a type S (Standard) parameter. The second parameter may be any changeable standard operand; its value will be put in $2, and when the subsequence has been executed, the value in $2 (which may have been changed by the subsequence) will

be put back in the operand. This is a type SR (Standard Return) parameter. The third parameter may be any standard tape designator; it will be put in $3. This is a type T (Tape designator) parameter. The fourth parameter is a type S, like the first.

Other types will be discussed later.

If we use XYZ and supply less than 4 parameters, the remaining ones will be understood to be zero. E.g., "XYZ 1 JOE T3," would set $1, $2, $3 as described above, and $4 to zero. The reader may recall that when no parameter types were specified, in the earlier discussion, then the $n's corresponding to missing parameters were not set to zero but were left unchanged. Sometimes this is desirable in general. Suppose instead of "DEFINE XYZ (S SR T S)," we had written "DEFINE XYZ (S SR/ T S),"; the "/" would have no effect unless XYZ is used with less than four parameters. Then, $1, $2 would be set to zero if the corresponding parameter was not supplied, but $3, $4 would be left unchanged from the previous use. Similarly if the / is in a different place. (If we had, e.g., "DEFINE XYZ (S /SR T/S)," the second "/" would be completely ignored.) Now suppose we use XYZ and give more than 4 parameters, say for example 6. The effect will be exactly as if we had originally defined XYZ with (S SR T S / S S) instead of (S SR T S); and similarly for other cases. (In particular

"DEFINE W (/ SSS)...,"

is equivalent to "DEFINE W,")

* * * * * * * * * *

Example 4.14.G

```
DEFINE ABC (S S / S), PRINT $1 $2 $3 $4, END
ABC 1 -2 3, ABC 2 5 7 8, ABC, ABC 3 4,
ABC -5,
```

would print as follows:

```
 1   -2    3    0
 2    5    7    8
 0    0    7    8
 3    4    7    8
-5    0    7    8
```

Example 4.14.H

Suppose that "TT τ α,"
will present a stimulus from tape τ, perform a certain test, and set α = 0 if the test fails, α = 1 if it succeeds. We wish to define a subsequence V so that "V τ α ☛ β γ," will perform error correction reinforcement with α stimuli from tape τ, then perform the above test on β stimuli from tape ☛ and set γ to the number of successes.

```
DEFINE V (T S T S SR),
     ZERO $5, EC $2 $1,
     (,$4, TT $3 $6, ADD $5 $6,) END
```

* * * * * * * * * * *

If we wish to use an extended operand as a parameter, we use a type E parameter. Suppose, for example, that the second parameter is type E; then $2 is not affected, but $B2 will be defined as a block of the right size, and the contents of the specified operand will be moved into it. Note particularly that we do not use "BLOCK $B2 ...," in the subsequence; this is done implicitly when the subsequence is used.

If we use the subsequence with less than two parameters then $B2 will become a 1-word block containing 0 (or if the / came before E [e.g., "DEFINE SUB (S/E),"]) then $B2 will be unchanged from the previous use.

If instead of type E, we use type ER, this will function exactly as described above and also, at the conclusion of the

subsequence, the values in $B2 (which may have been changed by the sequence) will be put back into the specified operand (which must of course be changeable).

(NOTE: If when a subsequence is used, SR or ER parameters are omitted, they will be treated as S or E respectively.)

* * * * * * * * * *

Example 4.14.I

```
DEFINE L (E E E E ER S),
IF ZERO $6,
    (AND $B1 $B2 $B5, AND $B5 $B3, AND $B5 $B4,)
IFN ZERO $6,
    (OR $B1 $B2 $B5, OR $B5 $B3, OR $B5 $B4,)
END
```

L $e\alpha_1$ $e\alpha_2$ $e\alpha_3$ $e\alpha_4$ $e\beta$ γ,

will set $e\beta$ to the "logical AND" of $e\alpha_1$ to $e\alpha_4$ if $\gamma = 0$, on to their "logical OR" if $\gamma \neq 0$.

* * * * * * * * * *

It is not allowed to have a GO-phrase, which goes to a named phrase in a lower level subsequence (that is, one used by this one, or used by one used by this one, etc.). However, it is permissable to go to a named phrase in a higher level subsequence or the main sequence. In other words, one can only get into a subsequence in the normal way; but one can get out of it by means of a GO.

*4.15 Subsequences (Exotic)

This section contains further information about subsequences, and may be omitted on first reading.

The phrase FINSUB may be used within a subsequence, and its effect when executed is to cause the immediate termination of the subsequence, and return to the main sequence or higher level subsequence which used it. This may also be used in the form

FINSUB a,

where a may be any simple operand, and in this case not only will the subsequence be terminated, but in the higher level sequence the next a phrases after the phrase using the subsequence will be skipped (see discussion of skipping of phrases in Section 4.13).

The phrase FIN will cause immediate termination of the current section, whether used in a subsequence or in the main sequence.

In addition to the types of parameters discussed in the previous section (S, SR, etc.) the following types may also be used.

Type B. If the i^{th} parameter is specified as Type B, then when the subsequence is used the i^{th} parameter specified should be a block or list name (not a block slice, but simply the name and nothing more). $Bi will be made synonomous to the named block or list; that is, all references within the subsequence to $Bi will act as if they were direct references to this block or list. (See discussion of LLSYN in Section 4.9.) Note the difference between this and a Type E operand, where $Bi is actually made into a block itself, with its own storage space, and the contents of the external operand are actually moved into it.

Type N. If the i^{th} parameter is specified as Type N (Name), then the parameter supplied should be a word of from one to six letters. The binary coded form of these letters (followed by blanks if there are less than six) will be placed into $i. Note that all that is dealt with here is the binary representation of the given name, with no reference to any meaning this name may have elsewhere

in the system (e.g., as name of a variable, a subsequence, a primary code, etc.). The only significance attached to this name is what is given to it by the use of this parameter in the subsequence. (The binary representation of alphabetic information may also be obtained by use of the $-operand "$(HS)bbbbbb" where bbbbbb may be any six characters (other than apostrophe). This is a simple unchangeable operand whose value is the code representation of these six characters.)

Type L. This type of parameter may be used to supply an indeterminate number of values (that is to say, the number supplied may vary from one use of the subsequence to another). If the i^{th} parameter is Type L, then when the subsequence is used, zero, one, or more standard operands are supplied, with the last one followed by an asterisk. $i will be set to n, the number of operands given; $Bi will be defined as an n-word block, and the values of the operands placed in consecutive words in it. Thus, the subsequence can determine how many values are supplied and get access to each of them.

* * * * * * * * * *

Example 4.15.A

We wish to define a subsequence, MAKE, which can be used as follows:

MAKE b xxx α β * zzz ,

or

MAKE b xxx α β γ * zzz,

b may be a block name; xxx may be one of the words BLOCK, LIST, or CLEAR; zzz may be the word ADD or may be omitted. If xxx is BLOCK, b should be defined as a block of dimensions α x β, or α x β x γ, respectively; if xxx is LIST, b should be defined as a list of dimensions α x β or α x β x γ; and if xxx is CLEAR, space used by b in a previous use as a block or list should be released. If zzz is ADD, then one item will be added to the list b (this should not be used unless xxx was LIST).

```
DEFINE MAKE (B N L N),
IF EQUAL $2 $(HS)CLEAR (, LLCLR $B1, FINSUB,)
IF EQUAL $2 $(HS)BLOCK ,
```

```
(, IF EQUAL $3 2, (, BLOCK $B1 $B3(1) $B3(2), FINSUB,)
      BLOCK $B1 $B3(1) $B3(2) $B3(3), FINSUB, )
IF EQUAL $3 2, LIST $B1 $B3(1) $B3(2),
IF EQUAL $3 3, LIST $B1 $B3(1) $B3(2) $B3(3),
IF EQUAL $4 $(HS)ADD      , ZERO $B1**NEW,     END
```

Note in the use of $(HS) how 6 characters are always left after the) . E.g., it would not be correct on the last line to have $(HS)ADD, ZERO etc; as the first six characters after $(HS), i.e., "ADD, Z", would be part of this operand.

* * * * * * * * * *

Type M. This is similar to Type L, but instead of supplying a list of standard operands, a list of extended operands may be supplied. $i will be set to n, the number of operands; $Bi will be defined as an n x 4 block. $Bi(j), which is a 4-word slice, will be used as a locator, and defined as a block of the same size as the j^{th} operand, and the contents of that operand will be moved into this block. Thus the subsequence can determine the number of operands supplied and have access (in a rather cumbersome fashion) to the contents of each. See the discussion of LLSYN in Section 4.9, particularly page 4.9.9 to 4.9.10

The following miscellaneous points may be noted.

(1) In the specification of parameter types, any E, ER, or L specification may be followed by a letter C; if so, the corresponding $Bi will be cleared when the subsequence is finished; that is to say, any space used by this block during the execution of the subsequence will be released.

(2) Normally, any explicit numbers provided as operands for a Type S, E, L, or M (and variations on these such as SR, EC, etc.) are considered as integers. However, if the specification letter is followed by integer k, then such numbers will be interpreted with k binary places; and if the specification letter is followed by * the number will be interpreted as floating point.

(3) In a Type L or M specification, the terminating * may be omitted if no further operands follow, that is, if the * would be immediately followed by the comma at the end of the phrase.

DEFINE A * B,

where B is the name of a primary code, or previously defined subsequence, will cause A to henceforth have the same meaning that B now has. (This is not to be followed by END.)

If a define phrase, either the normal kind or the one just mentioned, uses the name of a primary code or a previously defined subsequence, the new definition will replace the current meaning. (In case this was done unintentionally, a warning message will be printed.)

* * * * * * * * * *

Example 4.15.B

Suppose we have a program involving a rather complicated proceedure for generating a sequence of stimuli, which is not working correctly and we can't find out why. Suppose we decide we could probably discover our mistake if we knew exactly which stimuli were being chosen by the SSN phrases in the program. In other words, we would like each SSN phrase to print the number of the saved stimulus it is getting each time it executes. We could do this by inserting a print phrase after each SSN phrase, but if there are many of them it might be simpler just to add the following at the beginning of the program.

```
DEFINE ZSSN * SSN,
DEFINE SSN, PRINT $1, ZSSN $1, END
```

The first line defines ZSSN to mean the same as the real SSN; then SSN is given a new meaning. Note that even after that, ZSSN will still have the meaning that SSN had at the time ZSSN was defined.

* * * * * * * * * *

The material in sections 4.14 and *4.15 is summarized in 5.4.

4.16 Output (basic)

Any experiment we may perform must produce some form of printed output. This may range from a single number (e.g., number of correct responses to a certain group of stimuli) to a detailed description of the condition of the perceptron at each stage of the experiment, statistical analyses of a whole series of experiments, etc. As previously mentioned, certain testing phrases will automatically produce a standard line of output. However, in many cases this alone may be unsatisfactory. We describe in this section and the next a phrase which allows us to form and print any output we desire.

This phrase is made up of the code LINE, followed by one or more sub-phrases. Each sub-phrase consists of an alphabetic code, possibly followed by other information depending on the particular sub-phrase. (Note that the sub-phrases must not be separated by commas. A comma, as usual, marks the end of the phrase.)

The basic idea is that a line of output (up to 119 characters) is built up working from left to right. Various sub-phrases cause various kinds of information to be added to the line, while others cause the accumulated line to be printed and control the spacing between lines, etc. We now proceed to a discussion of some of the particular sub-phrases.

"V α" (α may be any standard operand) will cause the value of α (an integer) to be inserted in the line. The number of characters it will put in depends on the value; that is 0 to 9 will put one character, 10 to 99 or -1 to -9, two characters, 100 to 999 or -10 to -99, three characters, etc.

"B α" or "BLANK α" will cause α blanks to be inserted (α may be any standard operand).

"BCD *...*" will cause the characters appearing between the *'s to be put in the line. Any desired characters other than * or ' may be used. If we wish to have some *'s in the inserted material we may use any other non-blank character as a marker instead of *. The general

rule is that the first non-blank character after BCD is taken to be the marker; the second appearance of this character marks the end of the sub-phrase.

"START" initializes to the beginning of a line.

"PRINT" causes the line currently built up to be printed, and then reinitializes to start a new line.

Note that a LINE phrase does not automatically print a line unless the sub-phrase PRINT is given. Thus several phrases may be used to build a single line, if desirable.

* * * * * * * * * * *

Example 4.16.A

LINE START BCD *TEST * V2 B1 V-4 B1 V12796 PRINT V3 V57 PRINT,

would cause the following to be printed

TEST 2 -4 12796
357

Example 4.16.B

We wish to test a perceptron, presenting 1000 stimuli in groups of 10, and printing the number of correct responses in each group. We also wish to print the total number correct. Suppose "A t α," is a previously defined subsequence, which presents 1 stimulus from tape t, and sets α = 1 if response is correct, α = 0 if not.

```
SET TOT 0,
(5, LINE START,
 (20, SET COUNT 0,
  (10, A T1 N, ADD COUNT N,)
   LINE V COUNT BLANK 1, ADD TOT COUNT,)
 LINE PRINT,)
LINE BCD *TOTAL CORRECT = * V TOT,
```

Typical output from this might be

```
9 7 10 8 9 9 9 8 9 10 8 9 8 6 9 9 9 8 9 10
10 8 8 9 10 9 9 8 10 8 6 8 9 9 5 9 10 8 9 9
9 9 9 9 9 9 8 9 8 9 9 9 7 9 10 9 9 10 9 9
8 8 9 8 7 8 9 7 9 9 9 8 7 10 9 9 8 9 9 8
7 8 7 9 10 8 7 9 5 9 9 10 10 10 10 9 9 9 8 10
TOTAL CORRECT = 863
* * * * * * * * * * *
```

If the number of characters accumulated in the portion already formed exceeds 119, the first 119 will be automatically printed, (without need for a PRINT sub-phrase) and the rest will start a new line (indented by 3 spaces). The characters inserted by a single sub-phrase will not be split between 2 lines; if a certain sub-phrase will cause the line to overflow, then the printing will take place first and the entire insertion due to this sub-phrase will go on the new line. This same remark applies to most of the other sub-phrases discussed later. Exceptions will be noted when they appear.

"V0 α" where α may be any standard operand, will insert the value of α as a 12 digit octal number.

"E0 eα" where eα may be any extended operand, will insert the value of eα as an octal number (12 digits/word).

"EB eα" will insert the value of eα in binary (36 digit/word).

Since a single E0 or EB sub-phrase may conceivably insert more than one full line the remarks made above do not apply to these; that is, the octal or binary numbers inserted by these phrases may be split between lines.

If we wish to print only part of the octal number, we may use

"E0 (ia) eα" which will print only the first ia digits, or

"E0 (ia, ib) eα" which will omit the first ia digits and print the next ib.

ia and ib may be any I-operands (in particular, they may be any simple operands). "EB (ia) eα" and "EB (ia, ib) eα" are similar.

* * * * * * * * * *

Example 4.16.C

Suppose B is an extended operand holding a packed activity vector for a 1000 unit perceptron. We wish to print this in 10 rows of 10 units each.

```
(*I 900 0 100, LINE START EB(I, 100) B PRINT,)
```

Example 4.16.D

Suppose we wish to change the preceding example to print each 100 units in 10 groups of 10 separated by a space, instead of an unbroken row of 100 zeros and ones.

```
(* I 900 0 100, LINE START,
     (* J I+90 I 10, LINE EB (J, 10)  B  BLANK 1, )
     LINE PRINT,)
```

* * * * * * * * * *

The vertical positioning of the lines on the page can also be controlled by appropriate sub-phrases.

"PAGE" will cause the next line printed to go on a new page. It does not matter if the printing is done by the current LINE phrase or not until a later one.

"BOTTOM α" will cause the next line printed to go on a new page, unless there is still room for at least α more lines on the current page; in the latter case, the BOTTOM sub-phrase has no effect at all.

"SPACE" will cause 1 blank line to be inserted.

"SPACES α" will cause α blank lines to be inserted. (However, if there are less than α lines on the current page the remaining blank lines will not be inserted on the new page.)

We saw above that numbers printed by a V sub-phrase take up a varying number of spaces depending on the value of the number. The same will be true of some other phrases discussed in the next

section. However, sometimes it is desirable to place the numbers according to some regular spacing-- e.g., if printing a matrix.

"F α" causes the number inserted by the next sub-phrase to occupy a field of α characters. If it would normally use less than α characters, a sufficient number of blanks will be placed to the left of it. It must not use more than α characters.

* * * * * * * * * *

Example 4.16.E

```
LINE START V 2 B 1 V 473 B 1 V 6 PRINT
           V 59 B 1 V 1 B 1 V 6789 PRINT,
```

would cause the following printing

```
2 473 6
59 1 6789
```

but

```
LINE START F2 V2 F5 V473 F5 V6 PRINT
           F2 V59 F5 V1 F5 V6789 PRINT,
```

would cause the following printing

```
 2  473    6
59    1 6789
```

Example 4.16.F

We wish to write a subsequence which will print a matrix. We would like it to look like

```
PM eα β γ δ,
```

where eα holds the matrix (stored by rows), β is the horizontal dimension, γ the vertical dimension, and δ is an identification number to be printed before the matrix. Terms of the matrix are integers from 0 to 9999. Dimensions of matrix do not exceed 20. (If the reader has not looked at the first part at least of the material on blocks (4.9) he should skip this example.)

```
DEFINE PM (E S S S),
LINE START SPACES 2 BOTTOM $3 + 2
     BCD *MATRIX NUMBER * V $4 BCD * (*
     V $2 BCD *X* V$3 BCD *)* PRINT SPACE,
(*J $3, MPY J+-1 $2 JA,
     (*I $2, LINE F5 V $B1(JA + I), )
     LINE PRINT,)
END
```

Note how the use of BOTTOM insures that no matrix will split between two pages, yet does not prevent having several on the same page if they are small. Also note how the use of F insures a neat layout for the matrix.

* * * * * * * * * * *

Every page that is produced has a heading line, followed by a blank line. At any time we wish we may change to a new heading line. This is done by the phrase HEAD. This may contain the same sub-phrases as LINE, and builds up a line in the same way. This line, however, will then be saved and used as a heading for all future pages. Note the following points.

i) PAGE, SPACE, SPACES, BOTTOM are, of course, meaningless in a HEAD phrase, so may not be used.

ii) The line is limited to 110 characters. (This is to allow room for "PAGE n" which is automatically added at the end. Page numbering starts at 1 again where a new HEAD is given.)

iii) The first line printed after execution of a HEAD phrase will go on a new page.

iv) HEAD may be followed by an *. This will indicate that a line is partly built up by previous line phrases, and is to be used as the beginning of the heading line. If this is not used, the entire heading line must be built up by the single HEAD phrase.

v) When HEAD is given the line generated is printed once on the online printer.

See Example 3.4 for an example of use of HEAD.

Material in this section is summarized in 5.11.

*4.17 Output (Exotic)

This section contains further information about the LINE phrase, and should be omitted on first reading.

The following subcodes provide additional ways of entering information into the output line.

VB β α *a

will consider that the value of α is expressed with a fraction of β binary places and will insert it with "a" decimal places. (α and β may be any standard operands, a any simple operand.)

VF α *a

will consider that the value of α is in floating point form and will insert it with "a" decimal places.

VFF α *a

Like VF, this will consider the value of α to be in floating point form, but will insert it into the output line in floating point decimal form (that is to say, as a number with magnitude greater than or equal to one but less than ten, followed by a power of 10) with "a" decimal places.

V α *a

will consider the value of α as an integer, and will insert $\alpha \times 10^{-a}$ (in other words, it will insert the integer value of α but place a decimal before the last a digits).

In all of the above, "*a" may be omitted if "a" is to be zero. (NOTE. If in any of these forms α is a block or list name not followed by explicit subscripts nor an explicit item number; then the "*a" would appear to be an item number. Such a situation, however, is not likely to arise. In the case of a block, since α must be a standard operand, there would certainly be subscripts. The only trouble could come in the case of a list with one-word items; in such a case it would not be permissible to omit an item number of the form **NEXT, as can be done in most contexts.)

H eα

eα, which may be any extended operand, will be considered as containing alphabetic information, which will be inserted in the output line.

H(i) eα

will be the same, but only the first i characters will be inserted.

H(i, j) eα

will ignore the first i characters, and insert the next j. i, j may be any I-operands.

Note that alphabetic information is represented in the 7094 by a code which uses six bits per character; hence each computer word holds six characters. The $-operand $(HS)bbbbbb (where bbbbbb are any six characters, other than an apostrophe) may be used to indicate a word containing these six characters in coded form. This expression is considered a constant (hence may be used as a simple operand, standard operand, etc.) The expression

$(H *)bbb...bb*

may be used for an extended operand. bbb...bb may consist of any number of characters, and they may be any characters other than * or apostrophe. The operand will consist of the necessary number of words to hold these characters in coded form. If the number of characters is not an exact multiple of six, enough blanks will be added at the end to fill out the last word. If it is desired to have an * occurring in the string of characters, any non-blank character (other than apostrophe) which does not occur in the string may be used to replace the *, in <u>both</u> occurrences. (If the chosen character is a letter, it must of course be separated from the H by a blank.)

TAPE α

where α may be any standard tape designator, will cause the value of α to be inserted in the line as a tape address (e.g., A5, B3, etc.).

If the tape designator is of the form Tn, the actual tape address chosen by the system will be inserted in the print line, not "Tn".

C

will cause the current case number to be inserted in the output line.

CP

will cause the phrase "CASE n" to be inserted where n is the current case number.

(An,...,...,...)

may be used to insert one of several strings of characters, under the control of Counter An. These strings may contain any characters except comma, slash, or apostrophe; they do not need to be the same length. The strings must be separated by commas. If k is the current value of counter An, then the k^{th} string will be inserted. (NOTE: If the value of k is greater than the number of strings given, then the whole series is assumed to be repeated. However, if one of the separating commas is replaced by a slash, then only the strings following the slash are assumed to be repeated.)

Output produced by LINE phrases is normally written onto the standard output tape (A3) (as is most output under the FORTRAN monitor system) for later printing on an offline printer. In special situations it may be desirable to print a message on the on-line printer. In this case the sub-phrase PRINT may be followed by the word ONLINE; or it may be followed by the word BOTH to indicate that the line should both be printed on the online printer and placed on the output tape. PRINT may also be followed by "*a" where a may be any simple operand; printing will take place offline if the value of a is one, online if it is two, both if it is three, and neither if it is zero; other values should not be used. These qualifiers (ONLINE, BOTH, or *a) may also be used after subcodes SPACE, SPACES, PAGE and BOTTOM, discussed in the preceding section. (NOTE: One should write, e.g., "BOTTOM 3 ONLINE", not "BOTTOM ONLINE 3".)

The sub-phrase CLEAR will cause the entire print line to be made blank (in normal circumstances, it is not necessary to use CLEAR at the start of each new line as the line is automatically cleared after printing).

SETPP α

will cause the next item and any subsequent ones to be entered starting at position α in the print line,

$2 \leq \alpha \leq 120$.

SAVE

will cause the line which is currently being built up to be saved so that a new one can be worked on, and this one continued later (SAVE would have to be followed by CLEAR and START to commence a new line.

RESTOR

will bring back the line which was most recently saved. (NOTE: Any number of lines may be saved one after another and successive uses of RESTOR will bring them back in the reverse order to that in which they were saved.)

As was discussed in the last section, if the line being built up exceeds 119 characters, the first portion will automatically be printed (offline) and the rest will go into a new line. It was also mentioned that a single piece of information will not be split between two lines (except in the case of the sub-phrases EO, EB, and H) but rather that if the number will not fit into the current part of the line it will all be placed in the new part. Sometimes, however, it might be desirable to ensure that a section of line created by several sub-phrases will not be split in this fashion. This can be done by use of the subcodes G and GZ, which function as follows. The information inserted into the line is divided into pieces called groups. Normally a group consists of the information entered by a sirgle sub-phrase (except in the case of EO, EB, and H, where each six characters constitute a group). However, if the

subcode G is used, everything from this point to the next appearance of a G or GZ will be considered to constitute a single group (G may be used to start a group or to terminate one group and start another; GZ is used to terminate the current group and revert thereafter to the normal situation of one subphrase-- one group). If the material inserted by a single group is too long to fit into the remainder of the current line, it will not be split but the current line will be printed and the new material will go at the beginning of the next line. Warning: It is the user's responsibility to see that no single group creates output too long to fit into a single line even if placed at the beginning. If this should happen, it will cause the program to terminate completely.

As mentioned in the previous section, when this form of overflow takes place, each succeeding portion of the line will be indented by three spaces (thus limiting these portions to 116 characters). If desired, this value may be changed by the sub-phrase "INDENT α" which will change the indentation from three to the value of α. This new value will be permanent, that is to say, it will be used for the rest of this section unless a new INDENT is given.

The output line may be accessed directly by means of the following $-operands. Note that the output line consists of 20 words, each containing six characters (the first character has a special function internal to the system, which is why the line is limited to 119 characters as far as the user is concerned).

$LINE

This is a changeable, extended operand representing the entire output line.

$LINE(i)

where i may be any I-operand, represents the first i words of the output line.

$LINE(i,j)

where i and j may be any I-operands, represents words i + 1, i + 2, ..., i + j, of the output line.

$LINEPP

is a simple operand whose value is the position in the output line at which the next item to be inserted will begin. (Note that the output line runs from position 2 to 120 inclusive.) This operand is unchangeable. It may in effect be changed by use of the SETPP phrase as noted above, but the operand itself may not be directly changed.

Some of the concepts in this section are illustrated in the examples in sections 3.4 and 3.5.

The material in this (and the preceding) section is summarized in 5.11.

4.18 Intermediate Input-Output

In this section we will consider phrases which allow us to place information onto a tape unit and later get it back. As an illustration consider the following esample:

* * * * * * * * * *

Example 4.18.A

We wish to experiment on a certain perceptron by presenting 20 stimuli, saving the activity vectors and forming a G-matrix. Suppose the perceptron is very large, say 10,000 A-units. We could set up a deck in fairly straight forward fashion as discussed in preceding sections. The problem is that this may require more space than is available. One packed activity vector requires 278 words, so twenty require 5560. This alone is not excessive, but on top of the requirements of such a large perceptron, it will probably not be possible. The solution is to break the testing into two sections; the first forms the activity vectors one by one and writes them on a tape unit. The second reads them all in (at this time the perceptron is no longer taking up space and so there is no trouble finding room for them). Thus:

```
REW T2, REW T1, BLOCK B $NPACK 2, WRITE T2 $NPACK2,
(20, PRESNT T1, GETACT 2 B, WRITE T2 B,) REW T2,
END
READ T1 N, BLOCK BB 20 N,
(*I 20, READ T2 BB(I),)
--now form G-matrix from activity vectors in BB--
```

* * * * * * * * * *

The phrase

WRITE σ eα_1 eα_2... eα_k,

will write onto tape σ the contents of eα_1, eα_2, etc.

Similarly

READ σ eα_1 eα_2... eα_k,

will read what has been written on tape σ and put it into $e\alpha_1$, $e\alpha_2$, etc.

The information on a tape is divided into groups called "records". Each WRITE phrase makes one record (note that different records may contain different numbers of words). A READ phrase should read one record. The operands in the READ phrase should add up to exactly the number of words in the record to be read.

For most applications it will suffice to write certain information, REWind the tape, and later read back what was written, in the same order; but sometimes it may be desirable to read the records in irregular order, or write over previously written records.[1]

A tape can be positioned by a phrase consisting of

TAPE σ

followed by one or more of the following sub-phrases:

RECORD α
PRINT α
ENDFIL
FILE α
PRINTF α
REWIND
UNLOAD

α may be any standard operand
σ any standard tape designator

RECORD α — if $\alpha > o$, tape σ will be spaced forward α records; if $\alpha < o$, backward $|\alpha|$ records.

PRINT α — if $\alpha > o$, causes the contents of the next α records to be printed; if $\alpha < o$, the contents of the next $|\alpha|$ records will be printed and the tape will then be backspaced to its original position.

ENDFIL — will cause a special record, called an <u>end-of-file-mark</u> (EOF) to be written on tape σ. These marks may be used to divide the tape into groups of records called <u>files</u>.

[1] If a particular record is rewritten, the previous records are unaffected; however, all following records are lost.

FILE α — if $\alpha > o$, tape will be moved forward α files (i.e., until α end-of-file-marks have been passed); if $\alpha = o$, it will be moved backward to the beginning of the current file (i.e., as far back as possible without passing an end-of-file-mark); if $\alpha < o$, it will be moved backward to the beginning of the $|\alpha|$ file preceding the current one.

Note: If RECORD or FILE attempts to move the tape back before the beginning, it will simply move it to the beginning.

PRINTF α — if $\alpha > o$, the contents of α files will be printed. (If the tape is positioned partway through the current file only the remaining records will be printed for this file.) if $\alpha < o$, the contents of $|\alpha|$ files will be printed, and the tape will be moved back to the beginning of the file where it was at first.

Note: When tapes are printed by PRINT or PRINTF the printout will show the division into records and files.

REWIND — this will cause the tape to be rewound-- i.e., moved back to the beginning.

UNLOAD — this will cause the tape to be rewound and unloaded. (This is only used if it is necessary for some reason to save a tape and use it again later, e.g., the tape may contain stimuli which took a long time to generate and which will be needed for future experiments.)

A tape can be rewound either by

TAPE σ REWIND,

or

REW σ ,

If a READ phrase reads an EOF, nothing will be put into the operands in the READ phrase, but the special operand $QFILE will be set to a nonzero value; this may be checked and reset to zero whenever desired.

* * * * * * * * * *

Example 4.18.B

We wish to print all the remaining records in the current file on tape T1, and put the tape back where it is now. This can't be done by

```
TAPE T1 PRINTF -1,
```

as this will leave the tape at the beginning of the current file. It can be done as follows:

```
ZERO N $QFILE,
*LOOP READ T1, SET N N + 1,
     IF ZERO $QFILE, GO LOOP,
     SET M N + -1,
     TAPE T1 RECORD -N PRINT -M,
```

*4.19 Data

In some applications it may be necessary to supply a large amount of data to the program, e.g., the specification of a number of stimuli which cannot be generated internally. This can be actually built into the program, e.g., by use of the FILL phrase, by SET phrases, etc. However, in some cases this becomes quite cumbersome; and a special facility known as the DATA tape is supplied to overcome this. The DATA phrase is not executed like most phrases, but supplies data which, at the time the input deck is first interpreted, are placed on a special tape; this data may be obtained from the tape by use of the special $-operands discussed below.

The form of the DATA phrase is as follows.

