TEXreation — Playing Games with TEX's Mind

ANDREW MARC GREENE

Project Athena
Room E40-342
and
Student Information Processing Board
Room W20-557
Massachusetts Institute of Technology
Cambridge, MA 02139
amgreene@athena.mit.edu
Please address all correspondence to the Project Athena address.

ABSTRACT

TEX can be used to write applications that have little or no connection with document preparation. Games such as ANIMALS and BATTLESHIP are just a few of the recreational uses to which TEX can be put — mostly to show that it *can* be done, but also to provide an entertaining medium for both experimentation and presentation of programming techniques that can be used in more serious macro packages. Database management is exemplified by the program ANIMALS, and array handling is developed in BATTLESHIP.

1. Introduction

At some point, most TEX users find it necessary to extend the language of TEX to perform some task. Whether the task is as simple as defining a macro to alleviate repeated typing of some lengthy string, or as complicated as rewriting output routines, we sit at our keyboards alternately cursing and blessing Donald Knuth.

Few, if any, of us ever write a program in TEX that has nothing to do with typography. TEX is slower than C, more obtuse than assembler, and harder to trace than BASIC. Nevertheless, writing programs in TEX is possible and will occasionally yield results that are useful in "real" TEX programs (or macro packages).

This paper will present two such programs, both of which are games. ANIMALS, a simple "artificial intelligence" program, resulted in a set of TEX database management routines. BATTLESHIP, the classic game of naval battle on a grid, was a perfect candidate for implementation of array handling and indexed variables in TEX.

2. The Game of ANIMALS

ANIMALS was written in response to a dare from a friend at the Student Information Processing Board at MIT. It is a simple expert system, in which the computer asks questions and tries to guess which animal the user has selected based on the user's responses. An annotated listing of ANIMALS appears in Appendix A.

2.1 Rules

The user thinks of an animal which the computer will attempt to guess. On each round, the computer asks a yes/no question, which the user must answer truthfully. Eventually the computer will take its guess; if it is correct then the program ends, otherwise the computer will amend its database to include the new animal and a question distinguishing the new animal from the original guess.

These rules imply that the database should be arranged in a binary tree, such as the sample in Figure 1. Since TEX doesn't have random access files, this would be difficult to implement. However,

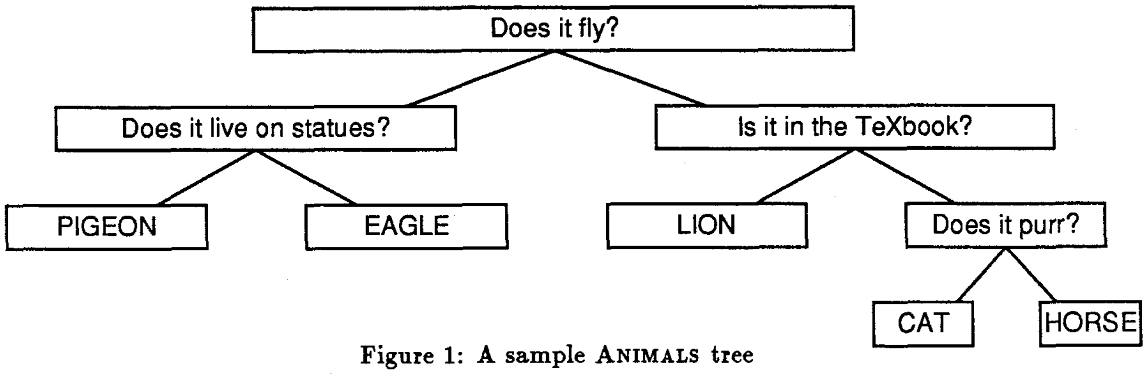


Figure 1: A sample ANIMALS tree

it should be noted that progress is always towards the bottom of the tree; therefore it is possible to simulate a random-access database whose records will always be read in a certain order using TEX's sequential file operators `\openin`, `\read`, and `\closein`.

2.2 Database Routines

For ANIMALS, three fields are required for each record: the question to ask, the record to go to for a "Yes" response, and the record to go to for a "No" response. These are stored in the data file as separate lines of text, with the record number prepended to each record. Thus, the data file for the sample tree in Figure 1 begins:

```
1
Does it fly?
2
3
2
Does it live on statues?
4
5
```

What is needed next is a routine to go to a specific record in the file, keyed by the record number. The `\Scan` routine does just that, assuming that TEX is already aligned at the start of a record. Once TEX is at the correct record, the `\Query` routine deals with extracting the data and selecting the next record.

