SESSION XI DATA GENERAL COMPUTER USERS GROUP

JOHN F. KING, *Professional Data Services, Inc., President*

A high-level language for control of psychology experiments

RICHARD F. DILLON, LOUIS R. DUMONTIER, WILLIAM R. BEZANSON,
and RODGER BRENNAN

Carleton University, Ottawa, Ontario, Canada K1S 5B6

A language for controlling learning, memory, and perception experiments is described. The language features the capability of manipulating large amounts of alphanumeric information with ease. A small, but powerful, command set uses the psychologists', rather than the programmers', logic and jargon.

A programming language must correspond closely to the manner in which we conceive experimental events. It must be simple, yet powerful. Rather than attempting to force the user to fit his thinking into the lower level logic found in typical high-level general-purpose languages such as FORTRAN, we must make the programming language fit the logic of the experimenter. We believe that most psychologists do not think about experimental control in terms of do loop, for-next, and jump statements. For example, if a user wants to perform 20 trials, he should be able to tell the computer to perform 20 trials, and not be concerned with loops that require decrementing counters to determine if 20 trials are completed. We cannot expect the occasional or one-time user to learn and remember the differences between object and source codes; between constants, variables (local and global), literals and labels; between operators and operands; between monitors, loaders, linkers, and assemblers. In short, a good language must allow the user to concentrate on what he wants to do, not how it is to be done.

The last paragraph illustrates how our thinking differs from that of the authors of many other languages. After emphasizing the need to make languages meaningful and easy to use for psychologists, other authors describe languages that are FORTRAN-like

(Halff, Scholz, & Walker, 1975; Kieras, 1973; Schneider & Scholz, 1973; Scholz, 1973), ALGOL-like (McLean, 1973), PL/1-like (Pinkus & Gregg, 1973), use assembler or subroutine calls in FOCAL (Link, 1975; Mullen, 1976; Siegel, 1972), use assembly language subroutines within assembly language (Kaplan, 1976), or use something higher level than assembly language but lower level than FORTRAN (Aaronson & Brauth, 1972; Burkhardt, 1976; Getty, 1975; Pitz, 1975). These authors assume either that (1) FORTRAN, etc., is easy to learn or, more likely, that (2) the user language will be easy to learn for the psychologist who already knows FORTRAN (or ALGOL, assembler, etc.), because of common similarities between the languages. We have attempted to design a language that is meaningful and easy to use for the psychologist who knows no programming language and nothing about computers, but who does know how to design experiments. Our language may best be described as "nonprocedural." Languages concentrating on what, not how, are known as very high-level languages (Leavenworth, 1974).

Early user languages suffered from hardware constraints that limited the opportunity to accomplish the goal of attracting the occasional user. These limitations have changed dramatically with the accelerating pace of computer hardware technology and the impact of the technology on marketing. A few years ago, the high cost of computer memory dictated languages which ran in 4K or, at most, 8K words of memory, often with an inconvenient and limited command set. Now that 16K word boards cost less than 4K boards did a few years ago, there

Development of this system was supported by grants from Carleton University and the National Research Council of Canada to R. Dillon, and from the National Research Council to W. Petrusic and to B. Jones. Requests for reprints should be sent to Richard F. Dillon, Department of Psychology, Carleton University, Ottawa, Ontario, Canada K1S 5B6.

is less reason why memory size should constrain a language's use.

The design of special-purpose languages before megabyte cartridge disk systems were affordable, and before floppy disks were even on the market, was constrained because most psychologists were struggling with paper-tape programs. Such programming was often done in a stand-alone environment or within fairly unsophisticated operating systems. A user could easily agree with Getty (1975) that a language should not require a special compiler. Now that most users with sophisticated operating systems can detect an error, make a minor change using an editor, and compile in less time than it takes to gather one's program off the floor and figure out which way the fan folds fold, a compiler/assembler approach, executing faster than an interpreter, makes a great deal of sense.

We have developed a system which, with many hardware constraints now removed, goes a long way toward fulfilling the needs of the majority of psychologists who do not want to be programmers. In our system a user prepares a list of commands using an editor and stores the sequence on a command file. He prepares a separate data file consisting of instructions to the subjects, learning items, durations, repeat counts, etc. He then activates the program which runs the experiment that was specified in the command and data files.

