THE EXCHECK CAI SYSTEM

by

JAMES MCDONALD

Institute for Mathematical Studies in the Social Sciences
Stanford University

EXCHECK is a CAI system which has been used at Stanford since 1975 to teach over 250 students per year in elementary logic (through predicate calculus), since 1976 to teach about 25 students per year in axiomatic set theory, and since 1980 to teach a small number of students each year in introductory Armenian. While it is a general purpose instructional system used principally to create and present entire university-level courses, EXCHECK can also be used to produce smaller modules for use as part of other courses.

A primary goal in the design of EXCHECK has been the creation of a system which can provide quality instruction in university-level courses, particularly those of a mathematical nature. An important means to that end has been the development of powerful packages which provide technological leverage to lesson authors by automating the dialog for large classes of exercises.

A strongly related goal has been that the system be readily accessible to both students and lesson authors. In particular, students are not presumed to have knowledge beyond that normally assumed for the course being taught, and lesson authors are not presumed to have skills beyond those of a good instructor for the course. In general, attempts have been made to attain as natural a level of dialog as possible, throughout the system.

The work described here builds upon earlier work by Robert Smith, and later work by Lee Blaine and James McDonald. Numerous people at IMSSS have contributed to the actual programming of EXCHECK, including Benjamin Cohen, Scott Daniels, David Ferris, Arvin Levine, Lawrence Markosian, Jos Marlowe, Vesco Marinov, Teri Pettit, John Prebus, Ron Roberts, and William Sanders.

Special thanks are given to Arvin Levine for substantial editorial assistance on an earlier draft. The author also wishes to thank Patrick Suppes for comments on the final draft.

The research reported in this article was partially supported by National Science Foundation Grants SED-74-15016 A02, SED-74-15016 A03 and SED-77-09698.

1. GENERAL STRUCTURE

For the reasons given above, it is not surprising that EXCHECK is a large system. Appendix A gives a rough description of the size of various components in the runtime configuration for a single user.

The runtime EXCHECK program (that which actually prepares and/or interprets lessons) has a modular design in which various functions have been segregated into separate programs, as seen in Figure 1. Using features inherent in the TENEX operating system (described in more detail below), these programs run as a single large program with interacting components. Lesson authors use the entire structure shown (and perhaps more), while students run a program that is some subset of the graph descending from the course driver.[1]

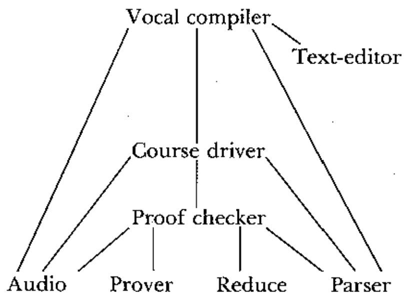

FIGURE 1. Structure of runtime EXCHECK system.

All EXCHECK lessons are written in VOCAL (Voice Oriented Curriculum Authoring Language). The VOCAL compiler is a program which accepts those lessons and produces files of lessons acceptable as input to the EXCHECK interpreter. The course driver interprets lessons prepared by the compiler and governs each student's progress through the course. It may invoke a proof checker to automatically present complex exercises such as the derivation of proofs or the construction of explicit mathematical examples. The proof checker has access to a theorem prover and to REDUCE, a program for algebraic reductions (Hearn, 1973). An audio program is available to access the speech synthesizing capabilities at IMSSS (Sanders & Levine, 1981), and each EXCHECK course has its own parser to interpret complex input from the student.

In addition to being composed of interacting sub-programs, EXCHECK also relies upon THRYEX, a program which prepares and tests the files

[1]For the Armenian course, the proof checker, prover, parser, and reduce are replaced by a component that provides drill and practice in Armenian. The audio facility is expanded to include Armenian as well as English dialog.

used to describe a mathematical theory, plus other programs which handle day-to-day maintenance and perform miscellaneous other chores. These support programs (as well as EXCHECK proper) in turn rely upon the TENEX operating system, as modified by IMSSS, and upon the excellent text editors, message handlers, etc., which have evolved under TENEX.

The program clearly has been influenced by the TENEX environment. In fact, many of the basic features of the program were patterned after similar TENEX conventions. Thus, it would be a mistake to analyze EXCHECK (as it exists at IMSSS) outside of this context.[2]

2. OPERATING ENVIRONMENT

The EXCHECK system currently runs under the IMSSS version of the TENEX (multiprocessing) operating system on a dual-processor PDP-10 computer system with 512,000 36-bit words of memory, 5 disk drives, 2 drums, and a multi-channel audio synthesizer (Sanders & Levine, 1981). A special part of the operating system, called the student executive, simplifies student contact with the computer and facilitates the management of information about the behavior of students in each course. The EXCHECK design reflects the TENEX environment by having independent programs, referred to as forks, for its different functions. While EXCHECK does make use of TENEX features specific to IMSSS, it does not have privileged access to the operating system.

2.1 *Student Executive*

Students are a special type of user for IMSSS TENEX, with a login procedure, called SEXEC, which limits them to exactly the course-program for which they are registered. This login procedure is started by the system whenever the space bar is typed on a free terminal. The student types his or her special student number, then first name or nickname. In this and following examples, student input is underlined. Other printing was done by the computer. The special keys 'space bar' and 'carriage return' are represented as '⟨space⟩' and '⟨return⟩', respectively.

```
<space>
Hi

Please type your number and name.
360John Smith<return>
```

[2]This does not imply that EXCHECK could not be run in other environments. Work on transporting a significant portion of EXCHECK is proceeding, and methods have been developed to minimize and compartmentalize system dependencies. However, our only substantial experience to date with EXCHECK has been that of running lessons for Stanford students under the TENEX operating system on a PDP-10 which is enhanced by special hardware for generating synthesized audio.

After checking the student's number and name, SEXEC logs the student in, updates information concerning use of the computer, and saves the student's number and name in a system buffer assigned to that job. It then determines the class the student is in, and overlays itself with the associated course-program. That program in turn reads the student's number and name from the system buffer and proceeds with the curriculum. In general, students are not given direct access to the TENEX system unless their course program explicitly provides such access.

Students are organized into classes associated with specific course-programs, and a separate program maintains the database for student users, creates classes, assigns students to classes, lists class members and login statistics, and searches for students by name.

