

By this time you've had quite a bit of programming experience in TI BASIC. You know what a program is, how it's structured, and how it's performed by the computer. Now we're ready to add a few more techniques to your programming skills.

In this chapter we'll introduce you to several new TI BASIC features. First, there's the very useful and versatile FOR-NEXT statement, which creates loops in programs. Next, we'll cover some "plain and fancy" printing, using the PRINT statement and the TAB function. Then we'll add some details about the "number power" of your computer: the way numbers are displayed on the screen and the order in which the computer performs mathematical calculations. Finally, we'll introduce you to the INTeger function.

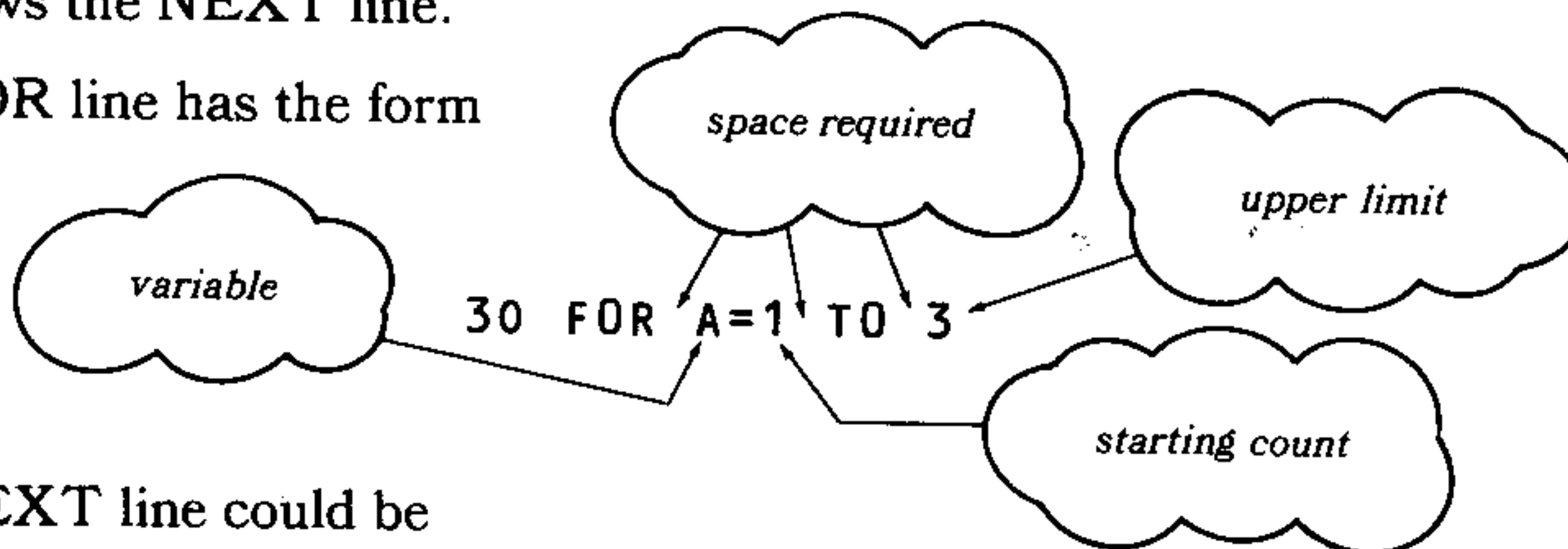
These new features will help you increase your programming skills, building on those we've already discussed in previous chapters. They'll also prepare the way for even more exciting things to come.

The FOR-NEXT Statement

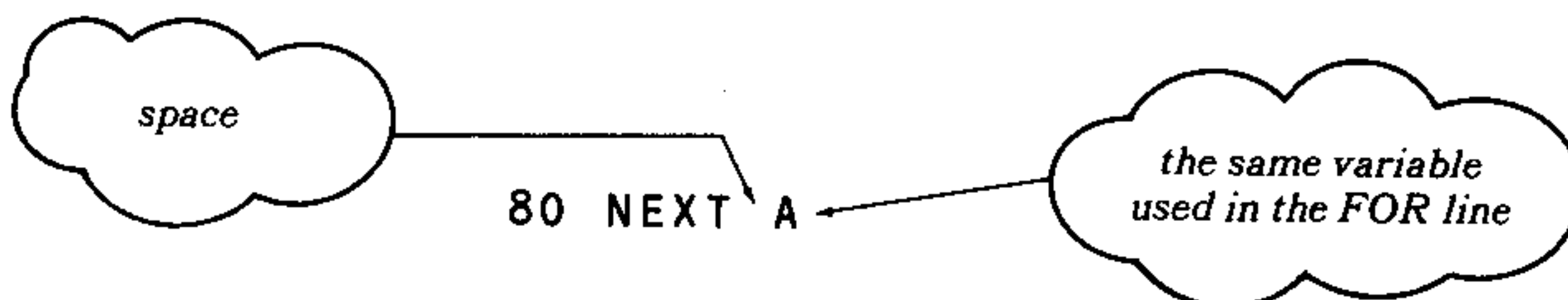
Chapter 2 presented several examples of the GO TO loop, which repeats a set of statements indefinitely – or until you press **CLEAR** to stop the program. The FOR-NEXT statement also creates a loop, but it's different from GO TO in two important ways:

1. The FOR-NEXT statement is actually a pair of lines in the program, the FOR line and the NEXT line, each with its own line number.
2. You control the number of times the loop is performed. After the loop has been "executed" the number of times you specify, the program moves on to the line that follows the NEXT line.

The FOR line has the form



The NEXT line could be



These two lines would cause the portion of the program between the FOR and NEXT lines to be performed three times. In this example the starting value of A is 1; after each pass through the loop, A is increased by 1. Its value is then tested against the upper limit (3, in this example). After the third pass through the loop, A is equal to 4, so the program "exits" (or leaves) the loop to the line following line 80.

To help you see the differences between GO TO and FOR-NEXT more clearly, let's compare two similar programs, one with a GO TO loop and one with a FOR-NEXT loop.

A GO TO Loop

Type NEW, press **ENTER**, and then enter this program:

```
10 CALL CLEAR
20 LET A=1
30 PRINT "A=";A
40 LET A=A+1
50 GO TO 30
```

Before you run the program, think for a few minutes about what it will do. First, the initial value of the variable A will be set to 1. Then, the computer will print out the current value of A. Finally, the value of A will be increased by 1, and the program will loop back to line 30. It will go on with this procedure until you press **CLEAR**.

Ready to run the program? Type RUN and press **ENTER** to see it in action. When you're ready to stop it, press **CLEAR**.

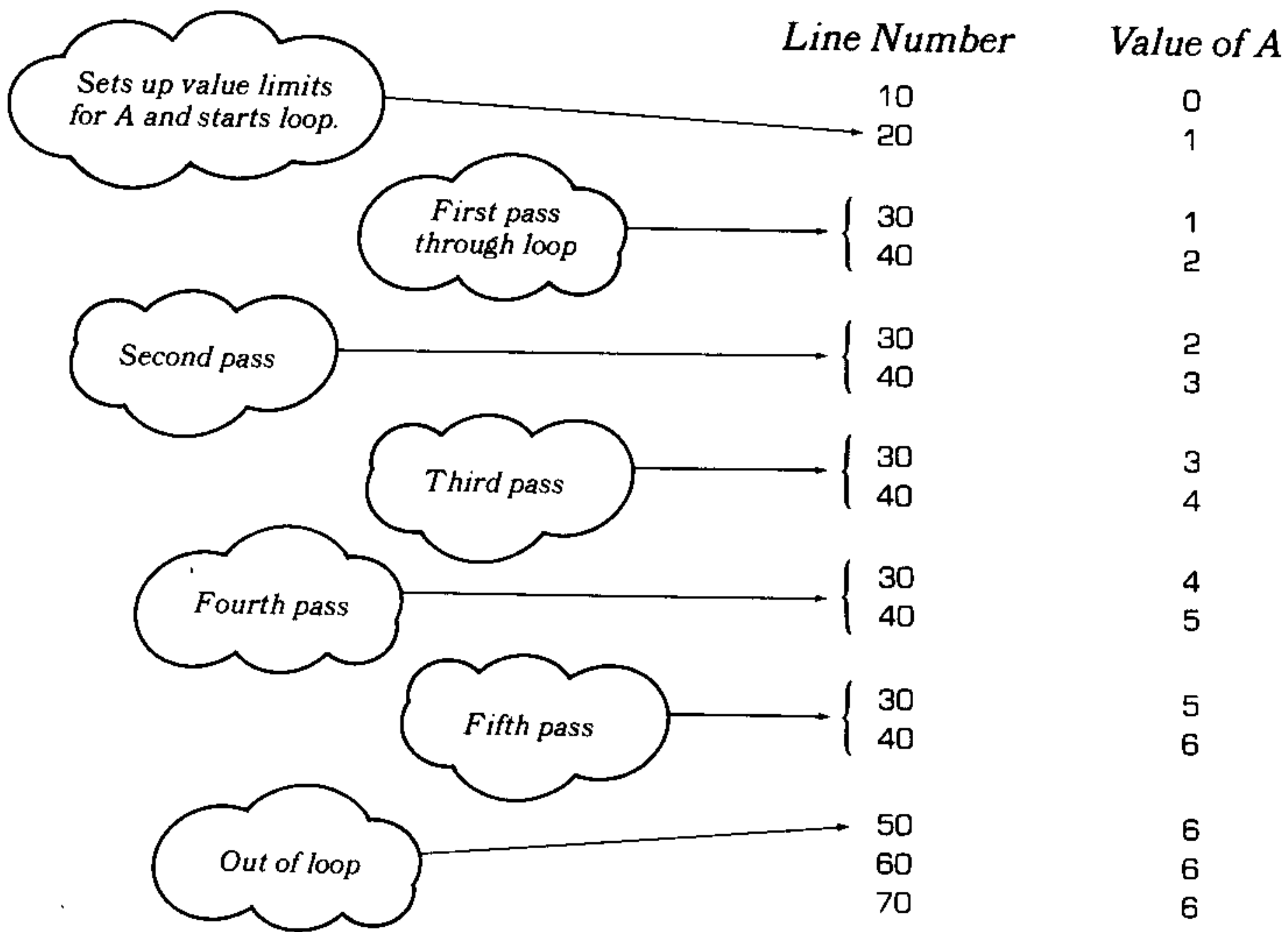
A FOR-NEXT Loop

Now let's examine a similar "counting" program with a FOR-NEXT loop. Type NEW and press **ENTER** to erase the first program. Then type these lines:

```
10 CALL CLEAR
20 FOR A=1 TO 5
30 PRINT "A=";A
40 NEXT A
50 PRINT "OUT OF LOOP"
60 PRINT "A=";A
70 END
```

Think about the way this program will be performed. The value of A will start at 1 and will be increased by 1 each time the program completes line 40. As soon as the value of A is *greater than* 5, the program will exit the loop and continue with line 50. If we listed the lines in their order of performance, along with the increasing values of A, this is what we'd have:

3



Run the program, and the screen should look like this:

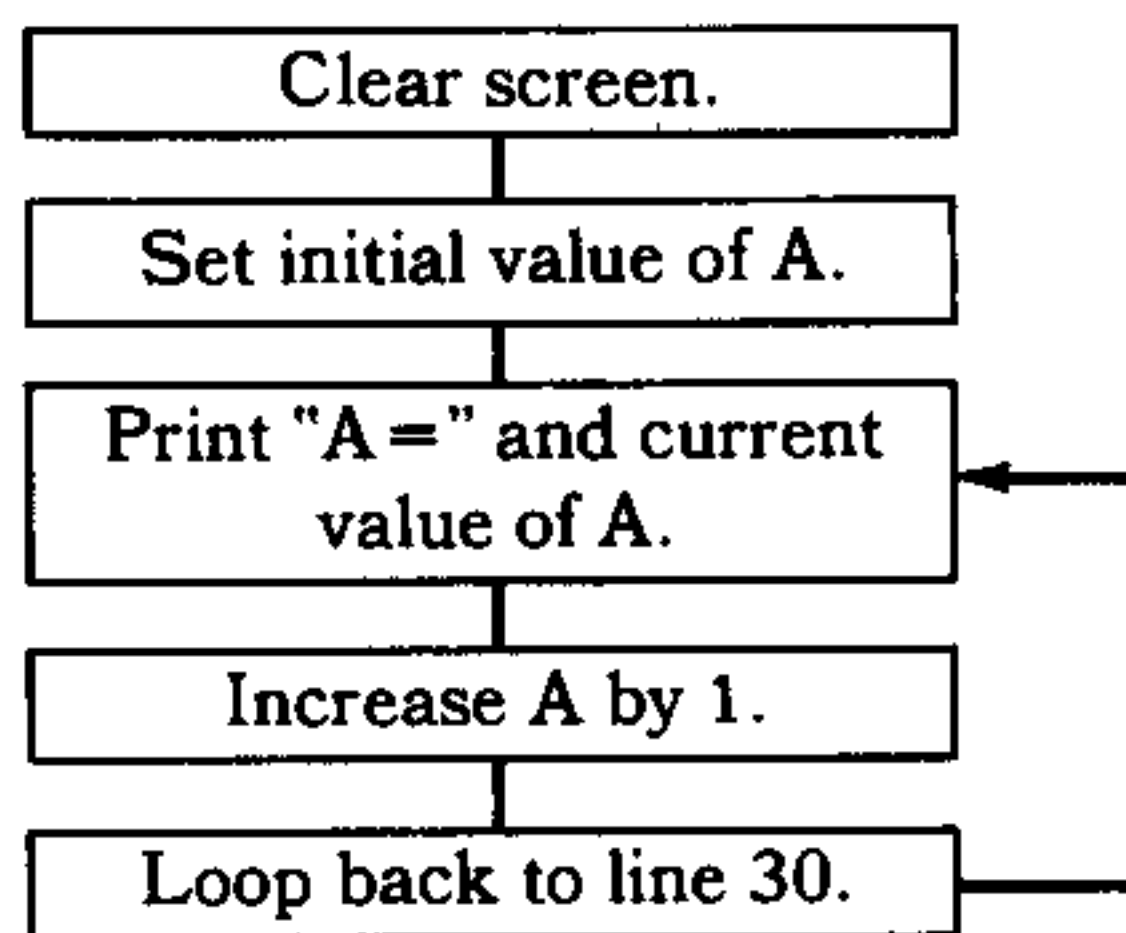
```
A= 1
A= 2
A= 3
A= 4
A= 5
OUT OF LOOP
A= 6

** DONE **

>□
```

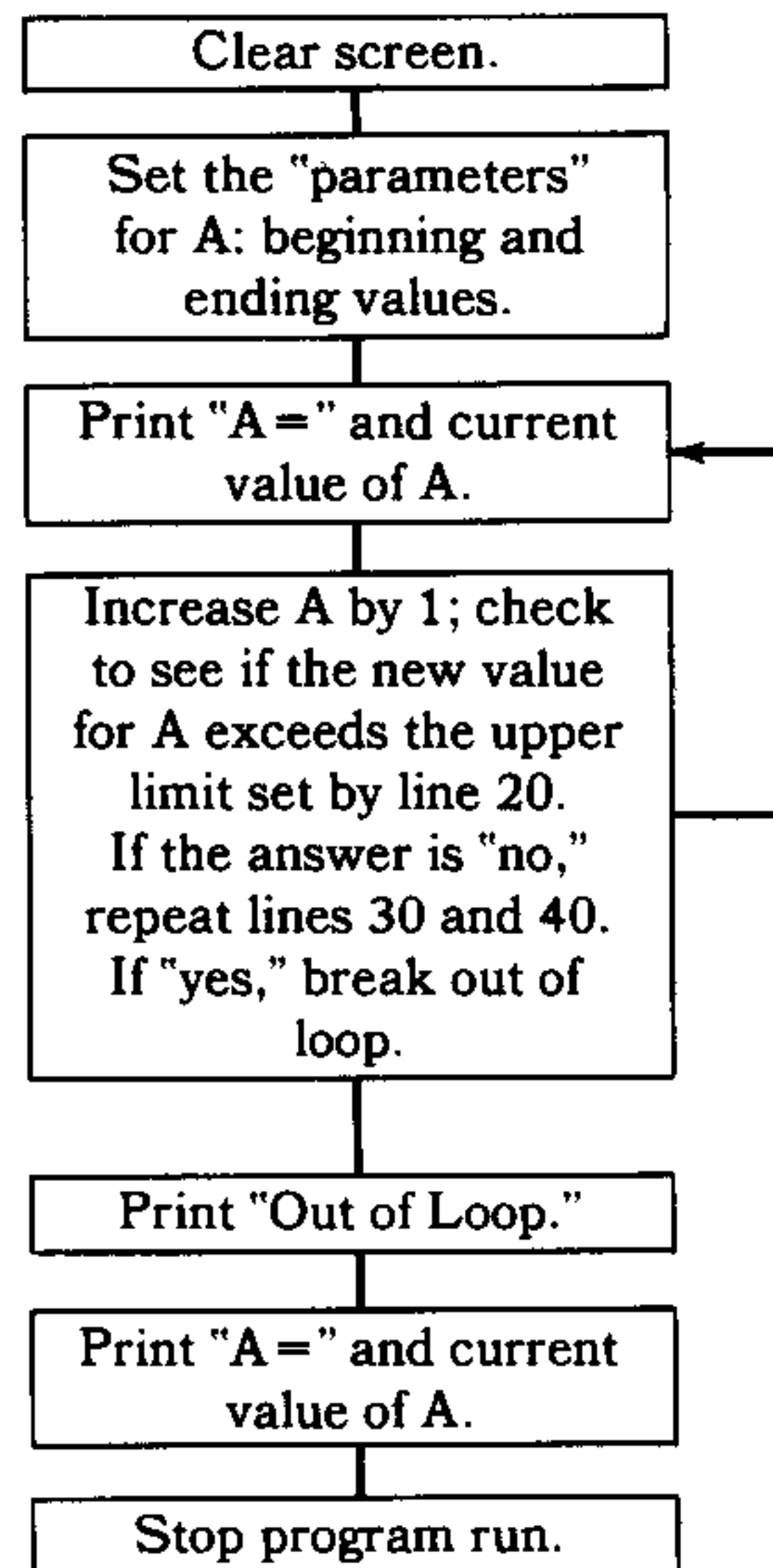
The following flowcharts illustrate the differences in the two programs.

GO TO Program



(Loop continues until you stop the program by pressing **CLEAR**.)

FOR-NEXT Program



In Chapter 2 we also used the GO TO statement in a CALL COLOR program to create a *delay loop*:

```
40 GO TO 40
```

This line caused the program to "idle" and hold the color design on the screen until you pressed **CLEAR**. Without some sort of delay loop, the color we used in the program would have blinked on the screen only for an instant before the program stopped and the screen returned to its normal Immediate Mode colors.