COMMAND DEFINITIONS

We have implemented a limited repertoire of commands, each of which incorporates, as completely as possible, the information involved in some logical high-level operation. For example, when setting up an experiment, we do not simply display. We (1) display (2) some information (3) on some display device at (4) some location on the display device for (5) some duration. We may or may not want to (6) blank the display when we are done, (7) sense a response to the display, (8) store the response and latency for later output, (9) terminate the command when the response occurs, and (10) start timing from the first or last item displayed. The ability to specify all of these components makes up an experimental entity, and they are all incorporated in a single command. Twenty multicomponent instructions are the basic logical units of the language. They are treated as "off-the-shelf" building blocks that the psychologist can tailor to his needs and assemble together as appropriate.

In addition to keeping the number of instructions small, we have attempted to minimize syntactic idiosyncracies. Three simple types of syntactic structures are used for commands. First, most commands consist of entering a command name with a number to identify it (e.g., DISPLAY47) and a series of parameters separated by spaces on a single line. Once

the nine parameters described above are specified, a reference to DISP47 is the only thing necessary to present, for example, an item to be learned in a memory experiment. Second, a few commands consist of a single word with no reference number and no parameters on a single line. WAIT, for example, will hold up everything unconditionally until any key is pressed on a keyboard. The third type of syntax, used for the SUBROUTINE and DECISION commands requires (1) the name and number of the command on one line, (2) a number of lines of information, and (3) the word END on another line to indicate the end of the command. If, for example, DISP47 presents a single recall item, DISP48 a recall test, and DISP49 an ITI, SUB1 might present 10 items, the recall test for one item and the ITI to make up a trial:

```
SUB1
  DISP47 10
  DISP48 1
  DISP49 1
END
```

Within a subroutine, the first entry on a line is the name of a command to be performed and the second is the number of times the command is to be performed.

Any number of commands may be combined in an ordered sequence within a subroutine, which may itself be one component of a higher order subroutine, etc. In the example above, once a basic trial is defined as SUB1, an experimental session of 24 trials can be defined as SUB2:

```
SUB2
  SUB1 24
END
```

A simple anticipation method paired associate learning experiment, the sort of thing one would do on a memory drum, could be programmed using only DISPLAY commands. Most of the experiments run so far have used only the DISPLAY, SUBROUTINE, DECISION, DRIVER, WRITE, and WAIT commands.

Although there are only a few commands, most require specification of a number of parameters. Two summary sheets, one for the six most commonly used commands (see Figure 1) and another for the remaining 14 (see Figure 2), are provided to all users to minimize the need to remember what parameters are required for each command. The command line

```
DISPLAY47 $TT01 WORD2 LOC4 DUR27 Y Y Y N N
```

seems obscure, but a glance at the summary sheet indicates that Display 47 will display on device

Basic Commands. Commands are defined by filling in boxes with at least the first three letters of the command name plus a 1 or 2 digit number plus parameters. VALUE refers to the number desired. ARRAY REFERENCE refers to an array which contains the desired information. COMMAND REFERENCE is the name of a command.

Figure 1. Summary of basic commands.

\$TTO1 (a video monitor) a line of array WORD2 at the location on the video monitor found in array LOC4 for the duration found in array DUR27. The screen is blanked at the end of the command; a response is expected and the latency and response are automatically stored (in WORD98 and WORD99,

respectively); the display is on for the full duration, even if the subject responds before the time is up, and timing starts after the last character is displayed. (See the section on Data Definition for an explanation of WORD98, etc.)

