Using a High-Level Language as an Aid in Writing TEX Documents

Harry L. Baldwin, Jr.
San Diego City College
1313 12th Ave.
San Diego, CA 92101

Abstract

Certain mathematical procedures associated with the preparation of a document are best handled by some high-level language other than TEX. One example is the generation of random numbers and the subsequent ordering of those numbers as a means of scrambling lines or groups of lines in the preparation of different forms of a test. Other examples involve the generation of curves and angles for inclusion in graphs and diagrams. An alliance of the True BASIC programming language and TEX has proven to be a useful combination for the efficient creation of TEX source files.

Introduction

Although TEX was designed for quality typesetting, and to reach that goal Don Knuth filled his program with many capabilities, most users would probably admit that number crunching is more easily done in some other high-level language. In my four years of using TEX as a mathematics teacher, I have found several applications in which my attempts to produce high-quality output are simplified by using TEX in close association with another language. This article will be a chronological account of some of those applications.

The high level language that I use is not one of the latest exotic creations to which *Byte Magazine* might devote an entire issue, but rather what has lately been looked upon as the Rodney Dangerfield of computer languages, BASIC. My first meshing of TEX and BASIC came when I implemented PCTEX and needed an editor capable of creating an ASCII file. Although I was doing True BASIC number crunching on the PC, all my word processing was being done on a NorthStar Horizon computer (anybody remember that old warhorse?). I decided to write my TEX source files using the True BASIC editor until something better came along.

The True BASIC editor has been working so well that I've never bothered searching for a better one. The only time it complains is when I occasionally accidently hit F9 (the "RUN" command) at which time it politely tells me that "`\documentstyle{book}`" is an illegal statement. All the editing features I need are available, such as search and replace, block move, copy, delete, and so forth, together with the capability to insert another file into the working file at any desired location.

Scrambling Questions and Answers

My first uses of TEX were handouts and tests (where "tests" means both long "examinations" and short "quizzes"). Since the number of students in my classes kept getting larger each semester, but the classroom sizes somehow stayed the same, roving eyes during tests became a problem. Giving different versions of a test seemed to be the appropriate action to take. The versions would differ from one another by either scrambling the order of choices to each multiple-choice question, or by scrambling the order of the questions on the test, or perhaps by using both of these techniques. Rather than have TEX create the different versions of a test — an approach adopted by Don De Smet (1991) — my strategy is to let True BASIC do the scrambling: A TEX source document (let's assume it is named `TEST.SCR`) is input to a BASIC program (called `SCRAMBLE.TRU`) that searches the lines for certain code characters. The codes identify lines whose positions will be changed in a manner that will be described below. A new source file (let's name it `TEST.TEX`) is written to the hard disk, all ready to be compiled under LATEX.

Two types of scrambling can be done: scrambling of questions and scrambling of answers. A multiple-choice question comprises a group of lines that contain the question's "root" and the five "answers" (the single correct answer and the four "distractors"). Scrambling of questions requires changing the positions of blocks of lines, while scrambling of answers involves shuffling a few contiguous lines within the block. The type of scrambling that will be done is determined by "code characters" that appear on some lines of `TEST.SCR`. The code characters

are placed to the far right end of the line for easy identification during screen editing of the source file. The leftmost character in any sequence of code characters is the comment character %, which will not interfere with the subsequent compilation of the document.

The beginning line and ending line of a question that is to be scrambled are identified by the code characters %B and %E in character positions 76-77. These lines, together with all lines in between, form a block whose position will be scrambled. The end of the last question in a group of questions that is to be scrambled is identified by %EL in positions 76–78.

A question that is to be scrambled need not be a multiple-choice question, but if it is, usually we want the answers to be scrambled also. Each of the answers in a multiple-choice question (we will assume there are five) will be an argument of a macro designed to output the answers appropriately. In TEST.SCR the answers must be written on five contiguous lines, one answer per line. (If an answer is too long to fit on one line, then that answer should be made into a macro, and the macro command put on the line.) To identify an answer that is to be scrambled, the comment symbol % is placed in character position 78 on that line. For example, if only the first four answers are to be scrambled (maybe the fifth answer is "all of the above") then four contiguous lines would contain a percent symbol in position 78.