This solves the problem of reading a quasi-random-access database. If the database needs to be modified, however, we run into difficulties. TEX allows a file to be open only for input or output, but not both. Furthermore, modifying a variable-length field within a file would be impossible, even if we could modify (rather than replace) an existing file.

The solution, of course, is to read in the original data from the beginning of the data file, copying each field to a temporary file. For ANIMALS, we find that one record needs to be replaced and two new ones added. Therefore, the record number is watched and, when the record to be replaced is reached, a modified version is output. Finally, the two additional records are output to the temporary file, which contains the revised database.

The process is now repeated in the other direction. The original file is replaced by a line-by-line copy of the temporary file. Since `\openout` overwrites the original file, this is an effective way to "modify" that original file.

Users of UNIX have another option to this way of modifying the data file, although it violates the spirit of this exercise. They can run, in the background, `tail -f /tmp/shell.tex : /bin/csh` and output editor commands to `/tmp/shell.tex`. This is cheating, however, since the goal is to write seemingly useless programs *entirely* in TEX.

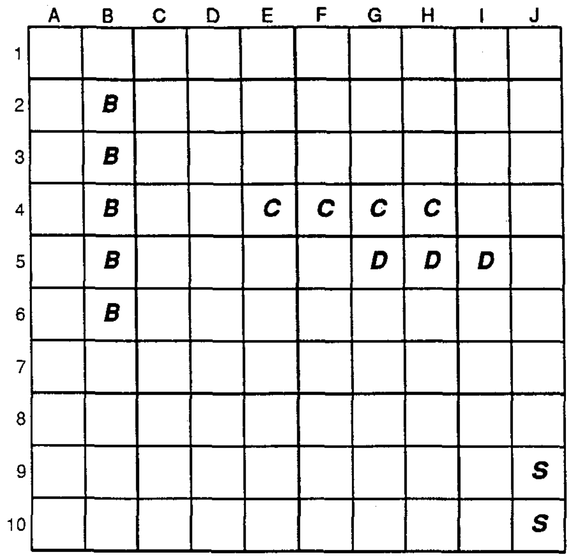


Figure 2a: A typical BATTLESHIP setup

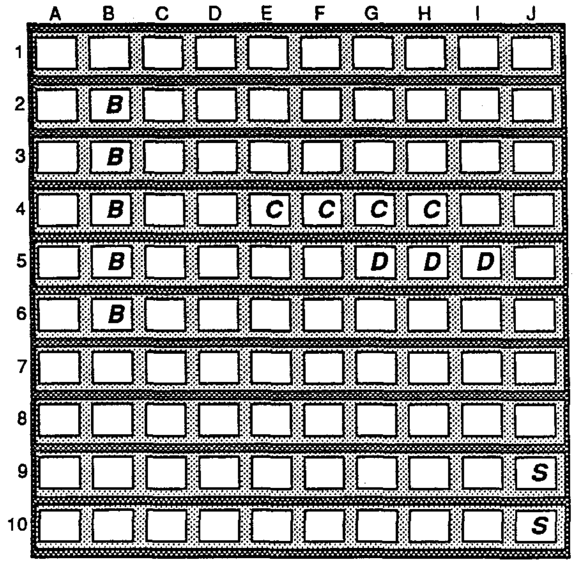


Figure 2b: Box representation

2.3 Other Database Applications in TEX

Despite the intentions of the author to avoid presenting any useful (i.e., typographic) code in this paper, there is one program that ought to be mentioned.

The author is in charge of "CokeComm" at the Student Information Processing Board at MIT. CokeComm is a debit-based system in which members deposit money, Coca-Cola products are delivered in bulk, and members mark a space on a list whenever they take a can. This calls for a program that can maintain balances and print out new sheets, properly formatted.

The code for `coke.tex`, which appears in Appendix B, should be compared to the ANIMALS code. The comments in `coke.tex` are intended to illustrate how to splice the database routines into other programs.

3. BATTLESHIP

BATTLESHIP is the most well-known variation of a game also known as "Salvo" or "Naval Battle." The implementation described in this section is based on the Milton Bradley version. BATTLESHIP was written as an excuse to experiment with arrays and indexed variables in TEX.