Eight dc drivers operate lights, tones, slide projec-

Figure 2. System operation.

tors, etc. Such devices all have interchangeable plugs that allow the user to connect any device to any driver. The DRIVER command allows the user to program such devices. Referring to Figure 1, it can be seen that the command line

```
DRIV6 1 167 Y 1800 Y Y Y
```

turns on Drivers 1, 6, and 7, options are desired to specify that the drivers should be on for 1,800 msec, a response is expected, the response character and latency are stored, and the drivers are turned off when the subject responds or after 1,800 msec, whichever occurs first. Drivers 1, 6, and 7 might be connected to a light, a tone, and a shock source, respectively. If a user does not want to operate all three devices simultaneously, three different DRIVER commands can be defined. For example,

```
DRIV6 1 1 Y 1800 N N N LIGHT
DRIV7 1 6 N TONE
DRIV8 1 7 Y 1800 Y Y Y SHOCK
```

can be grouped together as SUB3 as follows:

```
:C SHOCK SEQUENCE
SUB3
  DRIV6 1 LIGHT
  DRIV7 1 TONE
  DRIV8 1 SHOCK
END
```

A reference to SUB3 turns on the light for 1,800 msec, the tone unconditionally, and the shock until the subject turns it off by responding or until 1,800 msec are up. If the subject presses a key, the response character and response latency are stored for later output; if the subject does not respond, a special character and 1,800 are stored for output.

Since the most recent response and response latency are contained in WORD99 and WORD98, respectively, feedback or scoring subroutines can be accomplished easily using the DECISION command. Referring to the summary of the DECISION command in Figure 1, the following sequence is easily interpreted.

```
DECISION 1
WORD99 EQ WORD7 AND
WORD98 LT 501 PERFORM SUB4
WORD99 NE WORD7 OR
WORD98 GT 500 PERFORM SUB5
END
```

Assuming WORD7 contains the correct response, DECISION1 allows one to perform a subroutine (SUB4) if responses are correct and latencies short enough (<500 msec); DECISION1 allows for another subroutine (SUB5) if responses are incorrect or latencies too long. Decisions can be arbitrarily complex, with any number of commands performed, as long as their antecedent conditions are satisfied.

In the previous example, SUB4 and SUB5, in addition to scoring or feedback routines, might output data to any output device (disk file, Teletype, etc.) using the WRITE command summarized in Figure 1. For example,

```
WRIT46 $TT0 WORD99 RESPONSE
WRIT47 $TT0 WORD98 LATENCY
```

provide a record of response and latency for the current trial. In the summary of the WRITE command, an output device name or file name may be specified, or the user may provide the name of an array which, in turn, contains the name of a device or file. These examples illustrate that the summary sheet, with a more accurate "memory" than most of our users, is designed as a constant reference during program development.

In addition to a small number of commands, simple command structure, and a summary sheet, syntax within each line was designed to avoid unnecessary and error-provoking demands on the user. Parameters may be separated by space, comma, tab, or any combination of these. The name of a command may be DISP47, DISP 47, DISPLAY47, DISPX LAY 47. Such symbols as semicolon, slash, asterisk, and parentheses, which are common sources of error, are not used, although the colon is used with keywords (e.g., :C). To improve readability, command lines may be indented as far as wanted, blank lines may

appear anywhere in the program, and comments may be placed (1) on any line of which the first two nonspace characters are “:C” or (2) at the end of any command line with at least one space character preceding the comment. All of these facilities were used in the DRIVER command example above. In all examples in this paper, information to the right of the last parameter is a comment.

It is worth repeating that a high-level language must be readable to be useful. It has often been assumed that readable implies a command form that approximates English prose. We feel that a language can be readable even if it is not English-like, if it is structured logically, uses the jargon of the application, and allows the user to concentrate on what is to be accomplished rather than how to accomplish it. Moreover, the consistent use of conventions involving comments, blank lines, and indentations by all our users, together with a readable command summary, can go a long way toward making code readable. There is another fact which convinces us that our rather terse commands are readily acceptable to users. When developing a program, the user programs by thinking something like, “First I need a DISPLAY to . . . , then four DRIVERS to . . . , then a DECISION to see if the response is correct.” The point is that users think at the macroscopic level of command type rather than at the microscopic level of parameters to be specified. Our users indicate that they attend to the parameters only when commands are first defined and when errors in a command are detected. This “bottom-up” programming style corresponds nicely to our way of thinking about events in an experiment. Ease of programming is further enhanced by providing maximum flexibility in the order in which command definitions may appear. It is necessary, of course, to define all commands before specifying the final program sequence. For this reason, the command file is divided into two parts: the command definitions in the first part and the program sequence in the second. These are separated by the line :PROGRAM (or :PRO). Beyond this, there are no restrictions on order of command definition. A command may be referred to (e.g., within a subroutine) before or after it is defined. We recommend that users attempt to group components of subroutines together, either before the subroutine is defined or after, but not both, whenever possible.