The scrambling is done as follows: The source file TEST.SCR is read into memory, each line destined to become an element of a one-dimensional string matrix that we'll name TEST$. As each line is entered, it is examined for a percent symbol in position 78, which would identify that line as an answer to be scrambled. If such a symbol is not found, then the line is merely appended to TEST$, but any sequence of answer lines that are to be scrambled are first stored in another temporary array. Pseudo-random numbers are generated (the "pseudo" is for the purists — I'll just call them random numbers) and placed in another column of that array. The rows of the array are then reordered so that the random numbers increase as we go down the column, and the lines associated with those numbers have their positions changed as well. The reordered answer lines are then transferred to TEST$. This process continues until all lines of TEST.SCR have been entered.

The lines of the matrix TEST$ can now be output to TEST.TEX, and the only change from TEST.SCR will be the order of answers. However, if scrambling of questions also is desired, then some more juggling is required: As each line of TEST$ is output, it is examined for %B in positions 76–77, which would mark the "begin-line" of a question whose position is to be scrambled. If not found, then the line is merely output to TEST.TEX; if found, output is suspended while the program searches for the corresponding "end-line" of that question, identified by %E is positions 76–77. The line numbers that identify that question are stored in a row of a "line-number matrix", and search continues for another begin-end pair of line numbers, which will be stored in the next row of that matrix. After the end-line of the last question to be scrambled is identified (by %EL in positions 76–78), then a random number is assigned to each row of the line-number matrix, and those rows are reordered. The output of TEST$ is then resumed, with the order of the lines indicated by information contained in the rows of the reordered line-number matrix.

Sometimes a test should not have all questions scrambled together, but rather, say, the first twenty easier questions scrambled, and then the next thirty more-difficult questions scrambled. This can be done by the above scheme, by merely terminating each group that is to be scrambled with a line containing %EL in positions 76-78.

Each time a True BASIC program is run, the same sequence of random numbers is generated. To obtain a version of a test, SCRAMBLE.TRU asks for the date and the form number, and then discards the first n random numbers, where the number n is given by $n = 366 \cdot (\text{yr}+\text{form}) + 31 \cdot \text{mo} + \text{day}$. For example, to generate the fifth form of my last April Fool's test I entered the "seed" 0401925, and the first 35 627 random numbers were discarded (slowing the execution by less than a second). If, for some reason, I need to recreate the version I can input the same seed and obtain exactly the same test.

The macro chosen to output the answers to a multiple-choice question depends on the lengths of those answers. Five macros have accommodated all possibilities I've needed so far; letting r:s:t... represent r answers on the first line, followed by s answers on the next line, etc., the macros will print the answers following one of these patterns: 5, 3:2, 2:2:1, 4:1, and 1:1:1:1:1.

An example of what a couple of questions in the source file might look like, and what output might be produced, is shown in Figure 1 (in the Appendix). The command \QQQQR is the macro that outputs four answers on one line and the fifth answer on the second line (such an arrangement would only be used if the fifth answer is not to be included in the scrambling); the command \QQRRS outputs two

answers on the first line, two on the second, and one on the third; (can you guess what `\QRSTU` would output?). These macros begin by incrementing the question number. Then (in a `\vtop`, so that a question won't be split by a pagebreak) a horizontal rule is drawn for question separation, followed by printing the root of the question (the first argument of the macro) and then the answers (arguments two through six). The macro concludes with another horizontal rule.

Generating a Test

`SCRAMBLE.TRU` worked so well that when the City College Mathematics Department decided to implement a departmental final exam for the Beginning Algebra course (and I was drafted to create this final), it seemed reasonable to let True BASIC compose the entire test. The test-making philosophy our department adopted was this: the final exam would be formed by selecting questions from a test bank that is so large that test security is of no concern. Currently our test bank contains 215 "master questions" (we are aiming for 300), each master question having ten quite-similar "versions". Since a final exam would be formed by selecting 60 master questions from the bank of 215, *and* any of the ten versions of each question will be randomly selected, *and* the 60 test questions would be scrambled (in two groups), *and* the answers would be scrambled, we feel that any student capable of beating the system by remembering all the questions and their correct answers is probably bright enough to realize that it would be easier just to learn the algebra.

