Behavior Research Methods, Instruments, & Computers
1988, 20 (2), 119-128

TOOLS FOR RESEARCH IN PERCEPTION AND COGNITION

Mary Beth Rosson, *Presider*

A flexible programming language for generating stimulus lists for cognitive psychology experiments

STEVEN GREENE
(Student Award Winner for 1987)
Yale University, New Haven Connecticut
and Northwestern University, Evanston, Illinois

and

ROGER RATCLIFF and GAIL McKOON
Northwestern University, Evanston, Illinois

Listmaker, a programming language developed to simplify the production of stimulus lists for cognitive psychology experiments, is described. The user writes a description of the experiment in the language, and the computer program then generates stimulus lists for the experiment from files of materials. The language allows the user to manipulate easily a wide variety of stimulus materials (e.g., single words, sentences, or multiline stories). The language also provides simple commands that allow even very complex randomized and counterbalanced designs to be implemented. Two examples of programs written in the language are provided and explained in detail. Finally, an interface to a real-time experimental system is described.

Too often, an inordinate amount of an experimenter's time and mental resources is spent in the mundane task of creating stimulus lists for use in experiments. Translating the experimental design into a computer program to generate lists for individual subjects from sets of materials is not always a straightforward task, especially if the design calls for complex randomizations or counterbalancing. The process is one that is prone to error, and locating the source of errors can be extremely tedious.

To reduce the complexity of this task, we have designed a language to simplify and speed the development of stimulus lists for psychological experiments. We had three goals in designing the language: First, the language should be flexible enough to generate stimulus lists for a wide variety of experimental paradigms and kinds of materials. Second, the structure of the language should allow the design of the experiment to be expressed as directly as possible. Third, the language should provide a convenient interface to a real-time experimental system, such as that described by Ratcliff, Pino, and Burns (1986).

The language, Listmaker, is similar in approach to the EXPOSE system of Stevens, Levin, Olds, and Rumelhart (1977), but is considerably more extensive in its operators and, therefore, is more flexible. The EXPOSE system consists of a collection of nine FORTRAN routines that randomize and combine files containing experimental materials in order to generate stimulus lists. The EXPOSE programmer uses these routines in his or her own FORTRAN programs to simplify coding.

Listmaker, unlike EXPOSE, is a complete programming language with its own syntax. The entire design of

This research was supported by National Science Foundation (NSF) Grant BNS 85 10361 to Roger Ratcliff, National Institute of Health Grant HD18812 to Gail McKoon, NSF Grant BNS 85 16350 to Gail McKoon, and a "Research Experience for Undergraduates Supplement" to NSF Grant BNS 85 10361 for Steven Greene. Roger Ratcliff had the initial idea for the language and motivated 25% of the commands that were finally developed. Steven Greene, who is now at Stanford University, designed the other 75% of the commands and fully implemented the system while an undergraduate at Yale working as a summer research assistant at Northwestern. Gail McKoon is the primary user of the language and helped guide the final design. Reprint requests should be sent to Roger Ratcliff, Psychology Department, Northwestern University, Evanston, IL 60208.

ten in Listmaker. The Listmaker interpreter then processes the program and generates the stimulus lists for the experiments. The Listmaker interpreter itself is written in FORTRAN and runs on a Sun workstation. FORTRAN was chosen because it is the language used by most people in the laboratory and because of its ease of transportation to other computer systems. The Listmaker interpreter was written in standard FORTRAN 77, so it can be transported to other systems with relatively little modification.

Program Structure

The basic grouping of data in Listmaker is a "set." A set consists of a collection of similar items; for example, the sentences to be studied in the study phase of a study-test paradigm may be grouped together in a set. Each set is named with a capital letter from A to T, optionally followed by a single digit (e.g., A, B6, R7). Listmaker also provides for "index sets," which are similar to sets except that they contain numeric data and can be used in numeric calculations. Index sets are often used to keep track of the locations in one set into which items drawn from another set were placed. This allows related materials in different sets to be kept together. (See examples below.) Each index set is named with a capital letter from U to Z, optionally followed by a single digit (e.g., U1, X2). Individual items in a set are identified by appending a subscript to the set name [e.g., A(1), B6(102), X2(17)]. Finally, Listmaker also allows simple scalar variables, each named by a single lowercase letter. They are used for such purposes as counters in loops or to identify the current condition number in a design with multiple conditions.