The program sequence consists of a number of command names, one per line, in the order desired to perform the experiment. At one extreme, the program sequence can consist of the name of a single SUBROUTINE defined to contain many other SUBROUTINES. The DECISION command makes what might appear to be a fixed sequence remarkably flexible.

All of the above features are relatively useless without extensive error trapping and reporting. This

feature is one of the most important in determining the ease of using a language. A program (the Command Encoder) translates the commands into a special internal structure. As each command line is processed, syntactic errors are flagged by printing a message and echoing that line. A typical error message is:

****Illegal Option Specification in:**

DISP47 \$TT01 WORD2 DUR47 Y 1 N N N

After errors of this sort are detected and reported, the program tries to recover and continue. There are very few errors that cannot be detected at this stage. The few semantic errors that are fatal will return control to the computer operating system.

DATA DEFINITION

The emphasis so far in this paper has been on how and why we have attempted to simplify controlling the sequence of events with our system. We have also attempted to simplify the use of numbers and strings. The user, in fact, prepares two files—the command file outlined above and a parallel data file that is accessed by the commands. The need to be able to manipulate masses of alphanumeric information (instructions, learning items, durations, repeat counts, etc.) dictated an array structure. Five types of arrays are provided: WORD, DURATION, REPEAT count, GENERAL, and LOCATION. WORD arrays contain string information in ASCII form. DURATION, REPEAT count, and GENERAL arrays contain numeric information with which the user deals in ASCII decimal, but which is stored internally in binary. LOCATION arrays allow the user to specify absolute or relative locations on a video display. Although the differences between DURATION and REPEAT count arrays have disappeared, and the two are perfectly interchangeable, we have kept both. A number in ASCII form in a WORD array must be moved to one of the three types of numeric arrays to serve as a duration, repeat count, etc., or to be used in an arithmetic operation. Any conversion necessary is handled by the MOVE command summarized in Figure 3 and is of no concern to the user.

There are 99 of each type of array available, identified by at least the first three letters and a number between 1 and 99 (e.g., WORD25). For the GEN arrays, there can be a single entry between -32,000 and +32,000 for each array reference. For example, GEN9 might contain the value 300. The other four types of arrays can have any number of items within an array. For example, WORD10 might consist of 15 lines of instructions to subjects, and WORD41, 120 recall items (e.g., 12 trials of 10 words/trial). Any array with more than one entry is automatically provided with a pointer to the current item. If an

array name and number appear in a command with a plus sign (+) appended to it, the pointer is automatically incremented to the next item before the command is executed. A reference to an array with no plus sign causes the pointer to remain at the current item. If the end of an array is reached through use of the + mechanism, the pointer "wraps around" to the beginning of the array for the next execution, allowing repeated cycling through the array. Another way to alter the pointer for an array is with the SET command (see Figure 3), which moves it to the absolute or relative position specified.

The process of defining arrays has been made as simple as possible. The data file, like the command file, is prepared using the computer text editor. The keyword :DEF and then the array name on one line are followed by as many lines as desired with string information (WORD arrays), numeric information (DUR, REP, GEN arrays), or location information (LOC arrays). The end of an array definition is indicated by the word :END on a separate line. Arrays may be defined in any order. For example, in the command file a display command can be defined to present a ready signal, a recall signal, and then an ITI signal on the video monitor (refer to the DISPLAY summary in Figure 1).

```
DISPLAY1 $TT01 WORD1 + LOC1 DUR1 + Y N N N N
```

In the data file, WORD1, LOC1, and DUR1 are defined.

```
:DEF WORD 1
READY
RECALL
REST
:END
:DEF LOC1
20 12          row 20, column 12
:END
:DEF DUR1
2000          ready duration
10000         recall duration
4000          rest duration
:END
```