DATA$ ... $,

Between the two occurrences of the $ may come any number of data items to be put on the tape. These may be a number with or without a decimal point, or a symbol. Any symbol may be used other than $ (or, of course, apostrophe). Each of these data items will be placed on the tape in the order in which it occurs and also an indication as to whether the item was an integer, a fraction, or a symbol. (2.0 would be classified as a fraction, while 2 would be classified as an integer.) More than one DATA phrase may be used if desired. Each such phrase will create one "file" on the data tape. As was mentioned above, the DATA phrases have their effect during the interpretation of the program. Therefore, no matter where they occur, all the information will be available on the data tape when the first section begins executing. At the beginning of execution the data tape will be positioned to the first item in the first file. The phrase

DFILE a,

may be used at any time to position the data tape to the first item in the a^{th} file.

Information is obtained from the data tape by several special operands, of which the principal one is $DATA. This is a standard

operand and its value is the next item on the data tape. Successive references to this operand will obtain successive items from the tape. Other than the use of the phrase DFILE mentioned above, this is the only way in which the position on the data tape is changed. Several other $-operands to be discussed in a moment can be used to obtain further information about the current item; but successive uses of one of these will all refer to the same item, which will be the one referred to by the most recent $DATA.

We stated a moment ago that the value of $DATA is the value of the next item from the data tape. To be more precise, if the item on the data tape is numerical, the value of $DATA will be the integer part of its value. If it is a symbol (either a one-character special symbol or an alphabetic name of from one to six letters) then the value of $DATA will be the usual internal coded form of this symbol (with blanks added at the right if it contains less than six characters).

$DATAI — will have the same value as the most recent $DATA, but (as already noted) will not have the effect of moving the data tape.

$DATAF — will have for its value the fractional part of the most recent item, expressed with 35 binary places.

$DATAM a — will have for its value the last item (integer plus fractional part) expressed with "a" binary places.

$DATAQ — will have for its value

- 0 if the current item is an integer
- 1 if it is a fraction
- 2 if it is a symbol
- 3 if it is an end of file (see below)
- 4 if it is a binary card (see below)

(NOTE: The only difference between 2 and 2.0 as data items would be in the corresponding value of $DATAQ (zero or one, respectively). Each would have a value for $DATA or $DATAI of 2, and $DATAF of .0).

The values of $DATAF and $DATAM are meaningless unless the current item is an integer or a fraction.

The last item in a file on the data tape is followed by an end of file mark. An attempt to refer to this as a data item will cause a value of $DATAQ equal to three. Further uses of $DATA will obtain items from the next file if there is one. The operand $DFILE will at any time have for its value the number of the file in the data tape from which items are currently being obtained.

* * * * * * * * * *

Example 4.19.A

A certain experiment is to be controlled by two lists of numbers, each of indeterminate length. One of these lists consists of integers and the other of fractions expressed with 18 binary places. Let us suppose that the subsequence DOIT has been defined to actually perform the experiment and that it expects to find in list BA the list of integers and in list BB the list of fractions. We decide we would like to control the performance of this experiment by a series of data items as follows:

i) A series of integers followed by a comma will provide new values to go in list BA.

ii) A series of fractions followed by a comma will provide values to go in list BB.

iii) The word DO will indicate that the experiment is to be performed at this point.

This allows us to specify new values for one of the lists and leave the other unchanged, then repeat the experiment if we so desire; or we may specify new values for both lists if that is preferable. We may call for the experiment to be performed as many times as we wish and will stop when the data file is exhausted. Assuming the subsequence DOIT to be already defined, the following program will accomplish what we desire.

```
*S SET D $DATA,
JUMP $DATAQ,
     OPEN A, LIST BA,
          *AA MOVE D BA**NEW,
               SET D $DATA,
               IF EQUAL $DATAQ 2, GO S, GO AA,
               CLOSE A,
     OPEN B, LIST BB,
               *AB MOVE $DATAM 18 BB**NEW,
               SET D $DATA,
               IF EQUAL $DATAQ 2, GO S, GO AB,
               CLOSE B,
OPEN C, DOIT, GO S, CLOSE C,
END
```

Note we obtain the first item on the data tape following which the "JUMP $DATAQ", causes the compound phrase beginning with OPEN A to execute if that item was an integer, the compound phrase beginning OPEN B to execute if it was a fraction, the compound phrase beginning OPEN C to execute if it was a symbol; the section terminates if it was an end of file. In the compound phrase OPEN A we define BA as a list (which automatically removes any items previously on BA if it has already been used), put the current item into this list and all following items until we find a symbol (which can be nothing other than the comma terminating the list), and then go back to the beginning. Similarly in case the first item is a fraction. If the first item is a symbol, it can be nothing other than the word DO, so in this case we execute the subsequence DOIT.

This method assumes that the user will not make any mistakes punching the items in the DATA phrase and if there are such mistakes, heavens knows what will happen. It would be quite simple to make a few minor additions which would check for most mistakes and either correct them if possible, or print out a message.

* * * * * * * * * *

One more facility should be mentioned. Some information is most conveniently put in on binary cards, rather than as a series of numbers or symbols. A card is normally considered to have been punched in the usual fashion on a keypunch and is translated into the machine's own internal code, with each column being converted into a six-bit code. It is assumed that the actual pattern of punches on the card is of no interest to us but only the characters for which they stand, and in fact many possible combinations of punches are not even permitted under this arrangement. However, some kinds of information, e.g., a pictorial stimulus, may be best punched into a card as an actual pattern of punches on the card, with no relationship to the usual representation of characters. If such a card is converted by the machine in the usual fashion, the result will be meaningless. However, if column one of the card contains a 7 and 9 punch, this tells the machine that it is not to make the usual conversion but simply to read in the actual pattern of punches in the card. This will be read in reading the columns from left to right, and reading each column from top to bottom. Since a column contains 12 bits, the image of the card read into the computer will consist of 27 words, each containing 3 columns (except the last will contain only 2). This is referred to as a binary card image.

Binary cards may be included as items in a DATA phrase. Each binary card is considered as a single item. When such an item is obtained, the values of $DATA, $DATAI, $DATAF, $DATAM will be meaningless (however, of course $DATA must be referred to to get the item at all). The value of $DATAQ will be 4 and $DATAB will be an extended operand of 27 words containing the card image.

(NOTE: The fact that the data items are kept on a tape is really of no relevance to the user. He will have no occasion to access this tape by the READ/WRITE phrases or any other way than the means discussed in this section. In particular, if the reader is an experienced programmer he might note that the system does not literally put each data item into a separate record on the tape and read them one by one, which would of course be very time-consuming if large quantities of data were handled this way.)

*4.20 Perceptron Specifications, Exotic

This section contains most of the more involved information concerning the specification of a perceptron.

Type 8 linkages may be used to form many types of connection patterns which cannot be done by means of linkage types discussed in Section 4.3. It is especially useful when the user wants to give specific connections rather than allowing them to be chosen at random, although it is not limited to this. This type of linkage allows for fixed inputs, which may come from different layers. The initial layer specified in the LINK phrase is considered the normal input layer and will be referred to in the following discussion as the normal (or standard) layer; however, it is possible to have inputs from other layers if so desired. Basically the idea is that one supplies on a tape a file containing the specifications of desired inputs.[1]

The LINK phrase must contain the subcode

TAPE σ α

where σ may be any standard tape designator indicating the tape on which the information is to be found; and α may be any standard operand, telling which file on the tape the information is in.

The normal function of the type 8 linkage is similar to that of a type 2, that is, a unit has added to it a certain weight if the unit from which it receives the input is active. However, unlike a type 2, each input may, if desired, have a different weight; this is part of the information which is carried in the tables.

[1] One may refer to Section 5.8 (page 5.8.6) for the exact details of the arrangement of information in this file, create the file by any means desired, and put it on the tape using WRITE phrases (or even create the file by a completely different program). However, it is normally not necessary to do this; a special phrase, CONEC, which will be discussed shortly, is provided which should suffice to handle almost any possibility. The use of this phrase relieves the user of any concern for the actual details of the information placed on the tape.

Another variation, known as the direct option, works as follows. If the input unit is active, then the receiving unit will have added to it the value of the input unit (that is, the sum of its inputs minus its threshhold). If this option is desired, the subcode DIRECT should be included in the LINK phrase.

A further variation, called the unconditional direct option, causes the value of each input unit to be added to the receiving unit, whether or not the input unit is active. If this option is desired, the subcodes DIRECT and UNCON must both be included in the LINK phrase.

Important note: The direct and/or unconditional option is considered a part of the structure of the perceptron. Therefore, it is not subject to changing at the time the perceptron is being tested. Furthermore, it applies to the entire linkage, it cannot be applied to some inputs in the linkage and not to others. (However, the same layer may receive several linkages, some of which exercise this option and some of which do not.)

It should be carefully noted that when the perceptron specification phrases (LAYER, LINK, etc.) are executed, the only immediate effect is to place the information they provide where it will later be available to the perceptron generator. The actual generation of the perceptron does not take place until the end of a section in which the specifications are given. Therefore, in the case of a type 8 linkage it is irrelevant whether or not the tables are available on the tape at the time the LINK phrase is executed. It is simply necessary that they be there before the end of that section. Thus the CONEC phrase could come after the LINK phrase or on the other hand, it could come earlier, even in an earlier section.

We will now discuss various forms of CONEC and its associated phrases.

When one wishes to generate tables by means of the CONEC phrase one must first give the phrase

CONECI ☛ α,

This tells the system that tables are about to be generated and are to be put on tape σ in file number α. If α is greater than one, the previous files must already be written on the tape, either by earlier uses of CONEC or by WRITE phrases or any other means that the user desires. (NOTE: It is in the nature of the tape units that any file on the tape may be rewritten at any time, e.g., if the fifth file is rewritten, this effectively destroys the old fifth file and any following files, but files one to four are unaffected.) When one is finished describing the file by one or more CONEC phrases, one must give the phrase

CONECZ,

which tells the program to finish the file.

The CONEC phrase itself has many forms, of varying degrees of complexity, which we will now discuss.

CONEC SIMPLE α β_1 p_1 β_2 p_2 ...,

may be used to specify inputs to one unit in the terminal layer. α is the number of this unit. β_1 β_2 etc. are the numbers of the units in the initial layer from which the inputs come and p_i is the weight associated with the input from β_i. α, β_1, β_2, etc., may be any standard operands. p_1, p_2, etc., may be any 18-place operands. Note that the inputs come from the normal input layer, that is, the one specified in the LINK phrase.

CONEC DIRECT α β_1 β_2 ...,

This is similar to CONEC SIMPLE but is intended to create tables for a linkage which will exercise the direct or unconditional direct option as discussed above. In this case no weights are specified as none are required in the table.

Normally, the layer from which the inputs come is not specified in the tables; this information is obtained by the perceptron generator from the LINK phrase, just as in any other type of linkage. However, it is possible for the tables to override this for a particular input, and specify that it is to come from some other layer. If we wish to make use of this, we may use the phrase

CONEC LAYER α β,

which will cause all future inputs entered in the table to be specified as coming from layer α (delay β). α and β may be any standard operands; β may be omitted if zero is intended. The effect of this phrase will be canceled when CONECI is given to start a new set of tables, or may be canceled at any time by

CONEC LAYER 0 0, or simply CONEC LAYER,

In some applications involving specifically arranged patterns of inputs it is required to have several units in the output layer with essentially the same pattern of inputs, but shifted to a different position on the retina. This can be conveniently implemented by

CONEC COPY α β γ,

This specifies that unit α is to get the same inputs as unit β but shifted by γ; i.e., if β gets an input from unit n, α will get a similar input from unit n + γ in the same layer.

CONEC COPIES α β γ,

will have the same affect as the following sequence of phrases.

CONEC COPY α+1 α γ,
CONEC COPY α+2 α 2γ,
...
CONEC COPY α+β-1 α (β-1)γ,

In other words, it will have the effect of making β copies of unit α (counting unit α itself as the first one), each one shifted by γ from the preceding.

NOTE: The use of CONEC COPY and/or CONEC COPIES, instead of making successive uses of CONEC SIMPLE or DIRECT, is <u>not merely a convenience in writing</u>; it also results in a more efficient use of the space in the generated perceptron. When there are many inputs to a unit and many copies of the unit, it can lead to a quite significant saving.

* * * * * * * * * *

Example 4.20.A

We wish to have a linkage in which each layer receives five inputs from randomly chosen points in the initial layer. We would like n inputs to be positive and 5-n to be negative, where n is chosen at random from one to four, and is chosen separately for each unit in the terminal layer. Note that this cannot be done by a type 2 linkage because there not only the total number of inputs, but the numbers of positive and negative ones also, are constant for the entire linkage. We assume that NT is the number of units in the terminal layer and NI is the number of units in the initial layer.

```
BLOCK B 5,
OPEN A *I NT,
     (*J 5, MOVE $(B18) -1 B(J), )
     (*J $RAND 4, MOVE $(B18) 1 B(J), )
     CONEC SIMPLE I $RAND NI B(1) $RAND NI B(2)
          $RAND NI B(3) $RAND NI B(4) $RAND NI
          B(5),
     CLOSE A
```

Example 4.20.B

In certain experiments with a speech recognizing perceptron, units with the following type of input pattern were found useful. Inputs are received from 12 consecutive units starting at an odd-numbered unit in the initial layer, positive inputs from the odd-numbered units in the first 6 and the even numbered units in the last 6; and negative inputs from the others. It is desired to have units with this type of pattern in all possible positions on the input layer. The input layer has 80 units (therefore the terminal layer should have 35 units). Tables for this can be produced by the following.

```
CONEC SIMPLE 1  1 1  2 -1  3 1  4 -1  5 1  6 -1  7 -1  8 1
     9 -1  10 1  11 -1  12 1,
CONEC COPIES 1  35  2,
```

Example 4.20.C

In another version of the perceptrons mentioned above, units are required whose inputs could be described as the sum of inputs of three consecutive units of the above form. This could be implemented by working out the input patterns for the first unit, which would lead us to a CONEC phrase as follows.

```
CONEC SIMPLE 1  1 1  2 -1  3 2  4 -2  5 3  6 -3  7 1  8 -1
    9 -1  10 1  11 -3  12 3  13 -2  14 2  15 -1  16 1,
```

It could also be done by the following method.

```
LAYER 1 80,
LAYER 2 35,
LAYER 3 33 TH,
LINK 8 TO 2 FROM 1 TAPE T1 1,
LINK 8 TO 3 FROM 2 TAPE T1 2 DIRECT UNCON,
CONECI T1 1,
CONEC SIMPLE 1  1 1  2 -1  3 1  4 -1  5 1  6 -1  7 -1  8 1
    9 -1  10 1  11 -1  12 1,
CONEC COPIES 1 35 2,
CONECZ,
CONECI T1 2,
CONEC DIRECT UNCON 1 1 2 3,
CONEC COPIES 1 33 1,
CONECZ,
```

Note that layer 2 is a dummy layer which serves to accumulate the input patterns of the type used in Example 4.20.B, and then layer 3 combines these as required. TH is the required value of the threshhold for the final units.

* * * * * * * * * *

We now discuss the most general form of the CONEC phrases. This is slightly more cumbersome to use than the special forms given above, but is considerably more flexible. This consists of the phrase CONEC followed by one or more of the following sub-phrases.

U α (α may be any standard operand.) Indicates that all inputs specified after this are to go to unit α until another U subphrase is given.

Wp (p may be any 18-place operand.) Indicates that all following inputs are to have weight p, except when a weight is explicitly specified with the input as explained below.

L α β (α and β may be any standard operands.) Has the same effect as described above for "CONEC LAYER α β,"

$(t_1,t_2,\ldots,t_k)$ is used to specify inputs to the unit currently receiving inputs (as specified by the most recent U α). Each t_i may be one of the following forms (in the following α, β, γ and δ may be any standard operands):

γ	Input from unit γ
α γ	Input from unit γ in layer α
α β γ	Input from unit γ in layer α, delay β.

Any of the above may be followed by *p, p any 18-place operand, to indicate that this input should be weight p; if *p is not given, the weight specified in the most recent W subphrase will be used. NOTE: In the second and third forms the specification of input layer will override the normal input (or the input layer specified by a CONEC LAYER) for this input only. Note also that if a non-standard layer is in effect, it is <u>not</u> permissible for t_i to be "0 0 γ" to indicate an input from the standard layer.

The forms considered earlier (CONEC SIMPLE etc.) are complete phrases; that is, it is not permissible to have, e.g., SIMPLE and

COPIES following the same CONEC or to have these intermixed with the subphrases just discussed. The subphrases we have been discussing now, however, are true subphrases; that is, several of them may occur in the same CONEC phrase or may be split between different CONEC phrases as is most convenient in the particular application.

* * * * * * * * * *

Example 4.20.D

We will show another method for performing the job which was done in Example 4.20.A.

```
OPEN A *I NT,
     SET N $RAND 4,
     SET W $(B18)1,
     CONEC U I,
     OPEN B *J 5,
          IF MORE J N, SET W.$(B18)-1,
          CONEC ($RAND NI *W),
     CLOSE B,
CLOSE A,
```

Note how by separating the subphrase which specifies the terminal unit and the subphrase which specifies the inputs which it receives into two separate CONEC phrases, we are able to put the latter into a loop, which was not possible with the method used in the previous example. If the number of inputs were considerably larger than the five used here, this method would become much more convenient than the method of example 5.20.A. We now give another example which could not be done at all using the CONEC SIMPLE form.

Example 4.20.E

We would like each unit in the terminal layer to receive positive inputs chosen at random in the initial layer, with weight 1; but we would like the number of inputs to vary from one unit to another, being chosen at random between 3 and 10.

```
OPEN A *I NT,
    CONEC W 1 U I, SET J $RAND (3,10),
    (J, CONEC ($RAND NI), )
    CLOSE A
```

* * * * * * * * * *

If the input layer is the retina[1] then it may be more convenient to specify inputs in a coordinate form. This is not permissible in the special forms (CONEC SIMPLE and CONEC DIRECT), but it is in the general form; simply replace the single operand specifying the unit number by two operands enclosed in parentheses and separated by a comma. These give the first and second coordinates of the point on the retina.[2]

The following miscellaneous points should be noted about CONEC phrases.

(1) It is not permissible to make two separate tables for the same terminal unit (within a given linkage); that is, the number of a terminal unit may only appear once, either in a CONEC SIMPLE, a CONEC DIRECT, a U subphrase, or implicitly in a CONEC COPY or COPIES.

(2) If the general form of CONEC is being used to create tables for a linkage which will use the direct or unconditional direct option, "W 0" must be specified. If it is being used to create tables for a linkage not using either of these options, no input may have a weight of zero.

This ends the discussion of type 8 linkages and the CONEC phrase.

Some linkage types assume that the input layer has a two-dimensional structure, and are normally used on the retina which, of course, is usually specified in this way. However, if for some

[1] Or any other layer to which a two-dimensional structure has been given, see DIM below.

[2] If this is used when a non-standard layer is in effect, it will be assumed that this layer has the same dimentions as the standard input layer.

special reason it is desired to use these on another layer, it is necessary to specify the dimensions which will be assumed. This is done by including in the LINK phrase the subphrase

DIM α

This indicates that the horizontal dimension is to be α. The user should be careful that α is indeed a factor of the total number of units in the layer, and the program will determine for itself the appropriate vertical dimension. Note that this in no way affects the layer itself, but simply affects the way in which this particular linkage treats it. (If DIM is not used, the layer will be considered as having a vertical dimension of 1 and a horizontal dimension equal to its total number of units.)

Type 3 linkages may use one of two different internal methods for computing activities, sometimes referred to as "generation" (G) and "non-generation" (NG) methods. This is a detail which is normally of no concern to the user as the ultimate effect is the same; the only difference in the two is timing; the G method tends to be considerably more efficient if there are a large number of units in the terminal layer, whereas the NG method is more efficient when there are a very small number of units in the terminal layer. The system will normally use the G method if there are more than five units, or the NG if there are less than or equal to five. However, if the experiment is particularly time-consuming, the user may wish to go to the trouble of referring to the detailed timing notes in Appendix V, and to decide for himself which method is the more efficient. If he does this, he may add to the LINK phrase the subcode CGEN to force the linkage to use the generation method, or CGENX to force it to use the non-generation method. Similar remarks apply to reinforcement and in this case the relevant subcodes are RGEN and RGENX.

In a type 3 linkage the type of reinforcement may be specified, instead of by the subcodes LGAMMA, etc. discussed in Section 4.2, by the subcode

TYPE a b

where "a" and "b" may be any standard operands. If a = 0, b = 0, the reinforcement will be type α. If a = 1, b = 0, the reinforcement will be type γ; if a = 1, b = 0, the reinforcement will be type Γ. (The advantage of this subphrase is that "a" and/or "b" may be replaced by a counter-list, thus allowing a convenient way of performing the same experiment with different types of reinforcement if desired.)

Parenthetical paragraph: There are in the system certain special operands known as X-values. These are mainly a holdover from an earlier version of the system and most of their intended functions are now better handled by newer features; however, they are still needed in connection with the "linkage bypass" discussed in the next paragraph, and so we give here a brief discussion of their use. The form of one of these operands is $Xi where i may be a positive integer, or a simple operand whose value is a positive integer.[1] It is a changeable standard operand. A value placed in one of these operands is not destroyed at the termination of the section, but will be available to later sections.[2] The phrase

SETX n α,

will set $Xn to the value of the standard operand α. (NOTE that it is not permissible to use "SET $Xn α" as $Xn is not a simple operand.) In a LINE or HEAD phrase the subcode "Xc *a" will have the same effect as "V $Xc *a" and the subcode "TAPEX a" will have the same effect as "TAPE $X a". End of parenthetical paragraph.

If it is desired to have a linkage whose effect can be controlled at the time the perceptron is being tested (in other words, sometimes it is to be ignored but at other times it is to have its normal effect), this can be achieved by adding to the LINK phrase the subphrase

BYPAS n

[1] It is recommended that only small positive integers are used because if, e.g., "$X97," is used, the system will reserve space for $X1 to $X96 even if these are not used.

[2] This, in fact, is the major historical reason for the existence of this concept; but in this use it has been superceded by the later addition of the ALWAYS feature.

This specifies that this linkage is to be controlled by $Xn, in the following sense. Whenever $Xn has a non-zero value, this linkage will be ignored. That is, it will have no effect upon the computation of activities to its terminal layer, and if it is a type 3 linkage it will not be reinforced even though such reinforcement is called for in the normal course of events. When $Xn is zero, the linkage will function normally. Several different linkages may be controlled by the same $Xn if desired. Note $Xn will be set to zero during the generation of the perceptron if it is referred to by a BYPAS subphrase.

* * * * * * * * * *

Example 4.20.F

We wish to have a perceptron which is like a simple perceptron except that the A-units receive three positive and two negative inputs chosen at random on the retina; but some stimuli are to be treated using only the positive inputs while others are to be treated using both. We wish to present a sequence of 100 stimuli, computing the activity states and printing them out. For the purposes of the example we will assume that the subsequence "GET β," has already been defined and will get the next stimulus from the appropriate tape and put it on the retina and will set the standard operand β to 1 if this is a stimulus of the group which uses only the positive inputs and 2 if it is one of the group which uses the negative inputs also.

```
RETINA 10 10,
LAYER 2 100 1,
LINK 2 TO 2 FROM 1 POS 3,
LINK 2 TO 2 FROM 1 NEG 2 BYPAS 1,
END
(100, GET A, SUB 2 A $X1,
ACTIVE,
PACT 2,)
END
```

ACTIVE is a phrase which will be introduced in the next section whose effect is to cause the computation of activity for all layers in the perceptron after the retina, using the stimulus currently on the retina. Note that "SUB 2 A $X1" will set $X1 to zero if A equals two, which is the case where both linkages are to be used and will set $X1 non-zero in the case where we wish to bypass the second linkage.

* * * * * * * * * *

Instead of a single copy of a given layer, the linkage may use for its input several copies (at consecutive time steps in the past), considered as a single set of units. This is done by making the DELAY subphrase read thus

DELAY α * β

For example, if we have

LINK 2 TO 5 FROM 3 DELAY 2 *4 ...,

and if layer 3 has 4 units, then the input layer for this linkage will consist of 12 units, the first 4 being a copy of layer 3 two time steps back, the next 4 three time steps back, and the last 4 four time steps back.

It is necessary for the system to know the total number of layers in a perceptron at the time it has interpreted the cards for the section specifying it. Of course it would be no difficulty to determine this number when the section is executed, but when it is interpreted, the best it can do is count the number of times LAYER and RETINA phrases occur. Normally, this will give the correct value, but not if a LAYER phrase is used in a loop or in a subsequence or following a conditional phrase. In such cases the number of layers must be specified by the phrase

NLAYER n,

n must be an explicit integer and should be an upper bound on the number of layers the perceptron will contain (and preferably a

reasonably realistic upper bound). The same remarks apply to linkages, and in this case the relevant phrase is

NLINK n,

The remaining features discussed in this section are of doubtful utility to the normal user. They were incorporated in the system primarily as debugging aids for the system itself; however, we briefly mention them here for the sake of completeness.

PRTOPT α β γ,

will cause the description of the perceptron when constructed to be printed on the online printer if α is not zero. (This description is automatically printed as a part of the offline output.) If β is not equal to zero, details of the actual storage locations assigned to various parts of the perceptron mechanism will be printed out after the perceptron description; and if β and γ are both non-zero, this description will also be printed out on the online printer.

If a LINK phrase includes the subcode "SAVE n" then the location of the first word of the code which performs the activity computation will be placed into $Xn at the time the perceptron is generated. Similarly, "SAVEI n" will put into $Xn the first location of the initialization code for the activity computation for this linkage; "SAVER n" the location for the reinforcement code; "SAVCG n" the location for the code generated at computation time for activity computation; and "SAVRG n" the location of the code generated at computation time for reinforcement.[1]

The material in this section is summarized in 5.8

[1]These last remarks are not expected to be comprehensible unless one has looked at the implementation of the system.

*4.21 Perceptron Control (Exotic)

This section considers further details of the control of and communication with the perceptron during the training/testing phase. We will discuss a variety of phrases used for this purpose and also special $-operands.

We begin with the discussion of the STIM phrase, which is used to obtain a stimulus from a tape and place it on the retina. This phrase assumes that the stimuli on the tape have either been prepared by the stimulus generation features as discussed in Section 4.4 and 4.22, or have been produced in the same format by some other means.[1] Each stimulus may have associated with it the following information: (1) Class (positive and negative, or more generally, 18 bits which may be used to classify the stimulus into one of many different classes); (2) A name of up to 42 characters; and (3) A key number which may be an integer between 0 and 32,767. In general, the STIM phrase will bring in the next stimulus from the specified tape, place the class bits in the operand $CLASS, place the name in the operand $STNAME, the number of words in the name in the operand $STNAML (remember that each word holds six characters, so the number of words will be somewhere from 0 to 7), the key number in $KEY, and the stimulus itself in the retina, that is, layer one of the perceptron. It may also perform other actions, according to the specific variation as discussed below. When the class bits are placed in $CLASS, they are placed in the leftmost 18 bits. This means that the first class bit is in the sign bit of $CLASS. When stimuli are classified into two classes only, it is customary to use this first classification bit, so that $CLASS will in fact be plus for one class of stimulus and minus for the other class. (In particular, this is true when the classes are assigned to the stimuli by the SSPOS and SSNEG phrases, as

[1] For details of the format in which the stimuli are placed on the tape, see page 5.14.9. However, if one only produces the stimuli by the normal stimulus generation phrases and only reads them in by means of the phrases especially designed for that purpose (discussed in Sections 4.3 and the current section), then one does not need to know anything about these details.

discussed in Section 4.4.) However, if several classification bits are used, to indicate independent binary classifications (corresponding to an equal number of response units in the perceptron) then the convention followed is of using a "one" bit to indicate a desired positive response, and a "zero" bit to indicate a desired negative response. (In other words, the classification bits, if considered as a packed activity vector and unpacked into the response layer, would give the desired responses-- see discussion of packed activity vectors below.) Many of the other phrases discussed later in this section depend upon finding the class bits in $CLASS in the way that they are placed by STIM. The specific variations of the STIM phrase are as follows.

STIM τ,	This will perform the basic actions of bringing in the stimulus from tape τ as described above.
STIM (bx) τ,	bx should be the name of a list which has already been defined as a list with 7 word items. This phrase will perform the basic actions as above and will also place a new item on the list and put the name of the stimulus into this item. If the name has less than 42 characters, blanks will be added at the end.
STIM (*PRINT) τ,	This will perform the basic actions as previously described and will also print a message "STIMULUS PRESENTED" followed by the name of the stimulus.
STIM (*ADD) τ,	This will perform the basic actions as above and will add the name of the stimulus into the output line currently being built up by LINE phrases. Only the actual number of characters in the name itself will be added, followed by one blank.

In the last three variations, if the stimulus does not have a name, a name consisting of two asterisks will be used. If we add a $ immediately before the right parenthesis, then if the stimulus has no name, the action concerning the name will be omitted (that is to say, no new item will be added to the list, no message will be printed, or nothing will be added into the output line). We may, if we wish, put on the output tape an end of file mark after the last stimulus (see Section 4.18). If the STIM phrase finds the end of file mark when it attempts to read in a stimulus, it will set $QSFILE equal to one. Otherwise, it will set it to zero.

Having obtained the stimulus on the retina, the next thing we would like to do is compute the activity states of the other layers. The phrase

ACTIVE,

or

ACTIV,

may be used to cause the computation of the activity states of all the remaining layers. However, we may wish to compute the activities of some layers only (one reason for doing this may be that we wish to make some adjustment in the states of one of the intermediate layers by direct intervention, before using that as inputs to a later layer). To do this we may use

ACTIV α,

or

ACTIVE α,

This will cause the computation of activities for all layers which have not already been computed, up to and including layer α. To be more precise, the special operand $ACTOP has for its value the number of the highest layer for which activities have already been computed. (NOTE: If a stimulus is placed on the retina by means of the STIM phrase or one of the other phrases specifically designed for that purpose, then $ACTOP will automatically be set to one.

However, if the stimulus is placed on the retina by some non-standard method, then it is the user's responsibility to set $ACTOP equal to one.) The ACTIVE phrase will cause the computation of activities for all layers higher than the number currently specified in $ACTOP up to and including the one specified in the ACTIVE phrase; and will then adjust the value of $ACTOP to the new correct value. The phrase

REACT α,

will set the value of $ACTOP back to α unless it was already less than that.

REACT α β,

has the same effect as "REACT α, ACTIVE β,"

REACT α *,

may be used instead of "REACT α α," (this may be useful if α is a particularly complicated expression). Note that $ACTOP is a changeable, simple operand, and so its value may also be altered directly at any time if that is desirable.