Each master question (comprising ten versions) is stored as a separate file; for example, `Q130` is the file that contains master question 130. If that master question is selected for a test, then part of that file (a version of the question) is destined to be input by the BASIC program `MAKETEST.TRU` that will be composing the test. A random number determines which of the ten versions will be selected. Each version occupies 15 lines of the file, so version 4, for example, would extend from lines 46 through 60, with perhaps the last few lines being blank.

Refer to Figure 2 (in the Appendix) as we look at what `MAKETEST.TRU` does with the 15 lines of a question version that has been randomly selected. The first line contains five pieces of information about the question, with each of these pieces of information appearing in a specific location on the line:

Version number. A whole number, in character positions 9-10, identifies the version selected. During any subsequent scrambling, `MAKETEST.TRU` will keep track of the question master number (one of the 215 questions in the test bank) and the version number (1 through 10). After the test is constructed an answer key is printed that will include this information. If a bug in the question shows up then it is easy to locate the offending version.

Scramble code. A whole number, in character position 21, tells how many answers are to be scrambled: 0 (for no scrambling), or 3 (the first three answers), or 4, or 5.

Answerline code. A whole number (1, 2, 3, 4, or 5), in character position 36, tells `MAKETEST.TRU` what macro to use when outputting the answers. For example, code number 1 is five answers on one line.

Rootlines code. A whole number, in character position 49, tells how many lines contain the root of the question. The root lines immediately follow this first codeline, are written in TEX, and will be inserted directly into the TEX document being constructed. The maximum number of root lines is 9; although we have never needed that many lines, a macro could be defined and placed in the preamble if necessary.

Correct answer. A letter (A, B, C, D, or E), in character position 59, identifies the correct answer. The five answers are on the five lines that follow the root line, with exactly one answer per line. Before scrambling, answer A is the first answer listed, B is the second, and so forth. When writing a test question, I usually work the problem and write the correct answer on the first answer line, and then play the part of an inattentive student as I make up the distractors. During the subsequent scrambling of the answers, `MAKETEST.TRU` tracks the correct answer for inclusion on the answer key.

The construction of a test by `MAKETEST.TRU` proceeds as follows: The person creating the test selects the master numbers of the questions that will be on the test. These numbers are written in a file, perhaps named `FIN-S92` if they are the questions to be used on the final exam for the Spring semester in 1992. `MAKETEST.TRU` is then run, and begins by asking for the test title, subtitle, and special instructions. (Provision is made for placing this information in a file before running `MAKETEST.TRU`, and merely entering the name of the file when prompted.) `MAKETEST.TRU` next asks for the number of questions, for a random-number seed,

and for information on what groups of questions are to be scrambled. (If no question scrambling is to be done, then the questions will be output in the same order as they appear in the question file.) Then the master-question numbers are scrambled, and versions randomly selected.

The output file (let's again call it `TEST.TEX`) is now ready to be created. `MAKETEST.TRU` first calls a file named `PREAMBLE.TEX` that contains all the commands in a LaTeX document preamble. These lines are output to `TEST.TEX`. Then the title information and instructions are appended. Finally, the randomly selected version of each master question is input: the master number of a question is used to form the name of the file that holds the ten versions, the lines of that file are entered into memory, and the 15 lines that correspond to the selected version are retained while all other lines are discarded. Information is extracted from the codeline (the first line): how many of the following lines contain the root, how many answers are to be scrambled, how many answers should appear on each line, and which is the correct answer. The root of the question is then inserted as an argument of a macro written by `MAKETEST.TRU` using BASIC string functions, the answers are scrambled and inserted as arguments into the appropriate macro (also written by `MAKETEST.TRU`), and all of the lines are written to `TEST.TEX`. This procedure is repeated until the lines that will print all questions have been output to `TEST.TEX`.

The answer key is then constructed. After a `\pagebreak`, four columns of information are presented: the question number as it appears on the test, the corresponding master-question number, the version number, and the answer. The last hurrah of `MAKETEST.TRU` is to append `\end{document}`.

After the file containing the master question numbers has been created, the running of `MAKETEST.TRU` takes only a few minutes. Even though a lot of computation and scrambling is involved, on an 80386 running at 33 megahertz the questions are written to `TEST.TEX` at about one per second. For the final exam in Beginning Algebra this last Spring semester, all sections were to be given a test that used the same master questions. Since the tests were to be given at different times, eight runs were made using different seeds to obtain eight different forms, each form containing eight pages of questions plus an answer key. Creation of all eight forms of `TEST.TEX`, compiling under LaTeX, and then obtaining HP LaserJet II output ready for photocopying took about 40 minutes.