3.1 Rules

This version is simplified slightly by having fixed ship positions. The computer places four ships, each of a different length, in a ten-square grid. The ship designations and their lengths are "Carrier" (5 squares), "Battleship" (4), "Destroyer" (3), and "Submarine" (2). The ships are aligned either vertically or horizontally, so that each takes up the appropriate number of adjacent grid spaces. A typical arrangement is shown in Figure 2a.

Once the ship positions have been entered, the second player begins. On each turn, the player selects a grid location at which to shoot. The computer responds with the result — either the designation of the damaged ship or the text "You missed." If all cells of a particular ship are damaged, the ship is considered sunk; when all ships in the fleet are sunk the game is over.

It is obvious that the naval grid is a two-dimensional array. Unfortunately, TEX does not support array variables. The command string `\array` refers to typesetting a collection of elements in row-column format, surrounded by large delimiters. This is not what we want.

3.2 The Wrong Way

One's first impulse might be to implement arrays as nested boxes, as depicted in Figure 2b. An array variable might be declared with `\newbox\MyGrid` and initialized appropriately. In the diagram, `\box\MyGrid` is the large, dark `\vbox`, inside of which are ten light grey `\hbox`es, inside each of which are ten white \hboxes containing the array elements.

There are a number of disadvantages to this approach. An admittedly trivial objection might be that there are only 256 box registers, and some of those are reserved. Any programmer who needs more than 250 different arrays would have difficulties.

A more realistic objection arises when one considers how to extract element (i, j) from \MyGrid. \MyGrid needs to be \unvcopy'ed; the first $i - 1$ boxes need to be thrown away. The next box needs to be copied into a scratch box register, which is then \unhcopy'ed. Again we throw away $j - 1$ boxes, saving the next box, and tossing out $18 - i - j$ more boxes, before returning the saved value of $\texttt{\MyGrid}_{ij}$.

If that seems convoluted, consider how to *modify* the value stored in position (i, j). The neglected subarrays from the last paragraph now need to be stored up and re-combined correctly. An English description of how this can be accomplished is left as an exercise to the masochistic reader.

3.3 A Better Way

First, let's examine the simpler problem of a one-dimensional array. What is needed is a way to refer to a unique value pointed to by the two fields *ArrayName* and *Index*.

Fortunately, TEX has a pair of primitives (\csname and \endcsname) that allow us to do just that. Everything between the \csname and \endcsname is evaluated and formed into a control sequence. This control sequence can then be used like any other.

One needs to exercise caution in setting this up. If the *Index* is a \count, then it needs to be forcibly expanded into characters by using \the. If it's a constant (or a macro), however, using \the will cause a TEX error. The solution, of course, is to examine the category of the unexpanded *Index* and only use the \the if *Index* isn't already a constant.

In TEX code, that results in:

```
\def\elt#1#2{\csname #1.\if\relax #2\the\fi #2\endcsname}
```

The \if\relax #2 handles the category testing by comparing \relax, a control sequence (which is considered to be \char256 and have \catcode=16), and #2, the *Index*. If these match, then *Index* is a control sequence and the \if should insert a \the to force expansion of #2 to a constant. The period after the #1 serves to separate the *ArrayName* from the *Index*.

There are a number of advantages to this method. First, \csname allows digits to be part of the control sequence, which is normally not allowed. This protects the array elements from direct access.

Second, since TEX has to do minimal evaluation for any array reference, this method is the fastest for both reading and writing array elements.

Third, the contents of the array can be anything, and need not be the same between elements. One control sequence can be a macro, another can be a \count register, and yet another can be a \vbox.

Fourth, the *Index* of the array can be *any string*, including alphabetics. Space is allocated only for the elements that are used, and arrays can be indexed on, for example, the last names of students in a class.

3.4 Using \elt

Reading an array element is as simple as \elt{MyGrid}{17}. Writing to an array element is a bit more tricky, since \def\elt{MyGrid}{17}{value} will result in redefining \elt to be MyGrid and typesetting the string 17value. Using \expandafter, we can have the \elt macro expanded into the control sequence that we want to define:

```
\expandafter\def\elt{MyGrid}{17}{value}
```

To save typing \expandafter, it is convenient to define \put as follows:

```
\def\put#1#2#3{%
\edef\AINAME{\elt{#2}{#3}}%
\expandafter #1\AINAME}
```

\put is called with three arguments. The first is the version of \def to use (e.g., \def, \edef, \outer\def). The second and third are the *ArrayName* and *Index*, respectively. \put first finds the

Array-Index Name and stores it in `\AINAME`. It then constructs the correct version of the `\expandafter` technique, causing the token after the third argument to be taken as the value.