If any of the linkages in the perceptron uses a delay, then it is necessary before presenting a stimulus to move the activity states back a time step, that is, move the current copy into the delay number one copy and so on. This may be done by the phrase

HIST,

or

HISTOR,

The phrase

LEVEL α β,

will cause a suitably chosen constant (possibly negative) to be added to all units in layer α, so that exactly β of them are active. If several units have exactly the same value, it may be impossible to get exactly β active by this method. If this happens, the largest possible number less than β will be used.

If we wish to print out the activity states of any given layer, we may use one of the following phrases.

PACT α,

will print the activity vector of layer α as a series of zeros and ones (ones for active units).

PACT α β,

will do the same and will put exactly β units per line (β should not exceed 119). For a large layer, a suitable choice of β may make the output considerably easier to read.

PACTO α,

will print the activity vector of layer α in octal; that is, each digit will represent three consecutive units.

PACTO α β,

will do the same, and will put exactly β units (<u>not</u> digits) per line. It is also possible to print an activity vector which has been placed into packed form, as will be discussed later. This may be done by

BINPR α eβ,

where eβ is an extended operand holding the packed vector and the first α bits are to be printed.

BINPR α eβ γ,

will do the same and will arrange them in γ per line.

If it is desired to save activity vectors from several different stimuli, this may be done by using the MOVE phrase to put them in a block defined to be of the appropriate size. However, if the perceptron is large, there may not be enough space in the machine to save very many in this fashion. Hence it may be necessary to save them in packed form. In this form each unit is represented by a single bit (one for an active unit, zero for an inactive unit) rather than by an entire word; this of course requires a great deal less space.

GETACT a eα β,

or

PAKAC a eα β,

will get the activity vector for layer a (delay β) and place it in packed form in the extended operand eα. If eα is not big enough, only the first units will be placed into it. If it is too big, the rest of it will be set to zero. "a" may be any simple operand; β may be any standard operand or may be omitted completely if zero is intended. If it is desired to take a packed activity vector and put it back into a layer in the perceptron in its normal form, this may be done by

SETACT a eα β,

which will take the activity vector found in packed form in eα, unpack it and place it in layer a (delay β).[1] It is also possible to use another form of packing which may be useful for other purposes; in this a bit is set equal to one to show the unit has a nonzero value and equal to zero to show it has a zero value. This may be done by

GETACT * a eα β,

or

PAKAC * a eα β,

No phrase is supplied for unpacking in this fashion. It is necessary to know the size of the operand required to hold an activity vector either in unpacked or packed form if one has to define a block to hold it. If the experiment is only using a single fixed value for the number of units in the particular layer, then of course one can simply use explicit numbers for these values. However, it may be that experiments are being run on several perceptrons with different numbers of units. Even if

[1] Of course it is not possible to recreate the actual values held in the layer previous to packing. All that can be guaranteed is that the sign of each unit will be correct.

not, there is always the possibility that one may decide to change the number of units at some future time. If this number appears explicitly in many places throughout the deck, then to change the number of units it is necessary to change it in all of these places and it is very easy to make an error in such a situation. To get around this, there are several special operands which allow one to obtain, while the perceptron is being tested, the actual values currently being used for certain parameters. These are

$LAYERS

an unchangeable simple operand whose value is the number of layers in the perceptron.

$NUNITS a

an unchangeable standard operand whose value is the number of units in layer a ("a" may be any simple operand).

$NPACK a

an unchangeable standard operand whose value is the number of words required to hold a packed activity vector for layer a ("a" may be any simple operand).

$RETINH

is an unchangeable standard operand whose value is the horizontal dimension of the retina.

$RETINV

is an unchangeable standard operand whose value is the vertical dimension of the retina.

$THRESH a

is a changeable standard operand whose value is the threshold for layer a ("a" may be any simple operand). The value of $THRESH will have 18 binary places.

* * * * * * * * * *

Example 4.21.A

We wish to present stimuli from tape T1, compute the activity states of layer 3 and print out a line containing the name of the stimulus and the numbers of units in layer 3 which have a strictly positive, zero, and strictly negative value.

```
BLOCK BA $NPACK 3, BLOCK BB $NPACK 3,
    (100,
    LINE BCD *STIMULUS *,
    STIM (*ADD) T1, ACTIV 3,
    GETACT 3 BA,
    GETACT * 3 BB,
    AND BA BB,
    BITS BA P,
    BITSZ BB Z,
    SUB $NUNITS 3 P+Z N,
    LINE BCD * HAS * V P BCD * POSITIVE, * V N
          BCD * NEGATIVE, AND * V Z
    BCD * ZERO UNITS.  *PRINT,)
```

Note that BA holds the normal packed activity vector, that is, a one bit denotes a unit with a plus value; BB holds a vector with a one bit for each unit which has a nonzero value. Therefore, after these are ANDed together, the result will have a one bit for each unit with a strictly positive value.

* * * * * * * * * *

In many experiments we are interested in a perceptron with a single unit in the final layer, sometimes referred to as the response unit; we are interested in comparing the sign of the response unit to the class of the stimulus. The response is sometimes classified as correct (C) if it has a nonzero response of the same sign as the class; zero (Z) if it has a zero response; and wrong (W) if it has a nonzero response of the opposite sign to the class.

RESP,

will insert into the current output line (being formed by LINE phrases) a C, Z, or W (preceded by a blank) according to the current response and the class of the current stimulus.

RESPA,

is similar but it will insert a character plus, zero, or minus, according as the response is positive, zero, or negative. This phrase pays no attention to the class of the stimulus.

The COUNT phrase may be used to keep count of the number of correct, zero, and wrong responses of a series of stimuli. Several different sets of counts may be kept independently of one another. Normally up to five different sets, numbered from one to five, may be used. If more are required, then before COUNT is ever used we must give the phrase

COUNTS n,

where n is the number of different sets which are required.

COUNT α INIT,

will initialize the counts in set α to zero.

COUNT α,

will add one to the appropriate count in set α according to the current response and class of the current stimulus.

COUNT α PRINT,

will print the counts in set α. This phrase will assume that an output line has already been started by a LINE phrase and will complete the line and print it. Thus the user may add identification he desires at the beginning of the line.

COUNT α PRINT INIT,

will have the same effect as "COUNT α PRINT" followed by "COUNT α INIT".

COUNT (a) α PRINT,

will do the same as above but the printing will be done normally if "a"

equals one, on the online printer if "a" equals two, both if "a" equals three, and not at all if "a" equals zero. COUNT may also be used to count absolute responses rather than relative responses (that is, +, 0, and -, rather than C, Z and W). This is done by

COUNT α ABS,

Of course, a particular set of counts should not be used for both relative and absolute responses. "COUNT α PRINT" will know whether the counts were accumulated by "COUNT α" or "COUNT α ABS" and will modify the printout accordingly.

* * * * * * * * * *

Example 4.21.B

Tape T1 contains a set of stimuli which have been previously prepared. These stimuli are divided into three groups. The key number associated with a stimulus will be 1, 2 or 3 to indicate which group it belongs to. Stimuli from different groups may be intermixed. The tape is divided into segments and we do not know in advance how many stimuli will be in each segment. However, the end of a segment is marked with a dummy stimulus with a key number of four. After all the stimuli is an end-of-file mark. We wish to present these stimuli to the perceptron (which has already been specified) and to keep a separate count of the responses of stimuli in each group. We wish to count both the relative and absolute responses. At the end of each segment we wish to print the accumulated counts. At the end of all the stimuli we also wish to print cumulative total counts for all stimuli on the tape.

```
COUNTS 8,  (*I 8, COUNT I INIT,)
*A STIM T1, IFN ZERO $QSFILE, GO C,
            IF EQUAL $KEY 4, GO B,
    HIST, ACTIVE,
    COUNT $KEY, COUNT $KEY+3 ABS,
    COUNT 7, COUNT 8 ABS, GO A,
*B LINE SPACE,
```

```
        (*I 3, LINE START BCD *GROUP * V I BLANK 2,
            COUNT I PRINT INIT,
            LINE START  BLANK 9, COUNT I+3 PRINT INIT,)
        GO A,
    *C LINE SPACE START BCD *TOTALS* BLANK 3, COUNT 7 PRINT,
        LINE START BLANK 9, COUNT 8 PRINT,
    END
```

Note that sets 1, 2, and 3 are used to hold counts of relative responses for the three groups, sets 4, 5, and 6 are used for the absolute responses, and 7 and 8 for the grand totals (7 relative and 8 absolute).

Typical output produced by this might be

```
GROUP 1   C 80.00, W 10.00, Z 10.00, (8 1 1)
          + 50.00, - 40.00, 0 10.00, (5 4 1)
GROUP 2   C 75.00, W 25.00, Z 0.00, (15 5 0)
          + 50.00, - 50.00, 0 0.00, (10 10 0)
GROUP 3   C 40.00, W 40.00, Z 20.00, (4 4 2)
          + 80.00, - 0.00, 0 20.00, (8 0 2)

GROUP 1   C 75.00, W 20.00, Z 5.00, (15 4 1)
          + 50.00, - 45.00, 0 5.00, (10 9 1)
GROUP 2   C 43.33, W 23.33, Z 33.33, (13 7 10)
          + 33.33, - 33.33, 0 33.33, (10 10 10)
GROUP 3   C 100.00, W 0.00, Z 0.00, (10 0 0)
          + 100.00, - 0.00, 0 0.00, (10 0 0)

TOTALS    C 65.00, W 21.00, Z 14.00, (65 21 14)
          + 53.00, - 33.00, 0 14.00, (53 33 14)
```

Note that both percentages and actual numbers are given.

* * * * * * * * * *

If counts are accumulated by the COUNT phrase as above, then the totals are not directly accessible to the user, that is, all that can be done with them is print them as already described. We may, however, in any of the above forms of the COUNT phrase replace the

set number by an extended operand of three (or more) words. In this case, the first three words of this operand will be used to hold the counts (first word for C or +, second word for W or -, third word for Z or 0). The totals are then of course available to be used in any fashion desired.

* * * * * * * * * *

Example 4.21.C

We wish to present 100 stimuli from tape T1, classify the responses and print the numbers of correct and wrong responses, where each zero response is to be counted as one half correct response and one half wrong response.

```
BLOCK B 3,
COUNT B INIT,
(100, PRESNT T1, COUNT B,)
FLOAT B, DIVF B(3) $(F)2, ADDF B(1) B(3),
     ADDF B(2) B(3),
LINE START BCD* CORRECT * VF B(1) *1 BCD * WRONG *
     VF B(2) *1,
```

* * * * * * * * * *

If we are dealing with a perceptron with several respons' units and stimuli are classified using several class bits to indicate desired responses on the various units, we may wish to classify them as correct, zero, or wrong based on one particular unit. There is a variation of the RESP phrase which works as follows.

```
RESP α,
```

This will work like RESP, but the α response unit will be used, and the desired response will be taken from the α bit in $CLASS (counting the leftmost as one).[1] We may also use

```
RESP α eβ,
```

[1]Note the remarks on page 4.21.1 concerning the use of class bits to represent desired responses.

where eβ may be an extended operand; the correct response will be indicated by the sign of the α word in eβ.

RESPA α,

will classify the α response unit as plus, minus, or zero.

If it is desired to form cumulative counts for such cases, this must be done by special programming, as no code analogous to COUNT is provided.

In some of the following, it is useful to be able to refer to a particular linkage in the perceptron. For this purpose, linkages are numbered as follows: i) First they are placed in order according to their terminal layer, with those having the lowest numbered terminal layer coming first; ii) If there are two or more with the same terminal layer, these are placed in order according to their initial layer; iii) If there are two or more with the same terminal and initial layers, these are placed in the order in which they were specified (that is, the order in which the relevant LINK phrases were executed); iv) In this order they are numbered consecutively starting with one. If the structure of the perceptron is quite simple, one can easily see what the number of a particular linkage is and refer to it by using this number explicitly. However, if the structure of the perceptron is more complicated or is being frequently changed, or if we are writihg a testing procedure to be used with different perceptrons, this may not be so feasible. Therefore, the following $-operands are provided to assist us in this matter.

$NLINK a (i,j)

where a is any simple operand and i and j are any I-operands, is an unchangeable standard operand whose value is the number of the a^{th} linkage to layer i from layer j, or zero if there is no such linkage.

$NLINK a (i)

where "a" is any simple operand and i is any I-operand, is an unchangeable standard operand whose value is the number of the a^{th}

linkage to layer i, or zero if there is no such linkage.

$LINK a TYPE

is an unchangeable standard operand whose value is the type of linkage number "a".

$LINK a INIT

or

$LINK a FROM

is an unchangeable standard operand whose value is the number of the initial layer for linkage "a".

$LINK a TERM

or

$LINK a TO

is an unchangeable standard operand whose value is the number of the terminal layer for linkage "a".

$LINK a DELAY

is an unchangeable standard operand whose value is the delay for linkage number "a".

The weights for a particular type 3 linkage may be printed by the phrase

PW LINK α,

The weights for the type 3 linkages to layer α may be printed by

PW TO α,

If α is zero, this will cause printing of all type 3 linkages to all layers.[1]

If it is desired to print selected weights, or to get access to the value of the weights for other purposes, one of the following two methods may be used. The operand

$WEIGHT a (i,j),

[1]The phrase PW has not been completely tested, and may not be working correctly. See next example.

where "a" is any simple operand and i and j are any I-operands, is an unchangeable standard operand whose value is the weight from unit i in the initial layer to unit j in the terminal layer, for linkage number "a" (this value is given with 18 binary places). This is useful for obtaining a small number of weights but it is very wasteful of time to obtain all the weights for a given linkage using this operand. To do this, the following phrase is more convenient.

GETWGT γ eβ α β,

γ, α and β may be any standard operands, eβ may be any changeable extended operand. Weights for linkage number γ will be put into eβ, expressed with 18 binary places. The values of α and β will determine which weights are put into eβ, as follows:

(1) If α and β are both zero (or omitted), then all weights for the linkage will be put in eβ, in the order W_{11} $W_{21} \cdots W_{k1}$ $W_{12} \cdots$ (where W_{ij} is the weight from unit i to unit j).

(2) If α equals zero but β is not equal to zero, all the weights to unit β will be placed in eβ, in the order $W_{1\beta}$ $W_{2\beta} \cdots W_{k\beta}$.

(3) If α is not zero but β is zero (or omitted), all weights from unit α will be placed in eβ in the order $W_{\alpha 1}$ $W_{\alpha 2} \cdots$ $W_{\alpha m}$.

(4) If both α and β are nonzero, the weight from unit α to unit β will be placed in eβ.

Note: eβ must be big enough to hold all the weights which will be placed into it.

If it is desired to change the weights in a particular linkage, this may be done by

SETWGT γ eβ α β,

This is the exact reverse of GETWGT. In other words, the weights in linkage γ, specified as above by α and β, will be set to the values found in $e\beta$.

NOTE: If the linkage has a type γ-reinforcement or type Γ-reinforcement, then the values being put into it by SETWGT should meet the appropriate restrictions.

* * * * * * * * * *

Example 4.21.D

Suppose that the phrase PW does not work correctly, and we wish to define a subsequence of that name to fit the specifications given above. First we shall define an auxiliary subsequence, PWA, which will print the weights for a single linkage, and then will define PW as required.

```
DEFINE PWA,
     SET $2 $LINK $1 TERM, SET $3 $LINK $1 INIT,
     MPY $NUNITS $2 $NUNITS $3 $4,
     BLOCK $B1 $4,
     GETWGT $1 $B1,
     LINE PAGE START INDENT 7
          BCD *WEIGHTS FOR LINKAGE * V $1 PRINT SPACE,
     ZERO $5,
     (*$6 $NUNITS $2, LINE BCD *TO* F4 V $6 BLANK 1,
     (*$7  $NUNITS $3, SET $5 $5+1,
            LINE F 11 VB 18 $B1 ($5)*4,)
          LINE PRINT,)
     LLCLR $B1, END
DEFINE PW (N S),
     IF EQUAL $1 $(HS)LINK,  (,PWA $2, FINSUB,)
     SET $3 $2,
     IF ZERO $2, (,SET $2 2, SET $3 $NLAYER,)
     (*$4 $3 $2,
          (*$5 1000, SET $6 $NLINK $5 ($4),
               IF ZERO $6, JUMP 3,
```

```
            IF EQUAL $LINK $6 TYPE 3, PWA $6,)
        NOP,)
    END
```

Note that the use of DEFINE PW is perfectly legitimate even though PW is already a primary code (or a previously defined subsequence). A warning message will be printed to call our attention to this in case we did it unintentionally; then the previous meaning will be lost and the name will henceforth be the name of the new subsequence.

Example 4.21.E

We have a simple perceptron and would like to test it as follows. We have 100 stimuli on tape T1 and we wish to present these, using error correction reinforcement, as many times as necessary to obtain perfect response. We wish to print out the obtained responses to the 100 stimuli on the last round before perfection is attained. In the following illustration we do this by saving the status of the weights at the beginning of each round, and when we have attained perfection, using these to set the perceptron back to the state at which it was at the beginning of the round.

```
BLOCK B $NUNITS 2, BLOCK BC 3,
REW T1,
*A GETWGT 1 B, EC 100 T1, REW T1,
    COUNT BC INIT,
    (100 STIM T1, ACTIVE, COUNT BC,)
    REW T1, IFN EQUAL BC(1) 100, GO A,
SETWGT 1 B,
LINE START,
(100, STIM T1, ACTIVE, RESP,)
LINE PRINT,  END,
```

* * * * * * * * * *

The weights for a particular linkage may all be set to zero by the phrase

CLEAR LINK α,

The weights for all type 3 linkages to layer α may be set to zero by

CLEAR WEIGHT α,

The weights for all type 3 linkages in the perceptron may be set to zero by

CLEAR WEIGHT,

The actual values of units in the perceptron may be obtained (and if necessary changed) by means of the following $-operands:

$UNIT a (i)

is a changeable standard operand whose value is the i^{th} unit in layer "a" (with 18 binary places).

$UNIT a * j (i)

is the same but the value is not the current value but the value j time steps in the past. Of course a value of j should not be used larger than the largest delay specified for this layer (that is to say, specified in LINK phrases using this layer as initial layer).

$UNIT a

is a changeable extended operand representing all units in layer "a".

$UNIT a * j

is a changeable extended operand representing all units in layer "a", "j" time steps in the past.

INTAPE τ,

where τ is a simple tape designator, will specify that this tape is to be considered the standard tape for input of stimuli. If this has been given, then in any of the phrases in this section where a tape designator is specified, it may be omitted if the standard tape is intended.

Individual control over the reinforcement of particular linkages is available by means of the REINF phrase. Before presenting the specific variations of the REINF phrase we give a brief general description of the reinforcement process. There are two factors governing the reinforcement process; the reinforcement type, which is specified in the LINK phrase and is a permanent part of the structure of the perceptron, and the reinforcement procedure, which is specified at the time the reinforcement is performed and which may be allowed to change from one time to another. Essentially, a reinforcement procedure consists of a method of selecting certain units in the terminal layer of the linkage, and an increment (which may be positive or negative) corresponding to each. (This increment may be the same for all terminal units or may vary from one to another, according to the type of procedure.) Each weight coming to one of the selected units in the terminal layer from an active unit in the initial layer will have the appropriate increment added to it. Weights coming from inactive units in the initial layer will have no increment added to them. After these increments are added, any adjustments necessitated by the reinforcement type (as discussed in Section 4.2) will be made. (Some of these adjustments may of course also affect weights which come from inactive initial units.) If the reinforcement procedure is classified as quantized, then the increment will be a fixed constant and will in fact be the increment specified in the LINK phrase when the perceptron was defined. If the procedure is classified as nonquantized, then the increment will be selected separately for each terminal unit, and will be computed as follows. First e is chosen so that if the reinforcement procedure were carried through using increment e and then the input to the terminal unit were computed using the same activity states in the initial layer, the value of the unit would be zero (or in the case of continuous units, the desired value of the unit). Having determined

e in this fashion, its magnitude is then increased by the value specified as increment in the LINK phrase.[1]

In discussing the various kinds of reinforcement procedure, we will use the following terminology. We will say that a terminal unit is reinforced if it is one of the ones selected to have its input weights adjusted. In the case of quantized reinforcement, which will be the normal case, we will say that it is reinforced positively or negatively according as the increment is positive or negative.

We now consider various types of reinforcement procedure available and the specific forms of the REINF phrase which perform them (in the following discussions reinforcement procedures are quantized unless otherwise stated).

REINF LINK α SCON eβ,

This performs sign-controlled reinforcement of linkage number α (which should, of course, be a type 3 linkage). Every terminal unit is reinforced, the sign of the reinforcement being given by the sign of the corresponding word in eβ, which may be any extended operand. No consideration is paid to the actual activity states in the terminal layer.

REINF LINK α ECON eβ,

Error correction reinforcement. As in the preceding case, eβ is an extended operand, each word of which gives the sign of the desired response for the corresponding unit in the terminal layer. In this case, however, units with a nonzero value of the correct sign are not reinforced.

REINF LINK α ECON eβ NOQUAN,

[1] In the case of continuous responses, one would expect the increment specified in a LINK phrase would be zero so that the reinforcement tends to drive the final response to the desired value; while in the normal case one would expect a positive value which would provide an "overshoot", thus forcing a nonzero response of the correct sign. Nonquantized reinforcement should not be used on a linkage with a non-zero decay, or with type Γ-reinforcement. It is also not wise to use it on more than one linkage to the same layer.

The same as the preceding, but nonquantized reinforcement will be performed. This should not be used if linkage number α has a nonzero decay or was specified as having Γ-reinforcement.

REINF LINK α RCON,

In this case all the terminal units are reinforced, units with a plus value receiving positive reinforcement and units with a minus value receiving negative reinforcement. No "desired responses" are specified.

REINF LINK α SPOS eβ,

Those units corresponding to a plus word in eβ are given positive reinforcement; those corresponding to a minus word are not reinforced. No consideration is given to the actual values of the terminal units.

REINF LINK α RPOS,

Terminal units with a plus value will receive positive reinforcement and the others will not be reinforced. No "desired responses" are specified.

REINF LINK α CONTIN eβ p,

eβ is an extended operand containing the desired values for the terminal units, expressed with 18 binary places; p is an 18-place operand specifying a tolerance level. A terminal unit will be reinforced if its value differs from the desired value by more than the tolerance level, and in that case, the sign of the reinforcement will be such as to change the value in the direction of the desired value. If it is desired to have different tolerance levels for different units, p may be an extended operand, in which case each word of p contains the tolerance level for the corresponding unit.

REINF LINK α CONTIN eβ p NOQUAN,

Same as the preceding but reinforcement will be nonquantized. This should not be used if linkage number α has a nonzero decay or was specified as having Γ-reinforcement.

REINF LINK α SPEC v w (,..., REINFZ,),

This phrase may be used to perform any reinforcement procedures which have not already been covered. v and w should be variable names chosen by the user; ... may be any sequence of phrases desired, except it should not contain any conditional phrases or GO phrases which would send control outside the parentheses. During the reinforcement process, these phrases will be executed once for each terminal unit; their purpose is to decide whether this unit is to be reinforced, and if so, what increment to use. Each time the execution of these phrases commences, v will contain the number of the terminal unit currently being processed and w will contain its value. By the time REINFZ is executed, the phrases should have set v to 1 if no reinforcement is desired for this unit, 2 if quantized reinforcement is desired, and 3 if nonquantized reinforcement is desired. If quantized reinforcement is desired, w should have been set to the desired increment, with 18 binary places, or may be set to 0 if the standard increment (that is, the one specified in the LINK phrase) is to be used, or to -0 if minus the standard increment is to be used. If nonquantized reinforcement is specified, w should be set to the desired change in the value of the terminal unit, again with 18 binary places.

IMPORTANT NOTE: Reinforcements performed by use of this variation will be extremely time-consuming compared to those performed by the other forms. Note also that the entire expression "REINF LINK α SPEC v w (,...,REINFZ)," is considered as one phrase if it is being skipped over by a conditional phrase.

* * * * * * * * * *

Example 4.21.F

We have a perceptron in which layers 3 and 4 have the same number of units and each receives a type 3 linkage from layer 2 (these are the only type 3 linkages in the perceptron). We wish to present a series of 100 stimuli from tape T1, and compute the activities; we wish to determine which units in layer 3 have the same sign as the corresponding

units in layer 4, and to reinforce these units according to their sign, but reinforce the other units opposite to their sign; and the same for layer 4.

```
SET N $NUNITS 3, SET M $NPACK 3,
BLOCK BA M, BLOCK BB M, BLOCK BC M,
BLOCK B N,
(100, STIM T1, ACTIVE,
     GETACT 3 BA, GETACT 4 BB,
     EOR BA BB BC,
     EOR BA BC, EOR BB BC,
     UNPACK B BA, REINF LINK 1 SCON B,
     UNPACK B BB, REINF LINK 2 SCON B,)
```

This way of doing it is illustrative but a moment's thought shows us that it is extremely inefficient. What we are asked to do boils down simply to reinforcing layer 3 according to the actual response in layer 4, and vice versa. Hence,

```
(100, STIM T1, ACTIVE,
     REINF LINK 1 SCON $UNIT 4,
     REINF LINK 2 SCON $UNIT 3,)
```

Example 4.21.G

In the above situation we would like not to reinforce at all those units where the signs disagree.

```
(100, STIM T1, ACTIVE,
     (*I 2, SUB 5 I J,
        REINF LINK I SPEC N X (,
        EOR X $UNIT J(N) Y,
           SET N 1,
           IF PLUS N, (,SET N 2, ASHIFT 36 X,)
           REINFZ,)
     )
)
```

For some simpler examples, see Appendix II.

* * * * * * * * * *

REINF LAYER α...,

The remainder of the phrase may take any of the forms given above. This will perform reinforcement as described above for all type 3 linkages to layer α.

REINF UPTO α...,

This variation will perform reinforcement for all type 3 linkages to all layers up to and including layer α.

$REINFN

is an unchangeable simple operand. Whenever reinforcement is performed by any form of REINF, $REINFN will be set to the number of terminal units for which reinforcement was performed. (If REINF LAYER, or REINF UPTO is used and causes reinforcement of more than one linkage, $REINFN will be set to the number of units reinforced in the last linkage processed.)

* * * * * * * * * *

Example 4.21.H

Let us suppose we have a perceptron with 50 units in the retina and 10 response units. Furthermore, suppose that B is a block with dimensions 20 by 60, and suppose that each one-dimensional slice, B(1), B(2), etc., holds in its first 50 words a stimulus in unpacked form and in its last 10 words desired responses for this stimulus. We are interested in continuous response units so that the desired responses are values, not simply signs. We wish to present these stimuli using nonquantized continuous reinforcement, repeating the presentation until we find all responses within 0.2 of the desired value.

```
BLOCK BR 10,
*AGAIN ZERO C,
(*I 20, MOVE B(I) $UNIT 1, SET $ACTOP 1,
     ACTIVE, SHIFT -50 B(I) BR,
     REINF LINK 1 CONTIN BR 0.2 NOQUAN,
```

```
ADD C $REINFN,)
IFN ZERO C, GO AGAIN,
```

* * * * * * * * * *

The material covered in this section is summarized in 5.9.

*4.22 Stimulus Generation (Exotic)

In this section we will discuss in more detail the stimulus generation features, of which some aspects were discussed in Section 4.4. We begin first with a general discussion of how these features work. When the SSINIT phrase is executed to prepare for the production of stimuli, the following blocks are defined. (In the following we will use H to represent the horizontal dimension specified in the SSINIT phrase and V to represent the vertical dimension.)

$BSRETA is a block of dimensions H by V.
$BSRETB is the same.
$BSTIM is a one-dimensional block of the appropriate size to hold one packed stimulus.

When a stimulus is said to be "placed on the retina," it is either placed in packed form in $BSTIM (each bit represents one point of the stimulus and is one if that point is active; the points being taken from left to right, top to bottom); or else is placed in unpacked form in $BSRETA ($BSRETA (v, h) is plus if the point with coordinates (h,v) is active; otherwise is minus). The special operand $SSPAKQ will be zero if the stimulus is in unpacked form and non-zero if the stimulus is in packed form.

Normally one does not make explicit reference to these blocks but they are referred to implicitly by the special phrases. However, if the special phrases are not adequate for what one wants to do, one may at any time refer explicitly to these and do what one wishes with them. (Note that $BSRETA, $BSRETB, and $BSTIM are not operands but are block names.) Some of the stimulus generation phrases, e.g., SSTRAN, SSROT, use $BSRETB as follows. Keeping the original stimulus in $BSRETA a transformed copy is made in $BSRETB, and when complete this is moved to $BSRETA.

The following phrases may be of use if one is processing stimuli by nonstandard methods.

SSPACK,

will make sure that the stimulus on the retina is in packed form, that is to say, in $BSTIM.

SSUNPK,

will make sure that the stimulus is in unpacked form, that is, in $BSRETA. There is no harm done if SSPACK is given when the stimulus is already packed, or if SSUNPK is given when the stimulus is already unpacked.

SSWAP,

will interchange the retina and the alternate retina ($BSRETA and $BSRETB). Note that if the stimulus is in packed form, it will remain that way and will not wind up in $BSRETB. (Normally SSWAP is used to move a stimulus from $BSRETB to $BSRETA or vice versa, and not actually to interchange two different stimuli; but even so one should use this in preference to "MOVE $BSRETB $BSRETA," as SSWAP uses a special method to achieve its result considerably faster than MOVE could do.)

The following $-operands are also set up by the SSINIT phrase.

$SSHDIM

will have for its value the horizontal dimension of the retina.

$SSVDIM

will have for its value the vertical dimension of the retina.

$SSHCEN

will have for its value the horizontal coordinate of the center point of the retina (18 binary places).

$SSVCEN

will have for its value the vertical coordinate of the center point (18 binary places). (Note that the top left corner has coordinates (1,1) .)

The center coordinates are set to (0,0) by "SSINIT h v,"; however, we may use "SSINIT h v p q," where p and q may be any 18-place operands, to set the center coordinates to (p,q), (and the center coordinates may also be changed by "SSCENT p q," as mentioned in 4.4).

$BSTIML

is defined as a list of one-dimensional items. The size of these items is the size of a packed stimulus plus 8 more words (one for class information and seven for name). This list is used for internal saving of stimuli (SSDEFS).