We can also use the FOR-NEXT statement to build a controlled time delay into a program. Consider this example:

```
20 FOR A=1 TO 1000
30 NEXT A
```

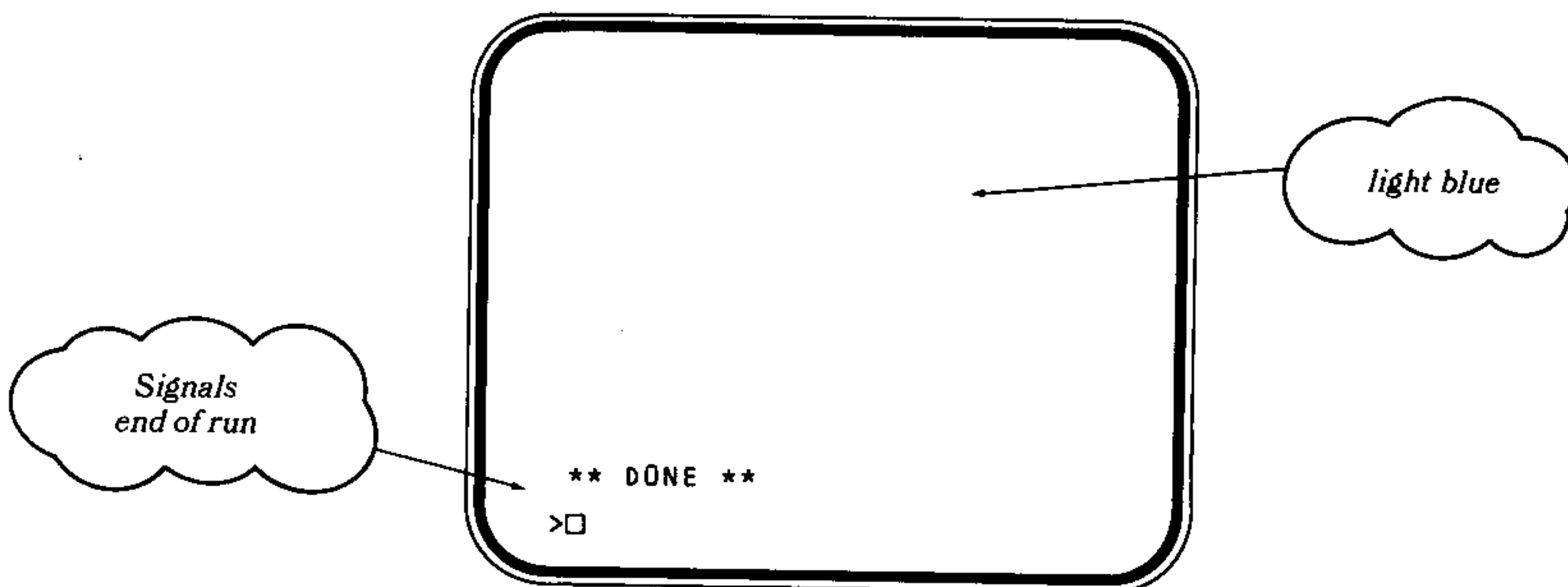
Better still, let's try it! Type NEW, press **ENTER**, and then type in the following program:

3

```
TI BASIC READY

>10 CALL CLEAR
>20 FOR A=1 TO 1000
>30 NEXT A
>40 END
>□
```

Now run the program. What happens on the screen? Not much, really; the screen changes to a light green, and the cursor disappears. After a short time delay (while the computer "counts" from 1 to 1000), the screen changes back to cyan (a light blue) and the cursor reappears:



Although no other lines are being executed between the `FOR` and `NEXT` lines, time passes while the computer counts the number of loops; in this example from 1 to 1000. The following program utilizes a `FOR-NEXT` time-delay loop in a `CALL COLOR` program.

CALL COLOR with a FOR-NEXT Loop

Clear the previous program (type `NEW`; press **ENTER**), and enter this program:

```
10 CALL CLEAR
20 CALL COLOR(2,7,7)
30 CALL HCHAR(12,3,42,28)
40 FOR B=1 TO 1000
50 NEXT B
60 END
```

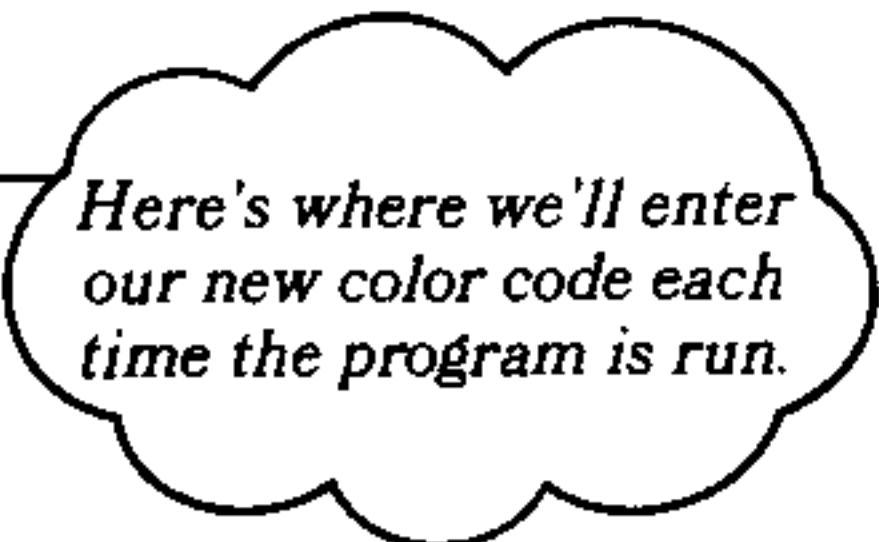
Color codes for foreground and background are the same: dark red

This program will print a row of asterisks on the screen. However, since the *foreground* color (the color of the asterisks) and the *background* color are both dark red, the screen will show a solid horizontal bar of dark red. The red asterisks blend into the red backgrounds.

Now run the program. Does the color bar stay on the screen long enough for you to observe it carefully? If not, change line 40 to increase the time delay (1 to 2000, for example).

Suppose we want to see a bar of a different color? We could retype line 20, inserting a new color code for the foreground and background colors. But there's an easier way to edit the program so that we won't have to retype line 20 every time we want to change colors. Type these lines:

```
15 INPUT A
20 CALL COLOR(2,A,A)
60 GO TO 10
```



Here's where we'll enter
our new color code each
time the program is run.

Well, well! A GO TO loop and a FOR-NEXT loop in the same program! Run the program, and see how it works. Remember, when you see the question mark on the screen, the program is waiting for you to "input" a color code from 1 through 16. If you enter a number that is outside this range, you'll see this error message on the screen:

```
* BAD VALUE IN 20
```

(Remember, also, that color 1 is transparent, and color 4 is the screen color in the Run Mode, so you won't be able to see these bars on screen.)

Experiment now with the color codes, and change the time delay in line 40 if you want to make the bar stay on the screen longer or disappear faster.

Experiment!

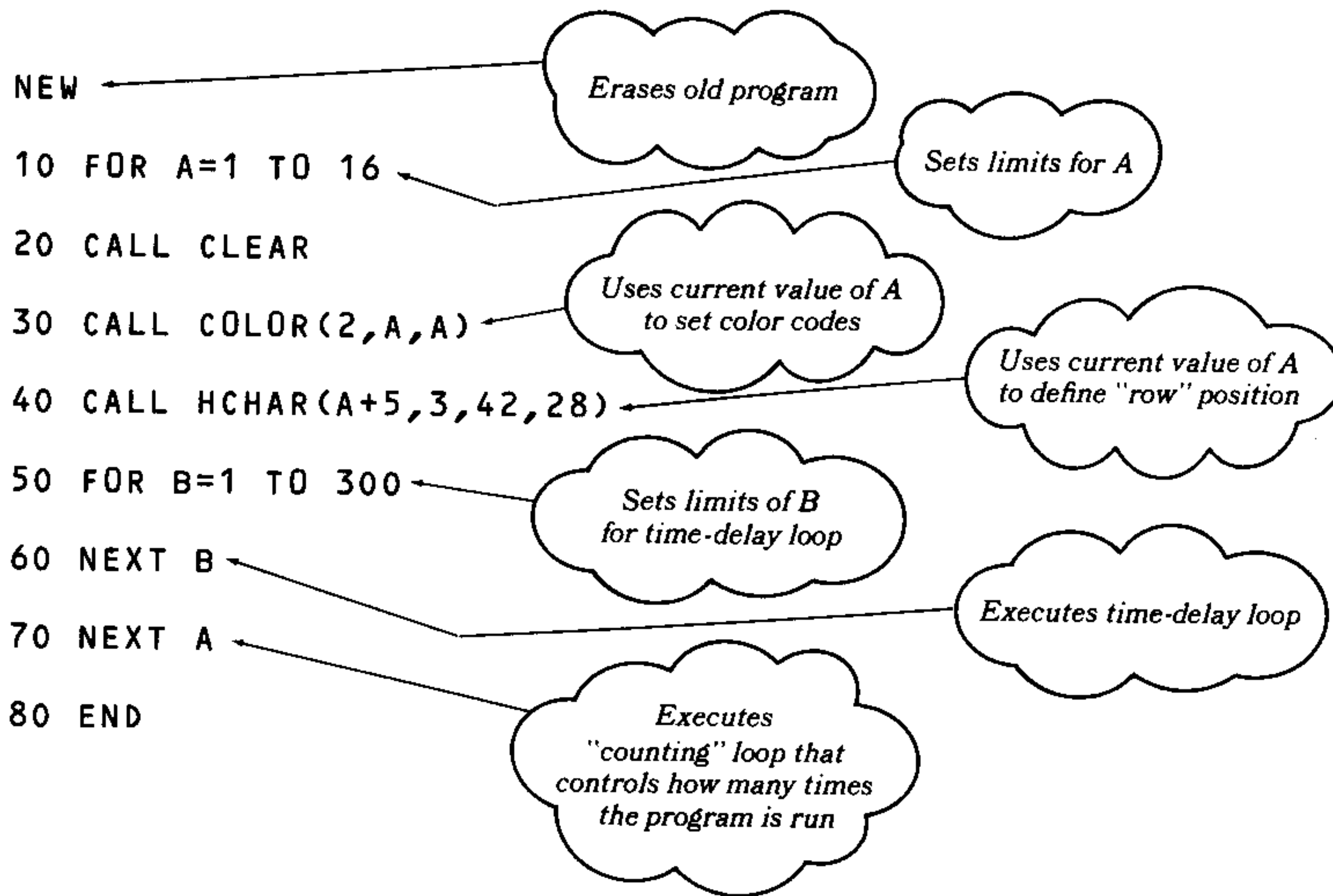
Here's a challenge for you! Can you change the program above to make a single small square of color appear on the screen, instead of a bar? (*Hint:* See Chapter 1, pp. 20-22, review using HCHAR or VCHAR to display a single character.)

"Nested" FOR-NEXT Loops

You've just seen that we can use both a FOR-NEXT loop and a GO TO loop in the same program. It's also possible for us to use more than one FOR-NEXT loop — one inside another — in a program. We call these *nested loops*.

As an example, let's experiment a bit with a program very similar to the one you've just completed. But this time, we'll get a little fancier. We'll make the bar "walk" down the screen, so that it appears in a different position each time the color changes. Type these lines:

3



Notice that one loop is wholly contained within the other loop. That's why these are called "nested" loops: one is nested inside another.

This program gets a lot of mileage out of the variable A. We're using it to control the number of times the program is repeated (a *loop counter*), to define the color codes for foreground and background, *and* to determine the row position of the color bar.

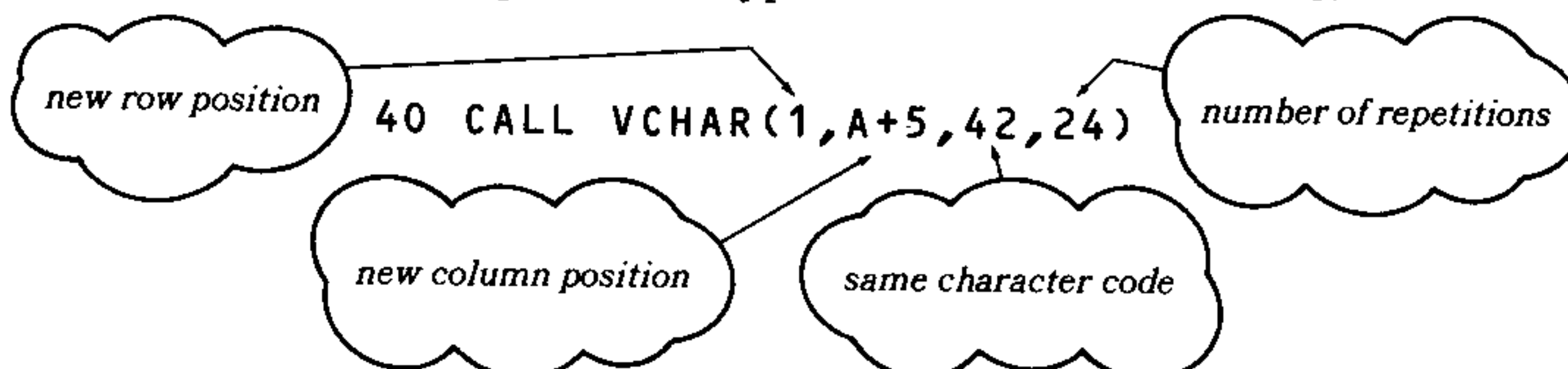
(Before you run the program, remember that color 1 is transparent and color 4 is the Run Mode screen color. You won't be able to see these bars.)

Now run the program. Does the bar appear to move down the screen? What happens if you shorten the time-delay loop? Try changing line 50 to

```
50 FOR B=1 TO 100
```

and run the program again.

Another interesting change would be to make the bar vertical instead of horizontal. We can do this easily by changing line 40. Type and enter this new line:



When you run the program this time, the bar will be vertical and will move across the screen from left to right.

Now let's examine another program with nested FOR-NEXT loops. The following program displays sixty-four of the alphanumeric characters, codes 32 through 95. (See *Appendix B* for a list of the character codes.) Enter these lines:

```
NEW
10 CALL CLEAR
20 LET CHAR=32
30 FOR ROW=7 TO 14
40 FOR COLUMN=13 TO 20
50 CALL HCHAR(ROW,COLUMN,CHAR)
60 CHAR=CHAR+1
70 NEXT COLUMN
80 NEXT ROW
90 END
```

The diagram consists of four cloud-shaped callouts with arrows pointing to specific lines of code:

- A cloud pointing to line 20: "Starting value for variable CHAR (character code)"
- A cloud pointing to line 30: "Beginning and ending values for row number"
- A cloud pointing to line 40: "Beginning and ending values for column number"
- A cloud pointing to line 60: "Increases numeric code for CHAR by 1"

The program will look like this on the screen:

```
TI BASIC READY
>10 CALL CLEAR
>20 CHAR=32
>30 FOR ROW=7 TO 14
>40 FOR COLUMN=13 TO 20
>50 CALL HCHAR(ROW,COLUMN,CHAR)
>60 CHAR=CHAR+1
>70 NEXT COLUMN
>80 NEXT ROW
>90 END
>□
```

There are several things we'd like to point out about this program. First, FOR-NEXT loops *do not* have to start counting at 1. They can begin with whatever numeric value you need to use. Second, the nested loop (FOR COLUMN-NEXT COLUMN) is not just a time-delay loop. It actually controls a part of the program repetition.

Finally, line 50 is called a *wrap-around* line. It has more than 28 characters, so part of it prints on another line on the screen. This is an important point: *program lines can be more than one screen-line long*. In fact, a program line, in general, can be up to four screen lines (112 characters) in length. (The exception is the DATA statement. See the "BASIC Reference" section of the *User's Reference Guide* for an explanation.) Notice that wrap-around lines (that is, the second, third, or fourth screen lines of a program line) are *not* preceded by the small prompting symbol.

3

Run the program, and the sixty-four characters will be printed in nice, neat rows on the screen:

```
!"#$%&'
()*+,-./
01234567
89:;<=>?
@ABCDEFGHI
HIJKLMNO
PQRSTUVW
XYZ[\]^_

** DONE **

>□
```

Hold on! There are only sixty-three characters on the screen! What happened to the other one? Well, there *are* actually sixty-four. Look at the top line, and notice that it appears to be indented one space. That's because *character 32 is a space*. Even though a space doesn't print anything on the screen, it does occupy room on a line, and it *is* a character, as far as the computer is concerned.

Experiment!

Let's add color to the character program above! Enter these lines:

```
22 FOR I=1 TO 8
24 CALL COLOR(I,7,15)
26 NEXT I
```

Try other color combinations until you find your favorite.

Error Conditions with FOR-NEXT

We mentioned earlier that a nested loop *must* be completely contained within another loop. If your program included lines like these,

```
20 FOR A=1 TO 6
30 FOR X=5 TO 10
```

...

```
80 NEXT A
90 NEXT X
```

Should be nested
within the "A" loop.

the computer would stop the program and give you this error message:

```
* CAN'T DO THAT IN 90
```

The computer can't go back inside the completed "A" loop to pick up the beginning of the "X" loop.

Another possible error condition with FOR-NEXT statements is accidentally omitting either the FOR line or the NEXT line. For example, if you attempted to run this program:

```
10 FOR A=1 TO 5
20 PRINT A
30 END
```

the computer would respond with

```
* FOR-NEXT ERROR
```

If you encounter an error message, just list the program (type LIST and press **ENTER**), identify the error, and correct the problem line or lines.

We've given you quite a lot of information now about FOR-NEXT loops, so it's probably time for a change of pace. Let's review a bit of the PRINT material we covered in Chapter 1.

Plain and Fancy PRINTing

While using the PRINT statement in the Immediate Mode, we saw that a difference in spacing occurred when we used a comma or a semicolon to separate numeric values in a PRINT statement. Let's take another look at this.

Spacing with Commas

Try each of the following examples. (In each, we'll assume that the screen has been cleared by typing CALL CLEAR and pressing **ENTER**.)

Type this and
press ENTER

```
>PRINT 1,2
1           2
>□
```

```
>PRINT 1,2,3,4,5,6
1           2
3           4
5           6
>□
```

3

So far we have used only small positive integers. Let's try some simple negative numbers.