This will be problematic once an element has already been assigned a value, since the `\edef` will expand the desired element all the way to that value. Therefore, we need to copy over the definition of `\elt` into the definition of `\put`, which also saves us the effort of using `\AINAME`:

```
\def\put#1#2#3{%
\expandafter #1\csname #2.\if\relax #3\the\fi #3\endcsname}
```

3.5 Two-Dimensional Arrays and the Naval Battle

It is not difficult to modify this code to use two index variables and act as a two-dimensional array, which is what the BATTLESHIP program, whose listing is in Appendix C, does. The program starts off by defining `\put` and its counterpart, `\get` (which is a better name than `\elt` now that we are not putting a call to `\elt` in `\put`). It then initializes the ten-square grid to all "Z" (for zero). This version does not feature random ship placement, so the four ships are hard-coded in the next section.

The macro `\damage` will record damage to the ship whose counter is passed; it then uses `\string` to use the name of the counter as the name of the ship in the message.

The main loop follows. One of its distinguishing features is the use of `^^C` to prevent the user from confusing TEX by inputting something other than a coördinate in the form <*letter*><*digit*>, with <*letter*> in the range A to J and <*digit*> any digit from 0 to 9. Another useful point is that `\csname` will make an unknown control sequence expand to `\relax`, which is how BATTLESHIP checks for invalid coördinates.

3.6 Potential Applications

There are a number of uses for array variables in TEX. The most significant of these is a combination of an array of records with the database routines discussed in Section 2. A file could be read into an array, manipulated as a truly random-access database, and then written out (over the original file) at the end of the session. Code to do this, as well as all the code in the appendices, is available for anonymous FTP from 18.72.1.4 (`gevalt.mit.edu`).

4. Conclusion

There are other games that offer interesting challenges to the TEX programmer. For example, a full implementation of BATTLESHIP, not to mention any card game, would require a fairly good pseudo-random number generator, using TEX's simple integer arithmetic facilities.

Why program games, or any non-typographical code, in TEX? First, as can be seen from the COKECOMM application, routines written for games can find use in "real" TEX programs. Second, one is more likely to experiment if the end result is actually fun. Third, the results are more interesting to other people, who can learn from one's experiment, as this paper testifies.

And, besides, it's nice to have something to show people who still use Scribe.