Within a data definition, a line not including comments, terminated by a carriage return, is an item. When DISP1 is first encountered in the program sequence, the word "ready" appears on the video display starting in row 20, column 12, and stays on the monitor for 2 sec. The next time DISP1 is encountered, the word "recall" appears on the screen for 10 sec. Finally, "rest" is visible for 4 sec. All three messages appear starting in row 20, column 12 since there is only one item in LOC1. However, DISP1 steps through the WORD1 and DUR1 arrays,

then the item pointers wrap around to the beginning of each array for the next trial because WORD1 and DUR1 are referenced with a + in the definition of DISP1. Note that comment lines, blank lines, and comments at the end of all lines, except text within WORD definitions, can improve readability of the array definitions.

In addition to defining arrays, it is possible to reserve arrays (e.g., for output buffers) in the data file. The entry

```
:C      Reserve output buffer
:RES WORD7 20 10
```

sets up WORD7 with pointers, etc., and room for up to 20 items of up to 10 ASCII characters/item. Like most characteristics of the language, once these provisions are made, the user does not typically know or care about the text string structure except how to access the contents of arrays.

Provision of the data array structure has an important advantage in addition to simplifying string handling. When faced with the decision to require the user to specify a parameter (e.g., the number of times a command is to be executed in a subroutine) as either a constant or a variable, we have provided a choice wherever possible. If the user knows that a command will be required four times within a subroutine every time the subroutine is used, the value "4" can be entered. On the other hand, the number of times a command is to be performed may vary (e.g., the positive set size from trial to trial in a Sternberg recognition experiment). The values can be built into an array, for example, REP61, and the name of the array with a +, rather than a constant value, entered in the command definition. The program simply steps through the array in order, taking the next entry each time, as illustrated above. The ability to move items between arrays (see the MOVE command summary in Figure 3), to move items conditionally (see the DECISION command), and to SET the item pointers of arrays conditionally or unconditionally makes this feature virtually unlimited in flexibility.

A processor must be relatively "smart" to be able to assume responsibility for understanding things that are often a burden on the user, things like where in a line the command starts, whether an entry is a constant or variable, etc. Our processor (a two-pass compiler) is smart enough to allow the user to include, for example, a LOCATION array reference in a DISPLAY command if a device which requires such information is plugged into the \$TT01 port, but to omit the LOCATION reference if such information is not needed. This, once again, illustrates our emphasis on providing mechanisms for specifying what, and not how.

Figure 3. Summary of advanced commands.

IMPLEMENTATION

Our system is implemented on a Data General Nova 1220 computer with 32K words of core memory, an 11.6-megabyte disk (5.8 megabytes fixed and 5.8 removable), two cassette units which are rarely used, a high-speed paper-tape reader and punch which are never used anymore, real-time clock, DECwriter II terminal, a 24 by 80 character video display and keyboard, a 32-character Burroughs Selfscan panel display, eight dc drivers for controlling lights, tones, projectors, etc., and interfacing to transfer files back and forth between the Nova and the Carleton University Xerox Sigma 9 computer. The language processor, written in assembly language, is designed to run a single subject at a time. We considered, and rejected, the possibility of a timesharing version. With the reasonable cost of microprocessors, separate subject stations, each with its own computer, are more practical than a time-sharing language.

We use Data General's RDOS operating system (Rev 3) and have used the RDOS tasking capabilities to simplify program coordination. Because of the use of RDOS task calls, the minimum time between any two commands is a nominal 2 msec (actually, 1.56 msec). Timing of driver durations, response latencies, etc., is to an accuracy of 2 msec. Users of more modern Data General computers can probably expect better timing accuracy. Reentrant coding makes it possible to have, for example, subroutines nested within other subroutines indefinitely. This feature, although useful, is provided at a cost in time between instructions. The time between instructions called within subroutines is given by 1.56 raised to a power equal to the highest level of nesting. Users who are aware of this limitation can avoid the use of subroutines if they want to maintain 2-msec timing between commands.