2.2 *Data Saving*

As Suppes and Sheehan (1981a, 1981b) indicate, a great deal of data has been collected on various aspects of student usage of EXCHECK. To facilitate system performance in maintaining student statistics, a special data area on the disks, known as the DATSV area, was provided. This avoids the overhead of creating files on the system's dynamic file structure for data that are saved regarding the student's interaction with a course-program. A DATSV command given by EXCHECK writes out a 128-word block of data to the DATSV area, including the date and time, information about student progress, and links to any previous block saved. The total DATSV area is accessed as a ring of four buffers, with each buffer holding approximately two or three days of data for all students using EXCHECK. The DATSV command automatically switches buffer areas as one becomes full, and signals the operator that the data can be sorted and dumped onto tape for long term storage and analysis.

2.3 *TV-EDIT*

TV-EDIT is a program developed at IMSSS for full screen text editing (Kanerva, 1973). It is not part of the operating system per se, but ease of use has made it the editor of choice at IMSSS. The VOCAL compiler is given runtime access to TV-EDIT to facilitate rapid editing of lessons during compilation sessions.

2.4 *Forks*

The ability of EXCHECK to run as a dynamic configuration of programs depends upon the multiprocessing facility of TENEX to coordinate many core images, called *forks*, as one program, without the need to program explicit overlays. The availability of forks is not special to IMSSS.

When a student logs in, if SEXEC chooses EXCHECK to run the course-program, then the EXCHECK course driver is started. The course driver, after checking the student's history, decides which forks to assemble for that student. Some forks, such as audio, must be included from the beginning of the student's session if they are to be used at all, while others, such as the theorem prover, can be added when they are actually needed, reducing the average size of the runtime system. Most forks in the EXCHECK configuration can be run either as independent programs or as subservient forks. Moreover, each fork can be written in a different programming language, allowing programs written at other sites, such as REDUCE (Hearn, 1973), to be readily incorporated for EXCHECK courses.

Another important use of forks is that programs too large for one core image (i.e., 256,000 words) can be split into parts and run in separate forks. This is in fact the primary reason that the course driver and proof checker reside in separate forks.

The fact that forks are largely autonomous is quite useful when using them to coordinate programs written in different languages, but this same feature can be a major disadvantage when using forks for the sole purpose of splitting a large program into manageable pieces. For example, since the course driver and proof checker do not have a common program control stack, EXCHECK does not attempt recursive calls from the proof checker back to the course driver. Thus features such as BROWSE (described below), which use the course driver, cannot be directly accessed while doing proofs in the proof checker. Instead, the student must first defer the given proof to return to the lesson level, then BROWSE, then return to the proof. Secondly, since the forks do not share complete data about the state of the display appearing on the terminal, they occasionally make mistakes when trying to restore that state, or when typing hints in a proof.

While both of these problems are merely inconveniences which can be remedied, they typify many problems inherent in using a small virtual address space.[3] The address space on a PDP-10 is 18 bits, or 256,000 words. A machine with a larger address space, for example 20 bits, or 1 million words, would enable both the driver and proof checker to reside in one core image, minimizing such problems. Having a larger address space, however, probably would not significantly enhance or detract from the ability to modularize the runtime configuration, particularly when interfacing modules written in alternate languages.

A more significant problem with the 18-bit address space is that one cannot arbitrarily divide a complex program into semi-autonomous pieces. Thus, an inordinate amount of effort was spent merely trying to make the

[3]The virtual address space of a machine is determined by the number of bits available to specify addresses in an instruction. It effectively limits the number of places (words or bytes) that one can directly access.

proof checker fit into one fork. Many features were truncated or abandoned simply because there was not room for them, which was often distressing. A larger address space would have freed that programming effort for more directly constructive tasks. It is encouraging to note that many newer machines have larger address spaces.

Much of the foregoing was intended to reflect some of the real problems encountered during the development of EXCHECK. Thus, it should be stressed that these problems, with the exception of the limited fork size, are currently perceived as correctable nuisances, and do not appear as significant problems for day to day usage of EXCHECK. In general, it seems clear that TENEX on a PDP-10 provided one of the best available operating systems in which the development of EXCHECK might have occurred.

3. VOCAL

The VOCAL compiler is designed to accept course material written in VOCAL, and to produce files suitable for interpretation by the EXCHECK driver and its subservient forks. Smith (1981) gives a description of the development of VOCAL.

A VOCAL course contains *lessons* consisting of *exercises,* which present audio-visual lectures, ask questions, comment about answers, assign proofs, check proofs, offer hints, ask for examples, check those examples, and perform other standard pedagogical operations.

The texts for lessons (and exercises) are created using TV-EDIT and then are processed by the VOCAL compiler, which produces a set of files in which each lesson is represented as a sequence of LISP-style S-expressions. A small, but somewhat typical expression as entered by an author might look like '(EXERCISE 10 (DERIVE "P → P"))'. VOCAL would process this to create the expression '(EXERCISE 10 (DERIVE (ARROW (P) (P))))', which is suitable input for the runtime EXCHECK program. For detailed descriptions of the structure of a VOCAL lesson, see the VOCAL manual (Hinckley, Laddaga, Prebus, Smith, & Ferris, 1977), or the VOCAL guide for lesson authors (Davis & Pettit, 1978). Weissman (1967) provides a description of the LISP programming language.

In general, each VOCAL expression has the general form '(OPCODE ARGUMENT . . . ARGUMENT)', where the OPCODE specifies a function to be performed, and the arguments provide information to that function about things to be done during its execution. These arguments can (recursively) be VOCAL expressions. VOCAL differs from LISP by providing only a fixed set of functions, all related to the presentation of curriculum, and by deliberately not providing the ability to define functions or declare arbitrary variables. See Smith (1981) for details, but note that VOCAL is not a general programming language.

Since the VOCAL compiler has runtime access to the text editor and to the course driver, the process of writing and testing a lesson is highly interactive. Once the curriculum author has written an exercise, it usually takes only minutes to compile it, test it in the driver, edit minor mistakes, and be ready to compile it again. For efficiency, the VOCAL compiler contains a highly truncated version of the course driver that can test major features of a lesson without bringing the course driver or the forks below it into core. The most important of these tests simply involves checking to see that the screen will be formatted properly.