A Listmaker program generally consists of two parts. The first part reads in the stimulus materials and organizes them into sets as required by the experimental design. A variety of set operators and control structures are provided to simplify this part of the program. The second part of a Listmaker program consists of one or more output blocks. An output block contains an explicit representation of the stimulus list it will generate. The format of that representation is described below. (The details of the language are included in the Appendix.)

Example 1: A Completely Random Design

In this section, we describe the development of a hypothetical priming experiment (see Ratcliff & McKoon, 1978) using Listmaker. Suppose we want a subject to read 32 sentences presented in random order. One word in each sentence is assigned to be the *target*; another is the *prime*. After reading all the sentences, the subject is presented with a test list of 64 words for recognition. The test list consists of 8 prime-target pairs from the same sentence (16 words), targets from 8 other sentences the subject read preceded by primes from different sentences (16 words), and 32 negative fillers from a list of words not in any of the sentences. Sentences are assigned to conditions randomly, so this is a design with no counterbalancing.

...ns materials into sets. There are four sets: S for the sentences, T for the target words, P for the primes, and N for the negative fillers. Assume that the lists are arranged in the disk file "EXP1MATERIALS," with each sentence followed by its target and prime. The negative fillers are in a list at the end of the file. The Listmaker lines that read in the data are as follows:

```
dim S(32),T(32),P(32),N(32)
open9:EXP1MATERIALS
(do i=1,32
read9:S(i),T(i),P(i)
next i)
read9:N
close9
```

The first line says that there are to be 32 items in each set. The next line opens the disk file and assigns it the number 9, which is used in the read statement that follows. The do-loop reads 32 items into each set S, T, and P. Since no subset specifier is included in the read statement for set N, the entire size of the set is read in automatically. The last line closes the input file.

Once the sets are defined, targets and primes need to be chosen from 8 of those sentences to be used for the same-sentence priming condition in the test line:

```
T1=8?T>X
P1=P<X
```

The first line says that set T1 gets 8 items chosen at random without replacement from set T. The "?" operator performs a draw without replacement; no item drawn from T will be drawn again until the set is explicitly reset using the "@" command (described below). Note that no "dim" statement is needed for T1; it is automatically assumed to be the size generated by the draw operation, in this case, 8. The ">X" appended to the first line says that the index set X stores the locations of the items in T that were put into T1. The next line says that set P1 gets those items from P that are in the positions stored in X. Thus, P1 contains the primes that go with the items in T1: Prime P1(1) is the prime for target T1(1), and so on.

Next, 8 targets and 8 primes are needed from other sentences. This is done as follows:

```
T2=8?T
T3=16?T>Y
P3=P<Y
P2=8?P3
```

The first line above chooses 8 more items at random from set T. These will be used as targets in the test list. The next three lines get 8 primes from the 16 as-yet-unused sentences. The second line puts the remaining 16 targets into set T3 and stores the locations from which they were drawn in index set Y. The third line puts the corresponding primes in set P3, and the last line draws 8 of those at random and puts them in set P2.

the sentences, T1 and P1 contain the eight target-prime pairs from the same sentences, T2 and P2 contain the eight target-prime pairs from different sentences, and N contains the negative fillers. We are now ready to begin generating stimulus lists. However, before we do, we must specify the file name to which the output should be written. If the lists are to be saved in a file called "EXPLIST," the following command is used:

```
output:EXPLIST
```

Next, the first output block, that for the study list of 32 sentences, can be created as follows:

```
[list=32
(do i=1,32
:S(i) : *
next i)
]
```

An output block always starts with a left bracket and ends with a right bracket. Following the left bracket is a statement specifying how many items will be in the output list generated by the block. Then we have a simple do-loop. Inside this do-loop is an output line descriptor. This line consists of two parts, delimited by colons. The part between the two colons says that at this position in the output list, put one item from set S. The part after the second colon identifies where in the output list this item can go. In this case, it can go anywhere, so we simply write "*."