Each stimulus has associated with it the following additional information (see page 4.21.1): (1) Class, which consists of 18 bits used for classifying stimuli; (2) Key number, which may be an integer from 0 to 32,767; and (3) Name, up to 42 characters (7 words). The class bits are kept in the first 18 bits of $CLASS (this means that the first of the class bits is in the sign bit of $CLASS), the key number is in $KEY and the name is in $STNAME; this is so whether the stimulus currently on the retina is in packed or unpacked form. $CLASS and $KEY are changeable simple operands and $STNAME is a changeable standard operand.[1] When a stimulus is written on the output tape by means of SSOUT, this information is put out with it in the format described later. When a stimulus is saved by means of SSDEFS, the class bits and name are saved with it but not the key number. When a stimulus which was previously saved is placed on the retina by a SSN or SSR phrase, the class and name which were saved with it will be placed back in these operands. When a new stimulus is created by means of a SSHBAR, SSVBAR, SSZ, SSRAND, SSRW, or SSPT phrase, the class name

[1]Note that these $-operands are also used by the STIM phrase discussed in the previous section. Normally the same section of a program will not be both creating stimuli and presenting them to the perceptron, so there is no conflict involved. If one is doing so, one must be very careful to watch exactly what is happening to these operands.

and key already set up in these operands are unchanged. If a stimulus is placed on the retina by means of SST or SSC, the class and name that were associated with it on the tape or cards respectively, will be placed into these operands. In all of these cases, $KEY will remain unchanged. The class, key number and name may be changed at any time by phrases intended for that purpose, which will be discussed later, or by direct use of $CLASS, $KEY, $STNAME.

When a stimulus is saved by means of "SSDEFS n," it will be done as follows. (1) If there are less than n items on the list $BSTIML, new items will be added until there are that many. (2) If the stimulus is not in packed form, it will be put into packed form. (3) The n^{th} item on the list will have placed into it the class (one word), the name (seven words), and the packed stimulus.

The format of a stimulus on tape is as follows. First word has the class bits in the left 18 bits, the number of words in the name in the next three bits,[1] and the key number in the last 15 bits. The packed stimulus begins in the second word and is followed by the specified number of words of the name. Note that the dimensions of the stimulus are not included, so that the program using the tape must know what dimensions were used when the stimuli were created.

We have discussed in Section 4.4 all the phrases used for getting a stimulus on the retina except SSRW and some forms of SSPT. Since there is some doubt as to whether SSRW will work correctly, and since in any case the description in Section 5.14 (pages 5.14.3 to 5.14.16) is fairly clear, we will not discuss this further here.

The phrase SSPT may be used to activate individual points in the stimulus.

SSPT a_1 b_1 a_2 b_2 ...,

[1] This will always be seven for stimuli generated by the stimulus generation phrases; however, the tape format allows any number from zero to seven if stimuli are created in some other way.

Each pair of numbers (a_i b_i) is considered as the coordinates of a point in the stimulus and this point is made active if it is not already. Note that the top left corner is (1,1).

SSPT * a_1 a_2 ...,

In this form each number represents a point, the points are numbered starting with 1 in the top left corner and working from left to right, top to bottom.

It is also permissible, in either form of the SSPT phrase, to have a list of extended operands, in which case the values in the first followed by the values in the second, etc., are taken as one series of numbers and then treated as described above.

SSPTB α β,

will cause points affected by future SSPT phrases to be translated by (α, β); in other words, if such a phrase would normally refer to a point with coordinates (x,y) it will actually refer to the point with coordinates (x+α, y+β).

SSPTB α,

will cause future uses of SSPT * to be translated by α; that is, a reference to point n will be made to refer to point n+α.[1]

The class of a stimulus may at any time be changed by one of the following phrases.

[1]If one is intermixing uses of SSPT and SSPT * then the following more detailed explanation of the workings of SSPTB should be carefully noted; otherwise, the rest of this foot note may be safely ignored. In the first form of SSPT the coordinates (x,y) are turned into a single number, x+(y-1)H; and this is used to identify the point exactly as in the second form. The effect of SSPTB is to set up a base value (which normally would be zero) which is added to the point number before it is used. In the case of "SSPTB α β," the appropriate value of this base number will be computed from α and β similarly to above. Thus it is not the case that "SSPTB α β," would not affect future uses of SSPT * or that"SSPTB α," would not affect future uses of SSPT; the two forms are provided only so that we may use the more convenient one in a particular instance. The value of the base number at any time is kept in the $-operand $SSBASE, which is a changeable simple operand.

SSPOS,

will set the class positive, that is, will set the first class bit to 0 (sign of $CLASS to plus).

SSNEG,

will set the class negative, that is, will set the first class bit to 1 (sign of $CLASS to minus).

SSCLAS α,

will set all 18 class bits to the last 18 bits of the value of the standard operand α.

The key number may be altered as follows. As we have already mentioned, $KEY holds the current key number and $SSKEYP holds a number we shall refer to at the moment as a "permanent key number." The phrase

SSKEY α,

will change the key number to α. When the phrase SSOUT is executed, the current key number will be put out with the stimulus; and then the key number will be set to the same value as the permanent key number. The permanent key number is set to zero by SSINIT; it may be changed at any time by

SSKEYP α,

The effect of this is that the permanent key number will in general go out with all stimuli but may be overridden for a single stimulus by the use of SSKEY.

The name of the current stimulus may be set by the following phrases.

SSNAME...,

The primary code may be followed by one or more of the following types of pieces of name: (1) A series of any characters other than slash (/), comma (,), apostrophe ('), or blank. These characters will be inserted into the name exactly as is. (2) / followed by a standard

operand; the significant digits (or the last six digits if the value is greater than 10^6) will be added into the name. (3) /* followed by an extended operand; the contents of the operand will be considered as coded alphabetic information and inserted into the name.

A single SSNAME phrase may contain up to a maximum of 15 such pieces.

SSNAME *...,

This is similar to the preceding but instead of throwing away whatever name was already there, the name specified in this phrase, as described above, will be added on to the end of the name already in $STNAME.

SSXNAM...,

Following the primary code may be the same type of pieces as described above for SSNAME. If "xxx" represents the name generated by these pieces and "sss" represents the name already attached to the stimulus, then the name will be changed to "xxx(sss)". Note the stimulus name may be a maximum of 42 characters long. If any of the phrases just described attempts to make the name longer than 42 characters, the 42nd character will be made into a $, and any further additions will be ignored.

* * * * * * * * * *

Example 4.22.A

We wish to generate on a 20 by 20 retina a series of stimuli as follows. Group 1 stimuli will consist of horizontal bars of width 3, 4, or 5, translated to all possible positions on the retina; group 2 stimuli will consist of vertical bars likewise; group 3 stimuli will consist of 3 by 3, 4 by 4, or 5 by 5 squares in all possible positions on the retina. We wish to make up the sequence as follows. We choose a group at random and pick the first stimulus at random from that group. After this we would like to repeat the same group with a 90% probability or switch to one of the other two

groups with a 5% probability each. We wish the key number associated with each stimulus to indicate its group; we also wish to have a 1% probability of following any particular stimulus by a zero stimulus with a key number of 4; but we do not wish this to interfere with the subsequent choices of stimuli as described above. We would like to attach to each stimulus a suitably descriptive name.

```
SSINIT 20 20,
SSOTAP T1,
SET G $RAND 3, SSKEYP G,
OPEN A 1000,
     SET W $RAND (3,5), SET HT $RAND 20, SET VT HT,
     IFN EQUAL G 1, GO AA,
        SSHBAR W, SSNAME HBAR/W, GO AC,
*AA IFN EQUAL G 2, GO AB,
     SSVBAR W, SSNAME VBAR/W, GO AC,
*AB SSHBAR W, SSTRAN (NT) W-20, SSNAME SQUARE /W,
                SET VT $RAND 20,
*AC SSTRAN HT VT, SSXNAM H /HT V /VT, SSOUT,
IF EQUAL $RAND 100 1,  (,SSZ, SSNAME ZERO, SSKEY 4, SSOUT,)
IF MORE $RAND 100 85, (,SET G $RAND 3, SSKEYP G,)
CLOSE A,
END
```

Names of typical stimuli produced by this might be

```
H5V5(HBAR4)
H19V19(VBAR3)
H2V2(VBAR5)
ZERO
H12V12(VBAR5)
H12V17(SQUARE3)
etc.
```

* * * * * * * * * *

Stimuli on the retina may be transformed by one or more of the following phrases.

SSTRAN α β,

will translate the stimulus α places horizontally and β places vertically (positive directions are right and down). A point shifted off the edge of the retina reappears at the opposite edge. (This is used in some experiments to avoid certain problems which would be caused if the stimulus changed its area as part of it disappeared over the edge.) If this effect is not desired, we may use

SSTRAN (HV) α β,

We may also use

SSTRAN (H) α β,

or

SSTRAN (V) α β,

to eliminate the "wrap around" effect in the horizontal or vertical direction respectively. SSROT and SSDIL were discussed in Section 5.4.

It is also possible to perform any arbitrary linear transformation. This is done by the phrase

SSLIN α_1 β_1 γ_1 α_2 β_2 γ_2,

The transformation performed will be the one given by

$$x' = \alpha_1 * x + \beta_1 * y + \gamma_1$$

$$y' = \alpha_2 * x + \beta_2 * y + \gamma_2$$

Note that the coordinate system used is taken to have its origin at the center point specified by the most recent SSINIT or SSCENT phrase.[1]

The stimulus on the retina may be printed out by the phrase

SSPRT,

[1] SSROT, SSDIL, and SSLIN have not been completely tested but are believed to be working correctly.

If desired we may use

SSPRT ONLINE,

to print it online; or

SSPRT BOTH,

to print it both online and offline; or

SSPRT * a,

to print it according to the value of "a" (any simple operand) as follows

a = 1 Print offline
a = 2 Print online
a = 3 Print both
a = 0 Print neither

When a stimulus is printed a heading line will be given containing the class, the key number and the name; it will be followed by the stimulus itself, with plusses for active points and no mark for inactive points, and the entire stimulus surrounded by a border of zeroes to indicate its extent.

The phrase

SSPRTH,

may be used to print the heading line only but not the stimulus itself (this is also subject to the variations given above for SSPRT.)

A packed stimulus which is not in the retina may be printed by using

SSPRT ($e\alpha$ β γ),

where the dimensions of the stimulus are β and γ and the stimulus is found in the extended operand $e\alpha$. (The name, class, and key are assumed to be in $STNAME, $CLASS, and $KEY as usual.)[1] This form may be followed by ONLINE, BOTH or * a as discussed for the previous forms.

[1]This form of SSPRT may be used even if the stimulus generation routines have not been initialized by use of SSINIT.

Stimuli may be written out on tape by the phrase SSOUT as discussed in Section 4.4. Actually this is only one of the possible functions of SSOUT, although it is the normal one. There are several "output modes" controlled by the phrase SSMODE; any arbitrary combination of these modes may be active at a given time and the action taken by the SSOUT phrase depends upon which modes are active. The various modes together with the subcode by which they are identified in the SSMODE phrase are as follows.[1]

OUT	Write the stimulus out on the output tape. (This is specified by SSOTAP; see below.)
OFF	Print the stimulus (offline).
ON	Print the stimulus (online).
OFFH	Print the stimulus heading only (offline).
ONH	Print the stimulus heading only (online).
DO	Move the stimulus onto the perceptron retina.[2]

The code

```
SSMODE
```

may be followed by one or more of the subcodes listed above and will make the specified modes active and the others inactive. The code

```
SSMON
```

may be followed by any of the subcodes and will make the specified modes active but will also leave any which are already active in that state.

[1]There is also a seventh mode PUNCH which is not described here as it is virtually useless; for the sake of completeness it is described in the appropriate place in Section 5.14.

[2]This, of course, can only be used if the stimulus generation features are actually being used in a section which is testing a previously defined perceptron.

SSMOFF

is similar but it will make the specified modes inactive and leave the others as they are. Note that SSINIT will make the OUT mode active and all others inactive.

If the stimuli are to be written on tape, before the SSOUT is used an output tape must be specified by

SSOTAP σ

where σ may be any standard tape designator.

$SSTAPE

is a changeable single operand holding the output tape number.

Material in this section is summarized in 5.14.

*4.23.1 Miscellaneous

We shall first discuss some $-operands which have not been previously mentioned.

$(OE)

followed by an octal number of any number of digits is an unchangeable extended operand consisting of a sufficient number of words to hold the octal number (12 digits per word). If the number of digits is not an exact multiple of 12, a sufficient number of zeroes will be added at the right end to fill out the last word.

$RETINA

is an unchangeable simple operand containing the dimensions of the retina (vertical in high order 18 bits, horizontal in low order 18 bits).

$PERROR

is normally zero but will be set nonzero if the perceptron generator detects errors in parameters specified for a perceptron.

$SUBN bv

where bv is the name of a block or list, is an unchangeable standard operand whose value is the number of subscripts on that block or list.

$SUB bv(i)

is an unchangeable standard operand whose value is the maximum value of the i^{th} subscript on block or list bv.

$CKEYS

is an unchangeable standard operand whose value is the current setting of the console keys[1] considered as a 36-bit binary number.

$DCKEYS

is similar but the keys are interpreted as a 6-digit decimal number, each group of 6 keys representing one digit.

[1] See Appendix IV.

$OCKEYS

is an unchangeable simple operand giving the console keys as they were set when the system was first loaded (binary).

$OPSIZ eα

where eα may be any operand at all (other than $OPSIZ or $OPLOC), is an unchangeable standard operand whose value is the number of words in the operand eα.

$DATE

is a simple unchangeable operand whose value is the current date (6 binary coded digits).

$CASE

is a simple operand whose value is the number of the current case.

$TIME

is a standard operand whose value is the current reading of the timeclock (an integer giving hundredths of a minute).

$HOUR

is a standard operand whose value is the current time of day expressed as 100 times hours plus minutes.

$RANCID

is a changeable simple operand whose value is the current seed in the random number generator.

$RAND (i,j)

is a standard operand whose value is an integer chosen at random between i and j inclusive (i and j may be any I-operands).

$RANDF a

is a standard operand whose value is a floating point number chosen at random between zero and the value of "a".

$RANDFT a

is similar. The difference between the two is that in the first form "a" is an integer or a variable whose value will be considered an integer, whereas in the second form "a" may be any decimal number or a variable whose value will be considered a floating point number.

$CORE a

is a changeable standard operand which represents the word in memory whose address is the low 15 bits of "a".

$CORE a (i)

is a changeable extended operand of i consecutive words starting at location "a". These operands permit direct access to any information in the machine. There are also several operands which allow one to determine the actual location of certain parts of the program and/or the generated perceptron, presumably for use in conjunction with $CORE. These will not be discussed here as their use requires some knowledge of the internal workings of the system. They are described briefly in Section 5.2 along with all other $-operands.

There are also a few phrases which have not been discussed in the preceding sections. Several of these are conveniently gathered together in section 5.16 and will not be discussed here. The others are as follows.

The settings of the sense switches may be tested by the following phrases.

IF SSW α,

or

IF SWITCH α,

will skip the next phrase unless sense switch α is down.

IF NSSW α,

will skip the next phrase unless sense switch α has been reversed from

the position it was in when the system was loaded. As usual, "IFN SSW α," and "IFN NSSW α," have the reverse meanings.

ONLY α β,

This will cuase the next phrase to be skipped except on the α, 2α, 3α, ...βα, times the ONLY phrase is executed. Note that this means the ONLY phrase will allow the next phrase to be executed a maximum of β times. If β equals zero, however, it will allow the next phrase to be executed an indefinite number of times. If the value of β changes, only the value it has at the first execution of the ONLY phrase will be relevant. If the value of α changes, the behavior is as follows. The value of α when the ONLY phrase is first executed determines the first time it will allow the following phrase to be executed. After that the value of α when the ONLY phrase allows the next phrase to execute will determine how long until it lets it execute again.

ONLY (a) α β,

This is the same as above except that the next "a" phrases will be governed by the ONLY phrase.

ERROR ...,

will print a message (as described in Section 4.13 for HALT and EXIT). It will then set $PERROR to a nonzero value and will terminate the current section.

There are some further odds and ends not covered in the preceding sections, but these fall into one of the following categories: (1) features whose purpose is to assist in debugging the system itself; (2) features which will be useful only to someone who is familiar with the internal workings of the system, or (3) features left over from an earlier version which have been effectively replaced by other better ones. Since these are of no use to the general user, they are not discussed here; however, for the sake of completeness they are mentioned in the appropriate parts of Section 5.

5. Reference Summary

Section 5 contains a concise summary of the material covered in section 4. It is not intended to be self-explanatory (even for an experienced programmer) but is simply intended to serve as a memory aid for someone who has read section 4. The material here is arranged, not in order of complexity, but in a (hopefully) logical fashion. Hence someone who has covered only part of section 4 should be prepared to ignore various incomprehensible material mixed in with the familiar.

To avoid excessive wordage, the following conventions will be used throughout section 5.

Except where the contrary is explicitly stated,

$e\alpha$, $e\beta$, $e\gamma$ ...	denote extended operands
α, β, γ , ...	standard operands
ia, ib, ic, ...	I-operands
a, b, c, ...	simple operands
v, w, x, y, z	variable names
bv, bw, bx, by, bz	block (or list) names
j, k, l, m, n	integers
p, q, r, s, t	18-place operands
σ , τ	tape designators

Anything printed in capitals is to appear on the card just as it is. #α, #c, #x, etc. denote the numerical value of α, c, x, etc.

5.1 General Description of the Input Cards

Input cards are subject to the following general rules:

a) All information is punched in columns 1 to 72 (columns 73-80 are completely ignored). Positioning on the card is completely free, i.e. there is no requirement to make any punches in specific columns. Excess blanks are simply ignored, unless the contrary is explicitly stated.

b) Cards are considered one continuous string of characters with no special significance attached to card boundaries. That is, column 72 of one card is followed by column 1 of the next, exactly as, say, column 38 is followed by column 39.

c) The character ' (apostrophe) (On some keypunches this character is - (dash) or@) may be used to terminate a card before column 72. E.g., if ' appears in column 45, then column 44 is followed at once by column 1 of the next card. (Whatever is punched after the ' is completely ignored--so these columns can be used for punching remarks, if desired.)
NOTHING SAID ELSEWHERE SHOULD BE INTERPRETED AS PERMITTING ANY OTHER USAGE OF THE CHARACTER ' ; there is no exception to this rule.

d) Normally the input is considered as a sequence of "items"; there are two kinds of items, "symbols" and "numbers". A number is more or less what you would expect it to be: e.g., 17 -23 14. 14.0 14.23 1935.246 A symbol is a string of one to six letters, or else one of the characters ( ) , $ * / (These latter are referred to as special symbols.) Two consecutive numbers, or two consecutive alphabetic symbols must be separated by at least one blank; a number and a symbol do not have to be separated from one another, nor does a special symbol have to be separated from any other item. However, as

noted above, it is never wrong to separate any two items by one (or more) blanks. Of course, a blank should never occur in the middle of a number or a symbol.[1]

The items on the control cards are grouped into "phrases"; each phrase begins with a symbol (called a primary code), and ends with a comma--what comes between may vary widely according to the particular primary code, and will be discussed in detail in the appropriate place below.

[1]There are two minor cases where the separating blank is not needed: these can safely be ignored, but are mentioned here for the sake of completeness--a six-letter symbol need not be separated from the following item even if that is a symbol. Two numbers need not be separated if the second starts with a plus or minus sign, or if the first contains a decimal point and the second starts with a decimal point.

5.2 Operands

Constants:

a) positive or negative integer (or zero). Maximum of 10 digits

b) $(Ø) followed by octal integer (maximum of 12 digits. If it has 12 digits and the sign, first digit may be only 0, 1, 2 or 3.)

c) $(k) followed by integer in radix k (binary representation must not exceed 35 bits (not counting sign))

d) $(Fk) followed by (positive or negative) decimal of not more than 8 significant digits. The constant is 10^k times this number in floating point form. k may be omitted if 0

e) $(Bm) followed by (positive or negative) decimal. This number will be used, with 35-m bits for the integer part, m bits for the fraction.

f) $(HS)bbbbbb bbbbbb are any six characters (other than ' of course): the constant is their binary representation.

g) $(T) followed by a tape designator (see below)

Variable:

a) name of 1-6 letters, not starting with B

b) $n

c) $$s$n, where s is a subsequence name (within the the subsequence s, $n and $$s$n are equivalent.)

Simple operand:

a) any constant

b) any variable

c) - followed by any variable

d) any $- operand designated as simple: see list below.

I-operand

a) any simple operand

b) any expression v+a (v a variable, a any simple operand)

c) any expression v+-a (v a variable, a any simple operand)

Block name:

a) 1-6 letters, first of which is a B

b) $Bn

c) $$s$Bn

List item:

a) bv*a bv a block name which has been defined as a list

b) bv**NEW

c) bv**LAST

d) bv**NEXT (or simply bv)

e) bv**SAME

Slice let bbb be either a list item, or a block name which has been defined as a block

bbb is a slice of size $i_1 \cdot i_2 \cdot \ldots \cdot i_k$

bbb $(ia_1, ia_2, \ldots ia_m)$, $m < k$, is a slice of size $i_{m+1} \cdot i_{m+2} \cdot \ldots \cdot i_k$

bbb $(ia_1, ia_2, \ldots ia_k)$ is a slice of size 1 (where $i_1, i_2 \ldots i_k$ are the maximum sizes of the indices.)

Standard operand

a) any I-operand

b) any slice of size 1

c) any $-operand not designated as Extended (see list below)

Extended operand

a) any I-operand

b) any slice

c) any $-operand

n-place operand: like a standard operand - but any integer constant involved will be considered to be preceded by $(Bn)

Tape designator

a) An tape n on channel A

b) Bn tape n on channel B

c) Tn nth tape assigned by the program

Simple tape designator

a) any tape designator

b) any simple operand which could not be confused with a tape designator

Standard tape designator

a) any tape designator

b) any standard operand which could not be confused with a tape designator

The most important of the above are

simple operands

I-operands

standard operands

extended operands

Note that each of these classes is completely included in the next. (E.g., any I-operand is automatically eligible as a standard or extended operand.)

An operand is changeable if it is

a) a variable (not preceded by -)
b) a slice
c) a $ operand designated as changeable

All others are unchangeable.

Any operand, in any context, may be preceded by $ØK. If this is done, no error message will be given for extended and/or unchangeable, even if the context does not normally allow it.

In a phrase accepting two or more standard operands, they may be (optionally) separated by a single slash (/).

The following is a list of all $-operands. The first column indicates simple (S), standard (ST), extended (E), or block name (B). The second column indicates changeable (C) or unchangeable (U).

SPECIAL CONSTANTS

E	U	$(OE)xxxxxx	xxxxxx is an octal number (any number of digits) left adjusted in necessary number of words.
E	U	$(H x)bbbbbbbx	x is a character, bbbb a string of characters other than x: binary representation will be left adjusted in necessary number of words, with blanks added to fill last word if needed. (NOTE, if x is a letter, must be separated from H by a blank, of course.)
ST	U	$CTV An	current value of auto-counter An
ST	U	$CTVA n	

ST	*	$(An, ia_1, ia_2,..., ia_k)	#ia_m where m is current value of counter An (commas are optional)
ST	*	$(La, ia_1, ia_2,...,ia_k)	#ia_m where m = #a (commas are optional)

(* These are unchangeable if any one of the ia_1, ia_2,...,ia_k is unchangeable; otherwise changeable.)

$-OPERANDS PERTAINING TO PERCEPTRON

S	U	$LAYERS	total number of layers in the perceptron.
ST	U	$NUNITS a	number of units in layer #a
ST	U	$NPACK a	number of words in packed form of layer #a
S	U	$RETINA	dimensions of retina (width in right half, height in left)
ST	U	$RETINH	horizontal dimension of retina
ST	U	$RETINV	vertical dimension of retina
ST	C	$THRESH a	threshold for layer #a (18 binary places)
ST	C	$UNIT a (ib)	Value of unit #ib in layer a (sum of inputs minus threshold), last time activity was computed for this layer (unless changed since). Value has 18 binary places.
ST	C	$UNIT a * ic (ib)	same as previous, but use unit from #ic time - steps back
E	C	$UNIT a	all units for layer #a
E	C	$UNIT a * ic	all units for layer #a, #ic time-steps back
ST	U	$LINK a TYPE	type number of linkage #a,
ST	U	$LINK a INIT $LINK a FROM	number of initial layer for linkage #a
ST	U	$LINK a TERM $LINK a TO	number of terminal layer for linkage #a
ST	U	$LINK a DELAY	delay for linkage #a
ST	U	$NLINK a (ia, ib)	number of the a^{th} linkage to layer #ia from layer #ib. 0 if no such linkage

ST	U	$NLINK a (ia)	number of the #ath linkage to layer #ia. 0 if no such linkage
ST	U	$WEIGHT a (ia, ib)	value of weight from unit #ia to unit #ib, for linkage #a. (18 binary places)
ST	U	$LAYLOC	core location of the VLAYEA list (first loc above)
ST	U	$THLIST	core location of the list of thresholds (first loc above)
ST	U	$LINKB	core location of linkage blocks
S	U	$LINKS	core location of the linkage description list (first loc on list)
S	C	$ACTOP	number of highest layer with activities computed since last stimulus presented
S	U	$REINFN	number of units reinforced for linkage most recently reinforced
S	C	$PERROR	nonzero if latest perceptron was suspended because of errors
ST	U	$WGTFIX α	location of fixed constants area for linkage #α

MISCELLANEOUS $ OPERANDS

ST	U	$SUBN bv	number of subscripts in block or list bv
ST	U	$SUB bv (ia)	maximum value for subscript #ia (from left to right) (Junk if no such)
ST	U	$LENGTH bv	length of list bv
S	U	$LØCL bv	location of "locator" for bv
ST	U	$CKEYS	console keys
ST	U	$DCKEYS	console keys, interpreted as a 6-digit decimal number
S	U	$OCKEYS	console keys, as they were at start of program
ST	U	$ØPLØC eα	location of eα
ST	U	$ØPSIZ eα	number of words in eα
ST	U	$ØPSIZ *	number words in eα on most recent $ØPLØC
ST	U	$ØPLØC *	location of eα on most recent $ØPSIZ
S	U	$DATE	monitor date
S	U	$CASE	number of current case

ST	C	$CORE a	core location #a (actually address part of #a)
E	C	$CORE a (ia)	a slice of #ia words, starting at core location #a
ST	C	$Xa	value of X-value #a
ST	U	$TIME	current time (hundredths of minute) (Absolute)
ST	U	$HOUR	current time (100 x hours + minutes, modulo 2400)
S	C	$RANCID	current random seed
ST	U	$RAND a	random integer, $1 \leq r \leq$ #a
ST	U	$RAND (ia, ib)	random integer #ia $\leq r \leq$ #ib (or vice versa, if #ia $\geq$ #ib
ST	U	$RANDF a	random number x (floating point form) $0 \leq r \leq$ #a
ST	U	$RANDFF f	(f is a decimal number, or a variable whose value will be considered floating point) random number, (floating point) $0 \leq r \leq f$
S	C	$QFILE	set nonzero when EOF is found by READ (NOTE: This is not reset to zero if EOF is not found)
S	C	$QDIVCK	this is set nonzero when divide check occurs, then stays that way until set zero by user
E	C	$LINE	the output line currently built up by "LINE" phrases (20 words - 1st word has 1 blank and first 5 characters, remaining words have 6 characters each).
E	C	$LINE (ia)	the first #ia words of $LINE
E	C	$LINE (ia, ib)	the next #ia words after the first #ib of $LINE
S	U	$LINEPP	current print position in "LINE"
S	C	$NEXREC	place to put control tape record number for special spacing
S	U	$RECNUM	last record number read from control tape is in decrement of this word
S	C	$ZLOCK	pseudo location counter
S	C	$ZNESX	loop or subsequence nesting location

$ - OPERANDS PERTAINING TO STIMULUS INPUT OR GENERATION

S	C	$STNAML	length of stimulus name (number of words)
E	C	$STNAME	stimulus name
S	C	$INTAPE	current stimulus input tape
S	C	$CLASS	class of current stimulus
S	C	$KEY	current key number
S	C	$QSFILE	this is set nonzero when EOF is found on stimulus tape, zero when stimulus or key is read
B	C	$BSTIM	packed stimulus
B	C	$BSRETA	first working retina (NOTE: point with coordinates (H, V) is $BSRETA (V, H))
B	C	$BSRETB	second working retina
B	C	$BSTIML	list of save stimuli
S	C	$SSHDIM	retina dimension, horizontal
S	C	$SSVDIM	retina dimension, vertical
S	C	$SSHCEN	center point, horizontal (18 binary places)
S	C	$SSVCEN	center point, vertical (18 binary places)
S	C	$SSHPAK	size of packed stimulus
S	C	$SSPAKQ	Flag, nonzero if packed
S	C	$SSTAPE	output tape
S	C	$SSKEYP	"permanent" key number
S	C	$SSBASE	base number for SSPT

$ - OPERANDS PERTAINING TO "DATA" TAPE

ST	U	$DATA	symbol or integer part of next item on DATA tape
S	C	$DATAQ	0 if item was integer 1 if fraction 2 if symbol 3 if end of file 4 if binary card
S	C	$DATAI	integer part of last item, or symbol

S	C	$DATAF	fractional ditto
ST	U	$DATAM a	last item, with #a binary places
E	C	$DATAB	image of the binary card just brought in (27 words)
S	C	$DFILE	file number of data file being read

5.3 Phrases

A simple phrase consists of a primary code, possibly followed by additional information (depending on the meaning of the primary code) and terminated by a comma. The comma may be omitted if followed by a ")".

A compound phrase may be one of the following two forms:

a) (a, ...)

... may consist of one or more (simple or compound) phrases. They will be repeated #a times. If a is omitted, it will be taken as 1. It is irrelevant if the value of a is changed by one of the inner phrases: it is the value at the beginning which determines the number of iterations.

NOTE: even if a is omitted, the comma must be there

b) (* v α β γ, ...)

Let c be the least integer > 0 with

$$\#\beta + c \cdot \#\gamma \geq \#\alpha \quad (\text{if } \#\gamma > 0)$$
$$< \#\alpha \quad (\text{if } \#\gamma < 0)$$

Then the loop will be repeated c times: v will be given the value $\#\beta$ at the start of the first iteration, and will have $\#\gamma$ added to it at the start of each succeeding one. (Thus, if it is not changed by any of the inner phrases, it will take the values $\#\beta$, $\#\beta+\#\gamma$, $\#\beta+2\#\gamma$, ...) Changes in the value of α, β, γ within the loop are irrelevant. Changes in v will not affect the number of iterations.

If γ is omitted (or zero) it is taken as 1 (if $\#\alpha \geq \#\beta$) or -1 (if $\#\alpha < \#\beta$). If β is omitted (or zero), it is taken as 1.

A phrase name (or place name) may be attached to a phrase by putting

*xxx

before the phrase, where xxx is the desired name.

If desired, the following alternate notation may be used for compound phrases; it is especially intended for complicated situations with much nesting of phrases.

"(" may be replaced by "OPEN name" and the matching ")" by "CLOSE name".

"Name" is any desired identification of from 1 to 6 letters. It should be the same for the matching OPEN and CLOSE, and preferably should be different for each different set. A warning message will be printed if the names do not agree.