4. EXCHECK COURSE DRIVER

The EXCHECK course driver interprets lessons prepared by the VOCAL compiler. It has three basic modes for curriculum presentation: LESSON, BROWSE, and HELP. The curriculum materials for all three modes are prepared by authors in VOCAL as described above. Because the curriculum material is interpreted as recursively as possible, EXCHECK admits powerful commands which allow students to BROWSE through other lessons, or to invoke special HELP routines on their own initiative, after which they return to their original lesson and exercise in the curriculum.

4.1 *LESSONs*

The LESSONs prepared for the main curriculum direct the general pattern of progress for each course. Each lesson roughly corresponds to a chapter in a book, and may be tailored to appear on a Teletype terminal, a display terminal, or a display terminal with an associated audio channel. If audio is generated, it may consist of prerecorded phrases, synthesized prosodic speech, or some combination. In the Armenian course, prosodic audio is used for English speech, while higher quality prerecorded phrases are used for Armenian speech.

Usually students proceed sequentially through every lesson in a course, but it is easy to set up special paths in which only selected lessons are given. This feature is occasionally used to provide students in other courses with a very brief overview of logic.

In the set theory course, there are currently three introductory and 18 normal lessons, all of which use prosodic audio. The basic logic course consists of 29 lessons, with additional lessons in Boolean algebra, probability theory, and social choice theory available for students who pursue grades of A or B. Each lesson in the basic logic sequence has both a display version and a version using prosodic audio. Later lessons have either a pure display version or a version using prosodic audio, but not both versions.

Where a choice exists among lesson modes, students can be automatically assigned some mode, or left to choose for themselves. Much of the data collected about the logic course pertains precisely to such choices. See Laddaga, Levine, and Suppes (1981) for details.

4.2 *BROWSE*

At most points in a course, a student can type 'BROWSE' to enter a command level from which any exercise in the course can be presented. A special flag is set at this level so that the interpreter presents the material in a lesson without requiring questions to be answered or problems to be solved. The students can also review and manipulate material they have created, such as stored versions of their proofs. The BROWSE facility is intended to simulate a student's ability to skim forward or backward while reading a textbook or notes.

4.3 *HELP*

Students who need particular advice or drill on some topic can type 'HELP' to enter the HELP system. Special lessons are available to address particular points not covered by the general curriculum, such as use of the computer system, hours for teaching assistants, and descriptions of the inference rules available for doing a proof. The HELP system is graph oriented, so that students can be guided toward the appropriate HELP lesson provided they have some idea of what they want. In the example below, a student wanted the computer to speak faster, and was able to find the appropriate information in a fairly direct manner. The 'escape' key, represented here as '$', does not show in actual use.

```
*HELP$

   **** HELP MODE COMMAND LEVEL ****  (Type OK to exit)

type                        for help on

ADMIN                 Administrative matters
SYSTEM                Use of the computer system
LANGUAGE              The language of Logic and Set Theory
EXCHECK               Use of the proof checker
ALPHA-LIST            An index to an alphabetical listing of all HELP
                        topics (with descriptions) is displayed

*SY$stem
```

[This will cause a new screen with the next frame to appear.]

```
   **** HELP MODE COMMAND LEVEL ****  (Type OK to exit)

type                        for help on

AUDIO                 Problems with the audio system
BROWSE                Using Browse Mode (^B)
CTRL                  Control commands
DERIV-FORMAT          How to change derive format
```

```
DISPLAY              How to use display features
GRIPE                How send a complaint or suggestion
HELP-HAS             What the Help System contains
HELP-USE             How to use the Help System
EXERCISES            The various types of exercises in the course
KEYS                 Functions of the special (black) keys
NEWS                 How to ask for news on the course
ADMIN-CMD-LIST       Some administrative commands

*AU$dio
```

[Again, this causes a new screen to appear.]

```
   **** HELP MODE COMMAND LEVEL ****  (Type OK to exit)

type                          for help on

SPEED                Controlling the speech rate
GRIPE                How to send a complaint or suggestion

*SPE$ed
```

[Now a brief lesson is presented showing how to use ↑S to change the rate of speech. When finished, it returns to this HELP level.]

```
   **** HELP MODE COMMAND LEVEL ****  (Type OK to exit)

type                          for help on

ADMIN                Administrative matters
SYSTEM               Use of the computer system
LANGUAGE             The language of Logic and Set Theory
EXCHECK              Use of the proof checker
ALPHA-LIST           An index to an alphabetical listing of all HELP
                       topics (with descriptions) is displayed

*OK$
```

[The screen is then restored and the student is returned to the exact prompt at which HELP was originally typed.]

In the next example, the student enters HELP and then goes directly to a HELP lesson on the ESTABLISH rule. Thus we can see that the graph is a guide, not a restriction.

```
*HELP$

   **** HELP MODE COMMAND LEVEL ****  (Type OK to exit)

type                          for help on

ADMIN                Administrative matters
SYSTEM               Use of the computer system
LANGUAGE             The language of Logic and Set Theory
EXCHECK              Use of the proof checker
ALPHA-LIST           An index to an alphabetical listing of all HELP
                       topics (with descriptions) is displayed

*EST$ablish
```

[The student enters a HELP lesson which explains how to use the ESTABLISH rule of the proof checker.]

Within HELP, lessons are presented as in the main curriculum, except that at any point the student may simply leave the HELP lesson. This allows curriculum authors to provide remedial lessons in such a way that students who do not need them never see them. The easy exit also prevents students from being trapped in a HELP lesson they no longer wish to pursue.

The set theory course makes extensive use of HELP lessons, since students come in with a wide range of backgrounds, dividing particularly into those who have and have not taken the logic course.

The logic course makes essentially no use of the HELP facility, primarily since its curriculum was complete by the time the HELP facility was developed. The logic curriculum, moreover, is meticulously structured to explicitly introduce students to every feature and concept they will encounter, and provides numerous review exercises which summarize important conventions. The ability to BROWSE through these review exercises has thus largely obviated the need for HELP lessons.

4.4 *Additional Driver Functions*

In addition to the three major modes of curriculum presentation, EXCHECK has some ancillary features which provide useful functions. Students can read news intended for the entire class, or personal messages intended for them alone. They can send *gripes* (messages) to the course authors, teaching assistants, or programmers. All news, from the first day of class, can be accessed in reverse chronological order. Recent personal messages are removed, but only when the student next logs out normally. Gripes range from real complaints about the course or program, to notifications of spelling or stylistic errors, to messages for particular staff members. Students are also encouraged to use it at the end of the course to indicate the grade they have earned (since a mastery approach is used, they know their grade). This provides a useful cross-check with the computer's records.