Next, an output block is needed for the test list. The test list is constructed according to the following rule: Words are placed randomly in the list, with the restriction that a target may not appear in one of the first three positions. A target is always immediately preceded by its prime. The output block will look like this:

```
[list=64
(do i=1,8
:T1(i) : *a,a>3
:P1(i) : a-1
:T2(i) : *b,b>3
:P2(i) : b-1
next i)
(do i=1,32
:N(i) : *
next i)
]
```

The first do-loop puts the targets in place. The first line of the do-loop puts an item from T1 at a random location, assigns that location to the variable *a*, and assures that *a* is greater than 3. The following line puts the prime at the location immediately preceding the location of the target. The next two lines repeat the procedure for the other prime-target pairs. The second do-loop assigns each of the negative fillers to a random location in the list.

A Listmaker program has now been written to generate a stimulus list that is fully randomized, subject to certain restrictions. Suppose we want to run in the experiment, we need to be able to produce many different lists, each subject to the same constraints. In Listmaker, this is easily accomplished by placing the relevant section of our program in a do-loop that is executed once for each stimulus list we want to produce. In the example, since we want to draw a different set of targets and primes for each list, the entire program must be repeated following the section that reads the materials from the disk. Therefore, the following two lines are added immediately after the "close" statement at the end of the input section:

```
(do s=1,8
@T,P
```

and this line

```
next s)
```

is added as the last line of the entire program. This loop changes the program to produce eight different random stimulus lists from the same set of materials. The "@" command tells Listmaker to rerandomize the order in which items from sets T and P will be drawn and to reset its internal count of which items have been drawn. Thus, all targets are available to appear in each stimulus list the program produces.

One last change is required. For each stimulus list to be saved in a different file, the output statement must be modified slightly to read as follows:

```
output:EXPLIST&s
```

The "&" replaces the variable that follows it (in this case, s) by its value at run time. Thus, this program will put one stimulus list in each of the files EXPLIST1, EXPLIST2, on up to EXPLIST8.

Thus, we have shown how Listmaker can be used to generate multiple stimulus lists that are fully randomized, subject to certain restrictions. Another short example is provided that demonstrates the ability of Listmaker to generate multiple counterbalanced stimulus lists.

Example 2: A Counterbalanced Design

For this example, suppose we want to present a subject with 18 stories of 10 lines each to read. Each story has three possible test sentences that are associated with it: one clearly true, one clearly false, and one plausible inference (see McKoon & Ratcliff, 1986). After reading all 18 stories, each subject will be asked to verify one of the sentences for each story. Since there are three conditions and any one subject can receive only one test sentence (i.e., one condition) for each story, a counterbalanced design is required. Since there are three test conditions, three groups of subjects are needed. Each group will receive the true sentence for 6 of the stories, the false sentence for 6 others, and the plausible inference for the remaining 6.

As above, we begin by reading in the materials from a disk file, in this case "STORYMATERIALS":

```
(do i=1,18
read9(F10):S(i)
read9:T(i),F(i),I(i)
next i)
close9
```

The only new feature in the above lines is the fourth line, which uses a special read statement. This line says to read in 10 lines from the file for each item in set S (the stories). The next line reads 18 test sentences each into sets T, F, and I.

We now want to generate three counterbalanced lists and put one in each of the files STORYLIST1, STORYLIST2, and STORYLIST3:

```
(do i=1,3)
output:STORYLIST&i
```

The do-loop is not closed yet, as it will encompass more of the program below.

Next, we assign the test sentences in sets A, B, and C as appropriate for the counterbalanced design:

```
if i=1 then
  A=T[1,6],F[7,12],I[13,18]
else
if i=2 then
  A=F[1,6],I[7,12],T[13,18]
else
if i=3 then
  A=I[1,6],T[7,12],F[13,18]
endif
```

Each assignment line above creates a new set, A, that consists of the appropriate six items from each of sets T, F, and I. Set A therefore contains one test sentence for each story; which sentence is chosen for each story depends on the counterbalancing condition.