Appendix A: Animals

```
% ---------------------------begin: animals.tex----------------------
%  Animals   (in TeX, no less!!!)
%
%  This is the program that uses a binary tree of questions to
%  guess the type of animal of which the user is thinking.
%
%  Andrew Marc Greene
%  <amgreene@athena.mit.edu>
%  Student Information Processing Board, MIT
%  March-April 1988
%
%  Cleaned up April 1989
%
%  Moral support (i.e., ``You can't do that!  Show us!'')
%  provided by the Student Information Processing Board
%  of MIT.
%
% Instructions on running this program:
%
%    tex animals
%
% Think of an animal.  The program will try to guess your animal.
%
% You will be asked a whole bunch of yes/no questions.  This is a
% spartan implementation, so answer with a capital Y or N.  When
% the program finishes going through its tree, it will either have
% guessed your animal or it will ask you to enter a question that
% it can ask to differentiate between your animal and its guess.
% It will then ask you which one is `yes.'
%
% Here's where I declare all my variables, etc.
%
% ``curcode'' is the current index into the data file.
% ``temp'' is a temporary holding variable.
% ``lc'' is a loop counter
% ``ifamg'' is a general-purpose flag.  amg are my initials.
% ``ifreploop'' controls loop repetitions.
% ``ifmainlooprep'' controls repetitions of the main loop.
% ``inp'' is the input file.
% ``outp'' is the output file.
% ``amgY'' and ``amgN'' are character constants.  Why I did it this way I
%                          don't remember.
%
\newcount\curcode\curcode=1\newcount\temp\temp=0\newcount\lc
\newif\ifamg\newif\ifyn\newif\ifreploop\newif\ifmainlrep
\newread\inp\newwrite\outp\def\foo{}
\def\amgY{Y}\def\amgN{N}
%
% The data file consists of records stored in the following format:
%
% Record Number <newline>
% Question <newline>
% If-Yes-Goto-Record Number <newline>
% If-No-Goto-Record Number <newline>
%
% The following routine scans the data file until it reaches the
% record requested in \curcode
%
\def\Scan{
{\loop
 \global\read\inp to \thisrecnum
 \ifnum\thisrecnum=\curcode\amgfalse\else\amgtrue\fi
 \ifamg
   \read\inp to \foo   % Discard unwanted record
   \read\inp to \foo
   \read\inp to \foo
\repeat}}
%
% The following routine displays the question and waits for a Y or N
% answer
%
\def\Query{
{\read\inp to \question
\immediate\write16{}
\message{\question}
\GetYN
\ifyn
  \read\inp to \foo\global\curcode=\foo\read\inp to \foo
\else
  \read\inp to \foo\read\inp to \foo\global\curcode=\foo
\fi
}}
%
% The following routines deal with the user's input
% \vread (verbatim read) ignores <newline>s and makes <space>s normal
% \GetYN gets input and repeats until it gets a Y or N response.
%
\def\vread#1{\catcode`\^^M=9\catcode`\ =12\global\read-1 to #1}
\def\GetYN{
{\loop
\vread{\bar}
\def\baz{\bar}
\reploopfalse
\if\amgY\baz\global\yntrue\else
 \if\amgN\baz\global\ynfalse\else\replooptrue\fi\fi
\ifreploop
\immediate\write16{Hey, you!  Answer Y or N, please.}
\message{Please enter Y or N -->}
\repeat
}}
%
% The following routine is called if the ``Goto-Record'' is -1,
% meaning that the program didn't guess correctly and is clueless.
% It gets the new animal and the differentiating question, and
% modifies the data file.  Actually, it makes a modified copy of
% the file, then copies the temporary new one over the old outdated
% one.
%
\def\NewAnimal{
\immediate\write16{Well, I'm stumped.  What animal did you have in mind?}
\vread{\usersanimal}
\immediate\write16{OK.  What question would let me tell the difference?}
\vread{\userquery}
\immediate\write16{Is the answer to that question Yes or No if I ask about}
\message{\usersanimal?}
\curcode=-1
\GetYN
\Scan
\read\inp to \lastcode\lc=\lastcode
\closein\inp
%
% Open up the files.  These names are system-dependent.   *FLAG*
%
\openin\inp=/mit/amgreene/TeXhax/animals.dat
\immediate\openout\outp=/tmp/animals-new.dat
%
% Read through the inp file, copying all records that don't need to
% be changed, outputting modified versions of the changed ones (and
% discarding the old), and appending the new records.
%
{\loop
 \read\inp to \foo
 \amgtrue
 \ifnum\foo=\temp\amgfalse\fi
 \ifnum\foo=-1=\amgfalse\fi
 \ifamg\immediate\write\outp{\foo}
  \read\inp to \foo\immediate\write\outp{\foo}
  \read\inp to \foo\immediate\write\outp{\foo}
  \read\inp to \foo\immediate\write\outp{\foo}
  \amgtrue
 \else\ifnum\foo=\temp
  \immediate\write\outp{\foo}
  \immediate\write\outp{\userquery}
  \immediate\write\outp{\number\lc}
  \global\advance\lc by 1
  \immediate\write\outp{\number\lc}
  \read\inp to \animal\read\inp to \foo\read\inp to \foo
  \amgtrue
 \else
  \lc=\lastcode
  \ifyn\WriteUsers\WriteAnimal
  \else\WriteAnimal\WriteUsers
  \amgfalse\fi
 \fi\fi
 \ifamg
\repeat}
\immediate\write\outp{-1}
\immediate\write\outp{\number\lc}
\closeout\outp
\closein\inp
%
% Now copy the temporary file over the original one
%
% These filenames are also system-dependent.          *FLAG*
%
\openin\inp=/tmp/animals-new.dat
\immediate\openout\outp=/mit/amgreene/TeXhax/animals.dat
{\loop
  \read\inp to \foo
  \immediate\write\outp{\foo}
  \amgtrue
  \ifeof\inp\amgfalse\fi
 \ifamg
\repeat}
}
%
% This routine is called by NewAnimal and writes the record for
% the user's new animal
%
\def\WriteUsers{
\immediate\write\outp{\number\lc}
\immediate\write\outp{Is it \usersanimal?}
\immediate\write\outp{0}
\immediate\write\outp{-1}
\global\advance\lc by 1}
%
% This one writes the modified old animal
%
\def\WriteAnimal{
\immediate\write\outp{\number\lc}
\immediate\write\outp{\animal}
\immediate\write\outp{0}
\immediate\write\outp{-1}
\global\advance\lc by 1}
\openin\inp=/mit/amgreene/TeXhax/animals.dat  %   *FLAG*
%
% Now we get into the main routine.
% It simply repeats the scan-query loop until it gets a 0 (right answer)
% or a -1 (wrong answer, I'm stumped), and calls the appropriate routine.
%
\loop
 \temp=\curcode
 \Scan\Query
 \mainlreptrue
 \ifnum\curcode=0
   \immediate\write16{Thank you for using Animals.  I'm glad I got it right.}
   \mainlrepfalse
 \else
   \ifnum\curcode=-1\NewAnimal\mainlrepfalse\fi
 \fi
\ifmainlrep
\repeat
%
% Ah, the joys of a job well-done.  We can now exit to the system, knowing
% that the world is a slightly better place for our efforts.
%
% The following line of code, probably the most profound in the entire
% program, sums up this philosophy of life in four characters.  The
% Puritan work ethic is embodied in this amazingly meaning-laden
% command designed by Donald Knuth.
%
\bye
-------------------------end: animals.tex-------------------------
```