4.5 *Recursive Interpreter*

The organization of EXCHECK to allow BROWSing and HELPing has advantages over tutorial texts and other sequential presentations of material, since students can decide when, where, how much, and what sort of help they need, without losing their position in the main curriculum. When students return from NEWS, GRIPE, BROWSE or HELP, their screen is usually refreshed to display the material that was there just before they selected one of the other modes, and they are able to respond to whatever prompt awaited them as if they had never left. The ability to return to the exact place which one left earlier is a general feature of recursive programs, and EXCHECK was designed as a recursive interpreter largely to facilitate this.

5. VINPUT

Because a student might be answering a question in one curriculum mode while a different question in some other mode is still pending, the routines which accept input need to be recursive to avoid losing their context. HELP, BROWSE, and related commands are designed to be available as generally as possible, so ideally every input routine in EXCHECK should be able to recognize and execute these commands. For these reasons and others discussed below, all the input in EXCHECK is obtained using one general routine called VINPUT.

One immediate advantage of using VINPUT is that student interactions now proceed in a uniform manner. This simplifies the documentation of features, and minimizes the knowledge students must have of EXCHECK. Features have been introduced to reduce the amount of typing required, to provide assistance in determining the options available, and to offer a means through which the student can access HELP, BROWSE, and other special features. Additionally, prompts, comments, and error messages are stored separately from the program's code, so it is easy to update them and to provide alternative styles such as 'terse' and 'verbose'.

5.1 *Input Recognition*

In every place where the permissible options can be listed, recognition is used to minimize the number of characters a student must type. A student can type just the first few characters of an option followed by the 'escape' key and VINPUT will attempt to recognize the option, extending the student's input as it does so. Thus, if very-long-option-a and very-long-option-b were the only alternatives, one possible student/EXCHECK interaction would be:

```
*V$ery-long-option-B$
```

The student's input is underlined, and '$' indicates the 'escape' key, which does not print in actual practice.

5.2 *Question Mark Response*

Coupled with the recognition feature is a question mark response facility that will list all possible options given the current input. Thus the example above could have proceeded as:

```
*V$ery-long-option-?
    very-long-option-A
    very-long-option-B
*very-long-option-B$
```

Of course, instead of continuing with B, the student could have deleted characters and tried again if some third command were intended.

The question mark facility works more elaborately if a question mark is typed without preceding characters. The assumption here is that the student does not know exactly what is available or what is expected. A new prompting level (indicated to the student by '?? ') is entered, and commands are treated as they would be at the main prompt, except that instead of generating actions, they generate descriptions of those actions. The motivating concept is that a command entered when the prompt is '?? ' will provide a quick description of what that command would do at the original place where question mark was typed. For example, 'NEWS' typed at any normal prompt will let the student read the news, but at the ?? prompt, it will generate:

```
?? NE$ws is always available and allows you to read the news.
```

Typing the 'escape' key at this prompt will give a list of all the options legal when the original question mark was typed. For example, at the top level of the proof checker, all the proof commands appear, followed by the general commands. The ?? prompt repeats until the student types 'ok' to leave it.

The question mark facility was added relatively late in the development of EXCHECK, and the online curricula do not really mention it. Moreover, data have not been collected on the use of this feature, so it is difficult to assess its utility for students.

5.3 *Control Commands*

VINPUT always recognizes certain control commands, such as HELP, BROWSE, NEWS, MESSAGE, and GRIPE. The appropriate routine is invoked recursively, if possible, after which the student's screen and position in the course are restored. VINPUT also recognizes certain control characters, 'control-A' (↑A), 'control-B' (↑B), etc. which allow the student to logout, abort the current operation, enter browse, refresh the screen, repeat the last action, etc.

Speech rate. Control-S permits the student to modify the rate at which the synthesized speech is spoken. Originally, the speech rate is set to be about 10% faster than the rate at which words were recorded. The student can vary this rate anywhere between a rate twice that of the recording and one 20% slower than the recording.

Line editor. Control-V invokes a line editor that allows a student to insert, delete, and append characters. If it is typed immediately following some input, that input is edited. Otherwise, each ↑V retrieves the next older input appropriate for the current prompt. The record of student input which ↑V accesses can also be used to reconstruct almost exactly what the student typed, a valuable aid in diagnosing any problems that arise.

Guess. Control-G can be used at many prompts to ask the computer to guess. This is most meaningful in exercises where some goal or subgoal is currently pending (see Blaine, 1981), so that some context exists for making a guess. Its use in such contexts can drastically reduce the number of characters a student must type, thus reducing typing errors as a source of distraction.

Hint. Control-H at any prompt will check to see if any applicable hints are available. Currently these hints are mainly entered by the lesson author, but dynamically generated hints are possible. Hints are offered in succession and many hints may be available for a given problem.

Uniform exit routine. Control-L was intended to allow the student a graceful exit from any situation that might arise in EXCHECK. It should be an immediate interrupt, to abort any other EXCHECK processing. Instead, for now only VINPUT and the theorem prover check to see if it has been typed. Occasionally, a student may present EXCHECK with an enormous inference not handled by the theorem prover. For example, problems given to BOOLE can grow exponentially in the time required to answer them, as new sentences are added for consideration. Students may also simply stumble onto a programming error which leads to infinite looping. Currently, the only remedy in such cases is to have a staff member log the student out remotely, after which the student can log back in. Unfortunately, work can be lost in this process, and if a staff member is not available, the student is helpless. This is the most significant problem left in the user interface. ↑ L does work well in those cases where students have begun to enter a command, only to change their mind halfway. Since VINPUT responds to ↑ L, it is recognized at any input point.

Logging out. Control-Z is designed to finish the current operation, record the student's status in the course, and terminate the session. The saved status will be used when the student next logs in to restore the context that existed prior to logging out. Files which keep track of a student's status are periodically updated while that student is working, to minimize the amount of work that the student must repeat if the computer crashes while the student is logged in. Even if a student's file is accidentally lost or garbled, it is possible for a teaching assistant to adjust that student to the proper course location.

6. PARSERS AND GRAMMARS

The language required for formal discourse in logic, and more dramatically, those needed for informal discourse about set theory or proof theory, require input grammars of significant complexity. Output grammars of comparable complexity are also needed to provide flexible presentations of the mathematical objects generated by EXCHECK.