There are two output blocks, one for the stories and one for the test sentences:

```
[list=18
(do j=1,18
:S(j) : *
next j)
]
[list=18
(do j=1,18
:A(j) : *
next j)
]
```

The first output block puts each story in a random position in the output list. The second output block does the same for a separate list of the test sentences. Now, all that is left is to end the initial do-loop:

```
next i)
```

Thus, using one conditional and three do-loops, we have a counterbalanced, randomized set consisting of 18 10-line stories followed by a test sentence for each.

Using Listmaker to Generate Paper-and-Pencil Tasks

In addition to allowing researchers to manipulate stimulus materials, Listmaker also provides features to make it easy to intermix text into stimulus lists. Using these features, we can insert instructions to the subject in the stimulus lists created by Listmaker. For example, if we had a disk file called "INSTRUCT" containing instructions for the story experiment above (Example 2), adding the line

```
include INSTRUCT
```

after the "output" statement in the program would insert the contents of that file at the beginning of the output file generated by Listmaker. Shorter instructions to the subject can be placed between output blocks. If, in the story experiment, we wanted to instruct subjects to wait for the experimenter to return after they finished reading the stories but before they start answering the test sentences, the following line could be added between the two output blocks:

```
write "Wait for the experimenter to return before
continuing."
```

This instruction would put the quoted text in the output file following the last story and preceding the test sentences.

Additionally, using Listmaker we can add short text strings directly to output lines. Still using the story experiment as an example, suppose we want to add a "True/False" response area for each test sentence. We simply modify the output line, which currently reads

```
:A(j) : *
```

to read

```
:A(j) "True _____ False _____" :*
```

Using this line, Listmaker will print, on the same line, the test sentence, followed by the quoted text.

It is often necessary for the experimenter to have a scoring sheet for each subject. In this example, the sheet might contain the counterbalancing condition that the subject was in and, for each test sentence, the item number of the sentence, the position of the sentence in the output list, the position of the corresponding story in its output list, and the experimental condition of the sentence (True, False, or Inference). To simplify scoring, we want the sentences to appear on the scoring sheet in the same order in which they appeared on the subject's test sheet. With minor modifications to the program, this is easily accomplished.

Listing 1 is the Listmaker program for the story experiment. Variable *i* identifies the counterbalancing condition. For any sentence, variable *j* identifies its item number. An index set, *X*, can be used to store the item number

Final Listmaker program for the story experiment
(Example 2)

```
dim S(18),T(18),F(18),I(18),X(18),Y(18)
open9:STORYMATERIALS
(do i=1,18
read9(F10):S(i)
read9:T(i),F(i),I(i)
next i)
close9
(do i=1,3
output:STORYLIST&i
include INSTRUCT
if i=1 then
    A=T[1,6],F[7,12],I[13,18]
else
if i=2 then
    A=F[1,6],I[7,12],T[13,18]
else
if i=3 then
    A=I[1,6],T[7,12],F[13,18]
endif
[list=18
(do j=1,18
:S(j) : *Y(j)
next j)
]
write "Wait for the experimenter to return
  before continuing."
[list=18
(do j=1,18
:A(j) "True _____ False _____" :*a
X(a)=j
next j)
]
output:STORYSCORE&i
write "Counterbalancing " i
(do a=1,18
j=X(a)
c=j-1/6+i|3
write a "Item: " j " Condition: " c
  " Story position: " Y(j)
next a)
next i)
```

of the sentence that is assigned to each position in the output list by further modifying the output block to read as follows:

```
:A(j) "True _____ False _____" :*a
X(a)=j
```

These lines will now store in the index set *X* the item number of the sentence that was assigned to each position in the output list. Thus, if sentence three were the fifth sentence in the test list, X(5) would equal 3. A similar strategy can be used to keep track of the order of the stories in the study list. However, in this case, given a story number, we want to be able to identify its position in the output list. (This is the converse of the situation above, in which, given a position in the output list, we want to be able to identify the item number of the sentence in that position.) Therefore, we need to modify the line in the output block for the stories as follows:

(These changes necessitate adding "X(18),Y(18)" to the "dim" statement at the beginning of the program.) The last number needed is the experimental condition of the sentence. Since it is known that *j* is the item number and *i* is the counterbalancing condition, the experimental condition of a sentence is $(j-1)/6+i$ modulo 3, using integer arithmetic. In Listmaker, this is written as follows:

```
j-1/6+i|3
```

(since Listmaker evaluates expressions from left to right). This expression returns 0 for the inference condition, 1 for the true condition, and 2 for the false condition.