NOTE 1: the entire expression "OPEN name" takes the place of "(" : thus we may have, e.g.

```
OPEN   A    5,
OPEN   JOE  V,
OPEN   SS   * I  5,     etc.
```

NOTE 2: the identifying names are completely independent of names used for place names, variable names, or any other purpose; hence it is not necessary to avoid duplicating one of these.

5.4 Subsequences

The general form of a subsequence definition is

DEFINE name (x_1 x_2 ... x_k / x_{k+1} ... x_m), ... , END

..... is one or more phrases which form the subsequence

A subsequence is used in a phrase of the form

name q_1 q_2 ... q_n ,

q_1 q_2 etc. are operands; the form and use of q_i is determined by the corresponding x_i in the definition.

if x_i is	q_i may be	use of q_i
S	any standard operand	$i will be set to #$q_i$
SR	any changeable standard operand	$i will be set to #$q_i$, and when the sequence is finished the final value of $i will be put back in q_i
E	any extended operand	$Bi will be defined as a (one-dimensional) block of the same size as q_i, and the contents of q_i will be put into it.
ER	any changeable extended operand	same as above; also the final contents of $Bi will be put back into q_i
B	a list or block name	$Bi will be made synonymous with q_i; i.e. all references to $Bi will function exactly as if q_i were used
T	any standard tape designator	$i will be set to #$q_i$
N	any name of 1 to 6 letters	$i will be set to the binary representation of q_i
L	a list of one or more standard operands, terminated by an asterisk (*)	$i will be set to n, the number of operands; $Bi will be defined as an n-word block, and the values of the operands placed in it.

M	a list of one or more extended operands, terminated by an asterisk (*)	$i will be set to n, the number of operands; $Bi will be defined as an n x 4 block. $Bi(j) will be used as a locator, and defined as a block of same size as the j^{th} operand; and the values of that operand will be placed in this block. (Note: See LLSYN, in section 5.10)

E, ER or L may be followed by C; if so, the corresponding $Bi will be cleared after exiting from the sequence.

S, E, L, or M (also SR, EC, etc.) can be followed by an integer k, indicating that the operand(s) are to have k binary places; or by *, indicating that the operand(s) are to be floating point.

In type L or M, the terminating * is not needed if no further operands follow.

If h > m (i.e. there are more operands than there are x_i's in the definition), then x_{m+1}, x_{m+2}, etc. will be assumed to be S.

If h < k, then q_{h+1}, q_{h+2}, ... q_k will be taken as zero
(If any of the relevant x_i is SR or ER it will be taken as S or E respectively; it <u>may not</u> be B or there will be an error.)

If h < m, then $i (or $Bi as the case may be) for the omitted operands beyond the / (that is, for i > h and i > k) will be left as they were at the end of the last use of the subsequence (zero if it has never been used).

In the definition, the / may be omitted. Furthermore, the entire (x_1 ...) may be omitted if there are to be no operands (other than S-type).

$i may be considered as an abbreviation for $$name$i, where 'name' is the name of the subsequence in which it is used. Hence $1 used in one subsequence is a <u>different</u> variable from $1 used in another subsequence. The long form $$name$i is rarely used, as the normal use of these variables is either by i) direct reference within the subse-

quence to which it belongs, (when the short form $i can be used) or ii) implicitly, by means of the subsequence definition, as described above. However, in special cases it may be required to make direct reference from another subsequence or the main sequence, and in this case the long form $$name$i is available.

Similar remarks apply to $Bi.

There is no objection to the subsequence making use of $i or $Bi's which are not used implicitly by the definition. Nor is there any objection to using both $1 and $B1, etc.

Note that when an ordinary variable or block or list (alphabetic name) is used, the situation is the exact opposite of the above; i.e. the same name refers to the same variable, no matter what sequence it is used in.

The following phrases may appear in a subsequence, in addition to any of the ones which may appear in any sequence (sub- or main):

FIN,	If this phrase is executed, it will cause immediate termination of the section.
FINSUB,	If this phrase is executed, it will cause immediate termination of the subsequence and return to the main sequence or higher level subsequence.
FINSUB α,	Similar to FINSUB, but the first #α phrases after the phrase using the subsequence will be skipped over. (See 5.7)

FINSUB, and FINSUB α, would not make sense in a main sequence; FIN could be used in any sequence, main or sub-; it is included here to point up the difference between FIN and FINSUB.

FINSUB, would be redundant if placed immediately before END
FIN would be redundant, <u>in a main sequence,</u> if placed immediately before END.

If a subsequence is defined with the name of a previous subsequence or a primary code, the new definition will replace the old one. (A warning message will be printed.) DEFINE A * B, will cause "A" to have the same meaning as is <u>currently</u> attached to the name "B".

5.5 Arithmetic Phrases

ADD eα eβ eγ,	Corresponding words in eα, eβ will be added together and the sum placed in the corresponding position in eγ. If eα or eβ (or both) has more words than eγ, the last ones will be ignored. If eα or eβ (or both) has fewer words than eγ, the last one will be repeated a sufficient number of times.
ADD eα eβ,	Same as ADD eα eβ eα,
SUB, MPY, DIV	Similar, but subtract (eβ from eα), Multiply or divide (eα by eβ) respectively.

DIV may also be used as

DIV eα eβ eγ eδ,	eδ will receive the remainder(s). eγ must have at least as many words as eδ, or strange errors may occur.
ADDF, SUBF, DIVF, MPYF	Similar to the above, but floating point arithmetic is used.

If, in either DIV or DIVF, division by zero is attempted, $QDIVCK will be set nonzero.

MAX α β_1 β_2 ... β_n,	Set α = max (#β_1, #β_2, ... #β_n) $n \geq 2$
MIN α β_1 β_2 ... β_n,	Set α = min (#β_1, #β_2, ... #β_n) $n \geq 2$
MAX α β,	Set α = max (#α, #β)
MIN α β,	Set α = min (#α, #β)
SET v α,	Set v to #α
SETX n α,	Xn is set to #α (Note: SET $Xn α is not permissable, as $Xn is not a simple variable)
ADK a β,	add #a to β
SBK a β,	subtract #a from β
RSBK a β,	set β to #a - #β
FLOAT (a) eα_1 eα_2 ...,	each word of each eα will be considered as a fixed point number with #a binary places, and will be converted to floating point. (a) may be omitted if 0 is intended.

FIX (a) $e\alpha_1$ $e\alpha_2$ ...,	each word of each $e\alpha_i$ will be considered as a floating point number and will be converted to fixed point with #a binary places. (a) may be omitted if zero is intended. (Note: excess binary places are truncated, not rounded.)
ASHIFT a β γ	contents of β will be shifted #a places (right if +, left if -) and put into γ. THIS is an <u>arithmetic</u> shift, i.e. the sign bit is not shifted. (see SHIFT in Section 5.6 for a logical shift).
ASHIFT a β,	same as ASHIFT a β β,
ZERØ $e\alpha_1$ $e\alpha_2$ ... $e\alpha_k$,	all words in all $e\alpha_i$ are set to zero
BRAKET $e\alpha$ $e\beta$ $e\gamma$ δ,	each word in $e\alpha$ is set equal to corresponding word in $e\beta$, if it was less, or equal to corresponding word in $e\gamma$, if it was more, or else is left unchanged. δ is set nonzero if any word in $e\alpha$ was changed; otherwise it is set zero. $e\alpha$, $e\beta$, $e\gamma$ should all be the same size

5.6 LOGICAL PHRASES

AND eα eβ eγ,	set eγ to "logical and" of eα, eβ
OR eα eβ eγ,	set eγ to "logical or" of eα, eβ
EOR eα eβ eγ,	set eγ to "exclusive or" of eα, eβ
COM eα eγ,	set eγ to complement of eα
LOGIC wxyz eα eβ eγ,	w,x,y,z may be either 0 or 1 eγ will be set to a logical function of eα, eβ as follows: each bit in eγ will be set to

w	if	the	corresponding	bits	in	eα, eβ	are	0 0
x	"	"	"	"	"	"		0 1
y	"	"	"	"	"	"		1 0
z	"	"	"	"	"	"		1 1

(e.g. "LOGIC 0001 eα eβ eγ " is equivalent to "AND eα eβ eγ " and " LOGIC 1011 eα eβ eγ " sets eγ to the logical or of eα and the complement of eβ)

NOTE: in any of the above, if eα and/or eβ is shorter than eγ, the last word (36 bits) will be repeated. If eα and/or eβ is longer than eγ the excess will be ignored.

NOTE: in any of the above, if eγ is omitted, it will be assumed to be the same as eα

SHIFT a eα eβ,	eα is shifted #a places (right if +, left if -) and put into eβ. <u>If necessary, eα is assumed to extend with zeroes on either side</u>.
SHIFT a eα,	same as SHIFT a eα eα,
MOVE eα eβ,	move eα into eβ: if eβ is longer, it will be filled out with zeroes.
BITS eα β, BITSZ eα β, BITZ eα β,	β is set to the number of 1-bits in eα β is set to the number of 0-bits in eα same as BITSZ
FILL n eα $e\beta_1$ $e\beta_2$...,	eα will be filled with the values in the respective $e\beta_i$, in order. If there are not enough words in the $e\beta_i$ to completely fill eα, the rest will be filled with 0.

FILLB n bv α_1 α_2 ...,

bv will be defined as a block of the appropriate size, and $\#\alpha_1$, $\#\alpha_2$, etc, will be placed in it.

LFILL n bv $e\beta_1$ $e\beta_2$...,

bv must already be defined as a list. A new item will be added to the list and the values of $e\beta_1$, $e\beta_2$, etc. put into it. If the item is filled, or if one of the $e\beta_i$ has an * before it, another new item will be added and the process will continue.

In either **FILL**, **FILLB** or **LFILL**, if any of the operands are explicit numbers, they will be interpreted as having n binary places, or as being floating point if n is *. n may be omitted if 0 is intended. n may be written (n), ie **LFILL** (n) bv etc. or **FILL** (n) $e\alpha$ etc.

PACK $e\alpha$ $e\beta$,

each bit of $e\beta$, starting from the left, will be set to 1 or 0, according as the sign of the corresponding word in $e\alpha$ is + or -

PACKZ $e\alpha$ $e\beta$,

similar, but the bit is set 0 or 1 according as the magnitude of the corresponding word is zero or nonzero

UNPACK $e\alpha$ $e\beta$,

the reverse of PACK; i.e. the words in $e\alpha$ are set + or - according as the corresponding bits $e\beta$ are 1 or 0. The magnitudes of the words in $e\alpha$ are left unchanged.

In **PACK** or **PACKZ**, excess bits in $e\beta$ are set to 0, or excess words in $e\alpha$ are ignored, as the case may be. In **UNPACK**, excess words in $e\alpha$ are set +, or excess bits in $e\beta$ are ignored, as the case may be.

NØTE: the choice of 1 for +, 0 for - is in agreement with the usual conventions for packed activity vectors. It is the reverse of the normal 7094 binary representation.

5.7 Conditional and Control Phrases

The "next" phrase after a given phrase is exactly what you would expect, with the following clarifications:

i) if the given phrase is followed by "(", the "next" phrase is the compound phrase running from the "(" to its matching ")".

ii) if the given phrase is followed by ")", i.e. it is the last in a compound phrase, the "next" phrase is the next phrase after the compound phrase, regardless of whether we are on the last iteration or not.

iii) if the given phrase is the last (other than END) of a subsequence definition, the "next" phrase is the next after the phrase using the subsequence.

iv) if the given phrase is END of the main sequence, the "next" phrase is still the END.

COMP (k m n) α β,	k, m, n are numbers from 0 to 7 if #α > #β, the next k phrases will be skipped if #α = #β, the next m phrases will be skipped if #α < #β, the next n phrases will be skipped
COMP α β,	same as COMP (012) α β,
IF LESS α β,	skip next phrase unless #α < #β
IF EQUAL α β,	skip next phrase unless #α = #β
IF MORE α β,	skip next phrase unless #α > #β
IF NEG α,	skip next phrase unless #α < 0
IF ZERO α,	skip next phrase unless #α = 0
IF POS α,	skip next phrase unless #α > 0
IF PLUS α,	skip next phrase unless #α is + (this includes + 0, which IF POS does not)
IF MINUS α,	skip next phrase unless #α is - (including -0)

IF LEQUAL eα eβ,	skip next phrase unless #eα and #eβ are the same. If lengths are different, shorter will be filled out with zero. (Note - this will distinguish between + 0 and - 0, which are considered identical by the arithmetic comparisons).
IF SSW α, **IF SWITCH α,**	skip next phrase unless sense switch #α is down
IF LZERO eα,	same as IF LEQUAL eα 0
IF NSSW α,	(IF Normalized Sense SWitch) skip next phrase unless sense switch #α is reversed from its original position.
IF LAST bv,	skip next phrase unless most recent use of list bv referred to last item on list
IFN ....,	exactly the opposite of "IF ...," that is, the next phrase will be skipped if the relevant condition is met.
SBSK v,	reduce value of v by 1; if it becomes 0 then skip the next phrase.
JUMP α,	skip next #α phrases.
GO xxx,	xxx may be any phrase name: control will go to that phrase. It <u>is not</u> permitted to go into a compound phrase or subsequence from outside. It <u>is</u> permitted to go out of a compound phrase or subsequence from inside.
#NLY α β,	next phrase will be done every #α times, but at most a total of #β times. (If value of #α changes, the value just before the phrase is done one time is used to determine next time. (β = 0 means indefinitely)
#NLY (a) α β,	Same, but next #a phrases are affected.
HALT, **HALT ...,**	the message, if any (...) will be printed on and offline, the online printer will be spaced up, and the program will halt - first comma terminates message.
ERROR ....,	Prints message, sets $ERROR nonzero, goes to next section.
EXIT ....,	Prints message, and terminates job (normal exit to monitor if running under monitor)

FIN,	This logically terminates the section, i.e. if executed the section terminates, but it does not mark the physical end of the section, as END does. FIN may be used in a subsequence, but its effect is still to terminate the section, not just the subsequence.
FINSUB,	This may be used in a subsequence only, and is similar to FIN, but just causes exit from the subsequence, not the section.
FINSUB a,	This will terminate the subsequence, and cause skipping of #a phrases after it.
NOP,	do nothing
END,	marks the physical end of a subsequence or main sequence. Also, if executed, has the same effect as FINSUB, if in a subsequence, or FIN, if in a main sequence.

5.8 Perceptron Specifications

Layers in the perceptron must be numbered consecutively from 1 (retina) up to the highest number.

Maximum of 15 layers.

Linkage with no delay must have a lower (numbered) initial layer than terminal layer.

Maximum delay is 15.

LAYER α β p, — layer #α is to have #β units, with threshold #p

RETINA α β, — layer 1 is #α units wide x #β units high

LINK n .... , — linkage of type n, further specified by one or more of the following subcodes:

subcode	significance	may be used on type
TO α	terminal layer is #α	all
FROM α	initial layer is #α	all
DELAY α DELAY α * β	delay (time lag) is #α compound layer, delay #β to #α	all all
WEIGHT p **W** p	Weight is #p	1,2,4,6,7,8
POS α **EXCITE** α **X**α	each A-unit has #α positive (excitatory) inputs	2,4,6
NEG α **INHIB** α **Y**α	each A-unit has #α negative (inhibitory) inputs	2,4,6
INPUTS α β	each A-unit has #α positive and #β negative inputs	2,4,6
NONREP	all inputs to a given A-unit must be from different units	2
ETA p **INCR** p	increment for weights is #p	3
DELTA p **DECAY** p	decay factor for weights is #p (p is a 35-place operand)	3
LGAMMA **LGAM** **LG**	reinforcement is type γ	3

LINK, (Cont.)

BGAMMA BGAM BG	reinforcement is type Γ	3
ALPHA A	reinforcement is type α	3
TYPE α β	reinforcement type is α if $\#\alpha = \#\beta = 0$ γ if $\#\alpha = 1$ $\#\beta = 0$ Γ if $\#\alpha = 0$ $\#\beta = 1$	3
HORIZ α H α	horizontal dimension of subretina is $\#\alpha$	4
HVAR α β	horizontal dimension of subretina is chosen at random in an interval of width $\#\beta$ centered at $\#\alpha$	
VERT α V α	vertical dimension of subretina is $\#\alpha$	4
VVAR α β	similar to HVAR (NOTE: VVAR should not be used if HVAR is not)	4
SQUARE	vertical dimension will have the same value as horizontal, i.e. the subretinas will all be square.	4
SQVAR α SVAR α	vertical dimension will be chosen in an interval of width $\#\alpha$ about the value used for the horizontal dimension	4
SAME α	$\#\alpha$ units will receive similar input patterns	6
DIM α	horizontal dimension of input layer will be taken as $\#\alpha$. (Usually used when a layer other than the retina is used in type 4 or type 6.)	4,6,8
SAVE α	location of computation code will be saved in X$\#\alpha$	all
SAVEI α	similar, for initialization code	all
SAVER α	similar, for reinforcement code	all

LINK (Cont.)

SAVCG α	location for second-generation code (computation) will be saved in X#α	all
SAVRG α	location for second-generation code (reinforcement) will be saved in X#α	all
CGEN	force computation method to be generation (G)	3
CGENX	force computation method to be direct (NG)	3
RGEN	force reinforcement method to be generation (RG)	3
RGENX	force reinforcement method to be direct (RNG)	3
TAPE $\sigma\alpha$	specifies tape unit (σ) and file number (#α) for input patterns for type 8 linkage	8
UNCβ	"unconditional" option in type 8 linkage	8
DIRECT	"direct" option in type 8 linkage	8
RANDOM α	#α will become the random seed at the start of generating this linkage. When finished the old one will be restored.	all
BYPAS α	During execution, this linkage will never be reinforced when X\alpha$ has a nonzero value; nor will it have any effect on activity computation at such times. (NOTE: X\alpha$ will be set zero during the generation of the perceptron.)	all

RANDOM α, set random seed to 2#α - 1

RANDOM * α, set random seed to #α

PRTOPT α β γ,
PRINTS α β γ,

if #$\alpha \neq 0$, print perceptron description online
if #$\beta \neq 0$, print storage allocations offline
if #$\gamma \neq 0$, print storage allocations online also (must have #$\beta \neq 0$)

NLAYER n, — normally the number of layers in the perceptron will be simply the number of times the code "LAYER" appears in the cards (+1 if "RETINA" is used). However, if you start getting fancy (putting LAYER in loops or subsequences, etc.) this may no longer hold. In such case you must give "NLAYER n" where n is at least equal to the number of layers (preferably not much more).

NLINK n — similarly, for linkages.

CONECI σ α, — prepare to put input patterns for type 8 linkage onto tape #σ, file #α. (Any information on tape #σ in files previous to #α (if any) is not disturbed. Files #α and following are destroyed.)

CONEC SIMPLE α β_1 p_1 β_2 p_2 ..., — unit #α is to have inputs from unit #β_1 (weight p_1), unit β_2 (weight p_2) etc.

CONEC DIRECT α β_1 β_2 ..., — unit #α is to have inputs from units #β_1, #β_2, etc. (Used for a linkage with "direct" option.)

CONEC LAYER α β, — inputs specified in future phrases (as above or below) will be from layer #α (delay #β) instead of the standard input layer specified in the LINK phrase. This will remain true until a new CONEC LAYER is given, or until the next CONECI. If #α = #β = 0, or if α, β are omitted, this restores to the standard layer.

CONEC COPY α β γ — unit #α gets the same inputs as unit #β but shifted by #γ (that is, where #β gets an input from unit n, #α gets a corresponding input from unit n + #γ (in the same layer).)

CONEC COPIES α β γ, — this has the same effect as the following sequence of phrases:

```
CONEC   COPY α+1   α   γ,
CONEC   COPY α+2   α   2γ,
....
CONEC   COPY α+β-1 α   (β-1)γ,
```

NOTE: the use of CONEC COPY and/or CONEC COPIES instead of specifying the inputs to each unit, is not merely a convenience in writing--it results in a more efficient use of space in the generated perceptron. When there are many inputs to a unit and many copies of the unit, this can lead to a quite significant saving of space.

CONEC ..., ... consists of one or more of the following subcodes:

Uα	following inputs are to unit #α
Wp	following inputs are to have weight #p, except when a weight is explicitly specified with the input
L α β	same effect as CONEC LAYER α β,
$(t_1, t_2 \ldots t_k)$	this specifies k inputs to the unit currently getting inputs. Each t_i may be one of the following forms:

γ	input from unit #γ
(γ, δ)	input from unit with coordinates #γ, #δ (assumes input layer is retina, or has been given 2-dimensional structure by use of DIM in LINK phrase).
α γ $\alpha(\gamma, \delta)$	input from unit #γ (or unit (#γ, #δ)) in layer #α
α β γ α β (γ, δ)	input from unit #γ (or unit (#γ, #δ)) in layer #α, delay #β
any of the above followed by *p	input has weight #p

NOTE 1: if a non-standard input layer is in effect (from a previous CONEC LAYER or L), t_i may not take the form

0 0 γ	to indicate an input from the standard layer.

NOTE 2: if a non-standard layer is in effect, and one of the 2-coordinate forms is used, the layer is assumed to have the same 2-dimensional structure as the standard layer.

CONECZ, this phrase must be given when all inputs for this linkage have been specified

NOTES: The effect of the CØNEC phrases when executed is to place information about the input patterns on tape; this information is incorporated into the perceptron when it is generated (which takes place after the execution of the section specifying the perceptron). Hence it is irrelevant whether the CONEC phrases come before or after the LINK phrase--indeed they could even come in an earlier section.

The CØNEC phrases should suffice to generate almost any desired input patterns. However, in case there should be exceptions, the necessary tables can be formed directly and put on the tape by WRITE phrases; the form of the information on tape is as follows:

1. The information must occupy 1 file.

2. The division into records is logically irrelevant, and may be arranged for the convenience of the writing program.

(However, most efficient timing is obtained with about 20 to 24 words per record). If any word is all ones (i.e. MTH -1, 7, -1) then that word and all remaining words in that record are ignored.

3. I, followed by one or more of the following combinations, followed by 0, specifies inputs to unit I.

3A. N, followed by weight (18 binary places), specifies an input from unit N in proper layer (weight omitted if to be used with a LINK with "direct" option.)

3B. $(2^{18}V + H)$, followed by weight (18 binary places) specifies an input from unit with coordinates (H,V) in standard layer. (Weight omitted if to be used with LINK with "direct" option)

3C. $-(2^{18}D + L)$ followed by either of the above - same as above, but input is from layer L, delay D. (NOTE: if 2-coordinate form is used, layer is assumed to have same dimensions as standard layer).

4. -I, followed by $(2^{18}K + J)$ specifies unit I is to have same inputs as unit J, shifted by K places.

5.9 Perceptron Test Phrases

During the training/testing of a perceptron, $ACTØP holds the number of the highest layer for which activities are currently computed. It is used implicitly by ACTIV and REACT phrases, as discussed below. It may be used explicitly, but there should rarely be any need for this.

When a stimulus is set up on the retina by means of STIM, (as discussed below) $ACTØP will automatically be set to 1; when a stimulus is set up by any other means, the user must explicitly "SET $ACTOP 1,".

NOTE: In the following discussion of ACTIV and REACT, CT is used to represent the value of $ACTØP.

ACTIV α, ACTIVE α,	if #α > CT, compute all activities for layers CT+1 up to and including #α, and set CT = #α (#α must of course be no greater than the highest numbered layer in the perceptron) If α is omitted (or zero), it is taken to have the highest possible value. i.e. "ACTIVE," causes all (hitherto uncomputed) activities to be computed.
REACT α,	if CT > or = #α, set it back to #α-1 (but not < 1) (this allows for recomputing activities of some layers, e.g., after reinforcing a linkage.)
REACT α β,	This has the effect of "REACT α" followed by "ACTIV β"
REACT α *,	This has the same effect as "REACT α α" (useful when α is a complicated expression).
HISTOR, HIST,	move activity states back one time step
LEVEL α β,	a suitably chosen constant will be added to all units in layer #α, so that #β of them will be active. (Note: if several units have exactly the same value, it may be impossible to get exactly #β active in this way; if so, the largest possible number < #β will be used.)

REINF LINK α xx....xx,

Reinforce linkage #α, according to the method specified by "xx....xx", which may consist of one or more of the following:

SCØN eα — S-controlled reinforcement, where eα contains the desired signs (each word of eα giving the sign for the corresponding unit).

ECØN eα — E-controlled reinforcement, eα similar to SCØN

RCØN — R-controlled reinforcement

SPØS eα — Positive S-controlled reinforcement. (Reinforce unit if corresponding word in eα is +)

RPØS — Positive R-controlled reinforcement. (Reinforce units which are +)

CONTIN eα p — Continuous R-units, eα contains desired values (18 binary places): p is the tolerance level, or p may be an extended operand, in which case each word gives the tolerance level for the corresponding unit.

NØQUAN — Reinforcement will be nonquantized; if this is not used, it will be quantized. (May be used with ECON or CONTIN, but not if linkage is type Γ or has nonzero decay.)

SPEC v w (, ..., REINFZ,)

... ... consists of 1 or more phrases, of any kind. They will be executed once for each unit, during the reinforcement; before the start of each execution, v will be set to the number of theunit, and w to its value. Before reaching REINFZ, v should be set to 1 if no reinforcement is desired for this unit, 2 if quantized reinforcement, 3 if nonquantized. w should be set to desired change in unit for nonquantized, or desired increment for quantized (0 if standard increment to be used).

NOTE: "REINF LINK α SPEC v w (,...., REINFZ)," is considered as <u>one</u> phrase for purposes of skipping phrases.)

REINF LINK α *β, — Same as above, using options last saved at this β(as in REINF SETUP ... below).

REINF LAYER α .....,

etc., as in any of the above; same as described for LINK, but will be done for all (type 3) linkages to layer #α

REINF UPTØ α ...., — etc., same again, but do for all (type 3) linkages to all layers from 2 up to #α

REINF SETUP β xx....xx,	the options (as in REINF LINK α) will be set up for later use by REINF LINK α *β, etc. No reinforcement be directly caused by this phrase.

* * * * * * * * *

INTAPE τ,	τ is a (simple) tape designator: hence-forth this tape will be considered the standard stimulus input tape (until changed by a new INTAPE, if any)

In the remainder of this section, τ and σ will be used to denote (standard) tape designators. It will be understood that when τ is used, the specified tape designator may be omitted if the standard stimulus input tape is desired.

The various forms of the phrase "STIM", to be discussed below, provide for reading stimuli from a tape prepared by the "stimulus preparation phrases" (see 5.14) or any other means which conform to the following format. Each has

1st word, left 18 bits	class bits
1st word, next 3 bits	length of stimulus name (0-7)
1st word, last 15 bits	"key number"

The first word is followed by the stimulus in packed form, and then the name. When a stimulus has been read by some form of STIM, we will have

$CLASS	class bits (in first 18 bits)
$STNAML	number of words in name
$STNAME	name

$KEY	key number
$QSFILE	0 if a stimulus was found 1 if an end of file was found

Layer 1 will have the stimulus in unpacked form

STIM τ,	a stimulus will be read from tape τ and placed on the retina
STIM (bx) τ,	same: also the name of the stimulus (or ** if it has none) will be saved in list bx
STIM (*PRINT) τ,	same (as STIM) and a message will be printed "STIMULUS PRESENTED" followed by the name (or ** if there is none)
STIM (*ADD) τ,	same as STIM, and the name (or ** if there is none) will be added into the current output line, followed by one space.
STIM (bx $) τ, STIM (*PRINT$) τ, STIM (*ADD$) τ,	same as the corresponding one above, except that if there is no name, the action concerning the name will be omitted.

* * * * * * * *

CLEAR ACT α CLEAR LAYER α,	set activities for layer #α to off (-0) (all layers if α is omitted or zero)
CLEAR LINK α,	set weights for linkage #α to zero.
CLEAR WEIGHT α,	set weights for all (reinforceable) linkages to layer #α to zero (to all layers if α is omitted or zero).
PAKAC a eα β, GETACT a eα β,	the activity vector for layer #a (delay #β) is placed in eα in packed form (one bit per unit, bit is 1 for active unit). If eα is not big enough the extra bits are lost. If too big the rest of it is set to zero. β may be omitted if 0 is intended.

PAKAC * a eα β, GETACT * a eα β,	similar, but use 0-bit if magnitude of unit value is zero, 1-bit if it is non-zero.
SETACT a eα β,	the activity vector found in packed form in eα is unpacked and placed in the units for layer #a (delay #β)
GETWGT γ eβ α β,	some or all of the weights for linkage #γ will be put into eβ (18 binary places) The weights will be selected according to #β and #α as follows If #α, #β are both zero (or omitted) then all weights, in the order $W_{11}\ W_{21} \cdots W_{k1}\ W_{12} \cdots$ (where W_{ij} is the weight from unit i to unit j). If #α = 0, #β ≠ 0, all weights to unit #β $W_{1\#\beta}\ W_{2\#\beta} \cdots W_{k\#\beta}$ If #α ≠ 0, #β is zero or omitted, all weights from unit #α $W_{\#\alpha 1}\ W_{\#\alpha 2} \cdots W_{\#\alpha m}$ If #α ≠ 0, #β ≠ 0, the weight from #α to #β NOTE: eβ must be big enough to hold all specified weights.
SETWGT γ eβ α β,	weights will be set to the values found (18 binary places) in eβ. Weights are designated as in GETWGT
COUNT α INIT,	set counts (set #α) to zero
COUNT α, COUNT α ABS,	add response into counts (as C , Z, or W) add response into counts (as +, 0, or -)
COUNT α PRINT,	print counts, set #α (assume LINE already started)
COUNT α PRINT INIT,	as above, then reinitialize counts.
COUNT (a) α PRINT, COUNT (a) α PRINT INIT,	as above, but printing will be done on output unit(s) #a (see 5.11)

NOTE: In any of the above, α may be replaced by eβ, in which case eβ (which should have at least three words) will be used to hold the counts.