Appendix B: CokeComm

```
-------------------------begin: coke.tex--------------------------
%
%  CokeComm program for SIPB
%
%  Andrew Marc Greene
%  <amgreene@athena.mit.edu>
%  Student Information Processing Board, MIT
%  March 1989
%
\newif\ifamg
%
%  Macros for typesetting each person's entry on the list
%
\def\person#1#2#3#4{%  Name, username, cans, paid
\vbox{%
\hbox{%
\hbox to 1.5in{\strut#1\hfill}\hbox to .5in{\hfill#3\quad}%
\hbox to .7in{\hfill\$#4\quad}\bubbles}%
\hbox{\hbox to 2.7in{\strut\tt #2\hfill}\bubbles}}\hrule}
%
%
\newbox\fivebubbles
\setbox\fivebubbles=\hbox{$\cal{O}\cal{O}\cal{O}\cal{O}\cal{O}\ $}
\def\five{\copy\fivebubbles}
\newbox\bubblebox
\setbox\bubblebox=\hbox{\five\five\five\five\five\qquad\five\five\five}
\def\bubbles{\copy\bubblebox}
%
\hsize=8in\hoffset=-.75in
%
\font\title=cmbx10 at 17.2667pt
\font\coltit=cmbx12
%
\headline={\hfil}
{\centerline{\title CokeComm Sheet}
\bigskip{}
\vbox{\hbox{
```

```
\strut\hbox to 1.5in{\coltit Name\hfill}\hbox to .5in{\coltit cans\hfill}%
\hbox to .7in{\coltit Balance\hfill}\hbox to \wd\bubblebox{\coltit
Soda\hfill {\sl --Fill in Circle--}\hfill Juice}}\hrule}}
\footline={\hfil}
%
\def\flush{\immediate\closeout3\closein2}
%
% Here's an old friend...  (from Animals)
%
\def\vread#1#2{\catcode`\^^M=9\catcode`\ =12\global\read#1 to #2}
%
%
% ---- END PREAMBLE ----
%
%  Last changed 20-Mar-89
%
\def\NextRecord{
\vread{2}{\pname}
\vread{2}{\obalance}
\vread{2}{\ocans}
}
%
\immediate\openout3=coke.dat
\immediate\openin2=oldcoke.dat
\newcount\balance\newcount\cans\newcount\numb\newcount\dollars\newcount\rcount
\loop
\immediate\write16{----------------Next Person-----------------}
\ifeof2\amgfalse\else\amgtrue\fi
\ifamg
\NextRecord
\balance=\obalance
\cans=\ocans
\immediate\write16{\pname (\uname)}
\message{Total Deposits:}
\vread{-1}{\adddep}
\advance\balance by\adddep
\message{Enter sodas: }
\vread{-1}{\sodas}
\advance\cans by \sodas
\numb=\sodas
\multiply\numb by 35
\advance\balance by -\numb
\message{Enter juices: }
\vread{-1}{\juices}
\advance\cans by \juices
\numb=\juices
\multiply\numb by 45
\advance\balance by -\numb
\edef\nbalance{\the\balance}
\dollars=\balance
\divide\dollars by100
\multiply\dollars by100
\numb=\balance
\advance\numb by-\dollars
```