6.1 *Parsers*

To parse complicated logical or set theoretical expressions from the student, the parser for that course is invoked as a separate fork. Our parsers for the mathematical courses currently use the Cocke-Younger algorithm (Aho & Ullman, 1972).[4]

Simple formulas parsed for various courses include:

Set Theory
$(\forall x)(x \in A \rightarrow x \in B)$
For all x, if x is in A then x is in B
Proof Theory:
$ZF^* \vdash (\forall x)x = x$
ZF^* proves that for all x, $x = x$

The mathematical parsers are context free for a given theory. However, EXCHECK provides the ability to switch theories within a course and thus alter the interpretation of expressions.

Beyond meeting the technical requirements for courses such as proof theory, the parsers provide reasonable flexibility in the style of input. Thus, "$(\forall x)(x \in B)$" and "for all x, x is in B" will both be parsed as the same formula. However, "everything is in B", which is also a good paraphrase, cannot currently be recognized because it violates certain restrictions inherent in context-free languages.

A conjecture this author endorses is that people conversing in a system with mathematically clean rules for dialog will adapt very quickly to it, even if they are not explicitly aware of the rules. Context-free languages seem to form a natural plateau at which people can converse without making mistakes, and without feeling unduly restricted.

6.2 *Output Grammars*

We have put a significant effort into presenting logical and mathematical material in a clear and flexible manner. As one small aspect of this, our terminals were modified to include an extended character set using Greek letters, $\forall$s, and other mathematical symbols. At a more fundamental level, we have designed grammars that provide for a number of modes of output. These output grammars reside in the proof checking fork since most formulas are generated there.

A student can specify several dimensions of output style. For example, there is a choice between formal and informal output. If a student wishes

[4]A system for writing and testing grammars using the Early algorithm (also in Aho & Ullman, 1972) was largely completed, but never actually incorporated into EXCHECK proper.

to print the definition of subset, EXCHECK will normally print the definition in a formal style:

$$A \subseteq B \leftrightarrow (\forall x)(x \in A \rightarrow x \in B)$$

The student can also request an informal style, in which case EXCHECK will print:

A is a subset of *B*
if and only if
for all *x*, if *x* is in *A* then *x* is in *B*

Students are free to alternate between formal and informal styles. They can also decide whether EXCHECK should print top-level universal quantifiers. In the above examples, EXCHECK has not printed the top level universal quantification of *A* and *B*.

For complicated terms or formulas, the student can specify parametric abbreviations to simplify the form of the expressions printed. The specified abbreviations are also recognized by the parsers, reducing the typing needed to input new expressions. For example, "x hits y" could be used to abbreviate "$x \in A$ and $y \in B$ and $F(x) = y$" in both input and output.

7. PROOF CHECKING FORK

A primary student activity in the EXCHECK courses is the presentation, in dialog with the instructional program, of mathematical reasoning (proofs) to be checked for correctness. Blaine (1981) gives an extensive description of this activity. In fact, the most interesting aspects of the EXCHECK system are the procedures and the underlying theories of mathematical reasoning that permit this interaction to take place at a fairly natural semantic level in a style approximating standard mathematical practice. These aspects of EXCHECK include natural language facilities, natural deduction based proof procedures, theorem provers, decision procedures for some simple mathematical theories, procedures for analyzing and summarizing proofs, and procedures for conducting dialogs about some elementary mathematical structures. See Blaine (1981) for a detailed analysis of these features, and for sample proofs.

7.1 *Proofs*

In the set theory course, student proofs are entered in a top down fashion, in the sense that proofs are recursively broken down into smaller proofs. This is in keeping with standard mathematical practice and reduces the degree to which students are likely to generate correct but pointless lines. It also vastly enhances the computer's ability to analyze the student's progress so far and to provide direction for further progress. The effective use of $\uparrow$ G (guess) within a proof depends strongly upon this principle. In the logic course, proofs are still entered in a linear fashion, primarily because the curriculum was complete before the goaling mechanisms were implemented.

Another feature of standard mathematical practice is that proofs are usually only sketches of the main line of argument. Distracting detail is omitted. The proof procedures in EXCHECK match natural procedures well enough that students need not enter significantly more lines than those which would occur in a natural proof. Again, see Blaine (1981) for a more detailed discussion of this matter, and Suppes and Sheehan (1981a, 1981b) for quantitative data on the lengths of proofs.

Since typing formulas is another form of distracting detail, EXCHECK has been designed to minimize the typing involved in entering formulas. Many proof procedures automatically generate the formula to be inferred. The students can access a substantial list of axioms, definitions, lemmas, and theorems by name using VINPUT recognition. Expression designators, such as FM3:2 and TM3:2:1, have been provided to allow a student to access terms or formulas from previous lines. For example, FM3:2 designates the second subformula of line 3 and TM3:2:1 designates the first term of that subformula. A line editor is available for modifying previously typed formulas, commands, etc. Effective use of the proof commands, theorem lists, expression designators, and line editor can reduce a student's typing by 75% or more.

In the course of a term, logic students will typically do about 150 nontrivial proofs. Set theory students are likely to finish about 50 substantial proofs. These figures will vary depending upon the grades desired, and the extent to which students perform extracurricular proofs for their own pleasure or curiosity. It can safely be said that those who finish a course have almost always improved their ability to give valid proofs. Students will gripe about bugs in the program, the time required to complete the course (logic is a five-unit class), even the condition of the terminal rooms, but I think we have never received a gripe to the effect that a student has not learned how to do proofs.

Of course, in set theory, the emphasis should properly be on the mastery of mathematical concepts, and not merely on the ability to prove various theorems, however adeptly. Students in that course, particularly those who are exceptionally well prepared, have often felt that they were spending relatively too much time learning how to do proofs in EXCHECK, and too little time learning the concepts of set theory.

Much of this frustration probably arises from simple lack of power in the EXCHECK proof checker. In particular, simple truths about functions can be extremely obvious to the student, and yet require many minutes to prove. Additional decision procedures (analogous to BOOLE, TAUTOLOGY, and

TEQ) which could quickly confirm such propositions would perhaps alleviate most of that frustration.

7.2 *Construction Exercises*

A different class of exercises require the student to specify some small finite model that satisfies a given set of conditions. A typical exercise might proceed as follows. As before, student input is underlined, and '$' represents the 'escape' key, which does not show in practice.