The score sheet can now be created for the experimenter:

```
output:STORYSCORE&i
write "Counter-balancing " i
(do a=1,18
j=X(a)
c=j-1/6+i|3
write a "Item: " j " Condition: " c
  " Story position: " Y(j)
next a)
```

These lines tell Listmaker to send all further output to a new file, called "STORYSCORE1," "STORYSCORE2," or "STORYSCORE3." Then, for each position in the test list, they write the position number, the item number of the sentence in that position, the experimental condition of the sentence, and the position of the corresponding story in the story list. (The complete program for the story experiment, with all modifications, appears in Listing 1.)

Interface to a Real-Time System

One of our goals in creating Listmaker was to provide an easy interface to a real-time experimental system. The output features described in the previous section make this task extremely straightforward. The priming experiment discussed above (Example 1) will be used to demonstrate how Listmaker can produce stimulus lists ready to run on a real-time experimental system. For this example, we use the system designed by Ratcliff et al. (1986).

For this experiment, subjects are required to press the space bar to begin the study phase. They are presented with each study sentence for 3 sec. After the last study sentence is presented, they are again instructed to press the space bar to begin the test phase. Each word in the test phase is presented until the subject presses a key, one key for old words that appeared in one of the studied sentences, another for new ones that did not appear in any studied sentence. The key pressed and the reaction time are recorded.

Of course, we need to keep track of the condition in which each word in the test list appears (prime, target

different-sentence prime, or negative filler). Since each condition has its own line in the output block for the test phase, a number can be assigned to each of these conditions directly on the output line. For each prime we also would like to know at what position in the study list the associated sentence appeared. As in the story experiment, an index set is used to keep track of the position in the output list of each sentence. The index set V is used for this purpose. (Again, set V must be added to our "dim" statement.) However, since the targets (sets T1 and T2) were themselves not used in the previous output block, the subscripts of set V will themselves be index sets that indicate which sentence from the materials set is in each position of these sets. Set X already serves this purpose for targets in T1; we must add an index set Z to do this for set T2, so the line "T2=8?T" becomes "T2=8?T>Z."

The following changes and additions to the program accomplish the additional tasks of recording the experimental conditions and list positions. (The complete program, with all modifications, appears in Listing 2.)

```
output:EXPLIST&s
write "Press space bar on keyboard to begin
  study phase#R@C"
[list=32
(do i=1,32
:S(i) "#W3000@C" : *V(i)
next i)
]
write "Press space bar on keyboard to begin
  test phase#R@C"
[list=64
(do i=1,8
l=X(i)
:T1(i) "#R@C#S/" 100+V(1) "/" : *a,a>3
:P1(i) "#R@C#S/200/" : a-1
l=Z(i)
:T2(i) "#R@C#S/" 300+V(l) "/" : *b,b>3
:P2(i) "#R@C#S/200/" : b-1
next i)
(do i=1,32
:N(i) "#R@C#S/400/" : *
next i)
]
```

In Ratcliff et al.'s (1986) system, #R waits for a keypress and records the key and the reaction time, @C clears the screen, and #W3000 waits for 3,000 msec. The #S command records the data between the slashes that follow. Thus, the reaction times to all primes will be associated with the number 200, all same-sentence targets with the number 100 plus their associated sentence's location in the study list, all different-sentence targets with the number 300 plus their associated sentence's location in the study list, and all negative fillers with the number 400.