COUNTS α,	If any form of COUNT is to be used with a set number greater than 5, then before the first use of COUNT we must use this, where #α is the largest set number which will be used.
RESP,	C, Z, or W will be inserted into current output line, preceded by one blank, according to the current response (correct response found in $CLASS)
RESP α,	same as above, but for #α response unit. (correct response found in $CLASS, #α bit, counting from left.)
RESP α eβ,	same, but correct response is given by sign of #α word of eβ. (Bit is 1 for positive response.)
RESPA,	print response (unit 1) as +, -, or 0
RESPA α,	print response (unit #α) as +, -, or 0.
TEST a τ, NOTE: TEST will not work correctly if any linkages have nonzero delay. See Appendix II.	present |#a| stimuli from tape τ (no reinforcement), count number of correct, zero and wrong responses, and add them into cumulative counts. If #a > 0, print counts and reset to zero.
TESTP,	print the cumulative counts and reset to zero. NOTE: if TEST and/or TESTP, are used with a perceptron with more than one response unit, the first one only is relevant.
BINPR α eβ,	print the first #α bits in eβ as 0 and 1 (α may be omitted if all bits in eβ are to be printed, and their number is greater than 36.)
BINPR α eβ γ,	same, arrange them #γ per line
PACT α,	print the activity vector of layer #α
PACT α β,	print the activity vector of layer #α putting #β units per line
PACTØ α,	print the activity vector of layer #α in octal
PACTO α β,	same putting #β <u>units</u>, not <u>characters</u>, per line
PW LINK α,	print weights for linkage #α;
PW TO α,	print weight for all linkages to layer #α-all layers if α = 0

(NOTE: It is possible that PW may not be working correctly.)

IMPORTANT NOTE: IF ANY OF THE FOLLOWING PHRASES ARE TO BE USED, ONE MUST TAKE NOTE OF APPENDIX II.

EC α τ,	present #α stimuli from tape #τ. For each one, compute activities and perform error-correction reinforcement (at the response unit).
EN α τ,	similar to EC, but use non-quantized error-correction reinforcement.
ES α τ,	similar to EC, but use sign-controlled reinforcement.

NOTE: EC, EN, and ES are intended for use with a perceptron with a single unit in the highest layer--this is referred to as "the response unit".

PRECON α τ,	present #α stimuli from tape #τ. For each one, compute activities up to the next to last layer, and reinforce weights in all (type 3) linkages to all but the last layer, using reinforcement as in REINF... RPOS, (This phrase is primarily intended for preconditioning cross-coupled perceptrons.)
PRESNT τ,	present one stimulus from tape #τ, and compute activities.

5.10 Blocks and lists.

BLOCK bv α_1 α_2 ...,	define bv as a block with dimensions $\#\alpha_1$, $\#\alpha_2$, ...
LIST bv α_1 α_2 ...,	similar, but a list
LLSYN bv bw,	make bv synonymous to bw, i.e., future references to bv actually use bw.
LLSYN bv * eα,	use eα to hold the locator information for bv (NOTE: if eα has been used for other purposes, it must be set to zero before doing this.) eα must have three words plus one per subscript.
LLSYN bv	allow bv to be itself again (does not preserve previous use of bv)
LLMOV bv,	remove first item on list bv
LLMØV bv bw,	move first item on list bv to end of list bw
LLMØV bv -bw	move first item on list bv to start of list bw
LLMØV bv *,	same as LLMØV bv bv,
LLMOV (a) ...,	same as before, but move first # a items
LLADD bv bw,	add the whole list bv at the end of list bw
LLADD bv -bw,	add the whole list bv at the start of list bw

NOTE: **NEXT or **SAME should not be used with any list manipulated by LLMØV and/or LLADD until an explicit item on the list is used (or LLSTRT is used).

LLCLR bv,	remove all items from list bv
LLSTRT bv,	if this is done, next use of bv**NEXT will refer to first item on list
LLSAV bv LLSAVE bv,	save the current position
LLUNS bv,	restore to the most recently saved position

5.11 Output

Output is in the form of 119 character lines; these can be built up and printed by LINE .....,

The remainder of the phrase is made up of one or more of the following sub-phrases.

START	prepare to start new line

NOTE: The following subphrases refer to the output unit, which will normally be the offline printer. When other situations are desired, the subphrase as given below may be followed by one of the following:

ONLINE	online printer will be used.
BOTH	both online and offline printer will be used.
n *n	1 for offline, 2 for online, 3 for both, 0 for neither
*v	same as preceding using value of #v

NOTE: These qualifiers, if used, come after the entire subphrase as given below, e.g. "SPACES 3 ONLINE", not "SPACES ONLINE 3".

PRINT	print line currently built up, on off-line printer (after printing, line will be cleared ready for starting a new one. START need not be given if another line is to be done immediately).
SPACE	space 1 line
SPACES α	space #α lines (or start new page if less than #α lines left) (must not be done while a line is partly built up)
PAGE	start new page with next line
BOTTOM α	start new page with next line, unless there is still room for #α lines on this page.
BCD x.....x	"x" represents the first non-blank character after "BCD". Everything between the first and second appearances of x will be inserted into the line.

(An, ..., ..., ...)	between the commas may occur any string of characters except comma (,) or slash (/) (or apostrophe, of course). If k is the current value of auto-counter An, then the k-th string will be inserted. If k is greater than the number of strings given, we assume the whole series is repeated.
(An, ..., .../ ...,)	This is similar to the preceding, except that only the portion after the / is assumed to be repeated.
Bα BLANK α	insert #α blanks (i.e. move #α spaces to the right)
V α	insert #α
V α *a	insert #α, but consider the last #a digits as fraction (i.e. put a . before last #a digits)
VB β α *a	consider #α as having #β binary places for fraction, and insert it, with #a decimal places.
VF α *a	consider #α as floating point and insert it as a fixed point number with #a decimal places.
VFF α *a	consider #α as floating point and insert it as a floating point number with #a decimal places
Xc *a	same as V $Xc *a
R	same as V $RANCID
F α	this can come before V, VB, VF, VFF, X or R and means that the next item will be inserted in a space #α positions wide. (Normally they are inserted in the minimum necessary space - which of course depends on the actual values involved.)

WARNING: In any of the above subphrases where *a is used, there is one point which must be carefully watched. If the operand immediately preceding is a block or list name with neither subscripts nor item number, then *a would be misinterpreted as an item number. Therefore, in such a case, the item number must always be given explicitly (**NEXT).

VØ α	#α will be inserted as a 12-digit octal number
EØ eα	#eα will be inserted as an octal number of the necessary number of digits
EØ (ia) eα	the first #ia octal digits of #eα will be inserted
EØ (ia, ib) eα	the next #ib octal digits after the first #ia will be inserted
EB eα	#eα will be inserted as a binary number of the necessary number of bits
H eα	#eα will be considered as binary coded alphabetic information, and will be inserted
EI n eα	#eα will be taken as n bit groups (n=1,2,3, or 6) and inserted (NØTE, EØ, EB, H are special cases of this, corresponding to 3, 1 and 6 respectively).

EB, H, EI also have variations similar to the second and third forms given above for EO.

TAPE α	#α will be considered as a tape address and inserted (e.g. as A5 or B3) (α may be any standard tape designator).
TAPEX α	Same as TAPE $Xα
C	current case number will be inserted
CP	"CASE n" will be inserted, where n is the current case number.
SETPP α	next item entered will start at print position #α ($1 \leq \#\alpha \leq 120$)
CLEAR	set the print line to blanks. NOTE: in normal circumstances, it is <u>not</u> necessary to use CLEAR at the start of each new line; the line is automatically cleared after printing.
SAVE	The line currently built up will be saved, so that a new line can be worked on and this one continued later. (Can not be done in the middle of a "group" -- see below)
RESTOR	Restore the line most recently saved.

WARNING: There is a minor bug in the program in connection with SAVE and RESTOR. They will work as specified, but each execution of SAVE will cause 22 words of storage to be used up irretrievably. This is cumulative with each execution. ("Irretrievably" means within the current section. The space is regained at the end of the section,) Hence excessive uses will cause the program to be terminated for lack of sufficient space.

G, GZ

Normally, if a line builds up to over 119 characters, the first part will be automatically printed (offline only) and the rest will start a new line, indented 3 spaces from the usual starting place (which means successive portions are limited to 116 characters).

If desired, the subphrases may be divided into groups; G marks the start of a group - and the end of the preceding one, if any. GZ marks the end of a group but not the start of a new one. The portion of line generated by the subphrases in a group will always appear on the same line.

WARNING: If a group tries to generate too long a line, this is a serious error and may stop the program completely.

INDENT α	this changes the indentation referred to in the last paragraph to α spaces. This is a permanent effect - that is, it will stay at this value in all subsequent uses of "LINE" (in this section) until a further use of "INDENT", if any.

Page headings may be formed by

HEAD	followed by subphrases as used for "LINE". The line generated will be used as a heading for all future pages. The heading line will automatically include "PAGE N" at the right hand end (this limits the generated portion to 110 characters. If it is too long the first 110 will be used). The heading line will be printed online at the time it is generated.
HEAD*	A line may be partly built up by "LINE" phrases and then finished by "HEAD"* etc., as on "HEAD".
	PRINT SPACE PAGE BOTTOM may not be used in a HEAD phrase.

5.12 Intermediate IØ

WRITE σ $e\alpha_1$ $e\alpha_2$ ...,	put the current contents of $e\alpha_1$, $e\alpha_2$, etc. onto tape σ
READ σ $e\alpha_1$ $e\alpha_2$ ...,	read from tape σ into $e\alpha_1$, $e\alpha_2$, etc. If an End Of File is found, then $QFILE will be set nonzero.
IØ $e\beta_1$ $e\beta_2$ ...,	One or more such phrases preceding a READ or WRITE phrase will have the effect of adding $e\beta_1$, $e\beta_2$ etc. to the beginning of the list on the READ or WRITE.

NOTE: Each WRITE phrase puts 1 "record" on the tape. All items in preceding IØ phrases (if any) together with all those in the WRITE phrase itself go consecutively in this record. Normally this record should be read by a single READ phrase with a list of items of the same sizes, in which case the following pedantic details can be safely ignored. If the above is not followed, then the following rules must be obeyed:

1) The READ phrase will read one or more records as required to fill all the items in the list. If the last item is filled and there are extra words in the last record, these will be skipped over.

2) However, if an END-OF-FILE is encountered before all items are filled, reading will terminate at that point.

REW τ,	Rewind tape τ (τ may be omitted if stimulus tape is intended) (see 5.9)
TAPE σ	position tape σ according to one or more of the following. σ may be * if the stimulus tape is intended.
REWIND	rewind tape
UNLOAD	rewind and unload tape
ENDFIL	write an end of file mark
FILE α	space #α files forward if + space back to start of current file if 0 space back |#α| files if - NOTE 1: 0 may <u>not</u> be omitted NOTE 2: e.g. FILE -2 means space back to the beginning of current file (if necessary) and <u>then</u> back two more.
RECORD α	space #α records (note; end of file mark counts as a record)

PRINT α	Print $\lvert\alpha\rvert$ records; if $\alpha < 0$ backspace to original position after printing.
PRINTF α	Similar to above, but files instead of records (Note: if backspaced, it will assume tape was positioned at beginning of a file.

5.13 "Data" Tape

DATA xxx .... xxx,	following items go on data tape until the symbol xxx is repeated
DATA *@ etc.	use @ for data tape - should only be specified first time
DFILE a,	set data tape to start of file a
PRDATA,	print data tape before starting to execute

See $DATA etc. in 5.2

5.14 Stimulus generation

SSINIT α β p q,

initialize stimulus generation routines
stimuli will be formed on a #α x #β retina
center of the retina (for rotations and dilations) will be at (#p,#q), where the top left corner is (1,1)
NOTE: coordinates on the retina run in "tabular" form, i.e. increasing to the right and down.
p and q may be omitted if they will not be needed.

SSCENT p q,

change center to (#p, #q)

SSPACK,

make sure stimulus is in packed form (#BSTIM)

SSUNPK,

make sure stimulus is in unpacked form ($BSRETA)

SSWAP,

interchange first and second working retinas ($BSRETA and $BSRETB)

SSOUT,

output current stimulus, according to current output modes

SSMODE ... ,

set output modes according to ..., which may consist of one or more of the following

OUT	write out on stimulus output tape (see SSOTAP, below)
OFF	print on offline printer tape
ON	print on online printer
OFFH	print heading only on offline printer tape
ONH	print heading only on online printer
DO	put stimulus onto perceptrons retina
PUNCH	punch stimulus on a binary card

NOTES: stimulus heading consists of class and name. It will be printed alone for OFFH, or preceding picture of stimulus for OFF. (Will not be printed twice if OFFH and OFF are both on)

Format of stimulus on stimulus output tape is thus:

first word, left 18 bits are class bits
tag is number of words in name (next three bits)
right 15 bits are key number (normally zero)
then follows stimulus in packed form,
then follows name

stimuli to be punched must be small enough to fit on one card,
i.e. $\leq$ 540 points
format is same as on tape, starting with second word of card.

NOTE: output mode is set to OUT by SSINIT

SSMON ..., specified output modes are turned on, in addition to any already on

SSMOFF ..., specified output modes are turned off,

SSOTAP σ , stimulus output tape is set to σ (a standard tape designator)

SSPRT xxx, print stimulus with heading

xxx may be omitted to print offline
ONLINE to print online
BOTH to print on and offline
n to print on "unit n"
#v to print on "unit #v"

where unit 1 = offline
unit 2 = online
unit 3 = both
unit 0 = neither

SSPRTH xxx, as above, but print heading only

SSPRT (eα α β) xxx, print stimulus with dimensions #α x #β, found in eα (packed stimulus is in eα: name and class are in $STNAME and $CLASS, as usual.)
NOTE: this may be used even if stimulus generation routines are not initialized (SSINIT)

SSDEFS α, save current stimulus as stimulus number #α

SSN α, put stimulus previously saved as number #α on the retina
NOTE: saved stimuli are lost if SSINIT is used again

SSR α,	choose m at random, $1 \leq m \leq \#\alpha$ and put stimulus number m on the retina
SSHBAR α,	set stimulus to a horizontal bar, of width $\#\alpha$, at the top of the retina (width will be 4 if α is omitted)
SSVBAR α,	set stimulus to a vertical bar, of width $\#\alpha$, at the left of the retina (width will be 4 if α is omitted)
SSZ,	set stimulus to zero (i.e., all points off)
SSRAND p,	form a random dot stimulus with N active points, where N = integer part of #p + size of retina x fractional part of #p (e.g., p = 5 means 5 points active p = 0.23 means 23 percent of retina active p = 7.1 means 10 percent + 7 more points) if p is negative, the retina will <u>not</u> be cleared before placing new points
SSRW eα p,	form a stimulus by a random walk process, governed by a set of frequency distributions in eα. the number of active points is determined by p, as in SSRAND if p is omitted, same number as last time will be used if eα and p are both omitted, the same distribution tables will also be used.

The random walk process is governed as follows; a point can be chosen in one of nine ways.

1 moving down and left from the previous point (DL)
2 moving left from the previous point (L)
3 UL
4 U
5 UR
6 R
7 DR
8 D
9 RAN (chosen at random from previously unused points)

For each possibility a frequency distribution is provided to govern the next choice.

The first point is chosen at random; from then on, after each point

is chosen, the next point is chosen among the permissable possibilities (i.e. those which do not lead to points previously chosen) according to the appropriate distribution. If there are no permissable possibilities with non-zero entries in the table, then the next point will be chosen at random.

The retina is considered torroidal; i.e., a point on the left edge is considered to be immediately "right" of a point at the right edge, and similarly for up and down. (This can be changed if desired, see below.)

The distribution tables are most conveniently set up by use of SSBIAS, as described below, in which case it is not necessary to know the details of the way the table is arranged. If for some reason it is desired to use some other way of setting one up, it should be arranged as follows:

> The first 9 words form the table to be used after possibility 1; the next 9 the table for 2, etc. Within each table, the 9 words give the (relative) probabilities of choosing the next point according to 1, 2, etc. Note that only the relative, not the actual values are relevant, so they may be scaled in any convenient way.

If p is negative, the retina will not be cleared before starting.

SSRW (H) eα p,	like before, but the retina will not "wrap around" in the horizontal direction
SSRW (V) eα p,	like before, but the retina will not "wrap around" in the vertical direction
SSRW (HV) eα p, SSRW (VH) eα p, SSRW (NT) eα p,	like before, but the retina will not "wrap around" in either direction
SSRWC eα p,	like SSRW, but the first point, instead of being chosen at random, will be chosen as if it were a continuation of the most recent use of SSRW or SSRWC
SSRWC (H) eα p, etc.	just what you expect

SSRWI α β γ,

this will cause the next use of SSRW or SSRWC to begin with the point #α, #β and proceed as if this had been reached by method #γ (see discussion under SSRW, above)

if γ is omitted 9 (random) will be used

if (#α, #β) is not permissable (i.e. it is outside the retina, or it is already active) then the first point will be chosen at random

- NOTE: SSRW, SSRWC, SSRWI and SSBIAS have not been completely tested. They may be working right, but more likely they are not.

SSBIAS eα $ ...,

form a set of distribution tables for SSRW etc. in eα

eα must be an extended operand of exactly 81 words

... consists of one or more distributions; each distribution consists of a series of codes enclosed in parentheses followed by a series of standard operands

the codes in the parentheses may be one or more of the following:

U	this distribution is to apply to the case where the previous point was obtained by moving up
UR or RU	ditto, up and right
R, DR or RD, D, DL or LD, L, UL or LU	similar
RAN	this distribution is to apply to the case where the previous point was chosen at random
ALL	this distribution is to apply to all cases except the ones specifically mentioned (U, UR, etc.)
REL	meaning of this is explained shortly

the values of the standard operands determine the actual distribution in one of two ways, as follows:

absolute (REL does not appear within the parentheses) the values give in order the relative probability of moving DL, L, UL, U, UR, R, DR, D, random.

relative (REL does appear within the parentheses) the values give the probabilities, not for an actual direction, but for a change of direction from the previous, in this order:

back left, right angle left, half left, straight ahead, half right, right angle right, back right, random choice.

In either case, zero values at the end of the list can be omitted

SSBIAS eα n,

Set the distribution tables in eα to one of several possible standard forms, according to the value of n, as follows:

n	Standard distribution
1	all choices equally probable
2	all choices equally probable, except random choice only made when forced
3	very high probability of turning sharply back on itself (3600-60-1-1-1-60-3600)
4	moderate bias towards turning sharply back on itself (5-3-1-1-1-3-5)
5	straight lines. Once started it can only continue in same direction until blocked, then make a new random choice
6	line segments. Like 5, but there is a 1 in 6 chance of making a new random choice, even if not blocked
7	zig-zag lines. A strong probability of continuing straight ahead, (1-3-5-50-5-3-1)
8	after random choice, must move horizontally or vertically: other distributions not affected (see note below)
9	after random choice, must move diagonally; other distributions not affected (see note below)

NOTE: In either form of SSBIAS, it is possible to change some parts of the distributions tables, leaving the others as before (e.g., a distribution for horizontal and vertical lines could be formed by

SSBIAS eα 5, SSBIAS eα 8,

while one for short vertical segments could be formed by

SSBIAS eα 6, SSBIAS eα (RAN) 0 0 0 1 0 0 0 1,

SSPT $e\alpha_1$ $e\alpha_2$ ...,

the numerical values in $e\alpha_1$, $e\alpha_2$, etc. form a list of numbers; each pair of numbers is considered as the coordinates of a point in the stimulus, and this point is made active (regardless of its previous status) (top left corner is (1,1))
NOTE: the stimulus is not cleared before starting.

SSPT * $e\alpha_1$ $e\alpha_2$ ...,

same as above, but each single number represents a point (starting with top left corner as 1 and working from left to right, top to bottom)

SSPTB α β,

in all future uses of SSPT, (x, y) will be changed to (x + #α, y + #β)

SSPTB α,

in all future uses of SSPT* N will be changed to N + #α

SSTRAN α β,

translate current stimulus #α places horizontally and #β places vertically. Positive directions are right and down. Translation takes place on a torroidal retina.

SSTRAN (H) α β,

as above, but retina does not "wrap around" in the horizontal direction

SSTRAN (V) α β,

as above, but retina does not "wrap around" in the vertical direction

SSTRAN (HV) α β,
SSTRAN (VH) α β,
SSTRAN (NT) α β,

as above, but retina does not "wrap around" in either direction

SSROT α,

rotate current stimulus by #α degrees (positive is clockwise)
retina is considered non-torroidal
rotation is about the center as specified in SSINIT or SSCENT

SSDIL α β,

perform a dilation by #α in the horizontal direction and by #β in the vertical (in other words, transform by

$$x' = \#\alpha * x$$
$$y' = \#\beta * y \quad)$$

retina is considered non-torroidal:
center is as specified in SSINIT or SSCENT

SSLIN α_1 β_1 γ_1 α_2 β_2 γ_2

perform a general linear transformation according to the formulae:

$$x' = \#\alpha_1 * x + \#\beta_1 * y + \gamma_1$$
$$y' = \#\alpha_2 * x + \#\beta_2 * y + \gamma_2$$

retina is considered non-torroidal; origin is as specified in SSINIT or SSCENT

NOTE: SSROT, SSDIL, and SSLIN have not been completely tested, but are believed to be working right.

SSPOS, — set class of current stimulus to positive

SSNEG, — set class of current stimulus to negative

SSCLAS α — set all 18 class bits of current stimulus to the low order 18 bits of $\#\alpha$

SSNAME ..., — set the name of the current stimulus to the name specified in ..., which may consist of one or more of the following types of pieces:

i) a string of BCD characters, not including slash comma or blank (nor, of course, ')

ii) /* followed by an extended operand, which will be used as a piece of BCD.

iii) / followed by a standard operand, whose numerical value will be used. Value is an integer, positive; significant digits only will be used, or low order 6 digits if $\geq 10^6$.

..., may have up to 15 such pieces

SSNAME * ..., — as above, but the specified name will be added on at the end of any name already attached to the stimulus

SSXNAM ..., — if sss is the name now attached to the stimulus, and xxx is the name specified by ... as above, the name of the stimulus is changed to xxx(sss)

NOTE: Stimulus name may be up to 42 characters long; if any of the above phrases attempts to make name longer, the 42nd character will be made $ and any further additions will be ignored.

SSKEY α — key number is set to $\#\alpha$ (next SSOUT will use this number, then reset to 0, or to whatever basic value has been given by SSKEYP, see below)

SSKEYP α, — key number and basic key number are set to #α (basic key number may be changed by another SSKEYP or is automatically set to zero by SSINIT. It will be used for all following stimuli, unless overridden for a particular stimulus by SSKEY)

NOTE: KEY is not saved with stimulus by SSDEFS

General comments on stimulus generation

Stimulus points are numbered (1,1) (1,2) (1,3) ...
(2,1) (2,2) (2,3).:::
.. ..

When the stimulus on the retina is in unpacked form, point (h,v) is in $BSRETA (v,h), which is positive if the point is on, negative if off.

When it is in packed form, it is in $BSTIM, in the usual form. In either case, the left 18 bits of $CLASS hold the class bits, and $STNAME holds the name. (If these are not changed when a new stimulus is put on the retina they will not be automatically cleared, but will hold their previous values.)

When a single classification is used, it is most convenient to use the leftmost class bit, which is the sign bit of $CLASS. When more than one are needed, if the leftmost ones are used it is easiest to use UNPACK to put these in the proper form for controlling reinforcement in REINF phrases.

STIMULUS on tape is in this form

1st word, left 18 bits	class bits
next 3 bits	number of words in name
right 15 bits	key number
2nd word, etc., packed stimulus followed by name	

IMPORTANT NOTE: IF ANY OF THE FOLLOWING PHRASES ARE TO BE USED, ONE MUST TAKE NOTE OF APPENDIX II.

SSTσ,

Read in a stimulus from tape unit #σ. It is assumed that the stimulus is in the same form as stimuli written by SSOUT.

SSC,

Read in a stimulus from the DATA tape. The stimulus may be in one of two forms. 1) Binary form. Each card must have a 7 and 9 punched in column 1. The stimulus is punched starting in column 4 with each place on the card representing one unit. The first column of the stimulus runs along the top row of the card, the next column along the next row, etc. If the horizontal dimension of the stimulus is more than 12, then additional cards must be used. The vertical dimension may not exceed 69. Column 2 (reading top to bottom) and the top 6 bits in column 3 may be used for class bits. (If the stimulus occupies two or more cards, class bits must be on the first of these.) A name may be given to the stimulus by preceding the above described card(s) with a card containing the word NAME followed by the desired name (1 to 6 letters). 2) Integer form. In this form the stimulus is specified by a series of integers, each giving one of the active points (the points are numbered sequentially starting with one in the top left corner and running left to right, top to bottom.) Do not use any commas or other separating marks. The number of the last active point should be followed by a zero. The integers giving the active points may optionally be preceded by one or more of the following:

i) NEG, indicating that the stimulus is to be classified negative.

ii) CLASS, followed by a 6-digit octal number, indicating that all 18 class bits are to be set to the binary equivalent of this octal number.

iii) NAME, followed by an alphabetic name of 1 to 6 characters which will be used as the name of the stimulus.

NOTE 1. There is no need to specify in the SSC phrase which of these forms is used, as it is capable of recognizing them for itself.

NOTE 2. When a stimulus is provided in binary form the card or cards on which it is punched belong to it alone. No part of any previous or following stimuli may be punched on one of these cards.

NOTE 3. The cards carrying these stimuli should be put into a file on the data tape, as in Section 5.13, and if necessary an appropriate DFILE should be given before using the first SSC phrase.

5.15 Counters and Skipping

This section provides a complete logical description of the action of counters, automatic section repetition and skipping. The following primary codes are covered in this section.

EXTEND
BIND
CTOP
SKIP
DO

The following points should be noted:

i) This description is certainly not so transparent as the more intuitive one in section 4.5, which should be preferable for casual use. However, this one should cover all situations, whereas the one in section 4.5 may not be precise in certain especially complicated ones.

ii) The following description parallels closely, in many respects, the actual internal processing of counters and SKIP phrases. This, however, is a matter of convenience; it should not be taken as a precise description of the internal workings of the system.

I. GENERAL

The input deck is divided into sections by the primary code END occurring in a main sequence. To be more precise:

i) every DEFINE is matched with the first following END (except for the form "DEFINE A * B,")
ii) every unmatched END marks the end of a section
iii) if an end of file is found on the input tape, a dummy END is supplied at this point.

Several "dummy" sections may be added to those actually appearing in the deck, as mentioned in several places below.

The last section is called the skip section. More precisely:

i) If any section contains either of the primary phrases SKIP or DO, that section is designated the skip section, (and the deck is considered to finish at the END of that section).

ii) If not, the deck is considered to end at the End of File, and a dummy section, containing nothing but an END, is added after the last real one, and is designated the skip section.

Immediately before the skip section, a dummy section is inserted, called the final section.

The skip section is executed first, then the others in order of appearance. This order may be modified by automatic repetition, and/or skipping, as below. However,

i) Neither the skip section nor the final section can ever be repeated or skipped.

ii) The skip section is always executed first.

iii) The execution of the final section always means the end of the job.

II. INTERPRETATION OF COUNTERS

A section which contains perceptron specifications is designated a P-section. A section immediately following a P-section is a Q-section. NOTE: The heart of any job will be, of course, a pair of P and Q sections--or possibly in exceptional cases, several such pairs. All other sections are merely auxiliary to these. Therefore, it should not be surprizing that these sections play a key role in the automatic repetition mechanism. Note also that it would be meaningless to have two consecutive P-sections, or to have a P-section with no Q-section following. Thus

i) No section is both a P-section and a Q-section.

ii) The skip section may be neither a P-section nor a Q-section.

During interpretation of the deck, a table is kept of counters. Each counter has associated with it two sections, known as its FS (first section) and LS (last section) and certain other information, as implicit in the discussion below. During the processing of a section, if a counter is found in any context, it is entered in the table (if not already there) and its FS is set to the first of the following sections

i) current section

ii) previous section if current one is type Q

iii) first of FS of lower numbered counters already in the table

If "EXTEND An," occurs, then table entry for An is marked "extended".

If "BIND An," occurs, then table entry for An is marked "bound".

At the end of any section other than a P-section, the following actions are taken:

1. If it is a Q-section, all table entries are marked "closable".

2. The lowest numbered counter in the table is found, and if it is marked closable and not marked extended, the following are done:

   i) If any higher numbered counter in the table is marked bound, this one is marked bound also.

   ii) The LS of this counter is set to the section just finished.

   iii) This counter is removed from the table. (If the same An is again encountered in later sections, it will therefore be reentered in the table, and considered as a different counter, with its own FS, etc.)

3. If the lowest numbered counter in the table was not closable, or was extended, or if there were no entries in the table, go on to 4. Otherwise go back to 2 and process the new lowest.

4. Remove the "extended" marks from all entries still in the table (if any). ("Bound" and "closable" marks are not removed.)

Now for each counter we have a scope; which comprises the sections from the FS to the LS inclusive. (Note the same A-number may represent different counters if the scopes are distinct.) If the scope of a counter includes any appearances of a lower numbered counter, the above rules insure that it must include the entire scope of that counter; in this case we say the higher one contains the lower one. A counter, a, immediately precedes another, b, if a is contained in b, but there is no third counter contained in b and containing a.

The range of a counter is the maximum of the following:

i) the length of the longest list in a $ (An,...) appearing in the scope, if any

ii) the length of the longest list in a (An,...) appearing (in a LINE or HEAD phrase in) the scope, if any

iii) the largest value m appearing in CTOP An m, if any.

The case count of an unbound counter is the sum of the total case counts of the unbound counters which immediately precede it, or is 1 if it contains no other unbound counters.

The total case count of an unbound counter is the product of its case count and its range. Case count and total case count are not defined for bound counters.

The following "dummy" sections are added.

After the LS of any counter, we add a closing section (CS) for that counter. If the same section is LS for more than one counter we add a separate closing section for each, putting that for the lower (inner) counter first.

After the LS of an unbound counter which contains no other unbound counter, and before its CS, we put a "case end" section.

Thus when interpretation of the entire deck is complete, each counter has the following

bound/unbound

FS - first section

CS - closing section

R - range

CC - case count
TCC - Total case count } unbound only

One counter is said to be higher than another if its CS comes later. (Note that this is related to the original numbering of the counters.) Note that the original numbers, the "extended" marks, and the "closable" marks have fulfilled their function and are no longer relevant.

The above information is now permanent, that is, it is not changed during execution. Each counter also has the following, which may change during execution, as discussed later:

An unbound counter will be active or inactive.

Bound counters, and unbound counters while active, have a value, V. This is an integer, with $1 \leq V \leq R$.

III. SKIP-LIST

The skip-list is a sequence $S_1, S_2, \ldots$, where each S_i may be 1 or 0. (This is not the actual form the list is kept in, but for a logical description of its functioning, this is convenient.)

Originally, each S_i is 0; during the skip-section, a position counter, p, is kept: this is initially set to 1.