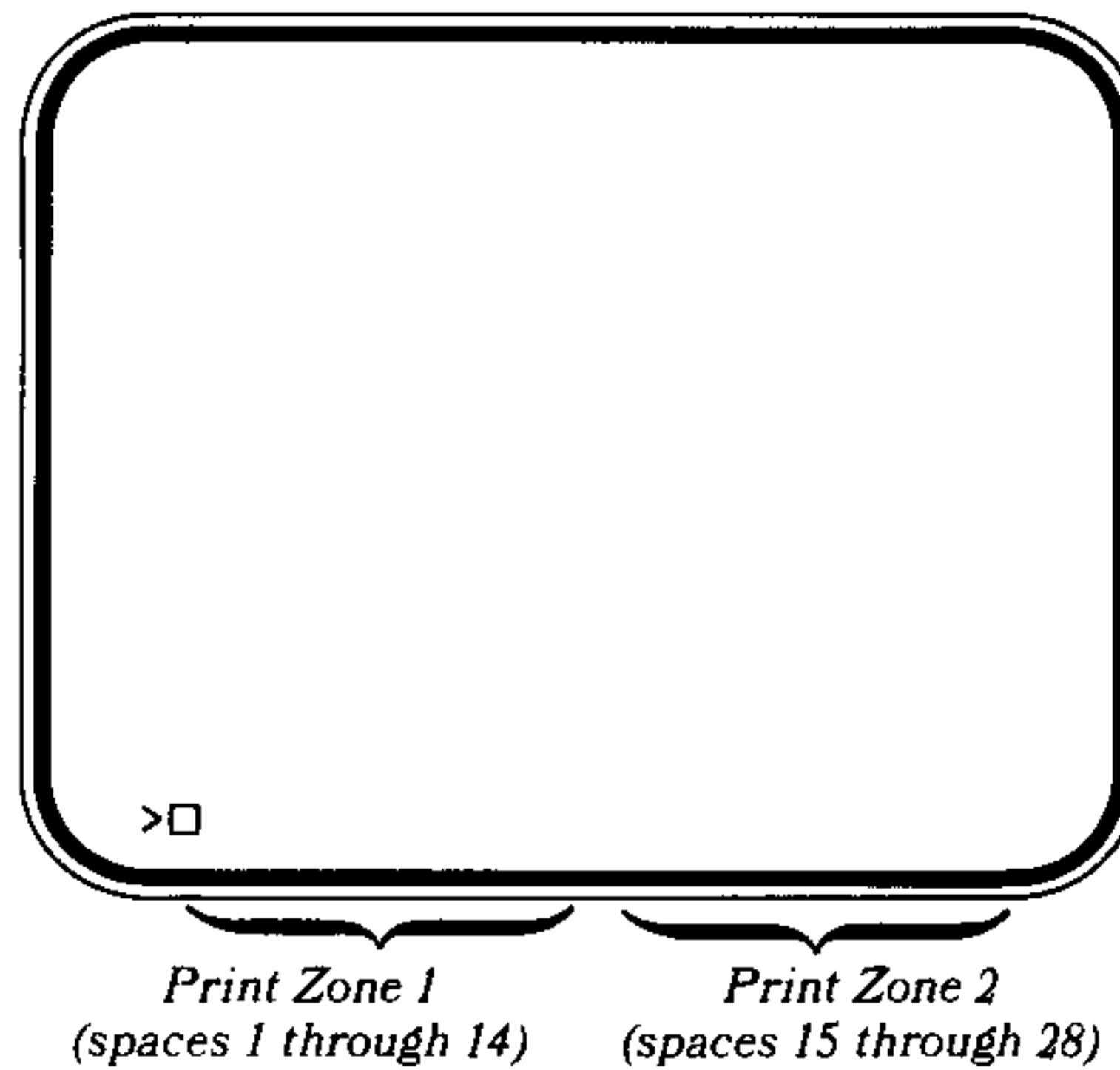
```
>PRINT -1,-2
-1          -2
>□
```

Now let's try a combination of positive and negative numbers.

```
PRINT 1,2,-3,-4
1          2
-3         -4
>□
```

Note that the computer always leaves a space preceding the number for the *sign* of the number. For positive numbers, the plus sign (+) is assumed and is not printed on the screen. For negative numbers, the computer prints a minus sign (–) before the number.

We mentioned in Chapter 1 that there are two *print zones* on the screen line. Each print zone has room for fourteen characters per line.



When you use a comma to separate numeric values or variables in a PRINT statement, the computer is instructed to print only one value in each zone. Therefore, since there are only two print zones on each line, the computer can print a maximum of two values per screen line. If the PRINT statement has more than two items, the computer simply continues on the next screen line until all the items have been printed.

Now let's try some examples with *string variables*, using commas as "separators." (See page 35 of Chapter 2 if you need to review string variables.)

```
>LET A$="ZONE 1"  
>LET B$="ZONE 2"  
>PRINT A$,B$  
ZONE 1      ZONE 2  
>
```

The *strings* (the letters and numbers within the quotation marks) are also printed in different zones on the screen when a comma is used to separate the string variables.

3

Try this example:

```
>LET A$="ONE"  
>LET B$="TWO"  
>LET C$="THREE"  
>LET D$="FOUR"  
>PRINT A$,B$,C$,D$  
ONE TWO  
THREE FOUR  
>□
```

(Note that, for strings, the computer does not leave a preceding space.)

Spacing with Semicolons

Now let's look at semicolon spacing. Try these examples:

```
>PRINT 1;2  
1 2  
>□
```

Aha! The numbers are much closer together.

```
>PRINT 1;2;3  
1 2 3  
>□
```

```
>PRINT 1;2;-3;-4;5;-6;7
 1  2 -3 -4  5 -6  7
>
```

The semicolon instructs the computer *not to leave any spaces* between the values or variables in the PRINT statement. Then why do we see spaces between the numbers on the screen? Two reasons! First, remember that each number is preceded by a space for its sign. Second, every number is followed by a *trailing space*. (The trailing space is there to guarantee a space between all numbers, even negative ones. The way numbers are displayed is discussed in detail in *Appendix D*.)

If the semicolon tells the computer to leave no spaces between variables in a PRINT statement, what happens when we use string variables, rather than numeric? Let's try some examples.

```
>LET A$="HI THERE!"
>LET B$="HOW ARE YOU?"
>PRINT A$;B$
HI THERE!HOW ARE YOU?
>
```

The two strings are run together. If we want a space to appear between them, then, we must include the space inside one of the sets of quotation marks! For example, let's change A\$. Type

```
LET A$="HI THERE! "  
PRINT A$;B$
```



3

```
>LET A$="HI THERE!"
>LET B$="HOW ARE YOU?"
>PRINT A$;B$
HI THERE!HOW ARE YOU?
>LET A$="HI THERE! "
>PRINT A$;B$
HI THERE! HOW ARE YOU?
>□
```

First example

Second example

Spacing with Colons

There is a third "separator" that can be used: the colon. The colon instructs the computer to print the next item at the beginning of the next line. It works the same way with both numeric and string variables. Enter these lines as an example:

```
LET A=-5
LET B$="HELLO"
LET C$="MY NAME IS ALPHA"
PRINT A:B$:C$
```

```
>LET A=-5
>LET B$="HELLO"
>LET C$="MY NAME IS ALPHA"
>PRINT A:B$:C$
-5
HELLO
MY NAME IS ALPHA
>□
```

To review for a moment, then, these are the three print separators we have used:

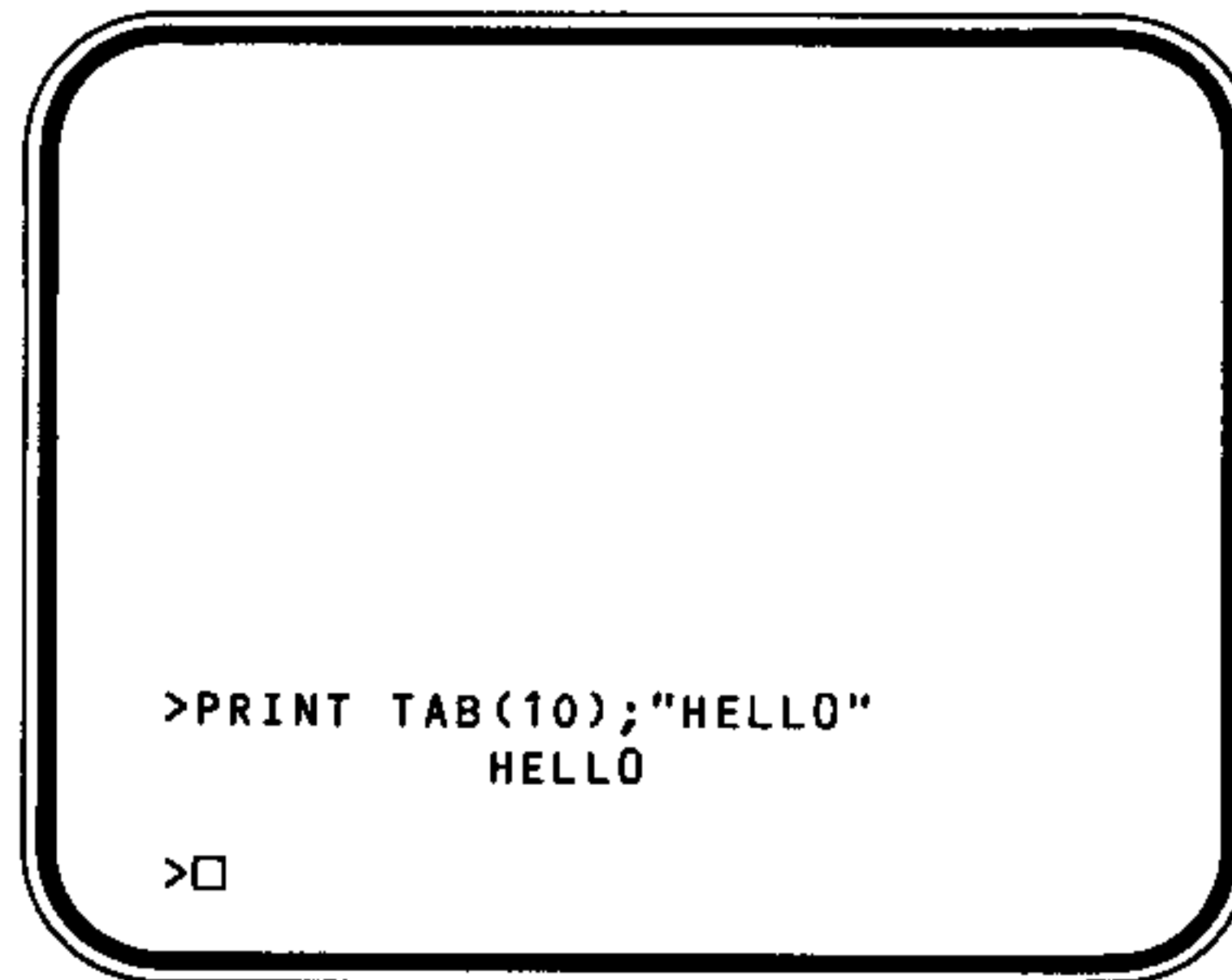
<i>Punctuation mark</i>	<i>Operation</i>
Comma	Prints values in different print zones; maximum of two items per line.
Semicolon	Leaves no spaces between items. (The spaces that appear between numbers are results of the built-in display format for numeric quantities.)
Colon	Prints next item on following line.

The TAB Function

Besides these separators there is another method you can use to control the printing on the screen. The TAB function operates very much like a typewriter TAB key:

```
PRINT TAB(10);"HELLO"
```

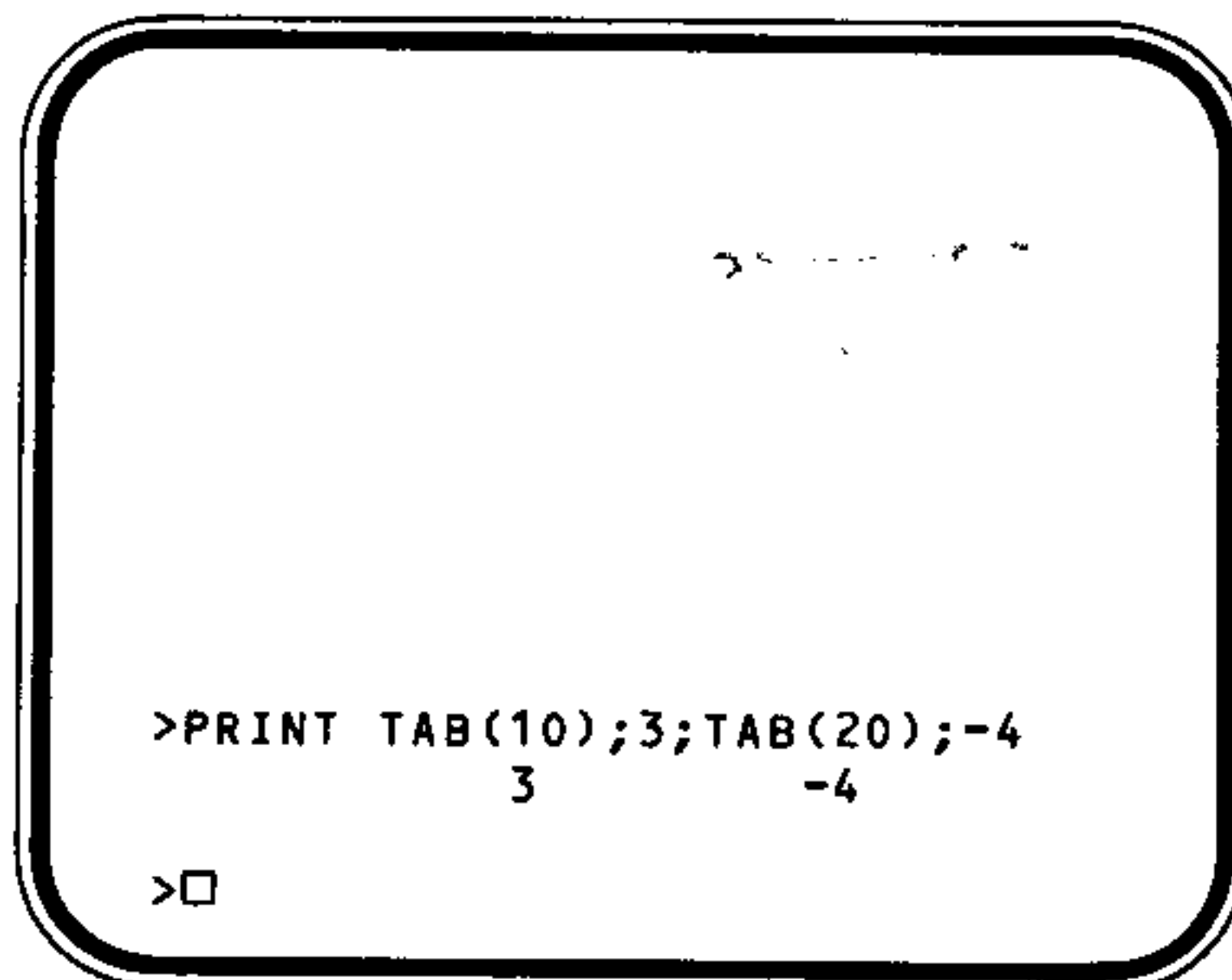
The statement would instruct the computer to begin printing the word HELLO in the tenth column on the screen.



```
>PRINT TAB(10);"HELLO"  
HELLO  
>□
```

Notice that the "print line" on the screen has 28 columns or character positions (unlike the "graphics line," which has a 32-column "grid"). Thus the first position on the print line counts as column 1. This is where the "P" appears in the word "PRINT" on the previous screen. The last print position on the line is column 28.

You can also use the TAB function more than once in a print statement:



```
>PRINT TAB(10);3;TAB(20);-4  
3 -4  
>□
```

Notice that the first number, 3, is actually printed in column 11, because the preceding or "leading" space (reserved for the sign of the number) occupies column 10, just as the minus sign of the second number occupies column 20.

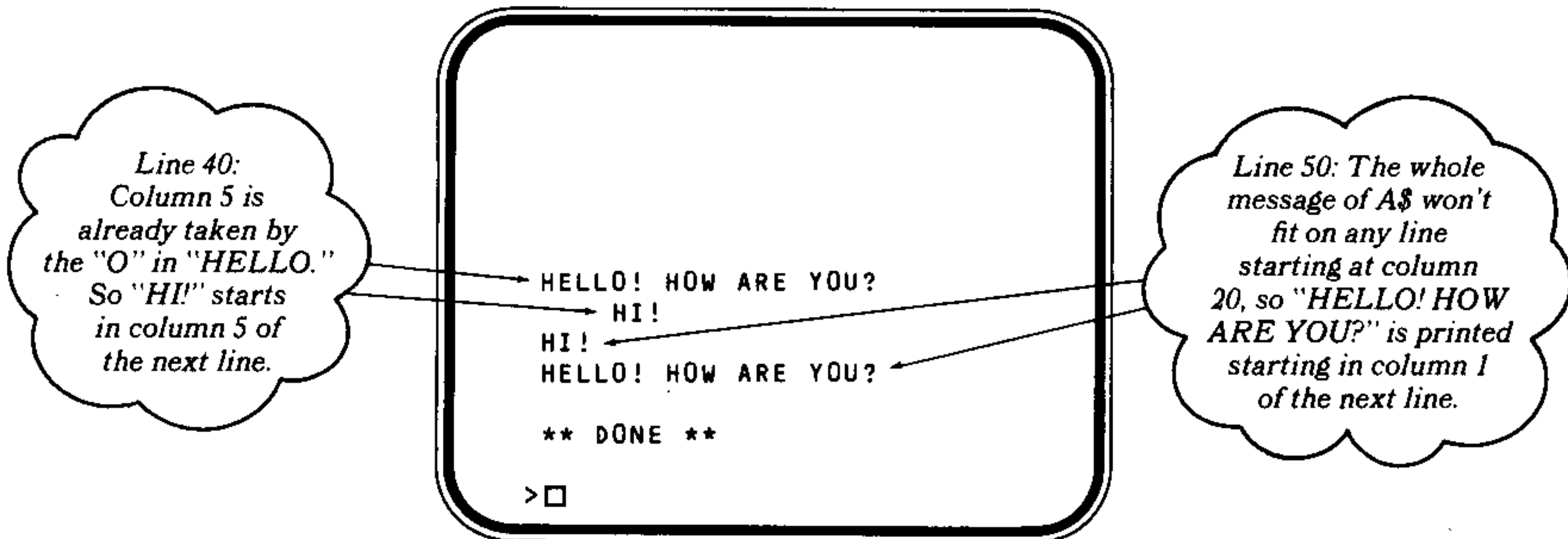
The TAB function always starts counting in column 1 (the leftmost print position on the line), regardless of where or how many times it appears in the PRINT statement. In the example above, the second number, -4, was printed starting in the twentieth column on the print line, *not* twenty spaces from the position in which the first number, 3, was printed.