The key feature that allows Listmaker to be used to generate real-time experiments is the ease of interfacing

(Example 1)

```
dim S(32),T(32),P(32),N(32),V(32)
open9:EXP1MATERIALS
(do i=1,32
read9:S(i),T(i),P(i)
next i)
read9:N
close9
(do s=1,8
@T,P
T1=8?T>X
P1=P<X
T2=8?T>Z
T3=16?T>Y
P3=P<Y
P2=8?P3
output:EXPLIST&s
write "Press space bar on keyboard to begin
  study phase#R@C"
[list=32
(do i=1,32
:S(i) "#W3000@C" : *V(i)
next i)
]
write "Press space bar on keyboard to begin test phase#R@C"
[list=64
(do i=1,8
l=X(i)
:T1(1) "#R@C#S/" 100+V(l) "/" : *a,a>3
:P1(i) "#R@C#S/200/" : a-1
l=Z(i)
:T2(i) "#R@C#S/" 300+V(l) "/" : *b,b>3
:P2(i) "#R@C#S/200/" : b-1
next i)
(do i=1,32
:N(i) "#R@C#S/400/" : *
next i)
]
next s)
```

to the Ratcliff et al. (1986) real-time system. In other real-time systems, the interfacing problem can be quite complicated (i.e., writing another program with real-time routines inserted). For a novice in microcomputer hardware, it would be difficult to build the system used by Ratcliff et al. (1986) because it requires building cables; ordering, installing, and setting interface cards; etc. For users without expertise, errors can be difficult to correct. However, it would be an extremely straightforward programming task to write a real-time system that emulates that of Ratcliff et al. on an IBM PC/AT, Apple Macintosh, or similar system using a compiled language. Benchmark timings on a Macintosh indicate that all features can be easily implemented. Listmaker can then be used on these systems.

Summary

Through the use of two examples, we have described some of the features of Listmaker. However, there are many more features that are not mentioned in the body of the paper, such as random draw with replacement, fancier control structures, and subroutines. The reader is

plete list of the features offered by Listmaker.

Listmaker provides a concise, flexible programming language to generate stimulus lists for cognitive psychology experiments. It reduces the amount of programming the experimenter must do by automating common operations, such as random draws without replacement. It also makes it easy to specify restrictions on the positions of items in the stimulus list. Another major benefit of Listmaker is its ability to treat multiline items as a single unit. We can assign an entire story to a single set item and never have to worry about its length again.

In addition to the reduction in programming time, the main benefit that Listmaker provides is clarity. The increase in clarity pays off in several ways. First, it reduces debugging time. Problems are easily identified because there are so many fewer lines of code than in a language such as FORTRAN. Second, since readability is improved, we can put down a Listmaker program, come back to it a month later, and be able to understand it in a matter of minutes. This leads to the third benefit: modifiability. If we want to modify the design of an experiment or add a condition to one already run, it is obvious what changes must be made to the Listmaker program.

Listmaker does have a few areas that could be improved. First, its syntax is very rigid compared with that of FORTRAN or Pascal, for example. Spacing, command formats, and so forth, are not flexible in Listmaker. This could be improved by developing a more sophisticated parser for the Listmaker interpreter. However, the syntax has not proven to be a significant problem, as the interpreter will identify the troublesome line, and the error is usually obvious. In fact, the rigidity may be a benefit to the experienced user.

The second area in which Listmaker could be improved is the range of legal variable and set names. Currently, scalar variable names are one letter, and set names are a letter and a digit. Obviously, mnemonic names would improve program readability even further. Again, a more sophisticated parser and a hashing scheme to keep track of variable and set names in use could solve this problem. A simplier approach would be to write a preprocessor that are described above. For example, the experimenter could store the mnemonic names in a file, and the program would replace these in the Listmaker file.

Despite these relatively minor limitations, Listmaker is an impressive tool. The examples provided in this article represent relatively simple experimental designs. To add more complex restrictions on output lists would require only minimal programming changes, because the materials sets are already defined and changes would be made only to the output lines. To add more complex counterbalancings or definitions of experimental conditions would require redefining the input sets, but again this is usually straightforward. As experimental designs become more complex, the advantages of Listmaker over other kinds of programs increase; in a Listmaker program, designs much more complex than those given in the examples do not look much more complex.