For any integer n, g(n) means the least integer $\geq n$ for which $S_{g(n)}=0$, or g(n) = n if all $S_i = 1$ for $i \geq n$

To say the position is advanced-a-skip means

i) q is set to g(p)+1

ii) $S_{g(p)}$ is set equal to 1

iii) p is set to q

To say the position is advanced-a-do means

i) p is set to g(p)+1

SKIP
DO These primary codes may be followed by one or more items, each of which may be one of the following

i) Any standard operand, except one involving a variable named SKIP, DO or REST

ii) One of the subcodes SKIP, DO, or REST

SKIP or DO will be considered the start of a new phrase, even though the current one is not terminated by a comma. The effect of

SKIP $item_1$ $item_2$ ...,

is the same as

SKIP $item_1$, SKIP $item_2$, ...

and similarly for DO

SKIP α, the position is advanced-a-skip m times if $\#\alpha = m>o$ or is advanced-a-do m times if $\#\alpha = -m<o$

SKIP REST, S_i is set equal to 1 for all $i \geq p$; then p is set to 1

DO α, the position is advanced-a-do m times if $\#\alpha = m>o$ or is advanced-a-skip m times if $\#\alpha = -m<o$

DO REST, p is set to 1

At the conclusion of executing the skip-section, the skip list has attained its final form.

IV. EXECUTION--BOUND COUNTERS

At the start of execution, the value of all bound counters is set to 1.

When the CS of a bound counter is executed, the following action is taken:

i) If V<R, increase V by 1 and go back to FS of this counter. Otherwise

ii) Set V to 1 and go to next section.

V. EXECUTION--UNBOUND COUNTERS

This part applies to unbound counters only and any reference to "counter" means "unbound counter".

During execution, an integer called the current case number, CN, is kept.

At any time, we say the skip-list allows k skips if

$S_i = 1$ for $i = CN, CN+1, \ldots CN+k-1$

At the start of execution, each counter is marked "inactive" and CN is set equal to 1

Before any section is executed, we do the following:

i) If there is no inactive counter for which this is FS, execute the section. Otherwise

ii) Find the highest inactive counter for which this section is FS and do the following:

a) Let t be the TCC of this counter. If the skip list allows t skips, increase CN by t, and go at once to the section after the CS of this counter. Otherwise

b) Mark this counter active, and set its value to 1

c) Let k be the CC of this counter. If the skip list allows k skips, increase CN by k, increase value by 1, and do c) again. Otherwise

d) Repeat the above steps, starting at i).

The action taken by a "case end" section is

i) Print "case finished" message (stating number of case just finished, number of cases done on this run and instructions for restarting from this point).

ii) Increase CN by 1

The action taken by a closing section is

i) If the value of the counter in question equals its range, mark the counter inactive and go on to the next section. Otherwise

ii) Increase value of counter by 1

iii) Let k be the CC of this counter. If the skip list allows k skips, increase CN by k and repeat the above steps starting at i). Otherwise

iv) Go back to the section which is FS for this counter.

VI. FINAL COMMENTS

If a counter is used within a subsequence definition, then for the purposes of the above discussion it is considered to appear in the section where the definition appears. Normally, one should not use a counter in a subsequence. Especially, one should not use a counter in a subsequence and then use the subsequence in a section other than the one where it was defined, unless (1) there is a particular reason for it, and (2) one has given careful thought to the exact consequences.

Under no circumstances should a counter appear in the skip-section.

5.16 Miscellaneous Codes

ALWAYS ...,
ALWAY ...,

This is used to specify one or more variables and/or blocks which are to be available to all sections. ALWAYS may be followed by

i) name(s) of variable(s)

ii) name(s) of block(s) each followed by dimensions (which must be explicit integers).

Block names specified in ALWAYS must never appear in a BLOCK or LIST phrase. Values placed into an "always" variable or block by one section will be available to the following sections. Two or more ALWAYS phrases may be used, if desired, but all must come at the very beginning of the deck. (Or more exactly, they must come i) before any other use of the names listed, ii) before any use of a counter [$(An].)

CODPRX,

When the deck is interpreted, information about the interpreted form is printed out. This is only of interest to persons familiar with the workings of the program, and then, only in special situations. Most of this printing will be omitted if CODPRX, is used.

CODPRT,

Resume printing information suppressed by preceding CODPRX.

LARGE,

Normally, sections other than the perceptron testing section are not expected to use large amounts of space, and this is taken advantage of to improve the timing slightly. In the exceptional case, the phrase LARGE, may be used in any section to indicate that that section needs the maximum possible amount of space.

MAXX n,

The program needs to know at the end of interpretation, the maximum X-number used. If all $Xn used (or implicit references to Xn, as in e.g., A BYPAS subcode in a LINK phrase) are explicit integers, the program can determine this itself. But if we use, e.g., $X I, then we must specify this largest number--and this is done by MAXX n,.

RANGE,	Normally any attempt to use a subscript value ≤ 0 or > maximum value given in BLOCK/LIST phrase, causes the program to stop with an error message. If RANGE, appears anywhere in the deck, this check will be omitted.
UNDEF name,	name may be the name of any previously defined subsequence. The effect of this code is to increase the available space during the interpretation of the deck. It should only be used after the end of of the last section which makes any reference (direct or indirect) to this subsequence.*
DUM,	take a core dump when this phrase is interpreted.
DUMX,	take a core dump when this phrase is executed.
ERR,	after this code is interpreted, any interpretation error will cause a full core dump to be taken
PCTAPE,	if this code appears anywhere in the deck, the control tape will be printed out before execution

NOTE: DUM, DUMX, ERR, PCTAPE are intended for use only by someone making changes to the system.

PRINT α_1 α_2 ...,	$\#\alpha_1$, $\#\alpha_2$ etc. are printed as decimal numbers.
PRINTØ α_1 α_2 ...,	$\#\alpha_1$, $\#\alpha_2$, etc. are printed as octal numbers
PRINTF α_1 α_2 ...,	$\#\alpha_1$ $\#\alpha_2$ are taken to be in floating point form, and are printed as floating point decimal numbers.

NOTE: PRINT, PRINTF, PRINTØ are primarily intended as debugging aids; they provide a somewhat simpler way to print out a list of numbers than LINE does, but allow no control over the format.

---

*WARNING; UNDEF may not be working correctly.

BLANK PAGE

6.1 Appendix I. System Operation and System Tape Preparation

This appendix discusses details of operation of the simulator system. Sections 1 and 2 provide the minimum information one must know to make use of it.

Throughout this appendix, to avoid possible confusion, we will use the words "SIM" and "program" in the following sense only. SIM will refer to the 7090 perceptron simulator system which processes the user's input deck and performs the desired experiments. Program will refer to the input deck made up by the user, in accordance with the main body of this report, to specify a particular experiment or set of experiments.

1. Normal Operation of SIM

SIM exists in the form of a symbolic (FAP) deck; a binary deck produced by assembling the symbolic deck; and a system tape (sometimes referred to as the SIM-tape) produced by executing the binary deck.[1] In normal operation, only the SIM-tape is used, not the symbolic or the binary decks. During operation the system will first read the entire program deck from the input tape (A2) and interpret it; during this process it will write on another tape (referred to as the control tape) a translated form of the program and will also copy from the SIM tape onto the control tape whatever portions of SIM will be needed when the program is being executed. Once this has been completed, the SIM tape is no longer used, the actual execution being governed by the control tape together with a small routine which remains permanently in the machine.

SIM is designed to run either alone or under the FORTRAN Monitor System.[2] To run under the Monitor, a job is made up consisting of a

[1] The SIM tape is not simply a copy of the binary deck on a tape.

[2] The testing of the program has taken place using the NYU version of the FORTRAN Monitor System. There is available with the program decks an editor deck which will create a copy of this version, starting from any other version. However, SIM makes no direct use of the facilities of the Monitor System, so that it will probably run equally well under another version, or alone.

loading program (discussed in more detail later) followed by the program deck. The loading program contains a special card telling which tape unit holds the SIM-tape; so it can be mounted on whichever unit is most convenient. When SIM is run alone, the SIM tape is mounted on A1, the program deck placed on A2, and the load tape button pushed.

We must discuss briefly the way in which SIM handles tape unit assignments. When it is first loaded into the machine, it checks all tape units to see which ones are in ready status, and makes a table listing these as "available" and the others as "unavailable". The unit on which the SIM tape itself is mounted is marked as "in use," as also are tapes A2, A3 and B2 (which are used for special purposes). If the SIM tape is not mounted on tape A1, then we must be operating under the Monitor, in which case A1 is reserved for the monitor system tape, so this also is marked as "in use". Whenever another tape unit is needed (e.g., for the control tape) the system will choose a unit from those marked as "available", mark it as " in use", and use it for the required purpose. Also, during the interpretation of the program, if tapes are specified in the program using the form Tn, then the system will choose the actual tape unit to be used from the "available" ones. If at any time all available tapes have been assigned, and another is needed, the system will print a message to the operator asking him to make more tape units available, if possible, or otherwise to terminate the job. If a specific tape unit is mentioned in the program (e.g., A5, B3, etc., rather than T1, T2, etc.) then the system will check whether that tape is actually available. If not, the system will print a message to the operator asking him to make that unit available.[1] If a specific tape,

[1] Specific tapes should not be used in the program unless there is a good reason for doing so; such a reason might be either (1) that information already existing on a tape is to be provided to the program, or (2) that a tape written by the program is to be saved and used for another purpose. In both of these cases it is necessary to know what tape unit is being used so that the appropriate instructions can be given to the operators. However, in the more usual situation that the tape is simply being used for intermediate results, there is no need for the user to know which unit will be used, and it is preferable to use the form Tn.

e.g., A5, is used in the program it is necessary to prevent SIM from assigning A5 to another use, e.g., for the control tape, before it encounters "A5" in interpreting the program deck. One way to do this would be to instruct the operator to mount the tape, but not to put it in ready status until a message requesting him to do so is printed. Another, probably more convenient, method is to place before the program deck a card reading "/SAVE A5".[1]

SIM can be divided roughly into three sections, referred to as setup, interpretation, and execution. Setup takes care of all details connected with the original creation of the SIM-tape, creation of a corrected copy of the SIM-tape, and other minor details which will be discussed shortly. It is governed by SIM control cards, which are distinguished by having a slash (/) as their first nonblank character, and which must all be placed before the first card of the program deck itself.[2] Normally there are no SIM control cards, and the setup section will pass immediately to the interpretation section. Interpretation reads and interprets the program deck itself and produces the control tape as mentioned above. The execution section contains various routines which are needed while the user's program is actually being executed (and which will be placed on the control tape where needed, as noted above).

The following is a summary of tape usage.

A1 Monitor system tape (or SIM tape when running independent of monitor)

A2 Input tape

A3 Output (offline printer) tape

[1]This is referred to as a SIM control card. It is not considered part of the program deck. SIM control cards will be discussed in a little more detail shortly.

[2]Note that it is impossible for the first card of the program deck to have a slash as its first nonblank character.

B2 Used by monitor when dumps are taken.
Also used by SIM (setup section) when original SIM tape is being created from binary deck (whether running under monitor or not).
Otherwise available for use by program.

any SIM tape (if running under monitor)

any control tape

any DATA tape, if used

also tapes required by program

2. Examples of Decks

To create a SIM tape from the binary deck, the following deck will suffice:

```
*ID etc   (monitor ID card, as used at this computer.)
*       XEQ
*       PLEASE MOUNT TAPE 278 ON UNIT A5
*
*
*       PAUSE
(binary deck of SIM)
/SIMTP A5 REW
/EXIT
```

To run a program, using a previously written SIM-tape, the following deck will suffice:

```
*ID etc   (monitor ID card)
*       XEQ
*       PLEASE MOUNT TAPE 278 ON UNIT A5
*
*
*       PAUSE
(binary deck of SIM-loader)
*       DATA
A5
```

Note 1. In each of these the comments are, of course, examples; the number 278 should be replaced with the appropriate number or other identification of the desired tape (in accordance with the usual proceedures at the computer installation where the job is being run).

Note 2. The tape unit need not be A5. Any unit (not already in use; see above) may be used; the appropriate number should be used in place of A5 wherever that is used above.

Note 3. It is not necessary to use the same tape unit when the SIM-tape is being created and when it is being used.

To create a SIM tape from the binary deck when not running with the monitor, use the following deck:

```
(absolute binary loader)
(binary deck of SIM)
/SYSTP A5 REW
/EXIT
```

To run a program when not running with the monitor,

i) place SIM tape on A1
ii) place program deck on tape and mount it on A2
iii) press load tape button.

The examples given above should be sufficient for most normal use of SIM.

3. Internal Subsequences

If certain subsequences are used very frequently, it is possible to have their definitions incorporated into the SIM tape, in such a way that it is unnecessary to supply a copy of the definition with each program which uses them. This is particularly useful in the case of certain features which are described in the main body of this manual as if they were primary codes, but are in actual fact implemented by means of subsequences, (e.g., see Appendix II).

Each internal subsequence may have associated with it one or more names (of 1-6 letters) called its use-codes. If it has no use-codes, it will automatically be interpreted every time a program is; however, if it has, it will only be interpreted if its use-code (or at least one of its use-codes if there are more than one) is mentioned on a special SIM-control-card which will be discussed shortly (/USE). It is, unfortunately, not possible for SIM to determine whether or not the subsequence will actually be needed and decide on that basis

whether to interpret it; it must be already interpreted before the first use is made of it. There is nothing wrong with having a subsequence interpreted which is never used. This does not cause any logical problems, and does not waste any space or time at execution time. However, it does make extra demands on space at interpretation time, which should be avoided if possible.

To add new internal definitions, we make up the cards bearing the definition just as usual. In front of them we place a card reading

/INPUT (followed by use-codes, if any, to be associated with these sequences)

after the subsequence(s) we put a card with

$$$ (in columns 1, 2 and 3)

This is all placed before the /SYSTP card in the deck used to create the SIM-tape.[1] Two or more such groups with different use-codes may be used if desired.

If a program is to use certain internal subsequences which have use-codes, the program must be preceded by one or more cards reading

/USE (followed by the use-codes of the desired sequences)

4. SIM-control cards

Besides the SIM-control cards already discussed (/SYSTP, /EXIT, /INPUT, /USE, /SAVE) there are a few others which we will now discuss.

All SIM control cards obey the following rules:

i) The first non-blank character on the card is a slash (/)

ii) The / is followed with no intervening blanks by a word of from 1 to 5 letters, which identifies the particular type of control card (/SYSTP, /EXIT, etc.)

[1] If a SIM-tape has been previously created, it is not necessary to start with the binary deck again; a new tape can be made starting with the old one, as will be explained later.

iii) Unlike the cards in the program deck itself, a SIM-control card ends at column 72; it does not carry over onto the next card. (Neither automatically, nor by use of the continuation character, ')

iv) Except for the above, the cards conform to the rules on page 5.1.1

Many SIM-control cards call for a tape unit to be specified (e.g., /SYSTP, /SAVE, etc.). On such cards, the following also hold.

v) The tape unit is indicated by the channel letter (A or B) followed by the unit number (1 to 9) (e.g., "A5", "B6", but <u>not</u> "5A")

vi) The tape unit may optionally be followed by one or more of the following; REW RUN WEF BSF SKIP which may be used to position the tape if this is desired. The actions are as follows

REW	rewind the tape
RUN	rewind and unload the tape
WEF	write an endfile on the tape
BSF	backspace the tape one file
SKIP	move the tape forward one file

(Note for example the use of REW in the first example under 2 above, to insure that the tape is rewound before SIM is written on it.)

The various SIM-control cards are as follows.

/SAVE followed by a tape unit. This indicates to SIM that the specified tape unit is going to be used for some special purpose and is not available for assignment by SIM.

/INPUT discussed in detail above

/USE discussed in detail above

/RUN this is used to control certain options which may be chosen during the interpretation and/or execution of the program; it will be discussed in more detail later.

These four types of SIM-control card may come in any order, but they must all come at the very beginning, before any of the others discussed below.

/SYSTP followed by a tape unit. This may be given when SIM has been loaded from the binary deck, and indicates on which tape unit the SIM tape is to be written. If /SYSTP is not given, SIM will choose an available tape to write it on.

/COREC optionally followed by a tape unit. This may be given when SIM has been loaded from a previously written SIM-tape, and indicates that a new SIM-tape is to be written. (If a tape is specified, it will be written on that unit, otherwise SIM will choose one.) If /COREC is not given, SIM will proceed immediately to the interpretation section; so the program should follow immediately.

/EXIT When a new SIM-tape has been written (either at original loading from the binary deck, or by means of /COREC) SIM will look for a /EXIT card. If it is there, this will be the end of the job. If not, SIM will rewind the newly written SIM-tape and load the (possibly changed) version of SIM which is on it.

/UNSAV followed by a tape unit. If /EXIT is not given, before the new SIM-tape is loaded, one or more /UNSAV cards may be given to specify that certain tape units previously in use are now available again. (There is no point to this if the /EXIT is given, as when the tape is reloaded at a future time, the available-tape-table will be made up anew; but when the new SIM-tape is loaded by the previous version of SIM which has just written it, the current table will be given to it.)

On page 6.1.11 is a simplified flow-chart of the SETUP section which shows the relationship between the various types of SIM-control cards.[1]

5. /RUN cards.

These SIM-control cards govern certain options which take effect during the interpretation and/or execution of the program.

Note that if these cards are used before the writing of a SIM-tape (either an original or a CORECted copy) then they will modify SIM as it goes on the tape and hence will affect any programs run with that tape; but if they are used when the tape is loaded to run a program, they will of course affect only that particular program. (This comment applies also to /INPUT and /USE cards.)

During the execution of a program, any use of a subscripted block or list name will be checked to see if the values used for the subscripts exceed the maximum given in the BLOCK or LIST phrase; if so, the program will be stopped and an error message given. It is conceivable that in special circumstances we might wish to allow this to happen. The check can be prevented by

/RUN RANGE

The use of <u>all</u> internal definitions can be suppressed, regardless of what use-codes and /USE cards are involved, by

/RUN INTER OFF

When internal subsequences are used, the "cards" with the definitions are not printed. If we wish them to be, we use

/RUN INTPR

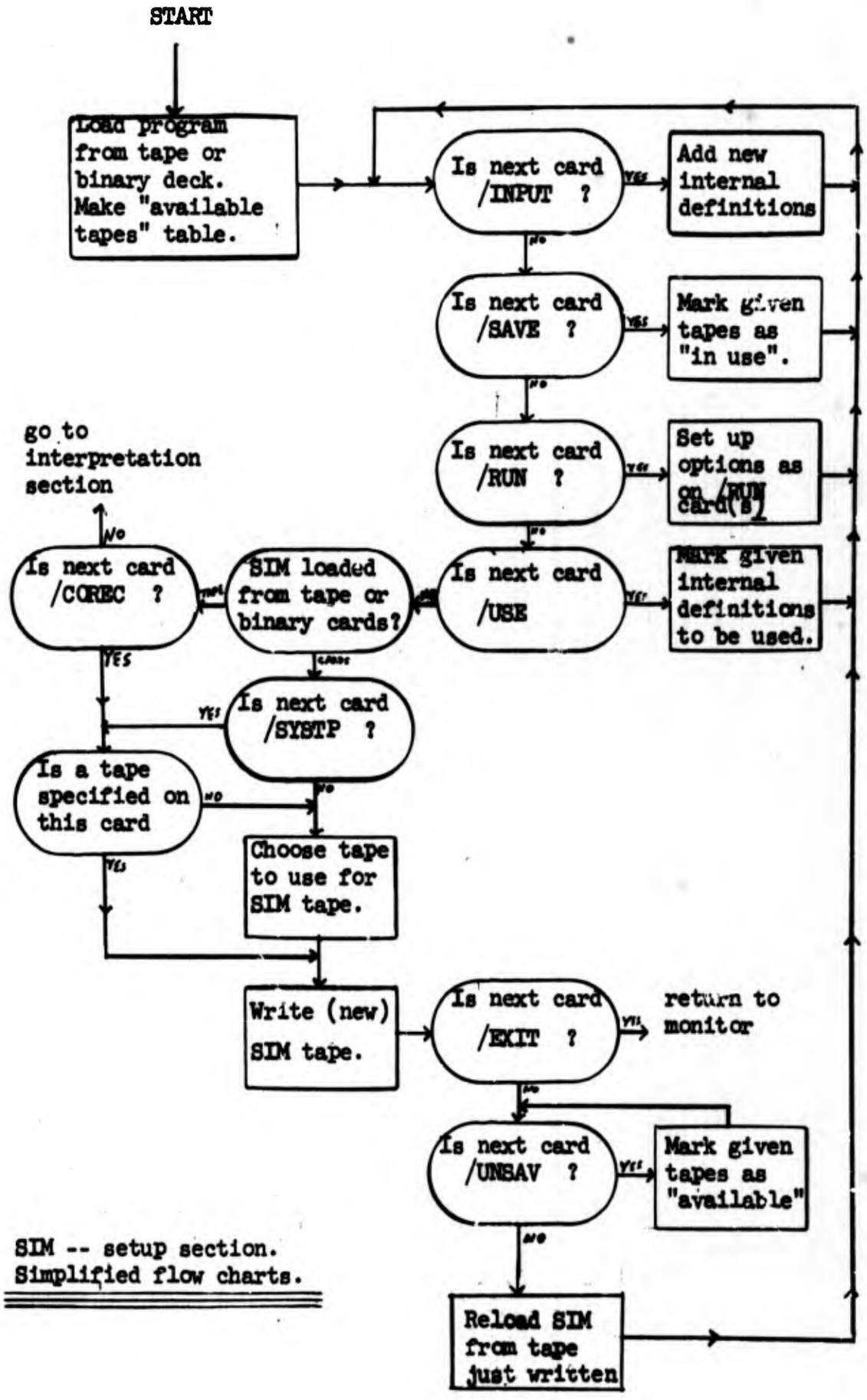
6. Example

Say that we have a SIM tape, and we wish to make a new one with some new subsequences we have made up added as internal definitions. Say we have two groups of subsequences, the first of which we wish controlled by the use-code "BABY", and the second to

[1] There are other types of SIM-control card, which are not discussed or shown on the flow chart. These are of use only if one is modifying SIM itself, in which case one can find out about them in the comments in the symbolic deck.

be used always. When we have written the tape we wish to run a program with it, using all the new subsequences, and printing all internal definitions used.

```
*ID  (monitor ID card)
*      XEQ
*
*     PLEASE MOUNT TAPE 278 ON UNIT A5
*      AND TAPE 279 ON UNIT B6
*
*
*       PAUSE
(binary deck of SIM-loader)
A5
/INPUT BABY
(first group of subsequences)
$$$
/INPUT
(second group of subsequences)
$$$
/COREC B6 REW
/USE BABY
/RUN INTPR
(program)
```



SIM -- setup section.
Simplified flow charts.

6.2 Appendix II. Special Subsequences

A small number of features discussed in the main body of this report (namely, the primary codes EC, EN, ES, PRECON, PRESNT, SSC, and SST) have not been incorporated into the system. Also, TEST and TESTP will not work correctly if any linkages in the perceptron have a nonzero delay. The subsequences given in this appendix will perform their functions as specified. Therefore, if these subsequences are added to each input deck, then the system will function as described in the main body of this report. It does not matter if the user has not read the sections on subsequences, as it is simply necessary to punch out the cards exactly as given here, and add them at the very beginning of each input deck. Alternatively, one may place these subsequences once and for all on the system tape. To do this, the deck given in Section 2 of Appendix I should be changed to the following.

```
*ID ... (monitor system ID card)
*      XEQ
*
*
*   PLEASE MOUNT TAPE 278 ON UNIT A5
*      PAUSE
(binary deck of SIM)
/INPUT
(subsequences)
$$$
/SYSTP A5 REW
/EXIT
```

Unfortunately, it has not been possible to check these subsequences in actual use; all but SST are extremely simple, and all have been carefully checked by two people, so they should be correct. However, it might be wisest to use them in a simple program where an error would be immediately detected before using them in a large production run. The subsequences are as follows:

```
DEFINE EC(ST),
     ($1, HIST,
          STIM $2, ACTIVE, REINF LAYER $LAYERS ECON $CLASS, )
     END,

DEFINE EN(ST),
     ($1, HIST, STIM $2, ACTIVE,
     REINF LAYER $LAYERS ECON $CLASS NONQUAN, ) END,

DEFINE ES (ST),
     ($1, HIST,
          STIM $2, ACTIVE, REINF LAYER $LAYERS SCON $CLASS, )
     END,

DEFINE PRECON (ST),
     ($1, HIST, STIM$2, ACTIVE $LAYERS+-1,
     REINF UPTO $LAYERS+-1 RPOS, )  END,

DEFINE PRESNT (T), HIST, STIM $1, ACTIVE, END,

DEFINE SSC, SSZ, ZERO $CLASS, SET $STNAML 7,
SET $1 $DATA,  IF ZERO $DATAQ+-2,
     OPEN A 100, IFN EQUAL $1 $(H*)NAME*,
          OPEN B, SET $2 400000,
               IF EQUAL $1 $(H*)CLASS*, SET $2 $DATA,
               ZERO $3,
               (6,ASHIFT 3 $3, DIV$2 10 $2 $4, ASHIFT -15 $4,
                    ADD $3 $4, )
               SHIFT -18 $3 $CLASS, JUMP 1, CLOSE B,
          FILL $STNAME  $DATA  $(H*) *  $(H*) * $(H*) *
               $(H*) *  $(H*) *  $(H*) *    ,
          SET $1 $DATA , IF ZERO $DATAQ+-2,  CLOSE A,
NOP, IF ZERO $DATAQ,
     (10C00, SSPT* $DATAI, IF ZERO $DATA, FINSUB, )
DIV $SSHDIM+35 36 $25,  BLOCK $B1 $25,
BLOCK $B2 27, MOVE $DATAB $B2,
SHIFT -12 $B2(1) $CLASS, AND $CLASS $(OE)777777,
SET $12 $SSHDIM, ZERO $BSRETB $2,
OPEN C 1000, MIN $11 12 $12, SBK $11 $12,  SET $13 $11+-12,
     SHIFT $13 $(OE)7777 $13,
     SET $21 2, SET $29 1,
     OPEN D *$31 $SSVDIM, SBSK $29,
          (, SHIFT -12 $22, JUMP 1, )
          (, SET $29 3, SET $22 $B2($21), ADK 1 $21, )
          AND $13 $22 $23, SHIFT $2 $23 $B1,
          OR $BSRETB ($31) $B1, CLOSE D,
     ADK 12 $2, IF ZERO $12, JUMP 3,
          SET $1 $DATA,  MOVE $DATAB $B2, CLOSE C, NOP,
(* $31 $SSVDIM, UNPACK $BSRETA($31) $BSRETB($31), )
ZERO $SSPAKQ,
LLCLR $B1,  LLCLR $B2,
END
```

```
DEFINE SST(T),
     READ $1 $2 $BSTIM $STNAME, SET $STNAML 7,
     AND $(OE)777777 $2 $CLASS,  SET $SSPAKQ 1, END

DEFINE TEST (S T ),
     IF ZERO $7, (SET $7 1, BLOCK $B1 3, COUNT $B1 INIT, )
     MAX $8 $1 -$1,
     ($8,  HIST,  STIM$2, ACTIVE, COUNT $B1, )
     IF POS $1, COUNT $B1 PRINT INIT,
     END

DEFINE TESTP,
     COUNT $$TEST$B1  PRINT INIT,
     END
```

6.3 Appendix III. Index of primary codes and subcodes.

This appendix lists all primary codes and subcodes; for each one, it gives all pages on which it appears. (Only references in the main body of the report, not in the appendices, are listed.)

A page number followed by E indicates that the code is used on that page in an example.

In some cases the same word is used in different contexts with different meanings; e.g., PRINT is a primary code, and also is used as a subcode in several different contexts (following the primary codes LINE, TAPE, or COUNT). These are not separated in the index; all uses of PRINT are given in a single list.

Names of variables, blocks, etc. used in examples are not indexed.

$-operands are similarly indexed in a separate list.

```
$(F )     3.5.2E, 3.5.8E, 4.11.2, 4.11.3, 4.11.4E,
          4.12.7E, 4.21.12E, 5.2.1,
$(H )     4.17.2, 5.2.3,
$(HS)     4.15.2, 4.15.2E, 4.15.3E, 4.17.2, 4.21.16E,
          5.2.1,
$(L       4.8.3, 5.2.4,
$(OE)     4.23.1, 5.2.3,
$(O)      4.8.1, 5.2.1,
$(T)      4.8.3, 5.2.1,
```

6.4 Appendix IV. The 7090 Computer

This appendix provides a few miscellaneous bits of information about the 7090 computer. It makes no attempt to be a complete description of the computer. SIM is intended to be useable by a person who is not familiar with the computer itself. However, in a few special situations some of the following facts may be useful.

The main avenues of communication between the computer and the outside world (that is, the user) are the card reader (which reads information punched on IBM cards), the printer, and the tape units (which write information on and read information from magnetic tape). One of the most important facts to know about these is that the operating times of the tape units are very slow compared to the computer itself; and the card reader and printer are in turn considerably slower than the tape units.

Once upon a time the card reader and printer were used for all input to and output from the computer, and the tape units were used during the course of the computation for storing intermediate information which was too bulky to be kept in the main computer memory. However, since the card reader and printer are so much slower than tape units, it is customary nowadays to use a special machine or a smaller computer to copy the input cards onto a magnetic tape. The large computer then reads the cards from the tape, and writes its output on another magnetic tape which will later be printed by the smaller computer. This is referred to as offline card reading and offline printing, respectively; the card reader and printer actually attached to the computer are referred to as the online reader and printer. SIM is designed to conform to these conventions. The online card reader is never used at all, and the online printer is used only if the user's program specially requests it.

The numberof tape units may vary from installation to installation. They are divided into two groups, referred to as channels.

The channels are referred to by letters, A and B.[1] The tape units on each channel may be numbered from one to nine. Hence a tape unit may be uniquely identified by a letter and a number, e.g., A5, B3, etc.[2]

Information may be written on the tape unit and later read from it. The information is arranged in groups called records, which in turn are arranged in larger groups called files by a special mark (called, of course, an end of file mark) which may be written on the tape. The tape can only be read and written while traveling forwards. It may be moved backwards over a certain number of records, a certain number of files, or rewound to the very beginning; but no reading or writing may take place during this backward movement. New information at any time may be written on the tape, either starting at the beginning or somewhere in the middle. If new information is written starting in the middle, any old information preceding this point will remain undistrubed; however, anything from this point will be lost. (In other words, it is possible to rewrite the last part of the tape but not to rewrite a small piece in the middle.) When a tape is rewound, once the instruction to start the rewind process has been given, the computer may proceed to do other things (either internal computation or manipulating other tape units) while the tape is rewinding.[3]

[1] A very few 7090 installations have more than two channels, but SIM cannot take advantage of this.

[2] A particular tape unit is always attached to a particular channel and this cannot be altered without physically changing the connecting wires, and so may be regarded as permanent. The number is determined by a dial on the individual tape unit, so that, for example, any tape unit attached to channel A can be made to be unit A3. However, this is of no real concern to the user, who may as well assume that the designations A3, B5, are fixed once and for all.