3

What happens, then, if we indicate a column that is already occupied by another message, or if there isn't enough room left on the line to print the message positioned by a TAB? Enter this short program to find the answer:

```
NEW
10 CALL CLEAR
20 LET A$="HELLO! HOW ARE YOU?"
30 LET B$="HI!"
40 PRINT A$;TAB(5);B$
50 PRINT B$;TAB(20);A$
60 END
```

Now run the program:



Notice that separators (semicolons) are also used in the PRINT statement above. Let's try a program to help explore the use of the TAB function and separators. Imagine for a few minutes that you are a loyal football fan, and it's time for the big game of the season. Since you are also a computer fan, you want to program your computer to cheer the team on to victory! So you enter this program:

```
NEW
10 CALL CLEAR
20 LET A$="GO"
30 PRINT TAB(13);A$::TAB(12);"TEAM"::TAB(13);A$;"!"
40 FOR Z=1 TO 10
50 PRINT
60 NEXT Z
70 FOR Z=1 TO 600
80 NEXT Z
90 GO TO 10
```

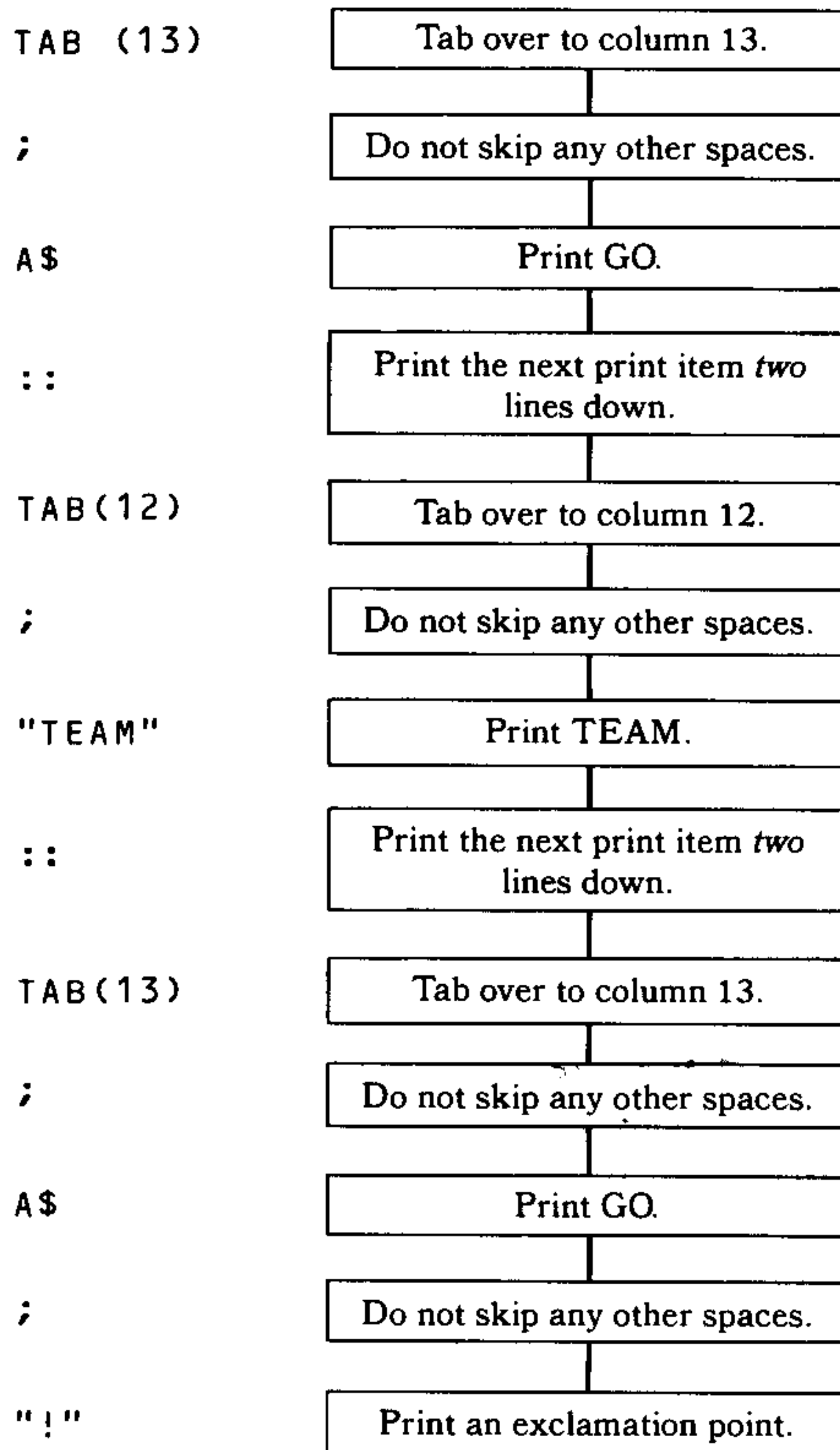
two colons!

Prints an "empty" line.

Note that this is a wrap-around line.

Before you run the program, let's analyze it. Line 10, of course, clears the display screen. Line 20 defines the string variable A\$ as GO.

Line 30 is a very, very hard-working line. It might be helpful if we drew a flowchart to describe what's going on here.

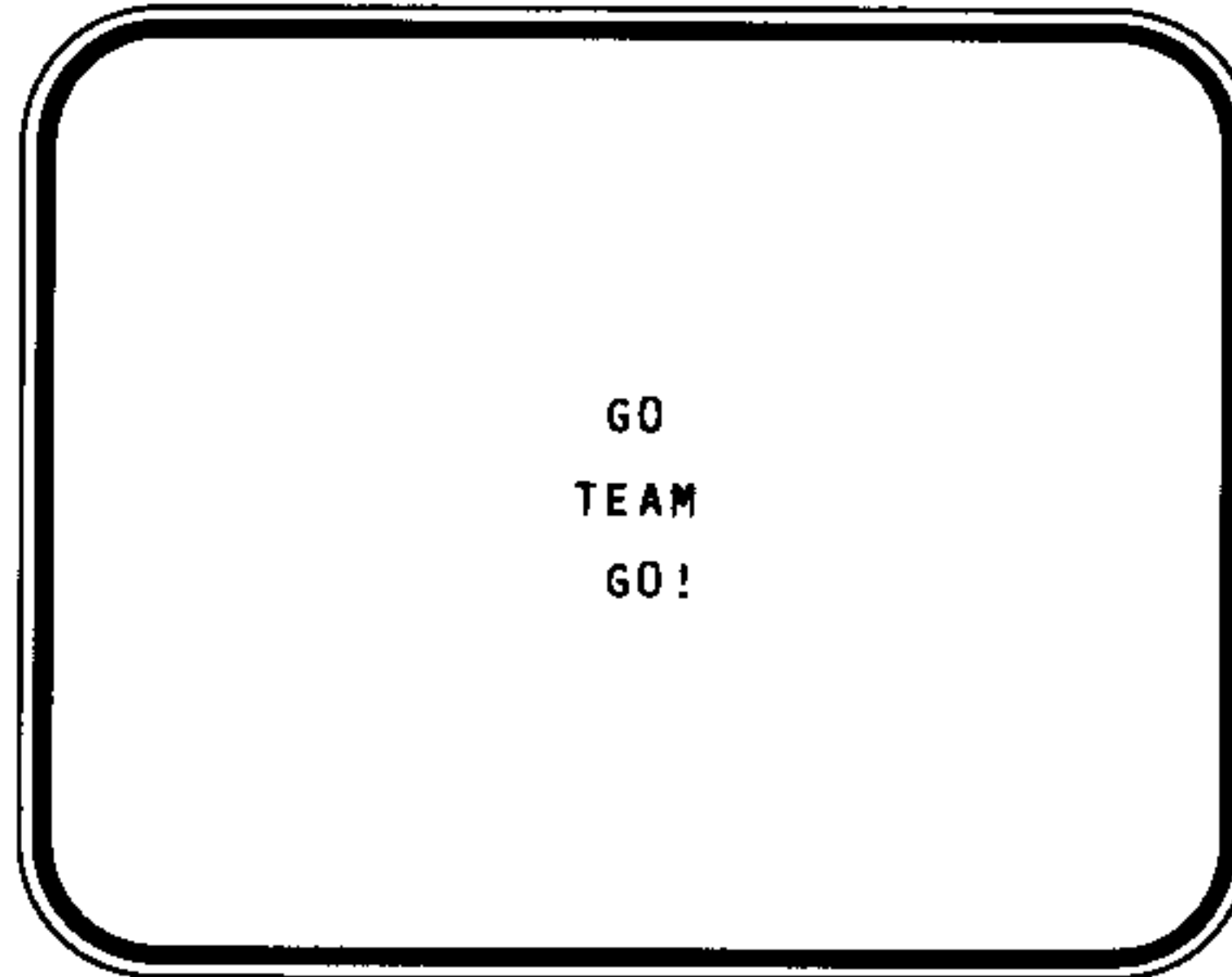


(You'll have to admit that's a lot of information to pack into one program line, even if it is more than one screen line long!)

The FOR-NEXT loop in lines 40 through 60 will print ten "empty" lines, to position your message in the middle of the screen. Next, lines 70 and 80 form a time-delay loop. Then line 90 instructs the computer to go back to line 10 and start all over again.

3

Run the program now, and watch your computer cheer!



The words come on at the bottom of the screen, one at a time, and scroll up to the center. Then the screen clears, and the whole process is repeated until you stop the program by pressing **CLEAR**.

By now, your team has probably won the game, and you're ready to try some other messages and formats. Experiment for a while with **TAB** and the three separators in different **PRINT** statements before we go on to discuss the arithmetic operations of the computer.

Arithmetic Power

You've been introduced before to the arithmetic powers of your computer, but it's time now to take a more detailed "tour" of some of its mathematical capabilities. For example, what is the answer to this problem:

$$4 + 6 * 5 = ?$$

*Remember, * means "multiply" to the computer.*

Let's say, for example, that the answer represents an amount of money you owe a friend. Your friend argues that you owe him \$50, because

$$4 + 6 = 10, \text{ and} \\ 10 \times 5 = 50.$$

You, however, don't agree. You say you only owe \$34, because

$$6 \times 5 = 30 \\ 4 + 30 = 34$$

Who is right? Why not ask your computer?

Type **PRINT 4+6*5**
and press **ENTER**.

The answer is 34. How about that! You win!

Order of Operations

There is a commonly accepted order in which arithmetic operations are performed, and your computer performs calculations in that order. In any problem involving addition, subtraction, multiplication, and division, the arithmetic operations will be completed in this way:

Multiplications and divisions are performed
before additions and subtractions.

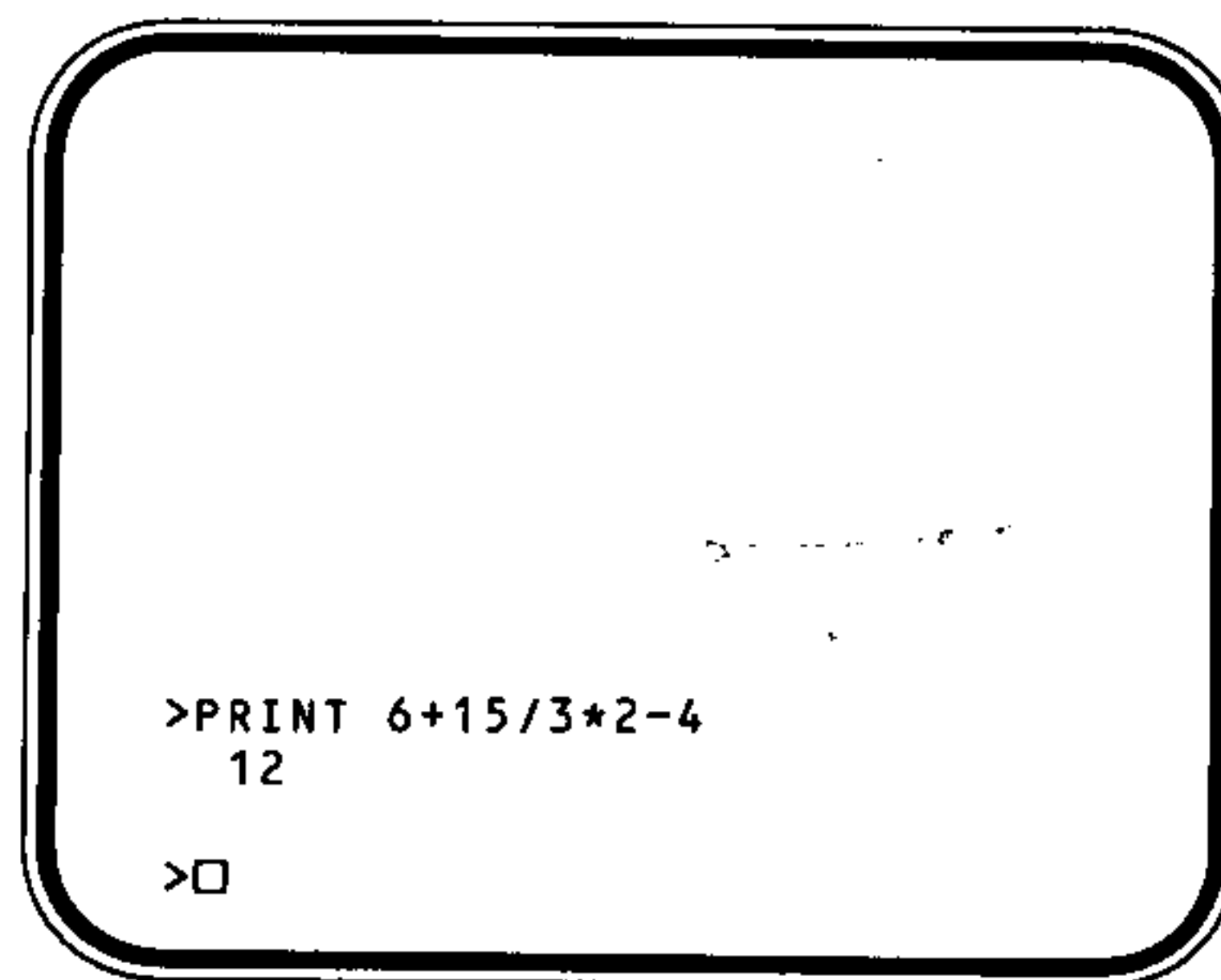
This is the method your computer used to solve the previous example. It first multiplied $6*5$ and *then* added the result to 4, giving you a final answer of 34. Now try this example:

```
PRINT 6+15/3*2-4
```

Before you press **ENTER**, let's think about the way the computer will evaluate this problem. Scanning the problem from left to right, the computer will solve it in this order:

```
15/3=5  
5*2=10  
6+10=16  
16-4=12
```

Your answer, then, should be 12. Press **ENTER** now, and see the result:



Using Parentheses

Suppose, however, that we want the computer to solve the last problem like this:

- (1) Add 6 and 15.
- (2) Divide the result by 3.
- (3) Multiply that result by 2.
- (4) Subtract 4, giving a final result of 10.

We can change the built-in computational order by using *parentheses*. Try this:

```
PRINT (6+15)/3*2-4
```

Press **ENTER**.

3

The answer, 10, is displayed on the screen, because the computer *has completed the computation inside the parentheses first*. So our new order of operations becomes:

- (1) Complete everything inside parentheses.
- (2) Complete multiplication and division.
- (3) Complete addition and subtraction.

Now try this example:

```
PRINT 8/2*4/2
```

The answer is 8, because

$$\begin{aligned}8/2 &= 4 \\ 4*4 &= 16 \\ 16/2 &= 8\end{aligned}$$

But suppose we entered the problem with parentheses, like this:

```
PRINT 8/(2*4)/2
```

This time, we get a result of .5, because the expression within the parentheses has been solved first:

$$\begin{aligned}2*4 &= 8 \\ 8/8 &= 1 \\ 1/2 &= .5\end{aligned}$$

Here's a slightly harder problem to try:

```
PRINT 274+10/2*100-30
```

If we enter the problem just like this, we obtain an answer of 744 because

$$\begin{aligned}10/2 &= 5 \\ 5*100 &= 500 \\ 274+500 &= 774 \\ 774-30 &= 744\end{aligned}$$

But by adding parentheses in different places we can get a variety of answers:

```
>PRINT (274+10)/2*(100-30)
9940
>PRINT (274+10)/(2*100)-30
-28.58
>PRINT (274+10/2)*100-30
27870
>□
```

Callout 1 (Left):

$$\begin{aligned}274+10 &= 284 \\ 2*100 &= 200 \\ 284/200 &= 1.42 \\ 1.42-30 &= -28.58\end{aligned}$$

Callout 2 (Right):

$$\begin{aligned}274+10 &= 284 \\ 284/2 &= 142 \\ 100-30 &= 70 \\ 142*70 &= 9940\end{aligned}$$

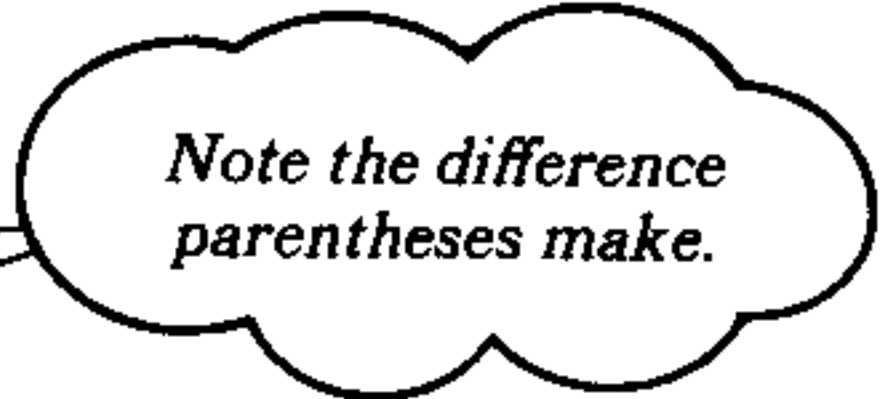
Callout 3 (Bottom Right):

$$\begin{aligned}10/2 &= 5 \\ 274+5 &= 279 \\ 279*100 &= 27900 \\ 27900-30 &= 27870\end{aligned}$$

Experiment!

Try the following for practice:

$38+6-4$
 $38+6-4*2$
 $(38+6-4)*2$
 $((38+6-4)*2)/(6+2)$



Note the difference
parentheses make.

Rearrange the parentheses in the last problem. How is the answer affected?

Scientific Notation

So far, all the examples we've tried have given results in a normal decimal display form. However, the computer displays very long numbers (more than ten digits) in a special way. Try this program:

```
NEW
10 CALL CLEAR
20 LET A=1000
30 FOR X=1 TO 5
40 PRINT A
50 LET A=A*100
60 NEXT X
70 END
```

When you run the program, the first four results are printed out in the normal form. The last result, however, looks like this:

1.E+11

We call this special form *scientific notation*. It's just the computer's way of handling numbers that won't fit into the normal ten-digit space allotted for numbers.

1.E+11 means 1×10^{11} or 100,000,000,000

As you can see, 1.E+11 represents a very large number!

You'll find a more detailed discussion of the mathematical capabilities and numerical displays of your computer in *Appendix D* (starting on page 127). Be sure to refer to this appendix when you want to explore the computational powers of the computer. For now, however, let's go on to another very useful feature, the INT function.

The INT Function

The INT function gets its name from the word *integer*, meaning a whole number, one that has no fractional part. Integers include zero and all of the positive and negative numbers that do not have any digits after the decimal point.