Construction Exercise:

```
    A = { 1, 2, 3 }     B = { 4, 5, 6, 7 }

Give an injection F from A into B.

F = { <1,4>, <2,5>, <3,6> <4,7> }$

Wrong, because the domain of F is not a subset of A.
* Why$ Because 4 is in the domain of F and 4 is not in A.
* So$ the domain of F is not a subset of A
* So$ the domain of F is not equal to A
* So$ F is not an injection from A into B
* ok$
```

Construction Exercise:

```
    A = { 2, 4, 6 }     B = { 4, 5, 6, 7 }

Give an injection F from A into B.

F = ↑V
F = { <1,4>, <2,5>, <3,6> <4,7> }
F = { <2,4>, <4,5>, <6,6> _____ }$

Very good.
```

The computer is asking the student to enter a set, to be called *F*, which will satisfy certain conditions involving the given sets *A* and *B*. Since the student's first answer is wrong, the program analyzes why the answer was incorrect and stores this information in the form of an internal graph. It is then ready to enter a simple dialog with the student which is based upon the chain of reasoning used to find the student's mistake. After the student understands the mistake, a very slightly modified exercise is presented, and the student uses the line editor to quickly modify the old answer to get a correct answer for the new exercise. The computer then acknowledges the correct answer.

The construction exercises were designed late in the development of EXCHECK and have not been incorporated into the curricula for any course. Nonetheless, the ability of EXCHECK to check this type of exercise has interesting potential. In particular, the ability to carry on graph oriented dialogs, while primitive here, seems to be an avenue strongly worth pursuing. Given the highly structured environments to be found in mathematics courses, it would seem to be quite natural to build graphs relating various notions such as rules, proofs, counterexamples, models, etc. Designing a program that could make effective use of such graphs, leading students from one related notion to another, could be a very productive project.

7.3 *Ease of Exercise Creation*

An important point to notice about EXCHECK (perhaps the most important) is that both construction and derivation exercises are extremely easy to create. The lesson author need only state the problem, leaving all interaction with the student under control of the proof-checking fork. The construction exercise above was generated from a VOCAL expression very similar to the following question and answer expression:

```
(Q INIT ((PROVIDE (A "[1,2,3]") (B "[4,5,6,7]"))
           (REQUEST "Give an injection F from A into B." (F)))
        A (ASSESS "F:A inj B"))
```

A fairly typical proof could be assigned by merely entering the VOCAL expression '(DERIVE "If $A < B$ and $B < C$ then $A < C$")', although in practice one would probably restrict the theorems available, and possibly provide hints.

The brevity of these expressions, contrasted with the range and variability of the dialogs they generate, provides a clear example of the technological leverage which is possible in CAI systems.

7.4 *Play*

In fact, EXCHECK does enough of the work involved in creating and evaluating derivation exercises that students can simply type 'PLAY' to enter a mode in which they themselves specify exercises. In the set theory course, they may choose an arbitrary formula to prove, or ask to prove one of the predefined theorems. When proving arbitrary formulas, they are allowed to use any theorem in the course, but when proving theorems chosen by number or name, they are restricted to using axioms, definitions, and theorems that precede the theorem to be proven. Many of the proofs which students do for credit are chosen through PLAY, providing flexibility in the assignments. The PLAY command is also useful when students are learning EXCHECK, and encourages curious students to explore. Construction exercises are not yet available within PLAY, partially for technical reasons, but primarily for lack of time.

8. RELATION TO CURRICULUM

It has now been mentioned more than once that certain features have not been made available to students because they were implemented after curriculum development was complete. This typifies a significant general problem for CAI courses—while it may be relatively easy to create new features, it can often prove quite difficult to introduce such features in a pedagogically sound manner.

For example, there is little doubt that the goaling mechanism is an important and useful feature of the set theory course. In the logic course, however, one can at least argue the point that providing such a powerful feature might detract from the students' participation in the organization of proofs. Even if we assume that goaling is a good feature for this course, there remains the formidable problem of rewriting hundred of pages of VOCAL lessons that make explicit assumptions about the structure of proofs. By paying meticulous attention to the detailed operation of the proof checker, lesson authors for the logic course have made it extremely easy for students to understand and use that machinery. Concomitantly, however, they have made it quite difficult to modify the proof checker without meticulous attention to the curriculum.[5]

This problem will undoubtably be a major one facing continued development in any system like EXCHECK. Given the largely unstructured nature of the text appearing in typical curricula, a detailed analysis of this problem would go far beyond the scope of this article. One partial solution, however, would be to mechanize the introduction of course features by implementing a *player piano* type of exercise. The main idea is that the program would reproduce sessions done by expert users, possibly generating questions along the way to see that students could predict what was legal and useful to do. New features would then be documented primarily by having experts run through enough such sessions to illuminate all the options. All such old sessions could be mechanically reexecuted to insure that they still worked. Fundamental pedagogical issues might remain, but the technology for implementing chosen features would be enhanced.

9. IMPLEMENTATION ISSUES

In implementing any program, the moment arrives when one must commit to specific languages, data structures, and algorithms, all too often before the experience is available to know what the best choices will

[5]It was distressing at one point to fix a deficiency in one of the commands, only to discover much later that one exercise described that deficiency in great detail, and explicitly advised students not to use that command in the manner in question!

be. In fact, there is a widely circulated bit of folk wisdom that to do a program right, one must first write it and make it completely operational, then start over from scratch and do it again. Of course, such luxury is rarely possible, and possibly never budgeted.

The design of EXCHECK has been in large measure quite good. The design of VOCAL, as described in Smith (1981), was well conceived, and the largely modular structure of the programs has facilitated development. In fact, problems have arisen not so much from flaws in the conceptual design of EXCHECK, as from lack of provision for the dynamics that occur when many people work on a large program. Thus, the final program works well and works reliably, but the path to that success has been more troubled than necessary. The primary problem stems from the use of a noninterpretive language for the general implementation. A second set of problems arose concerning the actual display of characters on the screen, and finally a myriad of minor problems have arisen from lack of programming conventions, or lack of adherence to them.

9.1 *SAIL*

While VOCAL has a LISPish syntax, the actual language used to implement the course driver and proof checker is SAIL (Stanford Artificial Intelligence Language), a dialect of ALGOL with numerous supported data types—in particular, convenient and efficient string routines (Reiser, 1976).