Writing the ten versions of a master question goes quickly. Since all versions are to be *very* similar, the roots of the versions usually will be the same, except perhaps for a mathematical expression or a few words. The codelines for each version usually differ only in the version number. Therefore, one version can be written, duplicated nine times, and then changes made to those parts where the versions differ.

Another BASIC program, `PRT-VER.TRU`, was written that will read in a master file and print all versions — a great help in proofreading the questions. Still another program, `PRT-ALL.TRU`, prints a single version from each master question. The output from `PRT-ALL.TRU` is what instructors look at when selecting master questions for a test, and what students look at when they are curious as to what might appear on a test.

This testing strategy can be a morale raiser for students, since they needn't fear being hit with a question of a type they have never seen before. But since the students would see only one version from each master question (printing all ten versions would make a booklet of several hundred pages), great care must be taken to insure that all versions are of the same difficulty. For example, a question that asks for one of the binomial factors of $x^2 + 7x + 12$ is not considered to be of the same difficulty as asking for one of the binomial factors of $x^2 - 4x - 12$, for this second trinomial involves both positive and negative integers. To insure that much thought is applied to the construction of the questions, a verbal description of the limitations is written for each master question. For example, the description of one trinomial-factoring question is as follows:

> A given second-degree trinomial has two first-degree binomial factors, one of which is to be selected from a list of five possibilities. The coefficient of the second-degree term is 1; the constant terms of the binomials will be non-zero integers, of different sign, having magnitudes less than 10.

Of course, a student who only knows his multiplication table up through fives might not consider the factoring of $x^2 + 3x - 54$ to be of the same difficulty as $x^2 + 3x - 10$, but at least we tried.

Figure 2 (in the Appendix) shows an example of a version from each of two master questions, as they would appear in the master-question files. Also shown is the output of these same questions as produced by `MAKETEST.TRU`.

Graphics Output

Other opportunities to mesh True BASIC and TEX arose from my efforts to produce mathematically-accurate drawings for my handouts, and for a geometry book recently completed (Baldwin, 1992). A brief description will be given here of only two of several BASIC programs that were helpful.

Most of the graphs I construct for tests and handouts require only a simple xy coordinate system framework, on which an arbitrary curve will be drawn. In the TEX document, a macro `\XYgrid` will construct the framework, label and place scales on the axes, and then leave the "cursor" at the left end of the top gridline of the framework. Another macro enters the LATEX picture environment, places the origin at the intersection of the x and y axes, and sets `\unitlength` equal to the unit distance on the framework. The TEX document is then saved while BASIC constructs the curve.

A True BASIC program named `CURVE.TRU` will write LATEX `\multiput` statements to a temporary file, which will then be inserted into the TEX document at the proper place. Each `\multiput` statement will place dots of a selected size along a straight-line segment defined by the endpoints. By making the segments quite short, and linking them together, a curve can be drawn.

The curve is defined by functions of a parameter — x and y each as a function of t — which are placed very near the beginning of `CURVE.TRU`. These functions are defined by editing `CURVE.TRU` before running (which is much easier that having `CURVE.TRU` ask for the functions during running). When run, `CURVE.TRU` first asks for the framework scale, the starting and ending values of the parameter t, and "delta t" (the change in t, which will influence the lengths of the straight-line segments. Finally, `CURVE.TRU` asks for the size and spacing of the dots that will form the curve. After a couple of seconds the program announces that it is finished, having written a series of `\multiput` commands to a file named `TEMP.TRU`. The TEX document is then reloaded, and the lines in `TEMP.TRU` are inserted at the proper location.

A dot diameter of 1 point (obtained by using the smallest possible LATEX `\circle*`) with a spacing between centers of 0.7 points creates a curve whose thickness matches a LATEX `\thicklines` line almost perfectly. Thinner curves can be made by using periods in smaller fonts.