Practically speaking, Listmaker provides large savings in time. An experimental design that might take 2 days to implement and debug in FORTRAN will take a couple of hours with Listmaker. Changes in a design that might take several hours in FORTRAN take 20 min in Listmaker. Over a period of 10 months, McKoon and Ratcliff have run about 25 experiments, with an average time to produce a debugged, real-time experiment of about 3 h. In summary, Listmaker is a deceptively simple system that translates into surprisingly large savings in time and effort for implementations of experimental designs.

REFERENCES

McKoon, G., & Ratcliff, R. (1986). Inferences about predictable events. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, **12**, 82-91.

Ratcliff, R., & McKoon, G. (1978). Priming in item recognition: Evidence for the propositional structure of sentences. *Journal of Verbal Learning & Verbal Behavior*, **17**, 403-417.

Ratcliff, R., Pino, C., & Burns, W. T. (1986). An inexpensive real-time microcomputer-based cognitive laboratory system. *Behavior Research Methods, Instruments, & Computers*, **18**, 214-221.

Stevens, A. L., Levin, J. A., Olds, R. R., & Rumelhart, D. E. (1977). A computer system for automatically constructing stimulus material. *Behavior Research Methods & Instrumentation*, **9**, 269-273.

APPENDIX
Description of Listmaker Syntax

Each set is named with a capital letter from "A" to "T," optionally followed by a single digit (e.g., A, B6, R7). Each index set is named with a capital letter from "U" to "Z," optionally followed by a single digit (e.g., U1, X2). Individual items in a set are identified by appending a subscript to the set name [e.g., A(1), B6(102), X2(17)]. Scalar variables are named by a single lowercase letter, "a" to "z."

The following are illustrative examples of Listmaker syntax. A formal description of the syntax is not provided because it would be much lengthier and far less readable. In almost all cases listed below, variables, integers, and arithmetic expressions combining both can be used interchangeably.

Control Structures:

(do i=1,10	Repeats the lines between the "do" and the "next."

next i)	Variable *i* takes on values from 1 to 10. Loops may be nested indefinitely.
(repeat x until a=b,x)	Repeats the lines between the "repeat" and the "until" until the condition is true. The variable *x* is used only to identify the loop.
if x=1 then y=1	If the conditional is true, then perform the operation following "then." Conditional expressions may be complex: (x=1)+(x=7) is true if x=1 or x=7; (x>1)*(x<7) is true if x>1 and x<7.

if x=1 then
else
endif
	If the conditional is true, execute the lines until the "else"; otherwise, execute the lines between the "else" and the "endif."

if x=1 then
else
if x=2 then
else
endif
	As above, but allows further conditionalization within the "else" clause. "Else/if" cycles may go on indefinitely.

call subname	Transfers execution to the subroutine named "subname."
subroutine subname return	Defines a subroutine: the lines between the subroutine name and the return are executed when the subroutine is called.
File Input/Output:	
open9:filename	Opens file "filename" for input. Any digit may be used in this and all other file I/O statements instead of the "9" used here.
read9:A	Reads all of a set from a file.
read9:A(1)	Reads one item of a set from a file.
read9:A[1,10]	Reads 10 items of a set from a file.
read9(F10):A	Reads 10 lines from a file for each item in set A. The same options are allowed as above.
read9(DX):A	Reads all of set A from a file. Each item in set A consists of successive lines from the file until the character "X" is encountered by itself on a line. The same options are allowed as above.
readL:A	Reads all of a set from the lines immediately following in the program. Again, the same options are allowed.
read9:x	Reads the value of a variable from a file.
close9	Closes an input file.
output:filename	Sends all further output from output blocks and write statements to file "filename." Remains in effect until another output statement is encountered.
include filename	Copies the contents of the file "filename" to the output file. Useful for including instructions or practice items in the output file.
Set Declaration and Randomization:	
dim A(10),X(10)	Declares set A and index set X to be 10 items long. This is only necessary for sets that are to be read in from disk files or assigned via partial set assignments or input statements (see below).
@A,B,C,...	For regular sets, resets the randomization so that further draws without replacement may draw any item in the set.