The best way to learn how the INT function works is by trying it. First, let's work a division problem that doesn't result in a whole number answer. Type

```
PRINT 16/3
```

and press **ENTER**. The answer is 5.333333333.

3

Now try this example:

```
PRINT INT(16/3)
```

Press **ENTER**.

new answer

```
>PRINT 16/3
  5.333333333
>PRINT INT(16/3)
  5
>
```

INT kept the whole number part of the answer and threw away the digits after the decimal point! Try another example:

```
PRINT INT(7/6)
```

$7/6 = 1.166666666$
 $INT\ 7/6 = 1$.

The answer is 1; all of the fractional part has been discarded.

How about a real-life problem? Let's say a salesclerk is giving \$1.37 in change to a customer. The customer wants as many quarters as possible. How many quarters can be given?

```
PRINT INT(1.37/.25)
```

The answer is 5. Five quarters can be given.

More than one INT function can be used in a PRINT statement. Here's an example:

```
>PRINT INT(1/3);INT(20/9)
  0  2
>
```

What would happen if you entered these values with the INT function: 8, 8.99, 8.34? Try them and see.

```
>PRINT INT(8)
8

>PRINT INT(8.99);INT(8.34)
8 8

>□
```

If you use INT with a whole number (integer), you just get the same number back. In the other two examples, no matter what digits are to the right of the decimal point, the INT function "truncates" or cuts off those digits – that is, it works this way for *positive* numbers. What happens with *negative* numbers?

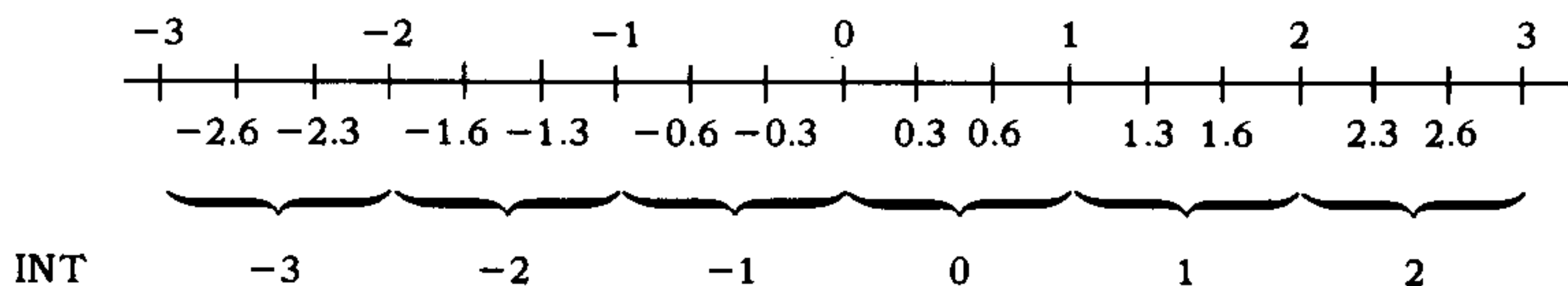
We'll use a program to explore INT and negative numbers. Enter these lines:

```
NEW
10 CALL CLEAR
20 FOR A=1 TO 7
30 PRINT -A/3,INT(-A/3)
40 NEXT A
50 END
```

Now run the program. The screen will show these results:

```
-.3333333333    -1
-.6666666666    -1
-1              -1
-1.3333333333   -2
-1.6666666666   -2
-2              -2
-2.3333333333   -3
```

So INT(X) – where X represents a number or a mathematical expression – computes the nearest integer *that is less than or equal to X*. Perhaps looking at a *number line* will help to explain.



3

As you see from the number line, when X has the value -0.3 , the nearest integer that is *less than or equal to* X is -1 .

One last feature associated with `INT` is very useful to know. It can appear on the right side of an equals sign in a `LET` statement. For example, try the next series of lines.

```
>LET A=INT(4/3)+2
>PRINT A
  3
>□
```

In the `LET` statement, `INT(4/3)` produces the integer result of 1. This result is added to the constant 2, yielding 3 as a final result. `A` is then assigned the value of 3 and printed.

Several applications of the `INT` function are shown in the chapters that follow. For now, try some other experiments with `INT` so that you become even more familiar with how it works.

Summary of Chapter 3

Chapter 3 has introduced you to some new and powerful TI BASIC capabilities:

- | | |
|--------------------------|--|
| FOR-NEXT | You've used this statement to build controlled loops that repeat a part of the program a specified number of times or create a time delay in the program. |
| PRINT formats | You've learned how to control the spacing of <code>PRINT</code> items using the three separators (comma, semicolon, and colon) and the <code>TAB</code> function. |
| Computation Order | You've discovered that your computer follows a certain mathematical order in solving problems: <ol style="list-style-type: none">1. Everything in parentheses is computed first.2. Multiplication and division are done next.3. Addition and subtraction are performed last. |
| INT function | You've learned how this function works on both positive and negative numbers that are not <i>integers</i> (whole numbers). |

These features will help prepare you for the programs that follow in the next chapters.

In this chapter we'll explore some features of the BASIC language that allow you to create exciting simulations and games.

Many computer programs are *simulations* that imitate some real-world event. With a computer simulation we can imitate an event as simple as the rolling of a single die or as complex as the patterns of animal migration in North America.

As an example of a simulation, we'll enter and run a dice-rolling program in this chapter. Other programs included here explore the games, graphics, and musical capabilities of your computer.

The heart of most games and simulations is the RND function, so let's begin there.

The RND Function

The letters in the name RND are taken from the word RaNDom. To find out what RND does, let's try a few examples in the Immediate Mode.

Clear the screen, and then enter this line:

```
PRINT RND
```

```
>PRINT RND
.5291877823
>□
```

Now try entering the line again:

```
>PRINT RND
.5291877823
>PRINT RND
.3913360723
>□
```

The second number is not the same as the first!

Here's an interesting situation! Every time we use RND, we get a different number. That's exactly what RND does — it generates *random numbers*.

4

Now let's try a program that will produce ten random numbers. Enter these lines:

```
20 FOR LOOP=1 TO 10
30 PRINT RND
40 NEXT LOOP
50 END
```

When you've checked your program for errors, run it. A list of ten random numbers will be printed on the screen. Look at the numbers closely. Are any two of the numbers identical?

You may have noticed that all the numbers generated by RND are less than one (1.0) in value. Also, there are no negative numbers. RND is preset to produce only numbers that are greater than or equal to zero and less than one ($0 \leq n < 1$).

Write down the numbers this program produced, and then run the program a second time. Check your written list against the numbers on the screen this time. Very strange! The list of numbers is the same!

This feature of the RND function is important to remember and can be very useful in certain applications. Within a program RND will produce the same sequence of random numbers each time the program is run.

UNLESS . . . !!

Unless the BASIC statement RANDOMIZE is used in your program.

The RANDOMIZE Statement

Add the RANDOMIZE statement shown below to the program that is still in your computer.

```
10 RANDOMIZE
```

Clear the screen now (type CALL CLEAR; press **ENTER**), and list the changed program on the screen:

```
>LIST
10 RANDOMIZE
20 FOR LOOP=1 TO 10
30 PRINT RND
40 NEXT LOOP
50 END
>□
```

Run the program again, and compare the new set of numbers with your written list from the first program run. Are they different this time? They should be!

Experiment!

Continue to experiment with the program until you feel comfortable with RND and RANDOMIZE. For example, try changing line 30 of the previous program to:

```
30 PRINT RND;RND
```

What result does this change have on the program?


If you want the program to generate more or fewer than ten random numbers, just change line 20.

Other Random Number Ranges

The program you just completed generates random numbers between 0 and 1 ($0 \leq n < 1$). Now let's examine ways to increase the range of the numbers we generate.

The RND function can be used as part of any legitimate computation. For example, $10 * \text{RND}$ and $(10 * \text{RND}) + 7$ are both valid uses of RND in TI BASIC. To show what is produced when RND is used in this way, try the following examples:

```
PRINT 10*RND
```



What number appears on the screen? Try the same example again. What number did you get this time?

In both these examples, you should see a decimal point followed by ten digits, or one digit to the left of the decimal point, followed by nine digits to the right of the decimal point. That's because $10 * \text{RND}$ produces random numbers in the range of 0 to (but not including) 10, or $0 \leq n < 10$.

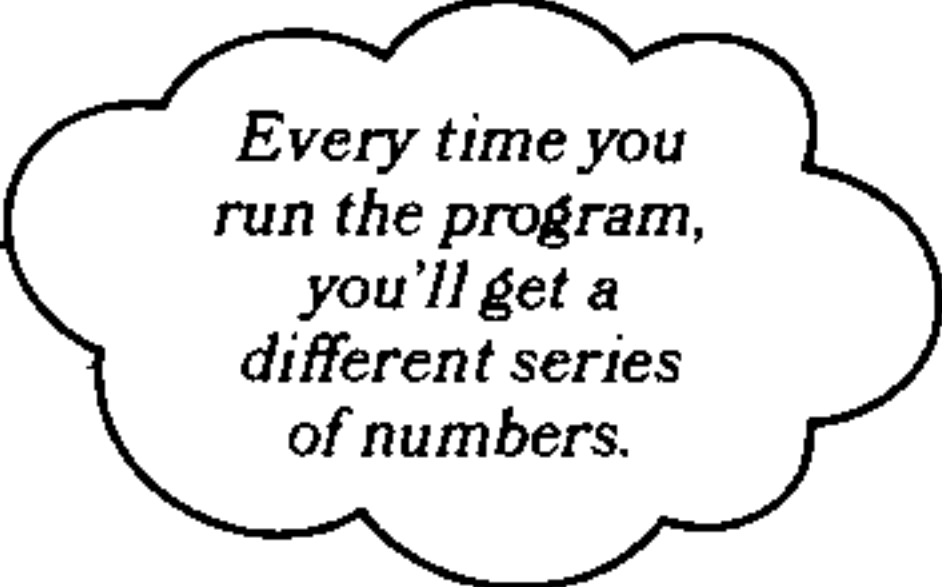
Now let's increase the range to this: $0 \leq n < 100$, or random numbers from 0 up to (but not including) 100. Try this:

```
PRINT 100*RND
```

and see what is produced. (Remember, this time you could get *one or two* digits to the left of the decimal point, in the range from 0 through 99.9999 . . .)

Let's use a program to generate some random numbers in the ranges 0 to 10 and 0 to 100. Enter these lines:

```
NEW  
10 RANDOMIZE  
20 FOR LOOP=1 TO 5  
30 PRINT 10*RND,100*RND  
40 NEXT LOOP  
50 END
```



Every time you run the program, you'll get a different series of numbers.



comma here

4

Now clear the screen and run the program. Although the numbers you generate on your screen will be different, they'll look something like this:

```
>RUN
 3.196128739    11.32761568
 6.233532821    9.502421843
 7.030941884    33.17351797
 .6689170795    86.40802154
 9.388957913    .7565322811
** DONE **
>□
```

Study the differences between the numbers in the left print zone on the screen and those in the right print zone. Can you see that the range is greater in those on the right? Run the program again to produce other numbers.

Suppose we'd like to eliminate all digits to the right of the decimal point and produce random whole numbers (integers). Well, do you remember the INT function we discussed in Chapter 3? This is a job for INT!

Change the program by typing and entering this new line:

```
30 PRINT INT(10*RND),INT(100*RND)
```

If you list the program now, it will look like this:

```
>LIST
10 RANDOMIZE
20 FOR LOOP=1 TO 5
30 PRINT INT(10*RND),INT(100
 *RND)
40 NEXT LOOP
50 END
>□
```

When you run the program, the screen will show two series of random whole numbers:

```

>RUN
 9          51
 0          14
 6          77
 5          9
 1          21

** DONE **

>□
  
```

Remember, you won't necessarily see these same numbers on your screen.

All the numbers on the left side of the screen will have values from 0 through 9, while the numbers on the right have values from 0 through 99. The INT function throws away the digits to the right of the decimal point. The following table summarizes what we have covered so far.

<i>Program Instruction</i>	<i>Range</i>
RND	0 through .9999 . . .
10*RND	0 through 9.9999 . . .
INT(10*RND)	0 through 9 (integers only)
100*RND	0 through 99.9999 . . .
INT(100*RND)	0 through 99 (integers only)

Notice that all these ranges begin with the value of zero. In many games and simulations, however, we need random numbers that start at some other value. For example, to simulate the throw of one die you need a random number generator that produces values from 1 to 6. You have seen that INT(10*RND) gives values from 0 to 9. What would INT(6*RND) produce? Change line 30 in the program to PRINT INT(6*RND) and run the new program.

Type:

```

30 PRINT INT(6*RND)

CALL CLEAR

RUN
  
```

```

>RUN
 4
 1
 5
 2
 3

** DONE **

>□
  
```

Random values within a range from 0 through 5

4

Your screen shows a list of five random numbers ranging from 0 to 5. What would happen if we added the value 1 to each item in this list? The resultant numbers would range from 1 to 6. That's just what we need to simulate the throw of a single die. Again, alter the program as shown below and run it.

Type:

```
30 PRINT INT(6*RND)+1
```

```
CALL CLEAR
```

```
RUN
```

```
>RUN
3
4
1
6
2

** DONE **

>□
```

Simulation of 5 rolls of a die.

That does it! The program now in your computer is a simulation (imitation) of throwing a single die five times.

A Two-Dice Simulation

At this point we can easily design a program to simulate the throws of two six-sided dice. Before you start, erase the old program by typing NEW. Then enter the following program:

```
5 CALL CLEAR
10 RANDOMIZE
20 INPUT "NUMBER OF ROLLS?":N
30 FOR ROLL=1 TO N
40 DIE1=INT(6*RND)+1
50 DIE2=INT(6*RND)+1
60 PRINT DIE1;DIE2,DIE1+DIE2
70 NEXT ROLL
80 PRINT
90 GOTO 20
```

Input number of rolls to simulate.

Simulate rolls.

Display each die and sum of "spots."