Several variations of `CURVE.TRU` have been written: dotted curves, dashed curves, curves where checking is done so plotting is restricted to a certain area, and some other variations. Rather than have one giant elegant program that will do everything, I have found it more efficient to select one of a number of special programs available, so that a lot of queries needn't be answered during running. For example, `CURVE-DU.TRU` ("dashes, unlimited") makes a dashed curve, without checking for out-of-bounds points, while `CURVE-DL.TRU` ("dashes, limited") asks for the limits on x and y.

Some curves result in *many* `\multiput` commands being inserted into the TEX file — perhaps a few hundred. Occasionally I get a "`Sorry, TeX capacity exceeded`" message. Since I don't understand any of the rest of the message, I usually just shrink the curve or increase the delta t and try again.

I often draw curves using the excellent curve-drawing capabilities of PICTEX, especially circles and ellipses. For other curves, PICTEX requires computing the coordinates of some points that lie on the curve and either entering these coordinates into the TEX document or storing them on a file. Rather than worry if enough points have been computed, or if something strange is happening on the curve between those points, I just use `CURVE.TRU`. The compiling times for TEX documents that produce curves by the two methods are about the same.

Figure 3, immediately below this paragraph, shows an example. The framework (axes and labels, grid lines, and the scale) was generated by `XYgrid` (a TEX macro), but the curve was formed from 72 `\multiput` commands generated by `CURVE.TRU`, and inserted into the TEX source file for the document you are reading.

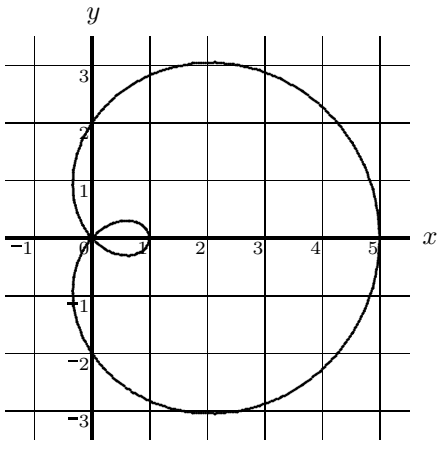


Figure 3: limaçon: $r = 2 + 3\cos\theta$

If an angle in a drawing is accurate to within about one degree of its labeled measure, then it takes a highly-trained eye to detect the error. Whenever

possible, I use the line-drawing capabilities of the `picture` environment of LATEX to draw an angle. The `\line` command in LATEX can draw a vertical line, or a line having slope $\pm y/x$, where x and y are whole numbers six or less. The available slopes allow an angle of any measure to be constructed with an error less than one degree, although not with an arbitrary orientation. If a figure has two or more angles, or if the orientations of one or more angles are constrained, then LATEX `\line` commands may not create a sufficiently-accurate drawing, so the sides of those angles may have to be drawn by another method. Furthermore, pictures of angles often require an arc to indicate the measure of the angle, and that arc might terminate with an arrow at one or both ends. Sometimes a picture should show the sides of an angle to be rays (an arrow pointing away from the vertex); sometimes an angle's sides are segments. A right angle might be indicated by a square placed at the vertex. All of these facts served as the incentive to let True BASIC do the job, by means of a program named `ANGLEARC.TRU`.

A LATEX picture environment is established, with the scale and the origin chosen, and the TEX source file is saved. When `ANGLEARC.TRU` is run, it asks (and I answer) these questions:

What is the scale?

What are the coordinates of the vertex?

What is the direction of the angle's side?

What is the length of angle's side?

Is the side a ray? If yes, a LATEX `\thicklines` vector of zero length having the closest-possible direction as the side will be placed at the end of the segment that forms the side.

Should a right-angle square be drawn at the vertex? If yes, then what is the size of the square?

What is the radius of the arc that will indicate the angle's measure? (An answer of zero indicates no arc is to be drawn, and the next two questions are skipped.)

Should arrows be placed at the terminal end of the arc, or at both ends of the arc? If yes, LATEX `\thinlines` vectors of zero length (and closest direction) are placed at the appropriate endpoints of the arc.

What is the central angle of the arc?

Is another side to be drawn? To construct the terminal side, answer yes and the questioning begins anew.

As the questions are answered, `ANGLEARC.TRU` constructs the proper LATEX `\multiput` and `\put` commands, and outputs them to `TEMP.TRU`. After no more angle sides are to be drawn, control returns to the BASIC editor and the TEX source file is reloaded. The cursor is moved to the proper location, and all the lines of `TEMP.TRU` are copied into the source file.