APPENDIX (Continued)

@X,Y,Z,...	For index sets of size *n*, assigns each item in the set to be a random integer between 1 and *n*, such that each integer is used exactly once.
@~A,B,C,...	For regular sets, resets the randomization so that further draws without replacement may draw any item in the set, with the restriction that the *i*th draw will not draw item *i*.
@~X,Y,Z,...	For index sets of size *n*, assigns each item in the set to be a random integer between 1 and *n*, such that each integer is used exactly once and item number *i* will not have the value *i*.
@!A,X,...	Actually rearranges the order of items in the sets to a random order. (@ does not change the order of the items in a set; it only changes the order in which they will be drawn by the ? operator, below.)
Set Assignment:	
A=B	Set A is assigned to be a copy of set B.
A=B(1)	Set A is 1-item set consisting of the first item in set B.
A=B[6,10]	Set A is a 5-item set consisting of items 6 through 10 of set B.
A=6?B	Set A is a set of 6 items chosen randomly without replacement from set B.
A=6!B	Set A is a set of 6 items chosen randomly with replacement from set B.
A=3?B[6,10]	Set A is a set of 3 items chosen randomly from items 6 through 10 of set B.
A=B,C,D	Set A contains all the items in sets B, C, and D.
A=B(1),C[2,8],D	Set A contains the first item in B, items 2 through 8 of C, and all of D.
A=6?B(1),C[2,8],D	Set A contains 6 items chosen randomly without replacement from the above group of items.
Partial Set Assignment:	
A(1)=B(7)	The first item of set A is set equal to the seventh item of set B.
A[1,5]=B[6,10]	Items 1 to 5 of set A are set equal to items 6 to 10 of set B, respectively.
Index Set Assignment:	
>X	Appended to any of the above full or partial set assignments, stores in the index set X the locations in the set on the right from which the items assigned to the set on the left were taken.
<X	Appended to any nonrandom full or partial set assignment, assigns to the set on the left those items in the set on the right in the locations specified in the index set X. (See text for examples of the use of these operators.)
Output Blocks:	
[list=10	Marks the beginning of an output block that will create an output list of 10 items.
]	Marks the end of an output block.
:A(1) x "zzzz": *	An output line consists of a materials section and a location descriptor. The materials section goes between the two colons and may include any combination of set items, variables, and quoted text. The values of set items and variables and the contents of the quoted text will be put in the output list in

the position specified by the location descriptor, which follows the second colon. An "*" indicates that the output from this line may go at any position in the output list. Note that an output line may generate more than one line of output; for example, A(1) might contain a multiline story in this case.

Location descriptors:

A location descriptor may be a numeric expression, in which case the output will be placed at that position in the list:

:A(1): 7 :A(1): a	Places the contents of A(1) at a fixed position in the output list.

A location descriptor may also be an "*," followed by a variable, followed by restrictions, in which case the output will be placed at a random position in the list as long as it meets the restrictions. The variable will contain the location to which the output was assigned:

:A(1): *a,a>7	Places the contents of A(1) at any position greater than 7 in the list.
:A(1):*b,(b>a)@(b<c)	Places the contents of A(1) in any position greater than a and less than c.
:A(1): *b,(b<a)+(b>c)	Places the contents of A(1) in any position less than a or greater than c.

Miscellaneous and Debugging Commands:

random(x)	Reseeds the computer's random number generator.
x=rnd(y)	Variable *x* is assigned a random integer value between 1 and *y*, inclusive.
x=rnd(y,z)	Variable *x* is assigned a random integer value between *y* and *z*, inclusive.
input A(1)	Input item 1 of set A from the keyboard at run time.
input x	Input variable *x* from the keyboard at run time.
print A(1) print "Wait" print x	The value of the set item, quoted string, variable, or any combination is printed on the screen.
trace on	The Listmaker interpreter will print out each line before it is executed.
trace off	Turns trace off (the default).
stats	The Listmaker interpreter prints out useful statistics about memory usage, etc.
stats,sets	The Listmaker interpreter prints out the same information, and also prints the size of every set in use.
tries=50	Tells the Listmaker interpreter how many times to try to construct an output block before giving up. Sometimes output blocks with complicated restrictions cannot be constructed on the first try. The default is 100.