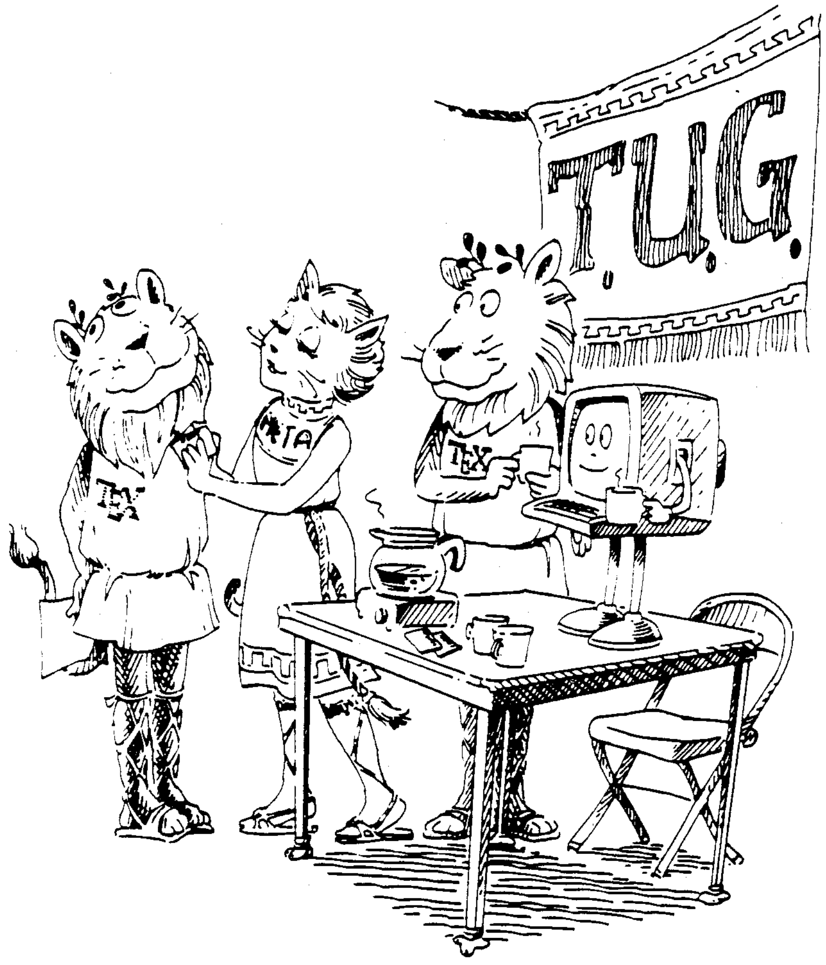
```
\divide\dollars by100
\def\sep{.}
\ifnum\numb<0 \multiply\numb by -1
\else
\ifnum\numb<10 \def\sep{.0}\fi
\fi
\def\bal{\the\dollars\sep\the\numb}
\edef\ncans{\the\cans}
\immediate\write3{\pname}
\immediate\write3{\uname}
\immediate\write3{\nbalance}
\immediate\write3{\ncans}
\immediate\write16{\pname : \uname : \ncans : \nbalance}
\person{\pname}{\uname}{\ncans}{\bal}\penalty-100
\repeat
\immediate\closeout3
\bye
------------------------end: coke.tex--------------------------
```