When EXCHECK was first being written, it was envisioned that string manipulations would be a predominant activity. SAIL and various LISP dialects were the most promising candidates for the implementation, and since LISP dialects in general were poor with strings, SAIL was chosen.

An *S*-expression package was added to accommodate the LISP syntax of VOCAL, and to facilitate logical operations on proofs, and other abstract structures. This package contains such LISP routines as CONS, APPEND, READ, and even MAPCAR, which basically manipulate *S*-expressions, but it does not perform operations such as PROG and LAMBDA binding, or ERRORSET protection, which pertain to the control of LISP programs.

The normal SAIL routines have proven quite nice for most system interfaces and string manipulations, while the *S*-expression package facilitates logical operations on proofs and other complex data structures.

This system does work (EXCHECK currently runs in it!), but in retrospect, it seems that some dialect of LISP would have been a far better choice than SAIL. The primary reason for this observation is that LISP has an interpreter, and supports the compilation and loading of individual functions. The time saved in the program debugging loop between function editing and testing would have been more than ample to put an efficient string package into whatever LISP dialect was chosen.

A second major disadvantage of SAIL compared to LISP is that it uses lexical, not dynamic, scoping of variables. As a result, the recursiveness of the course driver is not ideal, and it can become quite complicated to insure that the values of all appropriate variables are properly saved and restored when shifting to a different context.

A final unfortunate disadvantage of the current system is that the *S*-expression package resides in SAIL but is not known to the compiler. Programmers must therefore explicitly protect *S*-expression variables from the garbage collector, which would otherwise cannibalize their contents for use in other structures. Good programmers have made mistakes in this system, and confused or inexperienced programmers have wreaked havoc. Mysterious garbage collection problems accounted for perhaps 90% of all debugging effort during a two-year period. The SAIL compiler could perhaps have been augmented to be cognizant of the *S*-expression package, but the time for that almost certainly would have exceeded the time spent adding good string facilities to some LISP dialect.

As something of a postscript, a LISP version of the part of EXCHECK adequate to present the basic logic course is nearing completion.[6] Preliminary results with about 10 students seem to verify the intuitions stated above. The only major disadvantage to using LISP seems to be an increased use of memory and CPU time, but substantial optimizations are envisioned which may eliminate that disadvantage.

9.2 *Display Features*

The second problematic area in the implementation revolves primarily around screen management. The Datamedias we use have a 24 by 80 character screen, which limits the amount of text that can be shown at any one time. The template mechanism in VOCAL goes a long way towards clarifying screen management for lesson authors, but does not resolve detailed issues regarding the format of proofs, menus, and other internally generated structures. The ability to recreate the exact screen image or replay recent material also must be embodied by some definite implementation.

The current display code in the course driver and proof checker is adequate to the task, but contains many ad hoc routines written to address specific problems. The situation has improved considerably in the transportable LISP version being developed, but a truly elegant solution still eludes us. Languages such as Small Talk (Goldberg & Kay, 1976) and LISP Machine Lisp (Weinreb & Moon, 1979), which provide windows and screens as basic data types, seem to have made an important step in the right

[6]Professor Tryg Ager, of the State University of New York at Binghamton, has been working to develop a transportable version of the logic course.

direction. Given the time spent designing and implementing screen management for any substantial CAI course, this would seem to be an area worthy of further study. In fact, leaving graphic display aside, it would be most gratifying to see a general purpose LISP package evolve that has an intuitive and simple syntax, yet is suitable for presenting fairly arbitrary text patterns on a wide variety of terminals.

9.3 *Programming Conventions*

One cannot work on a large system for many years without being struck by the need for good programming conventions. To condense some of our experience, better adherence to the following rules (among others) would have significantly reduced the time spent debugging and modifying EXCHECK:

1. Use macros or defined functions to access data structures. Far too often, people explicitly took the first or third element of some structure on the assumption that they knew how that structure was organized, and that the structure would not change. The result now is that such structures dare not change.
2. Use descriptive function and variable names. Routines called F1 or F2 which used SE1 through SE12 as variable names rapidly became opaque. In fact, sometimes the only way to safely modify these routines was to replace all such names with mnemonic ones, tediously reconstructing the original programmer's intentions.
3. Copiously document side effects. The generation of delicate side effects by various routines seems to be a pervasive feature of large systems. Lack of proper documentation about such effects has caused tremendously obscure errors to occur when such a routine was altered in what seemed to be an innocuous manner. I/O routines seem particularly prone to this problem, since the state of the screen is a fairly hidden effect when looking at code.
4. Avoid delicate features, unless they are well understood. As mentioned before, the *S*-expression package was extremely convenient when used properly, but created code analogous to a minefield when used improperly.

While the direct cost of ignoring such conventions was significant and fairly obvious to appreciate, it is important to note that the worst effect of such problems may have been due, not to their direct costs, but to the sometimes debilitating effect they had on programmers' normal efforts. For example, when undocumented side effects were feared, it could take quite some time to verify that none of the hundreds of pages of code in other files were likely to be affected by a small intended change in one routine. Likewise, when a routine needed minor maintenance, but appeared opaque, it often seemed easier to completely rewrite it than to decipher the old code, an approach that sometimes introduced new classes of errors.

In short, our experience with EXCHECK, both positive and negative, has shown that adherence to good programming conventions is perhaps not critical to the implementation of an initial program, but quickly becomes a dominant consideration in programs intended for long use and evolutionary improvement. This is a warning readers should ignore at their peril.

10. SUMMARY

The EXCHECK system has proven moderately flexible and quite easy to use. Both audio and visual modes are available for lesson presentation, and copious data have been collected and analyzed on the effect of such redundancy.

For the students, powerful CAI modules perform detailed answer analysis, and lessons are interpreted recursively, giving them significant freedom in choosing what to do next. Proof procedures make it possible to do proofs in a rather natural fashion, while input parsers and output grammars provide multiple styles for representing and simplifying formulas.

For course authors, VOCAL allows lessons to be written and tested in interactive sessions, using sophisticated editors and debugging facilities. Moreover, forks are dynamically assembled at runtime, allowing each course designer important freedom to add novel features as they become available, even if no specific provision was originally made for them.

Problems as noted throughout still remain. EXCHECK was a prototype system, and thus bound to encounter mistakes in design, development, and documentation. However, the demands of providing ambitious students with courses worthy of university credit have clarified and limited such mistakes in an extremely sharp way.