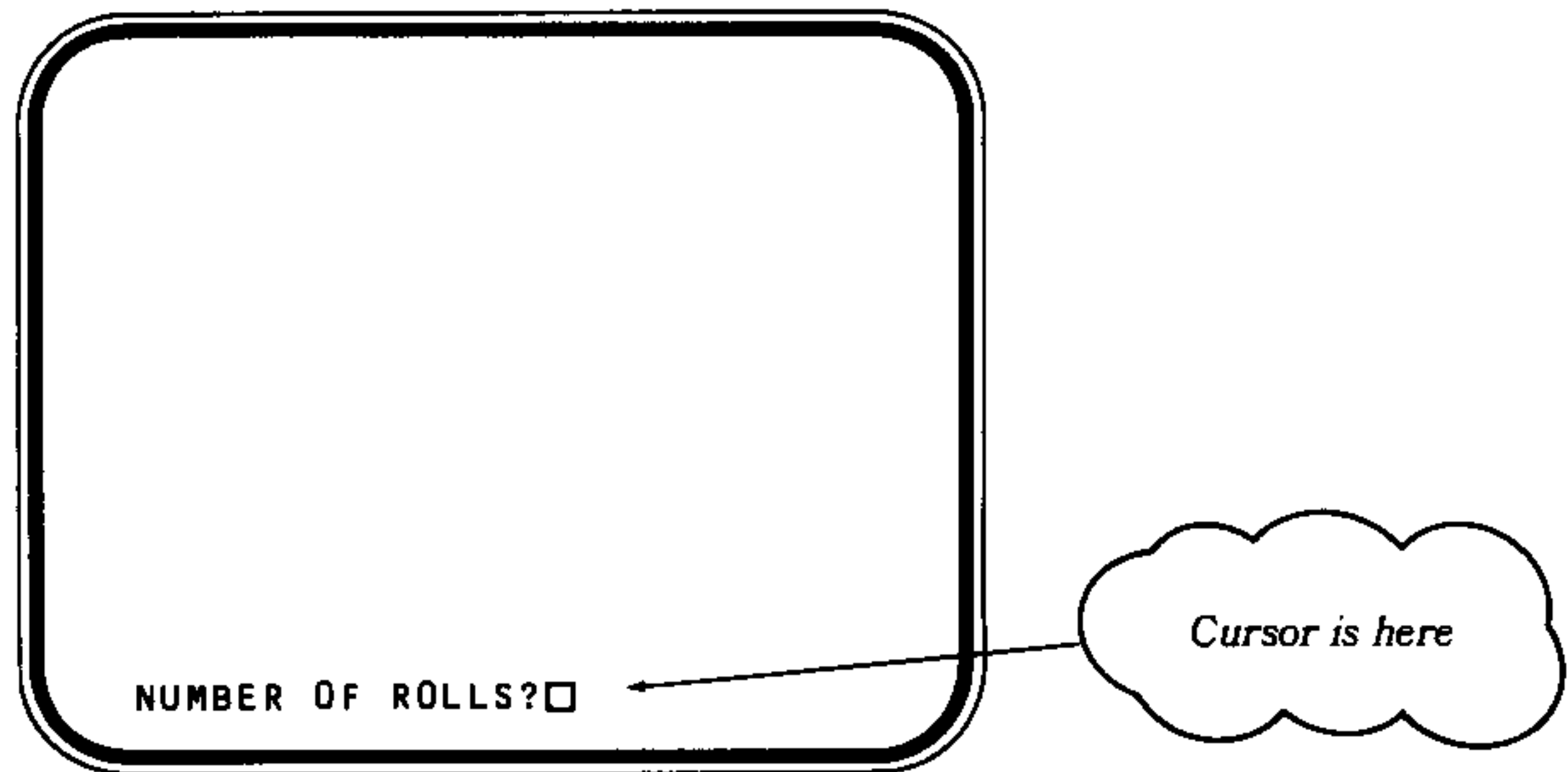
semicolon

comma

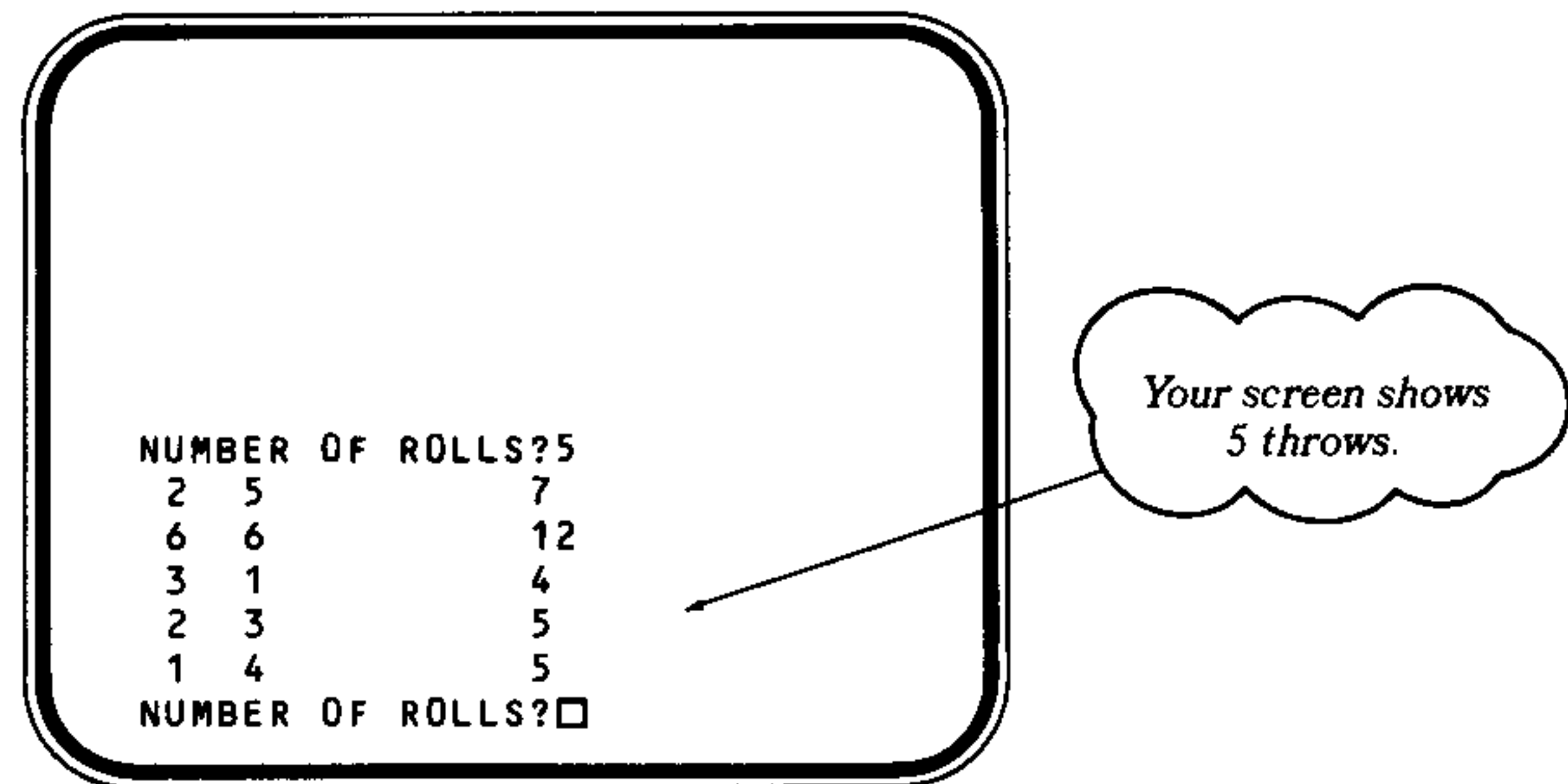
This program prints out the number of "spots" on each die and the sum of the spots on both dice faces. You are asked how many rolls you wish to make at the start of the program. Run the program now and watch what happens.

Type:

RUN



First, the program prints a request for the number of rolls to make. Enter a number (5, for example) and press the **ENTER** key.



The program keeps looping back to the INPUT request line. (If you want to stop the program, just press **CLEAR**.)

Experiment!

Try entering different values for the number of rolls. What happens if you try 30 rolls? Then make some changes to the program, if you'd like to experiment. For example, how would you alter the program to simulate the throwing of three dice? Two eight-sided dice?

4

Error Conditions with RND

The error messages produced by an improper usage of RND are essentially the same as the error messages we've mentioned before. Here are some examples:

Typing Errors

Error Message

missing operation

```
10 PRINT INT(10RND)
10 PRINT INT(10*RND)
```

```
*INCORRECT STATEMENT IN 10
*INCORRECT STATEMENT IN 10
```

missing close parenthesis

About the only new error condition we need to mention occurs if you try to use the letters RND as a numeric variable name in a LET or assignment statement. For example, if you type

```
LET RND=5
```

the computer will respond with

```
* INCORRECT STATEMENT
```

This occurs because RND is "reserved" to be used only as a function in TI BASIC. (For a list of all reserved words, see the "BASIC Reference" section of the *User's Reference Guide*.)

Randomized Character Placement

The following program utilizes the INT and RND functions to generate random screen positions for a character you input. First, type NEW and press ENTER to erase your old program; then enter these lines:

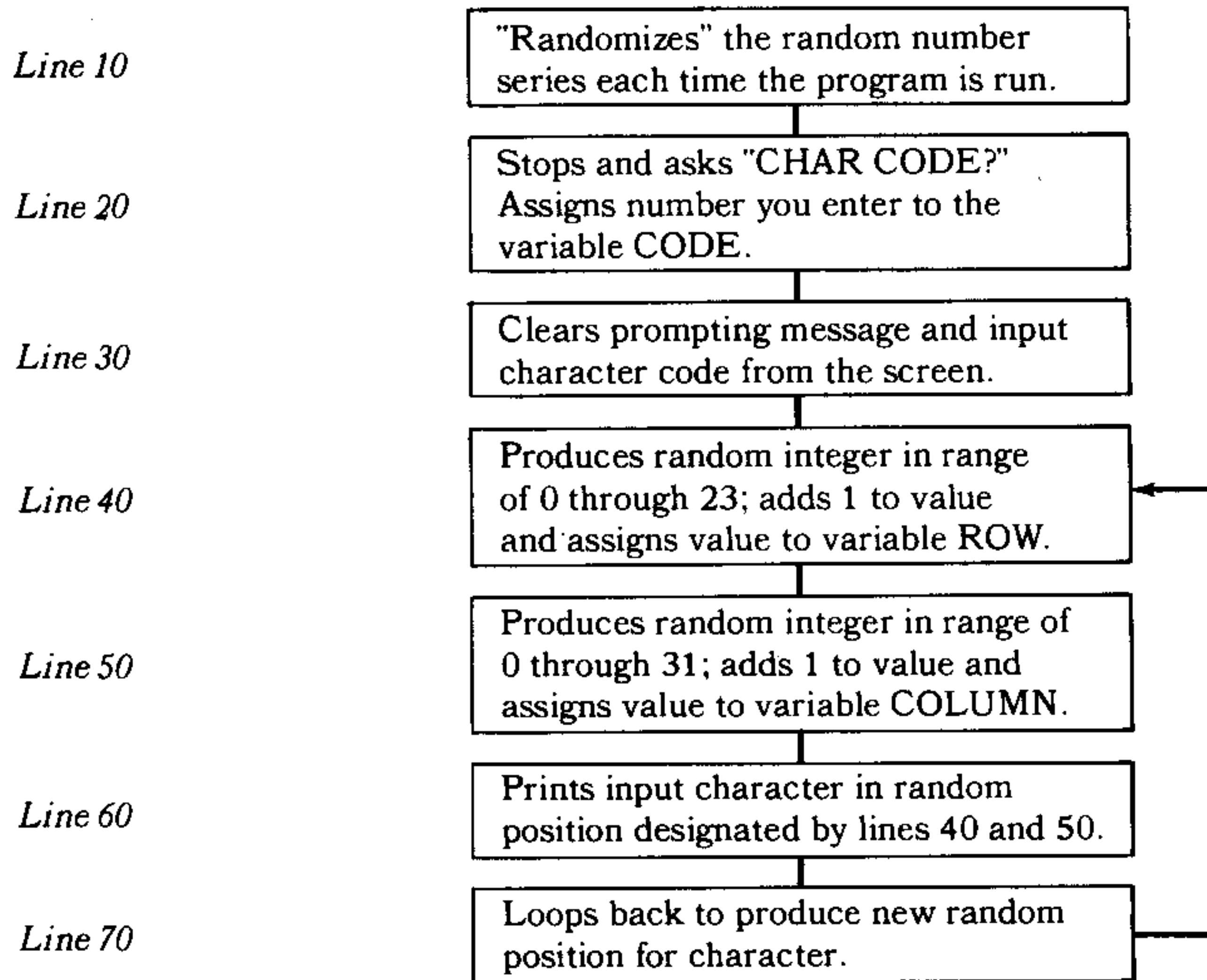
```
10 RANDOMIZE
20 INPUT "CHAR CODE?":CODE
30 CALL CLEAR
40 ROW=INT(24*RND)+1
50 COLUMN=INT(32*RND)+1
60 CALL VCHAR(ROW,COLUMN,CODE)
70 GO TO 40
```

*Produces
a row number
from 1 through 24.*

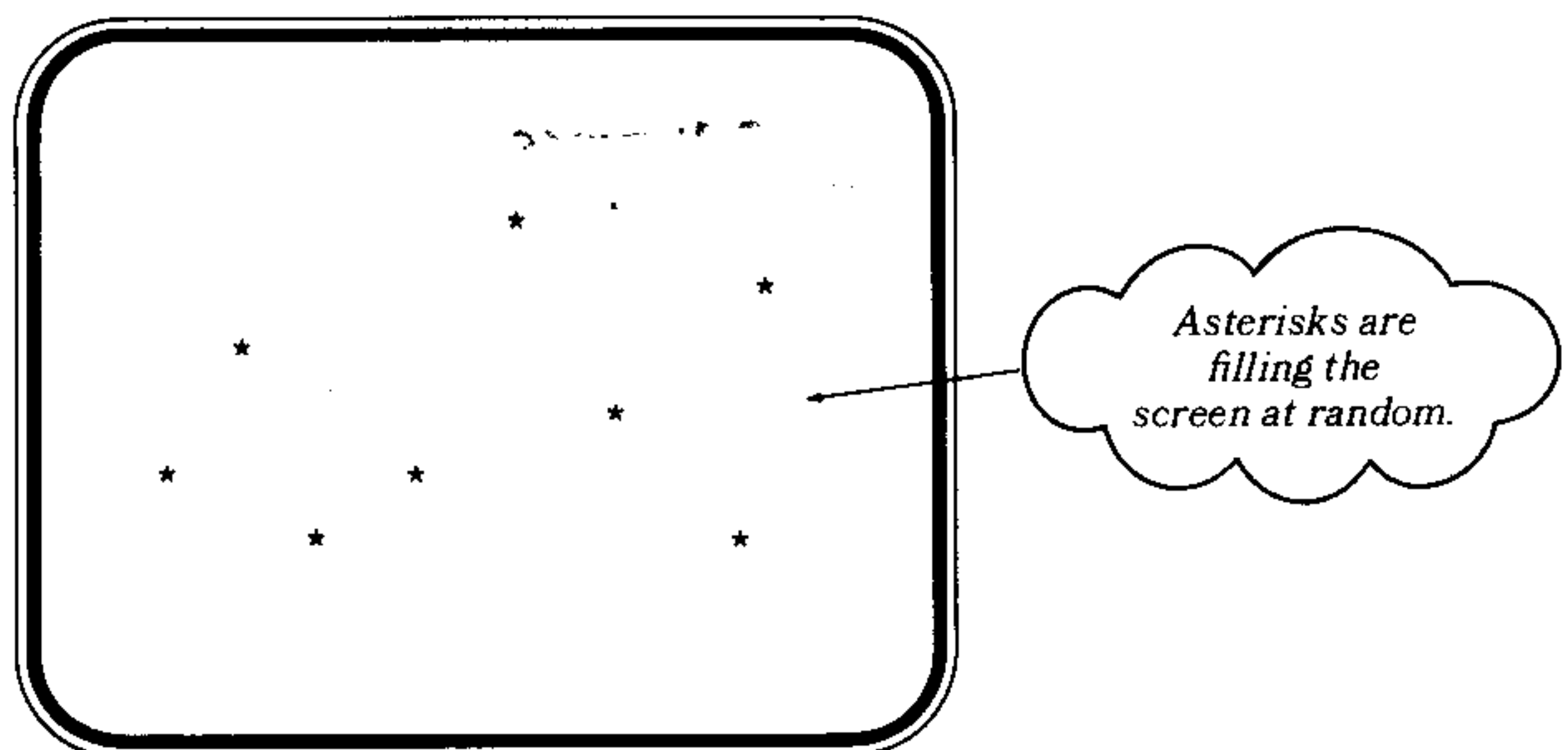
*Produces
a column number
from 1 through 32.*

We'll use the character codes 33 through 95; since character 32 is a blank space, we want to avoid entering it when the program asks for a code number.

Before running the program, let's examine a flow chart describing its performance.



Now clear the screen with `CALL CLEAR` and run the program. For this first example, enter 42 (the character code for the asterisk) as the input for CHAR CODE. The screen will look something like this:



To stop the program just press **CLEAR**. Then try running the program several times, putting in a different character code each time. See if any unusual designs are produced.

4

When you've finished experimenting with different characters, let's change the program to generate characters at random, as well as placing them randomly on the screen. First we'll have to decide how to set the limits we want for the character range. Here's a general procedure for setting the limits for use with RND:

Subtract the LOWER LIMIT from the UPPER LIMIT.

Add 1.

Multiply that result by RND.

Find the integer (INT) of this result.

Add the LOWER LIMIT.

Now we know that we want 63 characters, with character codes ranging from 33 through 95. So our LOWER LIMIT is 33, and our UPPER LIMIT is 95:

$$95 - 33 = 62$$

$$62 + 1 = 63$$

The number we want to multiply by RND is 63, and we must use the INT function:

`INT(63*RND)`

*Produces random integers
from 0 through 62.*

Now check the limits established when we add our LOWER LIMIT, 33:

$$0 + 33 = 33 \text{ (lowest possible character code)}$$

$$62 + 33 = 95 \text{ (highest possible character code)}$$

`INT(63*RND) + 33` will give us random whole numbers in the range we need. Type the following new line:

```
20 CODE=INT(63*RND)+33
```

and press **ENTER**. Now clear the screen and list the program to review this change.

```
>LIST
10 RANDOMIZE
20 CODE=INT(63*RND)+33
30 CALL CLEAR
40 ROW=INT(24*RND)+1
50 COLUMN=INT(32*RND)+1
60 CALL VCHAR(ROW,COLUMN,CO
E)
70 GO TO 40
>□
```

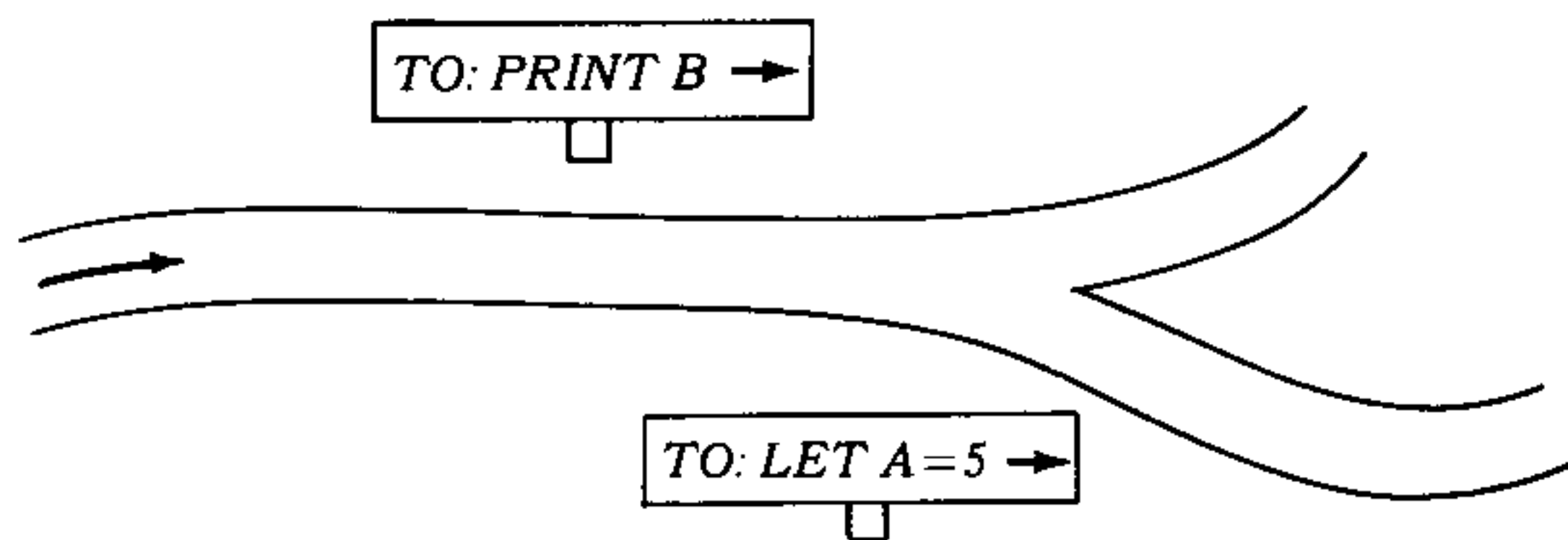
When we run the program this time, the computer will generate a random character code and then print the character in random positions on the screen. (Press **CLEAR** when you want to stop the program.) Run the program several times to see different characters.

Experiment!

By making changes in two lines, you can cause the previous program to print different random characters each time it loops. Try it! (*Hint*: Think about lines 30 and 70.)

The IF-THEN Statement

All the programs we've considered so far in this book have been constructed so that they either run straight through or loop using a GO TO or a FOR-NEXT loop. The IF-THEN statement provides you with the capability of making branches or "forks" in your program. A branch or fork is a point in a program where either one of two paths can be taken, just like a fork in a road.



The general form of an IF-THEN statement looks like this:

IF *condition* THEN *line number*

The *condition* is a mathematical relationship between two BASIC expressions. The *line number* is the program line to which you want the program to branch *if* the condition is true. If the condition is *not true*, then the program line following the IF-THEN statement is executed. For example,

```
30 IF K<10 THEN 70
```

The statement says: *If the value of K is less than 10, then go to line 70 of the program. If K is greater than or equal to 10, then do not branch to line 70. Instead, execute the line following line 30.*

Let's try a demonstration program. Enter these lines:

If new value of K is not less than 10, go on to next line.

```
NEW
10 CALL CLEAR
20 LET K=1
30 PRINT "K=";K
40 LET K=K+1
50 IF K<10 THEN 30
60 PRINT "OUT OF LOOP"
70 END
```

If new value of K is less than 10, go back to line 30 and repeat.

4

Now run the program.

```
K= 1
K= 2
K= 3
K= 4
K= 5
K= 6
K= 7
K= 8
K= 9
OUT OF LOOP

**DONE**

>□
```

Each time the program reaches line 50, it must make a "true or false" decision. When K is less than 10, the IF condition ($K < 10$) is true, and the program branches to line 30. When K equals 10, however, $K < 10$ is false. The program then executes line 60 and stops.

We mentioned earlier that the condition is a mathematical relationship between two expressions. In the example you've just seen, the mathematical relationship was $<$, or "less than." There are a total of six relationships that can be used in the IF-THEN statement:

<i>Relationship</i>	<i>Mathematical Symbol</i>	<i>BASIC Symbol</i>
Equal to	=	=
Less than	<	<
Greater than	>	>
Less than or equal to	\leq	<=
Greater than or equal to	\geq	>=
Not equal to	\neq	<>

Suppose we changed line 50 in the program to this:

```
50 IF K<=10 THEN 30
```

How would the program's performance be affected? Try it! Enter the new line, and then run the program again.