[3] It is also possible for the computer to read and/or write one unit in each channel and perform internal computations concurrently. However, SIM does not take advantage of this possibility, so there is no need to consider it further here.

The computer works internally with information stored in the form of 36-bit "words"; that is to say, each word has 36 positions, each of which can be zero or one. Such a word can be used to hold several different kinds of information, the difference being in the way the computer is programmed to use it. It can represent a number, by using a binary (or base two) number system. Conventionally, the first or leftmost of the 36 bits is used to indicate the sign of a number, a one indicating a negative number and a zero indicating a positive number. This number may be considered as an integer, in which case its magnitude may be anywhere from zero up to $2^{35}-1$. If we wish the numbers to have a fractional part, we may decide to allow the last so many bits to represent the fraction and the ones to the left of these to represent the interger; e.g., we may decide to allow the last 18 bits to represent a fraction, in which case we would speak of the number as being represented "with 18 binary places". Note that the computer does not know anything about this. It treats the numbers according to certain prescribed rules, and it is up to the user to interpret these rules in the light of the particular meaning he is attaching to the words. E.g., suppose we instruct the computer to multiply two numbers together. If we are thinking of these numbers as integers, then the result will be the product, also as an integer. However, if we are thinking of the numbers as both having 6 binary places, then the result would be the product, with 12 binary places. If we wish the result again to have 6 binary places, we would have to follow the multiply by a shift instruction to slide the result back 6 places to the right.

In the above, when we speak of the user, we mean the user of the computer, which in this case means the person making up the SIM system, and not the user of SIM. In most situations, these nasty details are taken care of within SIM and the user of SIM does not have to worry about them.[1]

[1]Unless he starts getting too fancy. See, e.g., the discussion in Section 4.8.

A 36-bit word may also represent a number in a different form, sometimes referred to as the floating point form. The exact details of this representation are of no concern. The general idea is that a floating point number carries the most significant part of the actual number (slightly more than 8 decimal digits) and an exponent to indicate its actual size. E.g., a number whose magnitude is approximately .01 could be kept in floating point form to an accuracy as high as 10 decimal places; whereas a number somewhere around 10^{10} could not be kept more accurately than the nearest hundred. Floating point numbers have several advantages over the other form, which are referred to as fixed point numbers. They can have a wider range of magnitudes, they make it easier to keep as many significant digits as possible, and the computer takes over the messy job of keeping track of the position of the binary point, thus freeing the user of the necessity of shifting products as in the previous example, etc. However, floating point numbers have the drawback that the computer is considerably slower in doing arithmetic with floating point numbers than with fixed point numbers. SIM makes use of fixed point numbers internally for all situations where a great amount of time may be used; and makes both forms available to the user of SIM (see Section 4.8).

Alphabetical information is customarily held in the computer using a code which uses six bits to represent each character (this allows $2^6=64$ possible combinations, which is sufficient to include the 26 letters, 10 digits, blank, and the special characters. $ ( ) . , * = ' + - /). Since there are 36 bits in a word, this means that one word may hold six characters.

While these are the most customary usages, any other interpretation may be placed on a word, provided the computer is programmed to treat it appropriately. The only other usage common in SIM is the so-called "packed" representation of activity states where each bit of a word is used to indicate whether a particular unit is active or inactive.

The operator's console of the 7090 contains two sets of switches, referred to as the sense switches and the console keys respectively. There are six sense switches, each of which may be set in one of two positions (up, which is the normal position, and down). A computer may be programmed to test the position of any sense switch, and perform different actions depending on its position. This provides a means by which the user may make a decision about alternative courses of action, while the program is actually running. The console keys are 36 in number, corresponding to the 36 bits of a word. The computer may be programmed to take whatever word has been set into the keys and use this as a piece of data. This provides another means by which a user may communicate with a running program.

SIM itself makes no direct use of the switches or keys, but it provides means by which the user of SIM may in his program make any use he desires of them (see Section 4.23).

Most computing centers make use of what is called a monitor system, or an operating system. This is a program provided by the center which allows a large number of users' jobs to be placed one after another on an input tape, and run consecutively with little or no human intervention required. This is done because on a very fast computer, if a number of short jobs are loaded individually by an operator, the amount of time spent in waiting for him to perform manual actions will be disproportionately large compared to the time the computer is able to spend actually working on the problems. In the case of SIM, it is expected that runs will generally be sufficiently long that this factor will not be important. However, most computing centers prefer to have all jobs run under the control of the monitor system if at all possible, so SIM has been written to conform to the standards of the FORTRAN Monitor System, which is the system most frequently used with the 7090.

6.5 Appendix V. Timing

1. Since some of the experiments run with this system may be quite time consuming, it may be desirable to estimate in advance just how long a particular experiment will take. This appendix gives some helpful information in that direction. To calculate the exact time that will be taken by a program is quite difficult; however, in all but the most complicated situations a few key phrases will account for almost all the time consumed. Foremost among these, of course, are those dealing with the actual simulation of the perceptron, that is, the activity computation and the reinforcement.

We shall give some rough and ready rules for obtaining a quick and fairly accurate estimate of the time. The following points should be noted.

(1) No attempt is made to give any estimate of time spent in interpreting the program deck, nor in reading and writing the system tape and control tape, nor of the time spent in the actual creation of the perceptron. In very short experiments, say three or four minutes or less, these may account for a significant, possibly even a major, amount of the total time. However, for a long experiment, the time spent in these activities will be negligible.

(2) Times are given as so many machine cycles (or simply cycles). To convert these to actual times, note that

1 cycle = 2.18×10^{-6} seconds for the 7090

1.20×10^{-6} seconds for the 7094[1]

(3) In situations which involve reading or writing of magnetic tape (including normal output, which is written on tape for later offline printing), part or all of the time will be given as so many ms. (milliseconds). These times assume that the tape unit being used is a 729-IV or VI set at density 556. For a 729-II or V these

[1]This figure takes into account the actual memory time and the best available estimate of gains due to internal overlapping.

times should be multiplied by 1.5. For a tape unit set at density 200, times for the same model at 556 should be multiplied by 2.8. For a tape unit set at 800, times for the same model at 556 should be multiplied by 0.7.[1] Times are always in machine cycles unless milliseconds is specifically stated.

(4) Timing for online printing, if any, is also given in ms. These times, of course, do not depend on the type of tape units attached to the computer.

(5) If there is a large amount of manipulation of lists, involving repetitive creation and destruction of new items, then timing of certain phrases will be increased by an amount which is difficult to calculate. At the very extreme this could go to a maximum of 100,000 cycles on a single phrase. This, however, is extremely unlikely, and would in any case almost certainly mean the program was on the very verge of running out of space and would soon fail to function at all. As a rough guess, 5,000 cycles per phrase might be suitable. However, we find it hard to imagine a practical case where list manipulations would be sufficient to cause this effect to play a significant role in the timing. Phrases affected by this note are indicated in the main body of this section by an * to the left of the phrase name.

(6) If on a particular run some cases are skipped, no significant amount of time will be used by these.

2. Timing for activity computation and reinforcement.

To determine the timing for activity computation, one determines the time for each linkage according to the following formulae and adds them together. In each of the following, "T" represents the number of units in the terminal layer of the linkage,

[1] Timing for density 200 will actually be slightly less than that obtained as above, while for 800 it will be more. The actual values will depend on the average number of words per record. If there are less than 10 words per record, times for all three densities will be approximately the same.

"I" represents the number of units in the initial layer, and "I*" represents the number of active units in the initial layer.

Type 1	6T
Type 2	2T(X+Y+6)+8I+2I* where X is the number of positive (excitatory) inputs per unit and Y is the number of negative (inhibitory) inputs per unit.
Type 3	10T(I+1) +25T is nonzero decay is used +8+8I+2T if Γ reinforcement is used +2+8I+23T if γ reinforcement is used +40+10I+2T+2I*(T+1)-10IT if computation is method G (see below under reinforcement)
Type 4	same as type 2
Type 5	there is no type 5
Type 6	same as type 2
Type 7	6I+2I*+7T
Type 8	11X+12+2T for the normal form 12X+12+2T if the "direct" variation is used 7X+12+2T if the "unconditional direct" variation is used

In the above, X represents the total number of actual inputs (not merely the total number specified explicitly in the tables.

Timing for the reinforcement computation is of course applicable only to Type 3 linkages. In the following formula, we have

I = number of units in initial layer
I* = number of active units in initial layer
T = number of units in terminal layer
δ = decay rate
η = increment
R = number of terminal units reinforced

Q = number of units which are reinforced nonquantized (normally Q qill be either R or zero, but in special cases it could be otherwise)

A = 1 if $I\eta\ (1-\delta)/\delta < 2^7$
= 0|otherwise

If $\delta \neq 0$, then we need

$$V = 6+ \frac{24+(A+8)\ (IT+I+T)}{\text{ent}\ (17-\log_2(I\eta/\delta))} \quad (-\log_2(1-\delta))$$

where "ent (X)" means "greatest integer $\leq X$" (for small values of δ, $-\log_2(1-\delta)$ is approximately 1.45δ).

Timing is also affected by the following considerations. There are two methods of performing the reinforcement, referred to as RG and RNG. Roughly speaking, RG is faster for large values of NT, and RNG is faster for small values. Normally the program automatically chooses RNG for $NT \leq 5$, RG otherwise. However, if timing is really crucial, one may compute times for both methods and force the use of whichever method he thinks best, as discussed in 4.20. Similar remarks apply to activity computation, where methods are referred to as G and NG. Timing for reinforcement is

94+6T+10I+12I*+6R(I*+1)+38Q	
+50+V	if $\delta \neq 0$
+32+6I+6I*+6R	if reinforcement is type Γ
+22+28Q+30R	if reinforcement is type γ
+8RI-2I-14I*-70	if method is RNG

This formula represents the timing required for the actual performance of the reinforcement, and does not take into account the timing for the particular phrase controlling the reinforcement. Normally this latter will be much smaller, and for first approximations may be ignored. It will be discussed in its proper place in the more detailed timing figures given later.

3. Timing for output.

4 ms. for each line printed in the normal fashion, (that is to say, written on tape A3 for later offline printing).

400 ms. for each line printed on the on-line printer.

This applies to all lines printed, whether they are done by LINE phrases, or by other phrases with their own special output, such as TEST. The number of lines should include 2 for the heading at the start of each page, and 1 for each blank line left by the subphrases SPACE and SPACES (but not for blank lines left at the bottom of the page by PAGE or BOTTOM).

For lines made up by "LINE" phrase, the following additional time (in cycles) is required.

200 for each subphrase

3000 for each number inserted by a V, VB, VF, VFF, VO, C, or CP subphrase

150 for each character inserted by an H subphrase

90 for each digit inserted by an EO subphrase

50 for each digit inserted by EB subphrase

10 for each digit inserted by a BCD subphrase

4. Timing for stimuli on tape.

If the dimensions of the stimulus are H by V, then the time required to read the stimulus from tape and put it on the retina is approximately

(.0027 HV + 8.2) ms.

+ (8.22HV + 400) cycles

5. Detailed timing considerations.

In many situations the rules given above will account for such a large portion of the total time that all the rest of the program

may be simply overlooked. However, in the event that some portion of the program is quite complicated, it may be necessary to make a more complete analysis. The figures given in this and the following sections are very rough, but should come within a factor of two of being correct.

For each operand in the program, allow 200 cycles. If the operand has subscripts, count each subscript as a separate operand. (NOTE: If an operand occurs in a phrase which is executed more than once, this must be counted for each time it is executed.)

For each compound phrase, 200 cycles plus 100 for each iteration performed (this is in addition to the time for the phrases within the compound phrase).

For each sequence executed, 100 plus 500 for each parameter, if any, plus 12N where N is the total number of words in extended operands used as parameters, if any.

200 for each phrase (each time it is executed) plus the adjustments for some special phrases, as given in the following sections.

6. Arithmetical, logical, and conditional phrases.

For the phrases listed below, add the indicated numbers of cycles in addition to the 200 cycles per phrase as specified in 5. All of these phrases admit either an extended operand, or an indefinite number of operands; in each case W represents the total number of words operated on by the phrase.

ADD	20W
SUB	20W
MPY	40W
DIV	56W
ADDF	34W
SUBF	34W
MPYF	34W
DIVF	56W
FLOAT	100W
FIX	20W
ZERO	5W
BRAKET	16W
IF LEQUAL	6W
IF LZERO	6W
AND	40W

OR	40W	
COM	100W	
EOR	100W	
LOGIC	100W	
SHIFT	40W	
MOVE	10W	
BITS	360W	
BITSZ	360W	
PACK	20W	(W is number of words in unpacked form)
GETACT	20W	
PACKZ	20W	
UNPACK	10W	
SETACT	10W	
FILL	16W	
*FILLB	16W+300	
LFILL	16W+300L	(L is number of new items added to list)

7. Blocks and lists.

In addition to the 200 cycles per phrase specified above, add the following.

*BLOCK	200	
*LLMOV	60	for each item moved, or
	260	for each item if removed completely (LLMOV bv,)
*LLCLR	200	for each item on the list

for each new item added to a list, add 200 cycles.

8. Perceptron test phrases.

For a stimulus of dimensions H by V, the STIM phrase takes

(400+8.22HV) cycles + (.0027HV+8.2)ms.

If stimulus name is printed (STIM(*PRINT)) add

1000 cycles + 4 ms.

If stimulus name is added to a list or to the print line

(STIM(bv) or STIM(*ADD)) add

200 cycles

In all the following perceptron test phrases, the following times will be understood whenever needed:

time to obtain stimuli	same as for STIM above
time for necessary activity computation and/or reinforcement	as discussed in 2. above
time to move activities into delayed copies of layers	for each layer 6ND cycles, where N is number of units, D is maximum delay.
time for printing output lines	4 ms. per line

Times given below are in addition to these. For phrases not mentioned here, the extra may be taken as zero.

REINF

SPOS or RPOS	10T
SCON or RCON	6T
ECON	24T
CONTIN	24T
SPEC	(160+P)T

T is number of units in terminal layer
P is number of cycles used by phrases forming the compound part of the REINF SPEC phrase.

CLEAR LAYER	200+4W	W is total number of words in layers cleared.
CLEAR LINK	300+80TI	for each linkage affected T is number of units in terminal layer I is number of units in initial layer
CLEAR WEIGHT	300+80TI	
SETWGT	300+80TI	
GETWGT	300+110TI	
GETACT	150+20W	
SETACT	150+10W	W is number of words in layer
COUNT	15000	if counts are printed
TEST	15000	
TESTP	15000	
BINPR PACT PACTO	3000+30N	N is number of units

PW 3000T(I+1)

and this phrase will print about $\frac{TI}{10}$ lines.

LEVEL between 17U and 17SU

probably about $17U + 4S^2(3+\log_e(U/S))$

U = number of units in layer
S = number to be left active by LEVEL

9. Stimulus generation phrases.

In the following discussion,

H = horizontal dimension of stimulus

V = vertical dimension of stimulus

P = 13 HV + 100 if the current stimulus is in unpacked form

= 0 if it is in packed form

U = 9 HV + 100 if the current stimulus is in packed form

= 0 if it is in unpacked form

Preceding each code in the following list is a single letter which indicates whether the phrase leaves the stimulus in packed form (P), unpacked form (U), or leaves it as it found it (X). The following list does not include times for printing stimuli, which should be added when applicable. These are:

to print heading	25 ms.
to print heading online	800 ms.
to print stimulus	10(V+2) ms.
to print stimulus online	400(V+2) ms.

Any phrases not included in this list require 200 cycles and leave the stimulus as they found it.

X	SSINIT	2000
P	SSPACK	P
U	SSUNPK	U

P	SSPRT	100+P
P	SSPRTH	100+P
P	SSDEFS	300+10N+HV+P
P	SSN	180N+HV
P	SSR	250N+HV
	N = number of stimulus	
U	SSHBAR }	10HV+10V+100
U	SSVBAR }	
P	SSZ	5HV+100
U	SSRAND	100+70M+5HV+U
U	SSRW }	14HV+200M
U	SSRWC }	
	M = number of points chosen	
X	SSBIAS	4000 per distribution
U	SSPT	150M+U
U	SSTRAN	400+12V+20HV+U
U	SSROT	1000+135HV+U
U	SSDIL }	400+135HV+U
U	SSLIN }	
X	SSNAME	3000
X	SSXNAM	5500
P	SSOUT	200+P+(.0027HV+8.2) ms.
U	SSC	(binary form) 300HV+2600V+9000 cycles + H+9 ms.
U	SSC	(integer form) 9HV+800N+700 cycles + .9N ms. +4000 cycles + .9 ms if NAME is used +10000 cycles + .9 ms if CLASS or NEG is used N is number of active points
P	SST	(.0027HV + 8.2) ms.

10. Miscellaneous

READ or WRITE	(7.3+.096N) ms. N = total number of words read or written
FILE or RECORD	(.067N+7.3R+34F) ms.

PRINT or PRINTF	(3N+20R+40F) ms. N = number of words R = number of records F = number of end of file marks over which the tape passes during the process
REWIND, UNLOAD, ENDFIL	require the same timing as given above for FILE and RECORD. However, if they are followed by phrases which do not affect tapes, they will proceed simultaneously, and effectively take no time.
DATA	Each item placed on or taken from the DATA tape requires approximately .87 ms., except that a binary card will require 9 ms.

11. Example

Consider the example 3.5, and let us compute timing figures for it.

We first note that the retina usually has 10% of its points active. (This is stated explicitly for the randomly generated stimuli, and it seems reasonable to assume it for the given stimuli as well.) On theoretical grounds which we will not go into here, we decide that the A-units will probably have an activity level of less than 5%; but to leave a margin for error, we will use 10%.

timing for activity computation for the type 2 linkage

T=100, X=3, Y=1, I=200, I*=10% of 200 = 20

time = 200(3+1+6)+8x200 + 2x20
= 3640 cycles

timing for activity computation for the type 3 linkage

T=100, I=100, I*=10 computation method is G (since T>5)

time = 1000(100+1)+2500
+40+1000+2x100+2x10(101)-10x100x100
= 6760 cycles

timing for reinforcement for the type 3 linkage

I=100, I*=10, T=100, R=10 (on the average), Q=0

$I\eta/\delta$ is between 200 and 400 depending on the case

so $I\eta(1-\delta)/\delta < 2^7$, so A=1
and ent$(17-\log_2(I\eta/\delta))$ is 8 or 9

$$V=6+\frac{24+9(10200)}{8 \text{ or } 9}\ (1.45\delta)$$

$$\cong 6+15000\,\delta = 156 \text{ or } 756$$

Putting this all together, and noting that since T>5 we have reinforcement method RG, we get

$$\text{time} = 96+6x100 + 10x100 + 12x10 + 6x10(10+1) + 50 + V$$
$$\cong 3000 \text{ cycles}$$

So total time to do activity computation and reinforcement for one stimulus is about 13000 cycles. If we assume we are using a 7094, this comes to

$$13000 \times 1.2 \times 10^{-6} = .016 \text{ seconds/stimulus}$$

For obtaining the stimuli from the tape, we have according to section 4,

$$(.0027x10x20 + 8.2) \text{ ms.} + (8.22x10x20 + 400) \text{ cycles}$$
$$= 8.74 \text{ ms.} + 2044 \text{ cycles}$$
$$= .00874 + 2044 \times 1.2 \times 10^{-6} = .011 \text{ seconds/stimulus}$$

For each G-matrix we have

220 stimuli at (.016+.0110) seconds each

22 lines of print at .004 seconds each

400 numbers inserted in print lines, at 3000 cycles = .0036 seconds each

which comes to

$$5.94 + .088 + 1.44 \cong 7.4 \text{ seconds}$$

So for a complete case (10 G-matrices) we have 74 seconds; and since the experiment as described comprises 10 cases (two values of δ, and for each one, five values of η) we get a grand total of 12 minutes.

The only factors not considered already which could conceivably affect the timing significantly are (i) the generation of the stimuli and (ii) the saving of activity vectors and computation of G-matrices. Neither of these is likely to make any major change in the figure already arrived at, but we will analyse them for the sake of further illustration.

Applying the figures in section 9 above, we see that the first loop (line 7 of the example) requires the following time per iteration:

SSRAND	100 + 70x20 + 5HV+9HV+100 cycles
SSTRAN	400 + 12V + 20HV + 9HV + 100 cycles
SSOUT	2(200+13HV+100) cycles +2(.0027HV + 8.2) ms.
Total	2700 + 12V + 69HV cycles + (.0054HV +16.4) ms.

and with H=10, V=20, this comes to

16740 cycles +17.14 ms. = .02 + .01744 = .037 seconds

This loop is iterated 100 times, and is enclosed in a larger loop which in turn is iterated 10 times, giving us

1000 x .037 = 37 seconds

The rest of the section is clearly negligible. (Note that this section is only executed once for all 10 cases.)

As for the other, the significant phrases are those on lines 24 and 25, which are performed 400 times per G-matrix.

We count 10 operands at 200 cycles each

3 phrases at 200 cycles each

40x3=120 extra cycles for the AND phrase

3x360 = 1080 extra cycles for the BITS phrase

56 extra cycles for the DIVF phrase

Total < 4000 cycles

This is done 400 times per G-matrix, 10 G-matrixes per case, total of 10 cases, giving

$$4000 \times 400 \times 10 \times 10 = 16 \times 10^7 \text{ cycles}$$

$$= 16 \times 10^7 \times 1.2 \times 10^{-6} = 192 \text{ seconds} = 3 \text{ minutes}$$

These two factors together increase the total estimate from 12 minutes to $15\frac{1}{2}$ minutes. The most careful analysis of the remaining phrases will not bring this over 16 minutes.

Note that considering <u>only</u> the timing for activity computation and reinforcement would have given about 6 minutes, which would have given a good indication of the order of magnitude; while considering as well the time to read the stimuli from the tape would bring it up to 12 minutes.

6.6 Appendix VI. Size Limitations

The experiments which can be performed are of course limited by the amount of space available in the 7090 memory. The system has been organized so as to make the best possible use of the available space. For example, no a priori assignment has been made of various parts of the memory for various parts of the perceptron (since if this were done, an experiment might fail to run for exceeding one of these areas, even thought there were plenty of surplus space elsewhere). Instead, the system assigns whatever space is available for whatever use it might be required for. Similarly, at the time the perceptron is actually being tested, only those portions of the system which will really be needed in this particular experiment are kept in the main memory, the others being kept on a tape. Hence, the only limitation is whether or not the total amount of memory available is exceeded. Unfortunately, of course, this very flexibility makes it more difficult to compute in advance whether a given experiment will or will not exceed the size limitations. We will give in this appendix some rules for determining approximately the space required.

The space requirements for each section of the program should be determined according to the following rules, and no section should exceed the total amount available, which is 32,768 words. The rules as given below give a pessimistic estimate of the amount of space required. Some numbers are followed by a second number in parentheses; by using the second numbers wherever they are given, one may obtain an optimistic estimate. If even the optimistic figure is too large, then the program will certainly not fit into the available space; if the pessimistic figure is small enough, the program almost certainly will fit. If the optimistic figure is small enough, but the pessimistic one is not, one will have to actually try a sample case to see if it will fit.

(1) The space requirements of a given section can be broken into three parts, which we will refer to as the permanent requirement, the perceptron requirement, and the variable requirement.

(2) The permanent requirement is A + B + C + D where A, B, C and D may be computed as follows.

In a perceptron testing section, or in any section containing the phrase LARGE,

A is 1400

+ 1300	if the section uses LINE or HEAD phrases or any other phrases which produce printed output (e.g., TEST).
+ 250	if there is any tape reading or writing (either explicitly by use of READ or WRITE, or implicitly by use of $DATA, STIM, SSOUT, etc. In particular, this will be required if there is any printed output, as above.)
+ 300	if information on a tape is to be printed using a TAPE phrase.

In any other section, A is 8200.

B is 1700 (450)

+ 1500 (600)	if stimulus generation features are used.
+ 350 (150)	if any LINE or HEAD phrase is used.
+ 200	if any CONEC phrase is used.
+ 250	if perceptron parameters are specified in this section.
+ 1000 (300)	if this is a perceptron testing section.
+ 350 (150)	if any blocks or lists are used, or if stimulus generation features are used.

Usually C may be taken as 500 (50). However, if the section is quite large, one should add to this as follows:

1 for each variable,
1 for each phrase,

1 for each operand (and if any of the operands are subscripted block or list names or special operands which contain a list of I-operands, one more for each subscript or I-operand).

1 more for each operand which is a constant, other than integer constants of magnitude less than or equal to 32,768.[1]

NOTE that no space at all is required by a DATA phrase.

D is zero unless the program contains an ALWAYS phrase. If it does, D is the total amount of space required by the variables and blocks named in ALWAYS phrases (for each variable, one word; for each block, the product of its dimensions). Note that D will be the same for every section of the program.

(3) The perceptron requirement is zero except for a perceptron testing section; there it is S + G, where S and G may be computed as follows.

For each layer in the perceptron add to S N(D+1) where N is the number of units in the layer and D is the largest delay specified in any linkage using this layer as the initial layer (D equals zero if no such delay is specified). In addition, add one third the number of units in layer one (retina).

G is 50 x the number of layers in the perceptron, plus an amount for each linkage, as determined by the table following. In this table, "I" will represent the number of units in the initial layer of the linkage, and "T" the number of units in the terminal layer.

Type 1 5

Type 2 I + T(X+Y+2) where X is the number of positive (excitatory) inputs per unit. Y is the number of negative (inhibitory) inputs per unit.

Type 3 150 + I(T+3) + I (if reinforcement is type) + T (if reinforcement is Γ).

[1] Note, for example, that according to these rules, one would have to add 12 for the phrase "FILL B 1 2 3 4 5 6 7 8 9 10," but 22 for the phrase "FILL 18 B 1 2 3 4 5 6 7 8 9 10,".

Type 4 same as Type 2.

Type 5 there is no Type 5.

Type 6 same as Type 2.

Type 7 T + 11

Type 8 2T + DC + 20 where C is the total number of connections actually specified in the tables (not counting those implicitly specified by giving the inputs to one unit as copies of those to another) and D is one if the "direct" option is used in this linkage, two otherwise.

(4) The variable requirement is the space used for any blocks and/or lists used by the section. Each block and each item on the list requires the appropriate amount of space as specified by the dimensions (e.g., a 5 x 30 block would require 150 words). The space required by a block is not needed until the BLOCK phrase is executed and may later be released by means of a LLCLR phrase. The space for a list item is not needed until the item is actually added to the list (using "**NEW"), and the space used by a given list may be released by LLCLR and/or LLMOV. Hence the total amount of space needed at one time or another may exceed the amount available provided that the amount actually in use at any given instant is not too large.

Note that if the stimulus generation features are in use, certain blocks and lists are implicitly used, in addition to those explicitly named in the program. Specifically, if H and V are the dimensions of the stimulus, then two blocks of dimensions H by V and a list of dimension $\frac{HV}{36} + 9$ will be defined at the time the SSINIT phrase is executed. The list will have n items added to it, where n is the largest numbered stimulus saved by a SSDEFS phrase.

(5) For a perceptron testing section, the following restriction must be satisfied.

$G + D \leq 30{,}450$ where G is as computed above under (3), and D is as computed above under (2). This is in addition to the usual

restriction that the total space requirement should not exceed 32,768.

(6) Consider the experiment of example 3.5. We would like to know if this example will exceed the space limitations, or more generally, we would like to know if we can use even more A units than are specified in the example given, and if so, how many. For the first section (lines 4 to 9 of the example), we compute a pessimistic estimate as follows.

$$A = 8200$$
$$B = 1700 + 1500 + 350 = 3550$$
$$C = 500$$
$$D = 0$$

So the permanent requirement is 12,250. The perceptron requirement is not applicable to this section, and the variable requirement consists simply of the two blocks and one list used implicitly by the stimulus generation features. The blocks are each 10 x 20; the list has 10 items, each of $\frac{10x20}{36} + 9 = 15$ words. This makes a total of 550 words, which added to the previous, gives 12,800. This is well below the permissible limit of 32,768 (and note that this does not depend on the number of A units). In a similar fashion we see that there is no problem for the second section. For the third section (lines 19 to 28) we have as follows.

$$A = 1400 + 1300 + 250 = 2950$$
$$B = 1700 + 350 + 1000 + 350 = 3400$$
$$C = 500$$
$$D = 0$$

A total of 6850 for permanent requirements. For an optimistic estimate, B would become 450 + 150 + 300 + 150 = 1050 and C would become 50, making a total of 4050. For the perceptron requirements, letting n be the number of A units, we have

$$S = 200 + n(1+1) + 70 = 270 + 2n$$

The Type 2 linkage requires 200 + n (3+1+2) = 200 + 6n. The Type 3 linkage requires (note that I and T both equal n) 150 + n (n+3). Hence G = 2 x 50 + 200 + 6n + 150 + n^2 + 3n = 450 + 9n + n^2 and the total perceptron requirement becomes 720 + 11n + n^2. Finally, the variable requirement is that needed for the blocks BSAVE and BX (line 19). Note that if the number of A units is changed, the "3" may be a different value; in general it will be $\frac{n}{36}$ + 1. The two blocks together will require 21 ($\frac{n}{36}$ + 1) which we may as well round to n + 21. Hence the grand total requirement for this section is 7591 + 12n + n^2 for a pessimistic estimate, and 4791 + 12n + n^2 for an optimistic estimate. For these expressions to be less than 32,768 would require $n \leq 152$ and $n \leq 161$ respectively. As for the requirement of rule 5, G + D $\leq$ 30,450, this requires that 450 + 9n + $n^2 \leq 30{,}450$ and this comes down to $n \leq 168$. We can conclude that the experiment can certainly be run with at least 150 A-units, and probably not with as many as 160. If we really wanted the largest number possible, we would have to resort to trial and error to pin it down.

We could increase this number slightly by rewriting the program to remove the LINE phrases from the third section into a fourth section. To do this we would write the activity vectors onto a tape in the third section and read them back in the fourth section and form and print the G matrices. This would allow the value of A to be reduced by 1300 and the value of B by 350, and would thus allow us to increase the number of A units by about 9 or 10.

(7) Space limitations during interpretation. The system was designed with the expectation that programs would be quite short (in terms of number of phrases used, not necessarily in terms of time), and as a result the use of space during the interpretation phrase of the system is not particularly efficient. Thus, if one has a very complicated program, one may find that it will not get

past the interpretation phase due to the lack of space, even though its space requirements computed as above are not excessive. In practice, this has only affected certain programs in which a large amount of information was being put in as a long list of numbers following a FILL phrase; in each such case the problem was avoided by rewriting the program to make use of the DATA feature (which makes no demands on space at either interpretation or execution time).