The author would in no sense claim that EXCHECK is the ultimate pedagogical vehicle. Nonetheless, it does provide a good vehicle given the current state of the art, and the experience gained from EXCHECK should prove most valuable in the design of newer systems.

REFERENCES

Aho, A. V., & Ullman, J. D. *The theory of parsing, translating, and compiling* (Vol. 1). Engelwood Cliffs, N. J.: Prentice-Hall, 1972.

Blaine, L. Programs for structured proofs. In P. Suppes (Ed.), *University-level computer-assisted instruction at Stanford: 1968–1980*. Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1981.

Davis, M., & Pettit, T. *Using VOCAL : A guide for authors* (Tech. Rep. 296). Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1978.

Goldberg, A., & Kay, A. (Eds.) *Smalltalk-72 instruction manual* (Tech. Rep. SSL 76-6). Palo Alto, Calif.: Xerox Palo Alto Research Center, 1976.

Hearn, A. C. *REDUCE 2 user's manual* (2nd Ed.). Salt Lake City: University of Utah, 1973.

Hinckley, M., Laddaga, R., Prebus, J., Smith, R. L., & Ferris, D. *VOCAL: Voice Oriented Curriculum Author Language* (Tech. Rep. 291). Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1977.

Kanerva, P. *Tvedit manual.* Unpublished manuscript, Stanford University, Institute for Mathematical Studies in the Social Sciences, 1973.

Laddaga, R., Levine, A., & Suppes, P. Studies of student preference for computer-assisted instruction with audio. In P. Suppes (Ed.), *University-level computer-assisted instruction at Stanford: 1968–1980.* Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1981.

M.I.T. Laboratory for Computer Science Mathlab Group *MACSYMA reference manual* (Version 9, 2nd printing). Cambridge, Mass.: Massachusetts Institute for Technology, 1977.

Reiser, J. F. *SAIL* (Stanford Artificial Intelligence Laboratory Memo AIM 289). Stanford, Calif.: Stanford University, Stanford Artificial Intelligence Laboratory, 1976.

Sanders, W. R., & Levine, A. The MISS speech synthesis system. In P. Suppes (Ed.), *University-level computer-assisted instruction at Stanford: 1968–1980.* Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1981.

Smith, R. L. VOCAL: A case study in the methodology of the design of authoring languages for computer-assisted instruction. In P. Suppes (Ed.), *University-level computer-assisted instruction at Stanford: 1968–1980.* Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1981.

Suppes, P., & Sheehan, J. CAI course in axiomatic set theory. In P. Suppes (Ed.), *University-level computer-assisted instruction at Stanford: 1968–1980.* Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1981. (a)

Suppes, P., & Sheehan, J. CAI course in logic. In P. Suppes (Ed.), *University-level computer-assisted instruction at Stanford: 1968–1980.* Stanford, Calif.: Stanford, University., Institute for Mathematical Studies in the Social Sciences, 1981. (b)

Weinreb, D., & Moon, D. *LISP machine manual.* Cambridge, Mass.: Massachusetts Institute for Technology, 1979.

Weissman, C., *LISP 1.5 primer.* Belmont, Calif.: Dickenson, 1967.

APPENDIX A

Structure and Size of EXCHECK

This appendix describes the size of the EXCHECK program on a module by module basis, and offers some translation into IBM terminology.

EXCHECK is an extremely large program. Including the compiler, there are seven forks in the system, where each fork can be thought of as a nearly autonomous program that has an entire virtual address space available to it. Since this virtual address space consists of 256,000 36-bits words, the system has available a combined address space of 1.8 million PDP-10 words, or about 8 million 8-bit (IBM) bytes. Only three of these forks are nearly full of code, and large segments of code are shared among many of the forks. Taking these factors into consideration, the program can be expressed as roughly 600 K (i.e., 600,000) words of code and another 300 K words of work space, for a total of about 900 K words, or 4.4 megabytes.

The following table gives a rather detailed analysis of two important forks, and then a grand summary for all forks. The detailed analysis performs rough divisions into code (sharable or reentrant memory) and variables (private or non-reentrant memory). Numbers are given in multiples of K (i.e., 1028) 36-bit words.

Size of Components in Course Driver

	Space used (in K words)	
Component	Code	Variables
VOCAL interpreter	12	0
Logical routines	3	0
Audio	7	1
Help package	2	3
High level input	8	1
High level output	2	1
I/O utilities	4	1
Fork communication	5	10
Basic SAIL routines	16	1
S-expression package	5	4
Global data structures	1	5
Workspace	0	26
I/O buffers	0	8
Misc (debugging, etc.)	15	14
Total size of course driver	80 K	75 K

Size of Components in Proof Checker

	Space used (in K words)	
Component	Code	Variables
Initialization	4	0
VOCAL interpreter	0	3
Proof checker	19	2
Logical utilities	15	0
Audio	6	1
High level input	11	1
High level output	15	1
I/O utilities	4	1
Fork communication	5	10
Basic SAIL routines	16	1
S-expression package	5	4
Global data structures	3	4
Workspace	0	28
I/O buffers	0	8
Theorem names	0	5
Misc (debugging, etc.)	25	1
Total size of proof checker	131 K	67 K

Sizes of Forks (sub-programs) in EXCHECK

Fork	PDP-10 words (36-bit words)	IBM equivalent (8-bit bytes)
Vocal compiler	153 K	690 K
Course driver	155 K	700 K
Proof checker	198 K	890 K
Theorem prover	211 K	950 K
Audio	132 K	600 K
Parser	56 K	250 K
Reduce	63 K	280 K
Total	968 K	4,360 K

The equivalents given are conservative, since they assume that the same number of bits will suffice for the same task. In actual practice, equivalent code seems to require about 33% more bits on an IBM-370.

On the other hand, about 130 K is redundant among the forks, so the size in one address space could be reduced to about 830 K, or 3.7 megabytes. EXCHECK without the VOCAL compiler could be further squeezed into about 700 K, or 3.2 megabytes. That portion of EXCHECK adequate to run the logic course can probably fit in about 240 K, or one megabyte.

Mere numerical comparisons fail to emphasize the ease with which EXCHECK's fork configuration can add new forks with new capabilities, even if they come from unanticipated sources and are written in different languages. For example, we occasionally run other experimental forks not described in this article, and could probably incorporate programs such as MACSYMA (MIT, 1977) quite readily under EXCHECK.