Now, the program prints the value of K all the way through 10, because the new line says, "If K is *less than or equal to 10*, branch to line 30."

Experiment!

The IF-THEN statement can be a powerful tool in program development. Try this program for a graphics application:

```
NEW
10 CALL CLEAR
20 CALL COLOR(2,5,5)
30 LET K=1
40 CALL HCHAR(K,K+1,42)
50 K=K+1
60 IF K<25 THEN 40
70 K=1
80 CALL HCHAR(K,K+3,42)
90 K=K+1
100 IF K<25 THEN 80
110 GOTO 110
```

Can you follow this pattern to create more than two diagonal lines?

Error Conditions with IF-THEN

Like most TI BASIC statements, the IF-THEN statement is pretty particular about its form. The main errors that can occur in using the IF-THEN statement are shown below:

20 IFA=B THEN 200	(No space after IF)
20 IF A=BTHEN 200	(No space in front of THEN)
20 IF A=B THEN200	(No space after THEN)
20 IF A==B THEN 200	(Invalid relational symbol combinations)
20 IF A= THEN 200	(No expression on one side of the relational symbol)

All of the above conditions produce an error message either when entered or during the running of the program, along with a reference to the line number of the statement in which the error occurs.

If the line number referenced in an IF-THEN statement does not exist, the program stops and produces a message saying that the line number referenced in the statement is not found in the program. For example (using the line above), if 200 is not a valid line number in your program, you see this error message:

```
* BAD LINE NUMBER IN 20
```

Games and Music

The remainder of this chapter explores color graphics and sound through special games applications. Several of the programs are based on a number-guessing game you may have played before. You'll also find that both the RND function and the IF-THEN statement are used extensively in the programs.

4

A Number-Guessing Program

In this game the computer generates a secret number from 1 to 100, using the RND function, and asks you to guess the number. The program tells you if your guesses are larger, smaller, or equal to the secret number. When you guess the number, the program chooses another number and begins the game again.

Type NEW, press **ENTER**, and enter these lines:

```
10 CALL CLEAR
20 SECRET=INT(100*RND)+1
30 PRINT "I HAVE A SECRET NUMBER!"
40 PRINT
50 INPUT "WHAT IS YOUR GUESS?":GUESS
60 IF GUESS=SECRET THEN 130
70 IF GUESS>SECRET THEN 100
80 PRINT "TOO SMALL!"
90 GOTO 110
100 PRINT "TOO BIG!"
110 PRINT "TRY AGAIN."
120 GOTO 40
130 PRINT "YOU GUESSED IT!"
140 PRINT "LET'S PLAY AGAIN!"
150 FOR DELAY=1 TO 1000
160 NEXT DELAY
170 GOTO 10
```

Select random number
from 1 through 100.

Your guess is
input here.

If you guess correctly,
the program
branches to line 130.

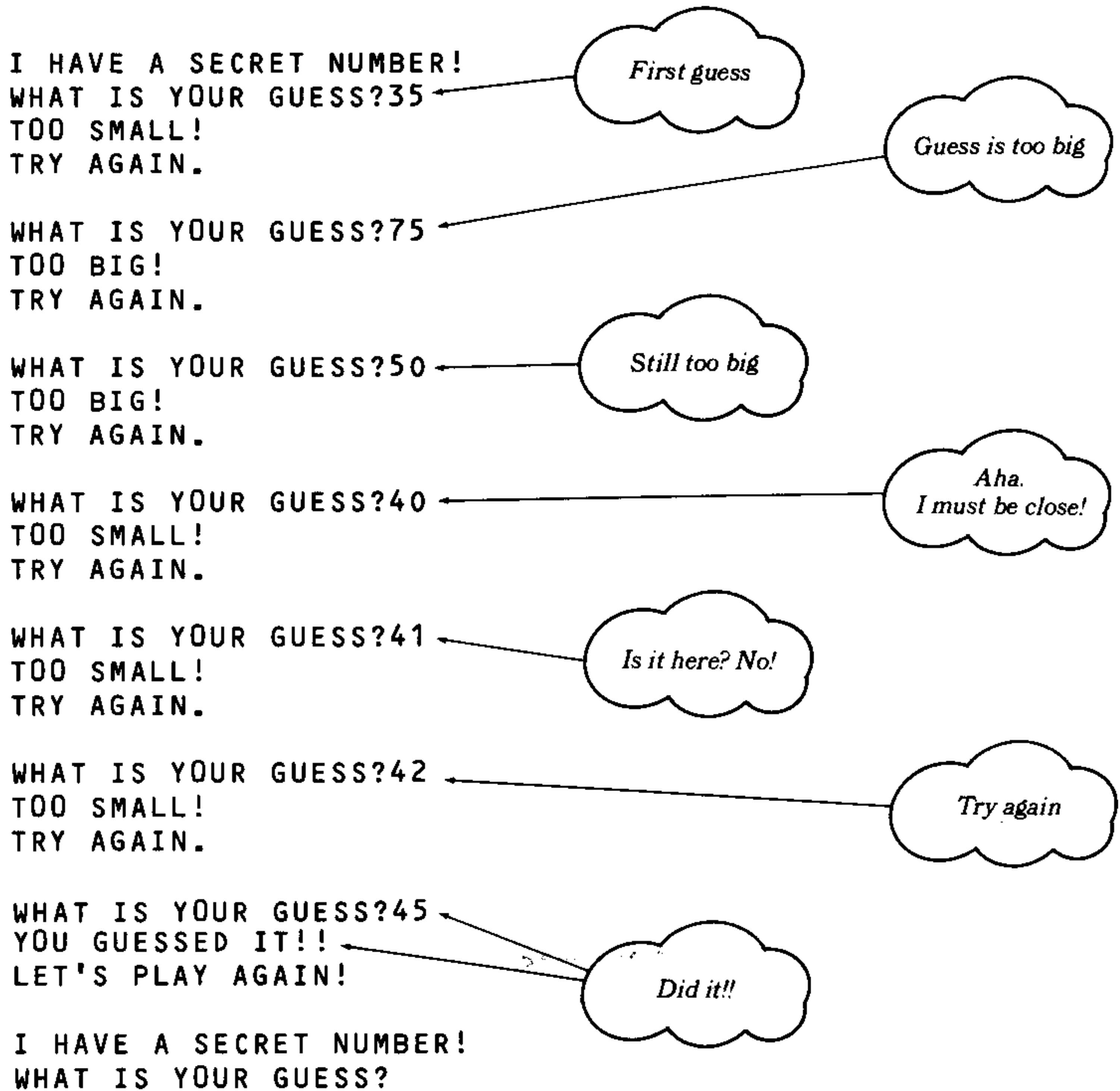
If your guess is too large,
the program
branches to line 100.

Delay so you can read
the "victory" message.

Notice that two IF-THEN statements are used in the program, at lines 60 and 70. In line 60, if the guess is *not equal* to the secret number, the condition in the IF-THEN statement is false, and the program proceeds to line 70. If the guess *is equal* to the secret number, the program branches to line 130 and prints the victory message.

At line 70, we test to see if the guess is larger than the secret number. If the guess is larger than the number, the condition is true, and the program branches to line 100. If the guess is smaller than the number, the condition is false, and the program proceeds to line 80.

Now run the program. When it asks "WHAT IS YOUR GUESS?" just type in a number from 1 through 100, and press **ENTER**. Here's an example of what might appear on the screen:

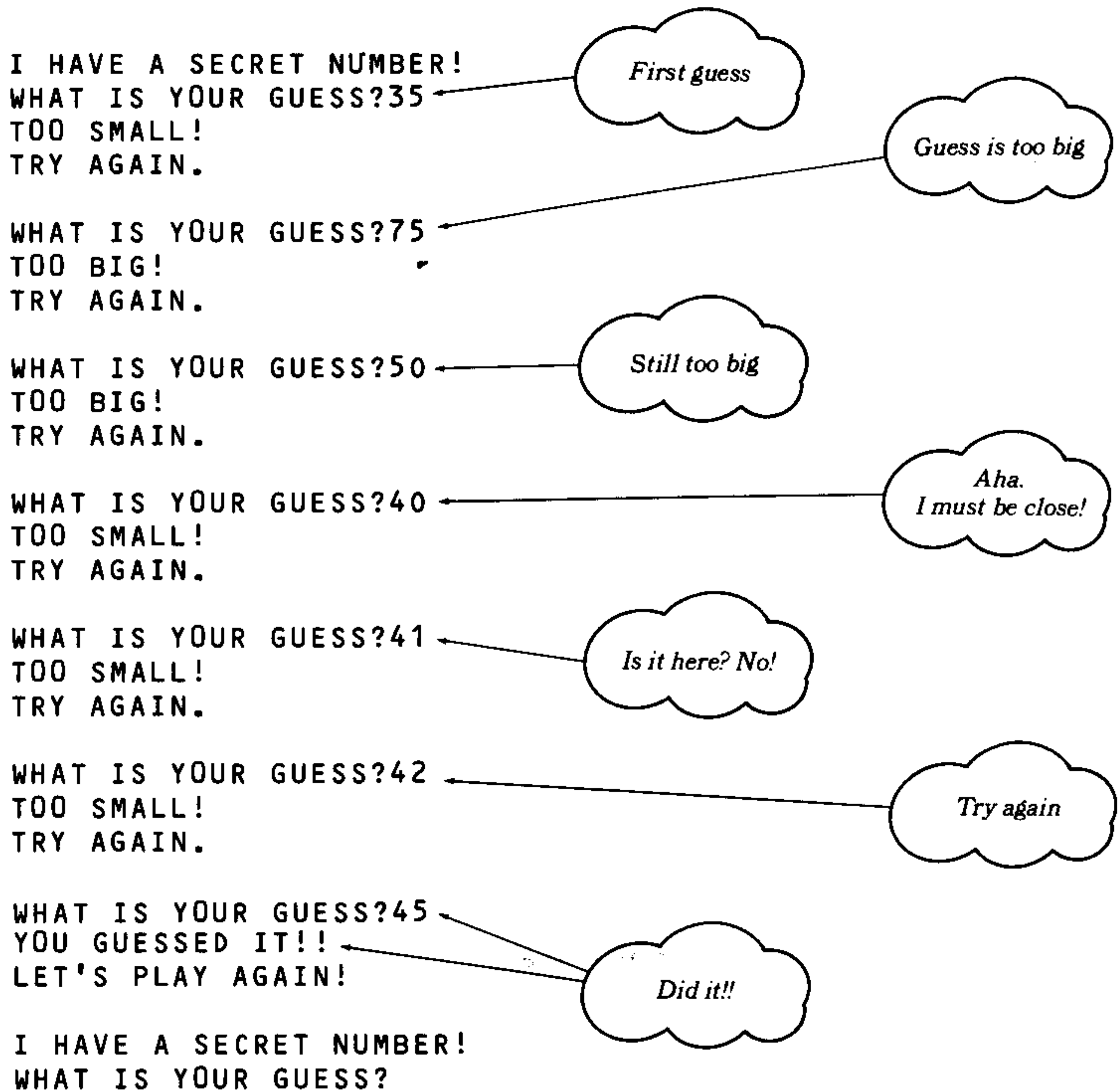


The computer will start a new game each time you guess the correct number. When you want to stop playing, just press **CLEAR**.

Notice also that we did *not* include the **RANDOMIZE** statement. Therefore, the program will generate the same series of random numbers each time you run it! If you want to make the program create a new set of random numbers each time, just add this line:

```
15 RANDOMIZE
```


Now run the program. When it asks "WHAT IS YOUR GUESS?" just type in a number from 1 through 100, and press **ENTER**. Here's an example of what might appear on the screen:



The computer will start a new game each time you guess the correct number. When you want to stop playing, just press **CLEAR**.

Notice also that we did *not* include the **RANDOMIZE** statement. Therefore, the program will generate the same series of random numbers each time you run it! If you want to make the program create a new set of random numbers each time, just add this line:

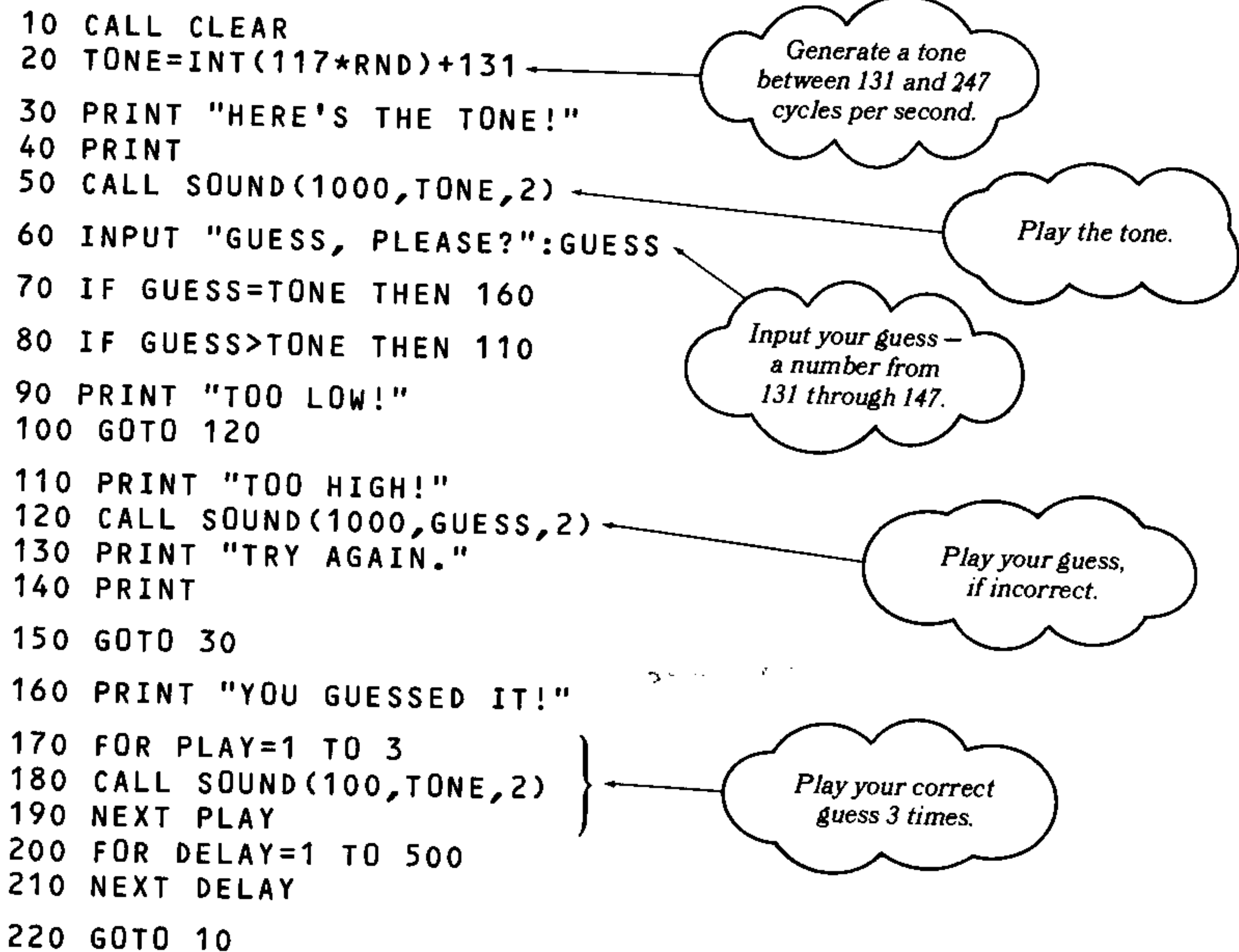
```
15 RANDOMIZE
```

4

A Tone-Guessing Program

A novel version of the number-guessing program can be created using the sound capabilities of your computer. This program generates a random tone from 131 cycles per second through 247 cycles per second. (If you need to review the `CALL SOUND` statement and the frequency limits of the computer, see Chapter 1, page 17.) Your job is to guess the frequency of the tone! The program lets you know if your guess is lower, higher, or equal to the frequency of the random tone that is generated. When you guess the correct frequency, the program plays the tone three times and begins the game again.

So type `NEW`, press **ENTER**, and enter the new program.



Line 20 may need a little explanation. If the lowest tone we want is 131 cycles per second and the highest is 247 cycles per second, how do we set our random number limits? Well, `INT(117*RND)` produces numbers from 0 through 116, and

$$0 + 131 = 131 \text{ (our desired lower limit)}$$
$$116 + 131 = 247 \text{ (Our desired upper limit)}$$

Now run the program. The information that appears on the screen is similar to the number-guessing program. The only difference is that in this program your guess is "played" back to you by the computer.

If you'd like to change the tone limits, you can do so easily by changing line 20. For example, suppose you'd rather hear a series of higher tones – perhaps in the range from 262 cycles per second through 392 cycles per second. How would you rewrite line 20 to generate these tones?