OPERATING PROCEDURES

An overview of the system operation is shown in Figure 2. Once the command file and data file are prepared using the text editor (Step 1), the user calls the Command Encoder (Step 2), which processes the command file. The user is asked where he wants error messages directed. He is then asked the name of the command file. If a line beginning with :X (for optional commands) is encountered, the user is asked whether or not these lines are to be included. The file is processed and any error messages are written out. Unknown to the user, the processed version of the file is written out to an intermediate file. The Command Encoder then automatically loads the Data Encoder program (Step 3). The user is asked the name of his data file, and then asked the max-

imum number of responses he wants to store in the experiment. The array file is processed, error messages written out, and an intermediate array file created. The Data Encoder then loads the Run-time Supervisor, which actually runs the experiment (Step 4). This program reads in the two intermediate files created previously and tells the user to strike a key to begin. Note that the user has only two explicit steps to perform; the rest are automatic.

Individuals accustomed to paper or cassette tape systems might think all this would be time consuming. On the contrary, the program is ready to run an experiment in a few seconds. Our users do not realize that there are several distinct programs and intermediate files. If errors are reported at any stage, the user interrupts the system, calls the editor, and makes the necessary changes before restarting.

DEBUGGING

Because error detection and reporting are extensive, debugging is remarkably straightforward. Nevertheless, two special debugging features are provided. First is a trace feature. Anytime Bit 0 of the computer switches is set, the computer stops at the beginning of each command (except DECISION and SUBROUTINE), disables interrupts from the real-time clock, writes the command name on the terminal, and pauses until a key is struck on the terminal. Users who run into serious debugging problems can use the language commands as debugging aids. For example, special previously prepared messages can be written to the video monitor or terminal using the DISPLAY and WRITE commands. These commands are entered with :X at the start of the line. Such lines can be included when debugging, but may not be included for the actual experiment once bugs are removed.

PORTABILITY

The language should function on any Data General computer with Rev 3 RDOS or higher. Running under Data General's RTOS requires some changes but, we are told, results in better timing accuracy. Since the programs are all modular and all relevant parameters (e.g., RDOS device code numbers) are included in tables, I/O modifications should be minimal. Only two substantial assembly language changes are necessary for the DISPLAY and the DRIVER commands. Special functions in our character generator (e.g., cursor addressing) differ from other units and require changes in the display module and in the handling of location arrays in the Database Encoder program. Our driver command works by sending a bit pattern (of no concern to the user) from an accumulator. Other techniques require mod-

ification to our code. The addition of new commands is relatively easy to implement, but requires a substantial effort by anyone unfamiliar with the internal logic and structure of the language.

In terms of hardware, a disk and a minimum of 24K words of memory are required. Although large data files require 32K, it is possible to program around this limitation by using the READ command (see Figure 3) to read a disk file into arrays as one progresses through the experiment. With the prices of core memory and disks so low the "bit bashing" associated with trying to operate with 4K or 8K words of memory is unnecessary. Having graduated from paper tape to cassette tape to disk, there is no doubt that a disk is a necessity.

ENHANCEMENTS

Improving error detection and reporting, and improving the flexibility and capabilities of existing commands, is an ongoing process. We still have features that have not been implemented. For example, we have designed and partially programmed session initialization capabilities for moving blocks of data among arrays. The randomization command summarized in Figure 3 has not been fully implemented because, with auxiliary randomization programs available to us, completion of the command always seems to have lower priority than something else.

We are currently adding A/D and D/A capabilities. It is expected that the data handling required for such hardware will fit nicely into our array structure and that A/D and D/A commands will be included in the language.

We are considering the possibility of attempting to eliminate the 1.56-msec interval between commands by programming our own tasking capabilities.

Finally, although technical documentation on the system is reasonably complete, user documentation, as is often the case, has lagged far behind. We hope to remedy this problem with a set of introductory and intermediate manuals on the use of the language, on selected aspects of the RDOS operating system, and on the text editor.

EXPERIENCE

Seven people have used the programming system since it became operational in early 1977. Only two had experience on minicomputers. All except one were familiar with FORTRAN programming. A number of experiments on auditory signal detection, duration judgment, line length judgment, and short-term memory using the Brown-Peterson paradigm have been programmed. Three facts are apparent.