An example of output produced mainly by commands generated by `ANGLEARC.TRU` is shown in Figure 4, immediately below this paragraph. Except for the two angle labels and the Figure label, all of the drawing was done by commands generated by `ANGLEARC.TRU`. The vector arrows, and the dots that form the arcs, were placed with `\put` commands. The angle sides were formed from `\multiput` commands, which placed 1-point dots with their centers spaced 0.7 points apart. The right-angle square was formed similarly, but using smaller dots. The three labels were placed "manually" after returning to TEX.

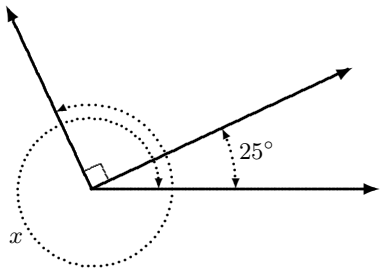


Figure 4: $x = ?$

Much use was made of `\ANGLEARC.TRU` and other True BASIC programs during the writing of my geometry book. Figure 5 (in the Appendix) shows part of a page from the book, one of many in which this program played an important role.

Every program described in this article could, of course, be implemented in other high-level languages — perhaps even in TEX. But for ease and convenience of use, True BASIC has earned my respect. I've enjoyed being a witness to the wedding of True BASIC and TEX, and I'm sure that the marriage will be a long and happy one in my computer system.

Bibliography

Kemeny, John G., and Thomas E. Kurtz. *True BASIC Reference Manual*. West Lebanon, N.H.: True BASIC, Inc., 1990.

De Smet, Don. "TEX Macros for Producing Multiple-Choice Tests." *TUGboat* 12(2),

pages 261–268, 1991.

Lamport, Leslie. LaTeX *User's Guide and Reference Manual*. Reading, Mass.: Addison-Wesley, 1986.

Wichura, Michael J. *The* PiCTeX *Manuel*. TeXniques 6, 1987.

Baldwin, Harry L. Jr. *Essential Geometry*. San Francisco, Calif.: McGraw-Hill, 1992.

Appendix

character position 1 ↓ … character position 77 ↓

```
\midexamplespacer      % Draws a horizontal rule                          %
%13                                                                      %B
\QQQQR{Which of the four complex numbers listed below has the
     greatest modulus?}
{$5+5i$}                                                                  %
{$7+3i$}                                                                  %
{$6+4i$}                                                                  %
{$8+2i$}                                                                  %
{They all have the same modulus.}
\midexamplespacer
                                                                         %E
%14                                                                      %B
\QQRRS{A circle of radius 1 is centered at the origin.  Starting at
     the point where $\,x=1\,$ and $\,y=0\,$, a distance $\,u\,$ is
     measured along the circle in a counterclockwise direction.  The
     coordinates of the location after moving this distance $u$ are}
{$x=\sin u,\ \ y=\cos u$}                                                 %
{$x=\cos u,\ \ y=\sin u$}                                                 %
{$x=\tan u,\ \ y=\cot u$}                                                 %
{$x=\cos u,\ \ y=\tan u$}                                                 %
{$x=\sec u,\ \ y=\csc u$}                                                 %
\midexamplespacer
                                                                         %EL
```

---

1. Which of the four complex numbers listed below has the greatest modulus?

A) $8+2i$ B) $6+4i$ C) $5+5i$ D) $7+3i$

E) They all have the same modulus.

---

2. A circle of radius 1 is centered at the origin. Starting at the point where $x=1$ and $y=0$, a distance u is measured along the circle in a counterclockwise direction. The coordinates of the location after moving this distance u are

A) $x=\cos u,\ \ y=\tan u$ B) $x=\sin u,\ \ y=\cos u$

C) $x=\cos u,\ \ y=\sin u$ D) $x=\sec u,\ \ y=\csc u$

E) $x=\tan u,\ \ y=\cot u$

Figure 1: An Example of part of `TEST.SCR` and the resulting output

```
        character   character     character     character   character
        position    position      position      position    position
          10          21            36            49          59
          ↓           ↓             ↓             ↓           ↓
version  2 scramble 5  answerlines 2  rootlines 3  answer B
The graph of the \U{intersection} of the
   equations\ \ $\cases{\hskip-.12in  &  $x-y=-3$  \cr \hskip-.12in
   &  $x+y=1$ \cr}$\hskip.2in is a point that is located
in Quadrant I.
in Quadrant II.
in Quadrant III.
in Quadrant IV.
on a coordinate axis.
```