Also, you may want to add the `RANDOMIZE` statement to create a new series of random tones each time you run the program. If so, just enter this new line:

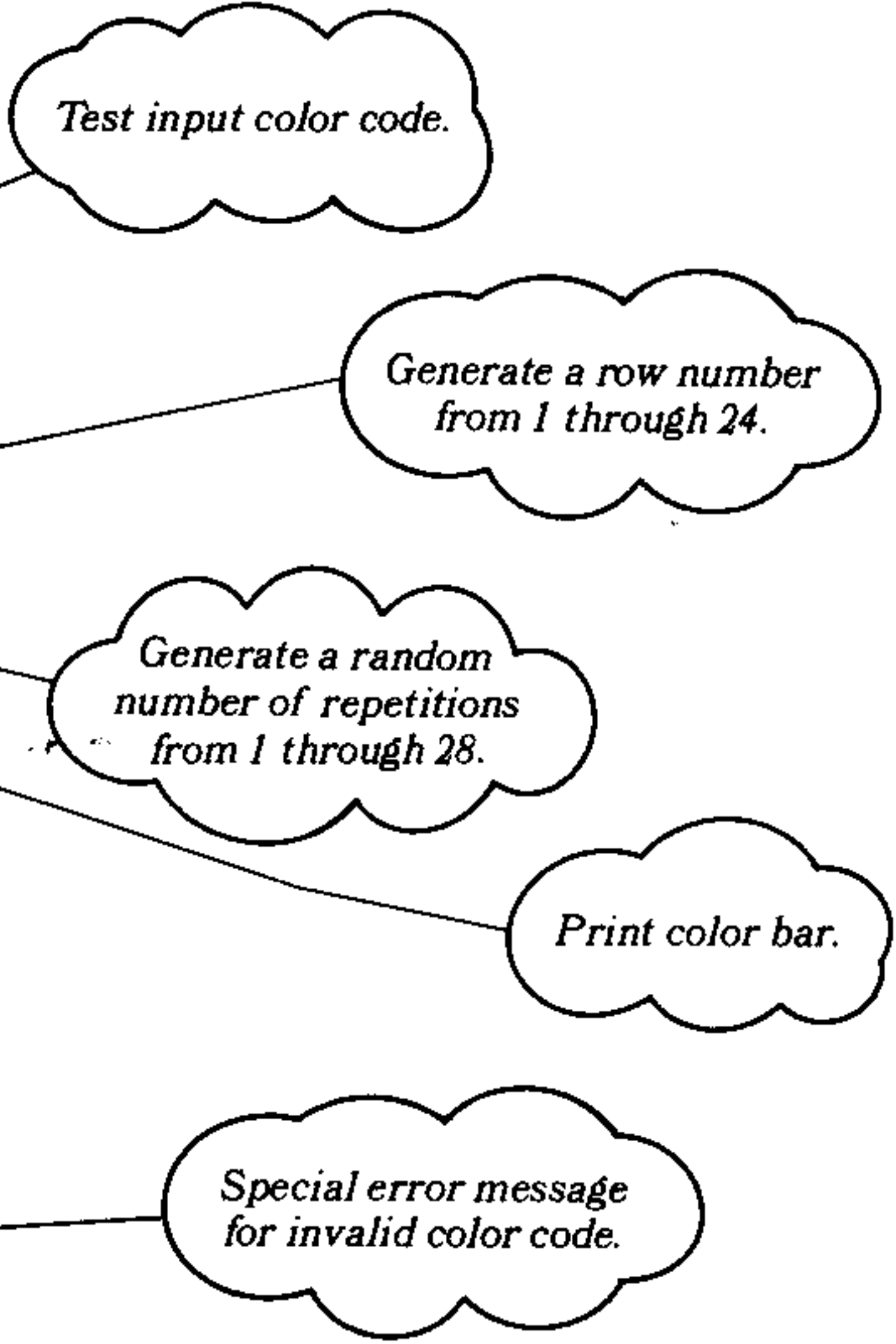
```
15 RANDOMIZE
```

Color Up!

Next, let's examine two color programs. The first program creates ten randomly placed horizontal bars – of a color you input, and of random lengths. Then the program stops for you to input a new color code.

You'll notice that we've used `IF-THEN` statements in a new way (lines 30 and 40). We test the input color code to be sure it's valid. If it isn't, the program gives you a specially written "error message."

```
10 CALL CLEAR
15 RANDOMIZE
20 INPUT "COLOR PLEASE?":C
25 CALL CLEAR
30 IF C<1 THEN 200
35 IF C>16 THEN 200
40 FOR LOOP=1 TO 10
45 ROW=INT(24*RND)+1
50 REPEAT=INT(28*RND)+1
55 CALL COLOR(2,C,C)
60 CALL HCHAR(ROW,3,42,REPEAT)
65 FOR DELAY=1 TO 100
70 NEXT DELAY
75 NEXT LOOP
80 GO TO 10
200 PRINT "BAD COLOR CODE!"
210 PRINT "MUST BE 1 TO 16."
220 PRINT "TRY AGAIN!"
240 FOR DELAY=1 TO 500
250 NEXT DELAY
260 GO TO 10
```

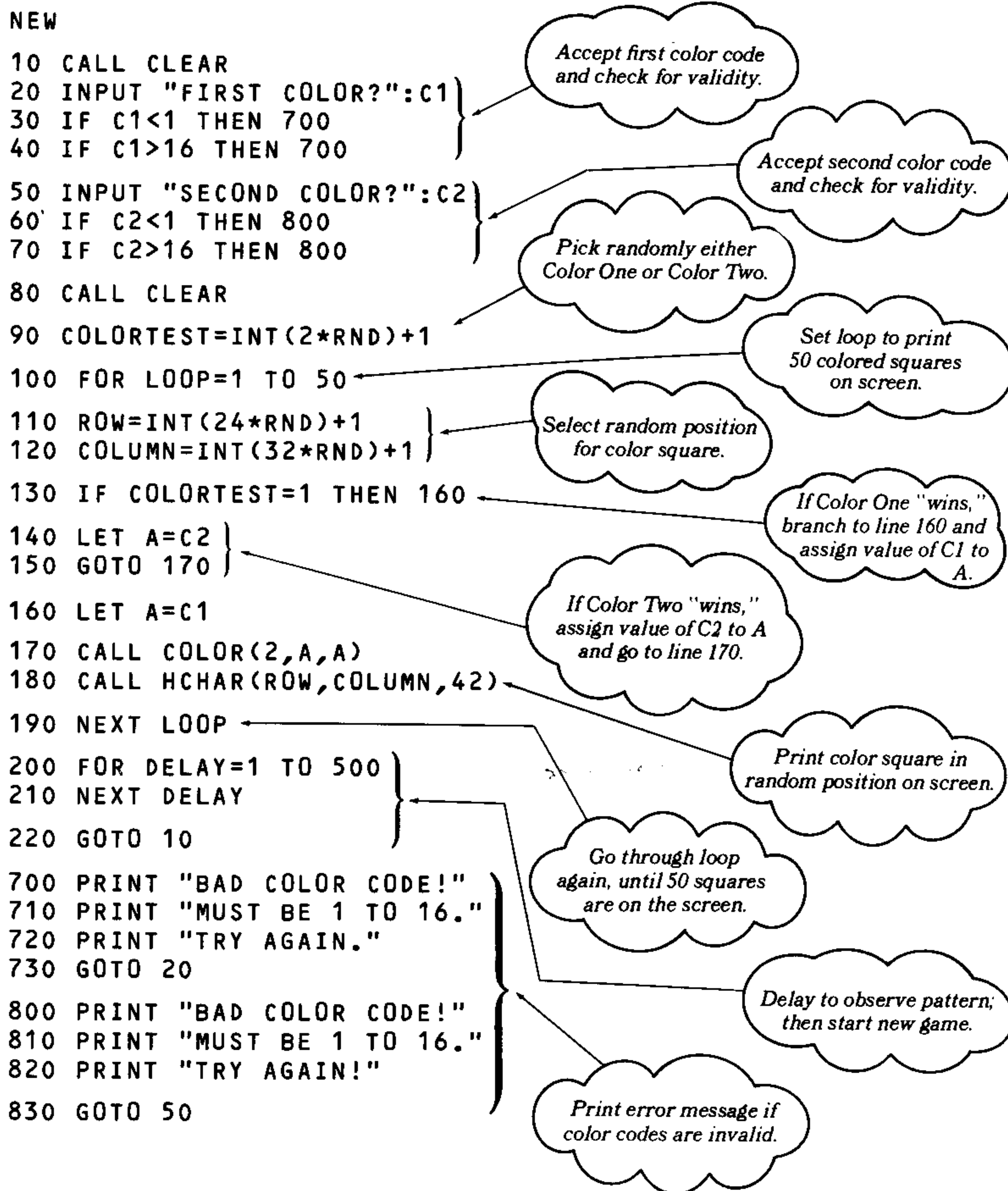


When you run the program, you'll see all of the bars begin at column 3, near the left-hand edge of the display. Their lengths, however, are random, as are their horizontal positions on the screen. After ten bars of the input color are placed, the program clears the screen and asks you for a new color code.

4

Remember to avoid putting in color codes 1 (transparent) and 4 (the screen color in the Run Mode). Although these are valid codes, you won't be able to see the bars.

The next program is a game that contests two colors against each other. A winning color is randomly chosen. The program is the longest you've seen yet, so we'll provide some explanations as we go along. Here's the program:



Two people can play against each other, or you can play against yourself by putting in both color codes, just to see which "wins" the game. (Again, avoid entering color codes 1 and 4.)

Random Notes

We've used CALL SOUND earlier in a program that played notes from a musical scale. (See Chapter 2, pages 39-40.) If we modify that program, adding the IF-THEN statement and the RND function, we can make the computer play some interesting (but not necessarily enjoyable) "music." Here's how:

```

NEW
10 LET C=262
15 LET D=294
20 LET E=330
25 LET F=349
30 LET G=392
35 LET A=440
40 LET B=494
45 LET C2=523
50 RANDOMIZE
55 NOTE=INT(8*RND)+1
60 TIME=INT(901*RND)+100
65 VOLUME=2
70 IF NOTE=1 THEN 200
75 IF NOTE=2 THEN 300
80 IF NOTE=3 THEN 400
85 IF NOTE=4 THEN 500
90 IF NOTE=5 THEN 600
100 IF NOTE=6 THEN 700
105 IF NOTE=7 THEN 800
110 NOTE=C2
115 CALL SOUND(TIME,NOTE,VOLUME)
120 GOTO 55
200 NOTE=C
210 GOTO 115
300 NOTE=D
310 GOTO 115
400 NOTE=E
410 GOTO 115
500 NOTE=F
510 GOTO 115
600 NOTE=G
610 GOTO 115
700 NOTE=A
710 GOTO 115
800 NOTE=B
810 GOTO 115

```

The diagram consists of several callout boxes connected to specific lines of code by arrows:

- A callout box: "Set up a musical scale from "middle C" through "high C."." points to lines 10-45.
- A callout box: "Randomly select 1 of the 8 notes." points to line 55.
- A callout box: "Randomly select a duration from 100 through 1000 milliseconds." points to line 60.
- A callout box: "Check which note to play. Notice that we don't have to check for the last note - "high C" (C2). It will automatically be selected if none of the first 7 notes are selected." points to lines 70-105.
- A callout box: "Play the note." points to line 115.
- A callout box: "Define NOTE." points to lines 200-810.

4

Now run the program and enjoy the "music." When you're ready to "stop the music," just press **CLEAR**.

You might like to experiment with this program in various ways. For example, do you notice anything different in the "music" if you change lines 60 and 65 to

```
60 TIME=500
65 VOLUME=5
```

A Musical Interlude

Now that we've let the computer play its "music," let's play some music of our own! With this program we can use the keyboard to input the notes we want to play. Enter these lines:

NEW

```
10 CALL CLEAR
```

```
15 LET C=262
```

```
20 LET D=294
```

```
25 LET E=330
```

```
30 LET F=349
```

```
35 LET G=392
```

```
40 LET A=440
```

```
45 LET B=494
```

```
50 INPUT "NOTE ":A$
```

```
55 IF A$="C" THEN 100
```

```
60 IF A$="D" THEN 200
```

```
65 IF A$="E" THEN 300
```

```
70 IF A$="F" THEN 400
```

```
75 IF A$="G" THEN 500
```

```
80 IF A$="A" THEN 600
```

```
85 IF A$="B" THEN 700
```

```
90 GOTO 50
```

```
100 NOTE=C
```

```
110 GOTO 800
```

```
200 NOTE=D
```

```
210 GOTO 800
```

```
300 NOTE=E
```

```
310 GOTO 800
```

```
400 NOTE=F
```

```
410 GOTO 800
```

```
500 NOTE=G
```

```
510 GOTO 800
```

```
600 NOTE=A
```

```
610 GOTO 800
```

```
700 NOTE=B
```

```
800 CALL SOUND(100,NOTE,2)
```

```
810 GOTO 50
```

This time, we'll only define 7 notes.

leave one space

Accept "note."

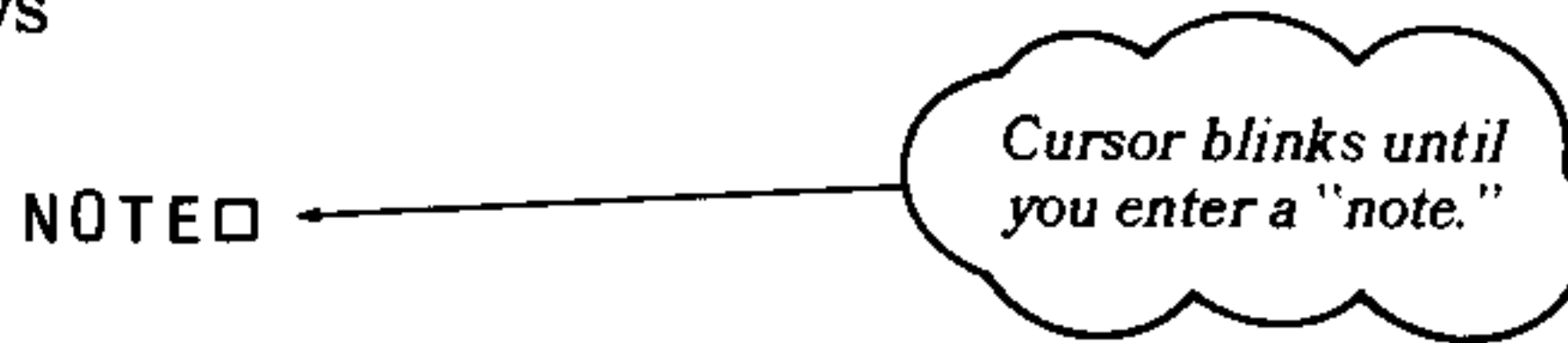
Check for the letter key pressed on the keyboard.

Not A-G! Do it again.

Play "note."

Return for new note.

When you run the program, the program will ask you for a note. You then press one of the letter keys (A,B,C,D,E,F, or G), followed by the **ENTER** key. For example, when the screen shows



and you press these keys:

A (ENTER)

the "note" A will play. The screen keeps a record of the keys you depress:

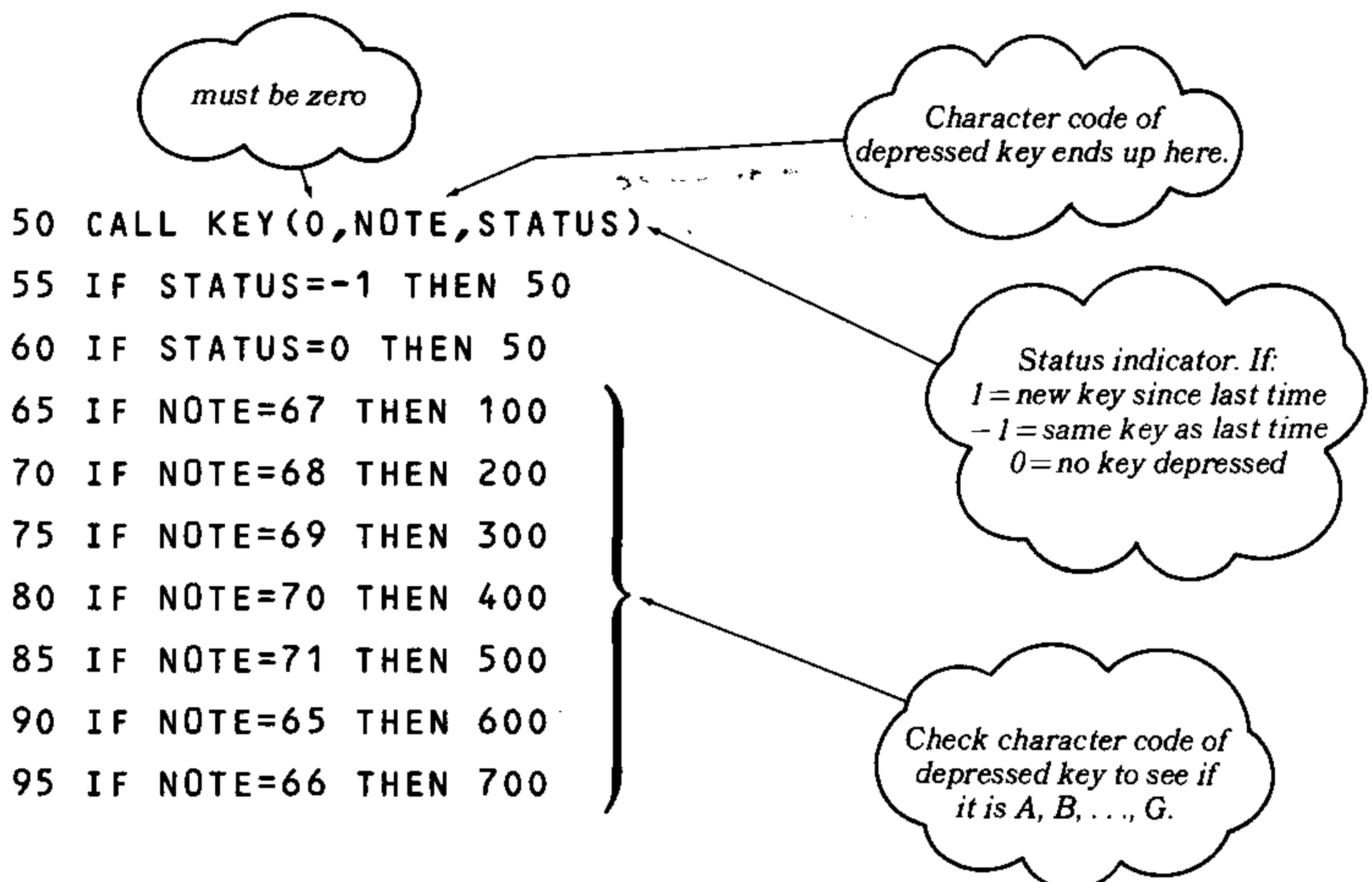
```
NOTE C
NOTE D
NOTE E
NOTE F
```

Having to press the **ENTER** key for each note slows down your musical performance a bit, doesn't it? What can we do about this problem?

The CALL KEY Routine

There is a routine that permits the transfer of one character from the keyboard directly into a program. The routine is **CALL KEY**. If you alter the current program in the following way, you don't have to press the **ENTER** key after hitting the key for each note.

Enter:



4

Here's how **CALL KEY** works. Each character on the keyboard has a numeric code. When a key is depressed, the character code of that key is assigned to the second variable in the **KEY** routine. In this example, the character code is assigned to the variable **NOTE**. The last variable in the **KEY** routine is a status indicator. The indicator lets the program know what has occurred on the keyboard. If you keep holding down the same key, the **STATUS** is minus one. If you press a key different from your last entry, the **STATUS** is one (1). If you don't press any key, the **STATUS** is zero (0). When you run the program, nothing appears on the screen as you press the keys. The program simply plays the note you request. So go ahead – make a little music!

The **CALL KEY** routine allows you to create "your own kind of music," and the routine can also be used in many games and simulations where single-character input values are requested. The **CALL KEY** routine speeds up the input of data by eliminating the need to press the **ENTER** key after your data entry.

Summary of Chapter 4

This chapter has given you an idea of the many interesting games and simulations you can develop with your computer. You've discovered these new features:

RND	Allows you to generate random numbers.
RANDOMIZE	Insures that each series of random numbers generated by a program will be different.
IF-THEN	Provides conditional branching capabilities in a program.
CALL KEY	Permits the transfer of a keyboard character directly into a program, without pressing ENTER .

Congratulations! You've accomplished a lot of computer programming!

The following chapter deals only with computer graphics. You'll learn how to define your own characters and how to make "animated" patterns on the screen. Just turn the page for some more exciting experiences!