First, even naive programmers are capable of fairly sophisticated programming if an appropriate language is available. Second, most users learn the programming system quickly, even though we currently have only limited teaching documentation. Third, the largest obstacle for all new users has been, not the language, but the necessity to learn how to operate the computer, how to use the RDOS operating system, and how to use the NOVA text editor.

CONCLUSIONS

Psychologists who want to use computers for experimental control have three options concerning the programming problem: (1) hire a competent programmer, (2) spend a lot of time becoming a reasonably competent programmer, or (3) program without becoming a programmer. Pilla (1973) advocates the first option, an approach that is very expensive and will not work for the occasional or one-time user. Our experience with statistics and mathematical statisticians provides some insight into the problems we can expect from such an approach. We agree with Pilla (1973) that the second option has a number of shortcomings. The third option, as contradictory as it appears, can be accomplished with an appropriate language for the occasional user.

Psychologists and computing scientists do not typically design new peripheral equipment and interfaces for each experiment. Most of us know very little about hardware except what some device accomplishes. We are rapidly moving toward a turn-key computer system with plug-compatible research modules that can be selected "off the shelf" as needed for different experiments. We envision, and are working toward, the time when psychologists will be able to program using plug-in off-the-shelf modules. The reader can evaluate the success we have had in designing such meaningful building blocks by examining the information contained, for example, in the DISPLAY command, and deciding what is included that should not be, and what is not included that should be.

REFERENCES

- AARONSON, D., & BRAUTH, S. SIMPLE guidelines for developing a computer-based laboratory. *Behavior Research Methods & Instrumentation*, 1972, 4, 257-264.
- BURKHARDT, K. J., JR. EMPP: An extensible multiprogramming system for experimental psychology. *Behavior Research Methods & Instrumentation*, 1976, 8, 239-244.
- GETTY, D. J. The PEPL system for control of experiments by a PDP-8 computer. *Behavior Research Methods & Instrumentation*, 1975, 7, 131-136.
- HALFF, H. M., SCHOLZ, K. W., & WALKER, J. H. An extension of Kieras' general experiment programming system. *Behavior Research Methods & Instrumentation*, 1975, 7, 464-470.

- KAPLAN, H. L. Some debugging strategies for assembler-based minicomputer operating systems. *Behavior Research Methods & Instrumentation*, 1976, **8**, 170-173.
- KIERAS, D. A general experiment programming system for the IBM 1800. *Behavior Research Methods & Instrumentation*, 1973, **5**, 235-239.
- LEAVENWORTH, B. (Ed.). *Proceedings of a symposium on very high-level languages* (Vol. 9). Association for Computing Machinery, Special Interest Group on Programming Languages Notices, 1974. P. 4.
- LINK, S. W. 4K laboratory FOCAL. *Behavior Research Methods & Instrumentation*, 1975, **7**, 137-142.
- MCLEAN, R. S. PSYCHOL: A computer language for experimentation. *Behavior Research Methods & Instrumentation*, 1969, **1**, 323-328.
- MULLEN, S. FOCLAB: A language for computer-controlled psychology research. *Behavior Research Methods & Instrumentation*, 1976, **8**, 229-232.
- PILLA, M. A. Co-existing with on-line systems. *Behavior Research Methods & Instrumentation*, 1973, **5**, 80-82.
- PINKUS, A. L., & GREGG, L. W. APCOL: A programming system for computer-controlled psychology laboratories. *Behavior Research Methods & Instrumentation*, 1973, **5**, 165-172.
- PITZ, G. F. Building a programming language for a small computer: Reinventing the wheel. *Behavior Research Methods & Instrumentation*, 1975, **7**, 42-46.
- SCHNEIDER, W., & SCHOLZ, K. W. Requirements for mini-computer operating systems for human experimentation and an implementation on a 4K PDP-8 computer. *Behavior Research Methods & Instrumentation*, 1973, **5**, 173-177.
- SCHOLZ, K. W. PROSS: A process control programming language. *Behavior Research Methods & Instrumentation*, 1973, **5**, 245-247.
- SIEGEL, W. Combining FOCAL and assembly language. *Behavior Research Methods & Instrumentation*, 1972, **4**, 105-106.