Appendix C: Battleship

```
------------------------begin: battle.tex----------------------
%  Battleship in TeX
%
%  Andrew Marc Greene
%  MIT Project Athena
%  and
%  Student Information Processing Board
%  Version 1.0 April 1989
%
%  Battleship is a registered trademark of the Milton Bradley Corp.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Array-handling code (modified to handle two-dimensional arrays)
%
%  \put    takes four arguments:
%             the variant on \def
%             the array name
%             the two index values
%
\def\put#1#2#3#4{\expandafter #1%    \def -- but first find the AINAME
\csname #2%                          begin the \csname and use the arrayname
.\if\relax #3\the\fi #3.%            first index
\if\relax #4\the\fi #4\endcsname }%  second index and end the \csname
%
%
%  \get    takes three arguments:
%             the array name
%             the two index values
%
\def\get#1#2#3{\csname #1%          same as above....
.\if\relax #2\the\fi #2%
.\if\relax #3\the\fi #3%
\endcsname}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  \say is a useful shorthand for ``output to screen''
%
\def\say#1{\immediate\write16{#1}}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Initialize the array to all ``Z''
%
\newcount\idx\newcount\idy
\idx=0\idy=0
\loop
\edef\Idx{\ifcase\idx A\or B\or C\or D\or E\or F\or G\or H\or I\or J\fi}
\put{\def}{MyGrid}{\Idx}{\idy}{Z}%
\advance\idx by 1
\ifnum\idx=10
\advance\idy by 1
\idx=0
\fi
\ifnum\idy<10
\repeat
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Display welcome message
%
\say{Welcome to Battleship.}
\say{(Battleship is a trademark of Milton Bradley)}
\say{}
\say{This version uses fixed-position ships.}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Position the ships.  Future versions will use a random-number
%  generator to provide a different game each time.
%
\put{\def}{MyGrid}{D}{4}{C}%        Carrier has 5 spaces
\put{\def}{MyGrid}{E}{4}{C}
\put{\def}{MyGrid}{F}{4}{C}
\put{\def}{MyGrid}{G}{4}{C}
\put{\def}{MyGrid}{H}{4}{C}
\put{\def}{MyGrid}{B}{1}{B}%        Battleship has 4
\put{\def}{MyGrid}{B}{2}{B}
\put{\def}{MyGrid}{B}{3}{B}
\put{\def}{MyGrid}{B}{4}{B}
\put{\def}{MyGrid}{G}{7}{D}%        Destroyer has 3
```

```
\put{\def}{MyGrid}{G}{8}{D}
\put{\def}{MyGrid}{G}{9}{D}
\put{\def}{MyGrid}{J}{6}{S}%        Submarine has 2
\put{\def}{MyGrid}{J}{7}{S}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Initialize some counters
%
\newcount\turns\turns=0%             Number of turns it takes to win
\newcount\hits\hits=14%%             Number of hits it takes to win
\newcount\carrier\carrier=5%         Number of hits to sink each ship
\newcount\battleship\battleship=4
\newcount\destroyer\destroyer=3
\newcount\submarine\submarine=2
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Routines to handle damage to a ship
%
%  \gobble (as defined in the TeXbook, page 308)
%
\def\gobble#1{}%                     Remove one token
%
%  \damage takes the name of a counter and damages that ship:
%
\def\damage#1{\advance#1 by -1 %     Lose one 'hit point'
\ifnum #1=0 %                        If there are no more,
%                                    Print a 'sank' message:
\say{You sank my \expandafter\gobble\string #1!}
\else%                               Otherwise, a 'hit' message:
\say{You have hit my \expandafter\gobble\string #1.}
\fi}
%
%  Note that the above messages used \expandafter\gobble\string #1
%  to get the name of the counter and strip the \escapechar off the
%  front of it.  The resulting string (because of the way we named
%  our counters) is the name of the appropriate ship.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Give the player a chance to specify a coordinate.
%  If it's already been used, or is out-of-bounds, \say an error.
%  Otherwise increment \turns and print an appropriate message;
%    If it is a hit, decrement \hits;
%    Whether it is a hit or not, blat the space.
%  If \hits is non-zero, keep going.....
%
%
\def\defab#1#2#3^^C{\def\a{#1}\def\b{#2}}
\loop
\message{Your turn: }
\read-1 to \usrinp
```

```
\expandafter\defab\usrinp^^C
\edef\c{\get{MyGrid}{\a}{\b}}
\if\c\relax
\say{Sorry, that's not a valid coordinate.}
\else
\if\c X\say{Sorry, you already shot there.}
\else\advance\turns by 1
\if\c Z\say{You missed.}
%
%  Drat!  The user hit one of our ships!  Record damage to the correct one.
%
\else\if\c C\damage{\carrier}
\else\if\c B\damage{\battleship}
\else\if\c D\damage{\destroyer}
\else\if\c S\damage{\submarine}
\fi% submarine
\fi% destroyer
\fi% battleship
\fi% carrier
%
%  And record that there was a hit someplace:
%
\advance\hits by -1
\fi%                                 End of the hit-or-miss section
%
%  Record that this space has been shot:
%
\put{\def}{MyGrid}{\a}{\b}{X}
\fi%                                 End of the 'shoot here' section
%
\fi%                                 End of the {in}valid spot section
%
%  We've finished a cycle.  If there is any part of the fleet left,
%  we go around again:
%
\ifnum\hits>0\repeat
%
%  Otherwise, we display the player's score and exit.
%
\say{You have destroyed my fleet.  It took you \the\turns\gobble1 turns.}
\bye
--------------------------end: battle.tex---------------------------
```



TEX Users Group

Tufts University, July 20–23, 1986

P. T. Barnum Auditorium