(The remaining 6 lines of the 15 lines that form this version are blank.)

```
        character   character     character     character   character
        position    position      position      position    position
          10          21            36            49          59
          ↓           ↓             ↓             ↓           ↓
version  6 scramble 5  answerlines 1  rootlines 1  answer A
$\displaystyle\frac{x(x+5)+2(x+6)}{x+4}\ =$
$x+3$
$x+2$
$x+1$
$x+4$
$x+5$
```

(The remaining 8 lines of the 15 lines that form this version are blank.)

---

1. The graph of the intersection of the equations $\begin{cases} x-y=-3 \\ x+y=1 \end{cases}$ is a point that is located

A) in Quadrant IV. B) in Quadrant III. C) in Quadrant I.

D) on a coordinate axis. E) in Quadrant II.

---

2. $\dfrac{x(x+5)+2(x+6)}{x+4} =$

A) $x+1$ B) $x+3$ C) $x+2$ D) $x+4$ E) $x+5$

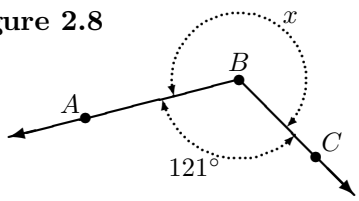
Figure 2: Examples of a version from each of two master questions, and the resulting output

Sometimes knowledge of the measure of one or more angles in a geometric drawing will enable you to determine the measures of other angles. The examples below show some of the techniques that can be used.

---

1. Rays BA and BC determine an obtuse angle having measure 121° (see Figure 2.8 at right). These same rays also form a reflex angle, which has been labeled x in the drawing. From knowledge of the measure of the obtuse angle ABC, determine the measure of the reflex angle ABC.

Figure 2.8



The sum of the measures of the obtuse angle and the reflex angle must be 1 revolution, or 360°. Although x is a name of the reflex angle, we will also let x represent the measure of this angle. Therefore,

$$121° + x = 360° \implies x = 360° - 121° \implies x = 239°.$$

---

2. In Figure 2.9a, angle XYZ is a right angle. Determine the approximate location of point P on the arc, if $\angle XYP$ is to have measure 45°.

Figure 2.9a

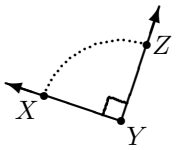
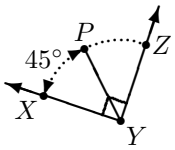


Figure 2.9b



Since 45° is half of 90°, point P must be located so that a rotation of ray YX to ray YP would be half of the rotation of that ray to ray YZ. If P is located halfway along the arc from X to Z (see Figure 2.9b), then the measure of $\angle XYP$ will be half the measure of $\angle XYZ$.

---

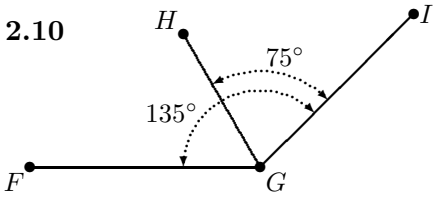
3. In Figure 2.10, these angles are shown:

$\angle FGI = 135°$ $\angle HGI = 75°$

a) What is the measure of $\angle FGH$?

b) What is the measure of reflex $\angle FGH$?

Figure 2.10



Unless information is given that an angle is a reflex angle, we assume that the angle we are interested in will be the one having the smallest possible measure. In part (a), therefore, $\angle FGH$ is referring to the acute angle, whose measure will be the difference between the measures of the obtuse angle FGI and the acute angle HGI. In part (b), we can determine the measure of the reflex angle by subtracting the acute angle's measure from 360°.

a) $\angle FGH = \angle FGI - \angle HGI = 135° - 75° = 60°$

b) reflex $\angle FGH = 360° -$ acute $\angle FGH = 360° - 60° = 300°$

Figure 5: Example of part of a page from *Essential Geometry* that made use of `ANGLEARC.TRU`