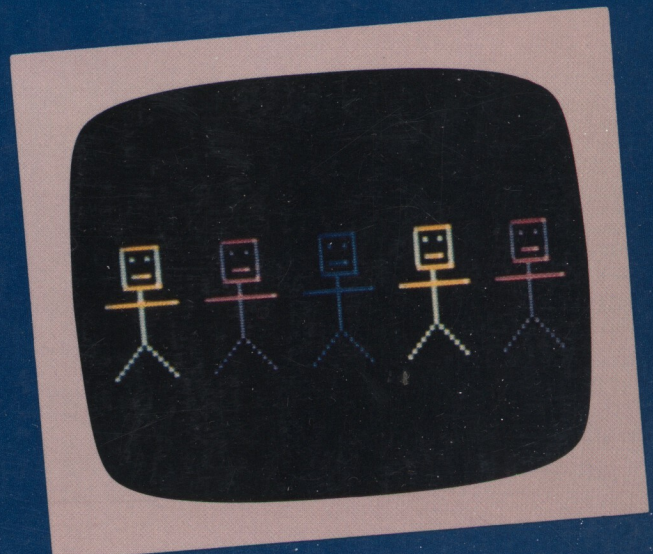
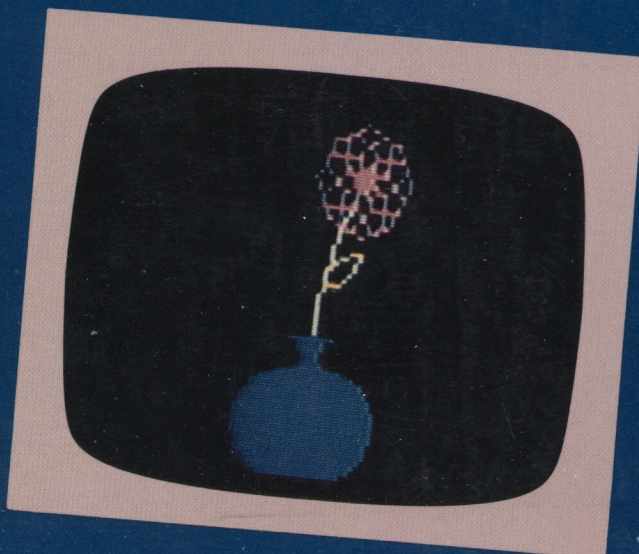
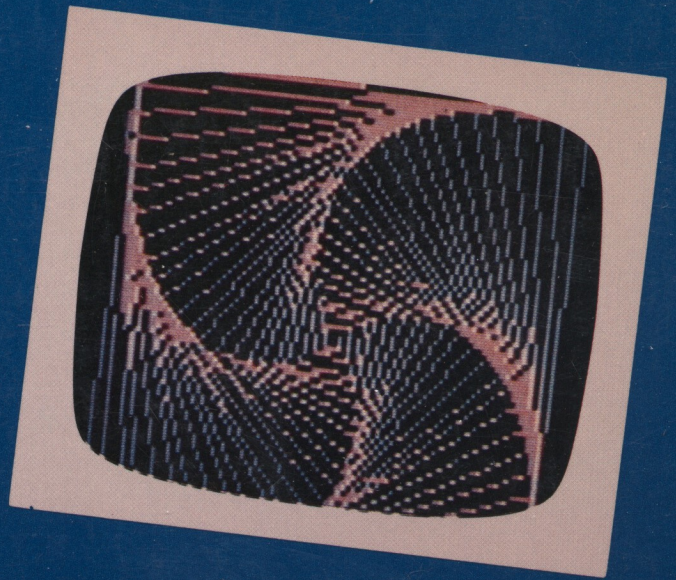
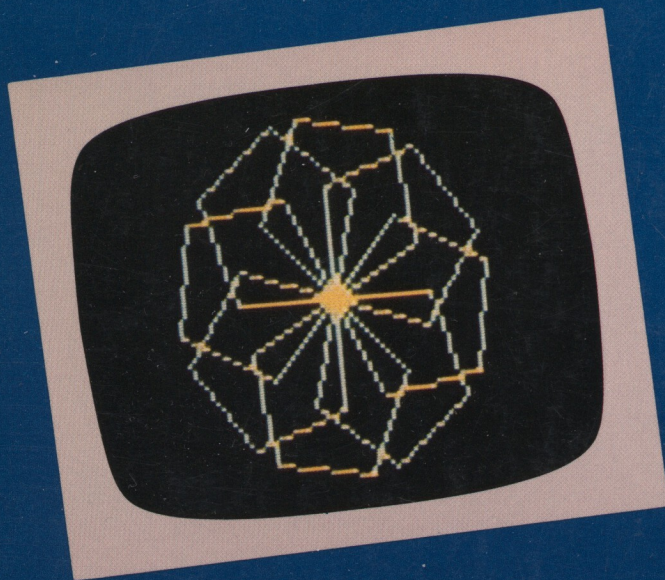


David D. Thornburg

PICTURE THIS!

An Introduction to
Computer Graphics for
Kids of All Ages

Including Atari™
PILOT Turtle Geometry



CG&D LIBRARY

CATEGORY: 5. COMPUTERS

DATE: 5/94

PICTURE THIS!

PILOT TURTLE GEOMETRY

An Introduction to Computer
Graphics for Kids of All Ages



DAVID D. THORNBURG



ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts • Menlo Park, California
London • Amsterdam • Don Mills, Ontario • Sydney

Other Microbooks from Addison-Welsey . . .

BASIC AND THE PERSONAL COMPUTER *by Thomas Dwyer and Margot Critchfield*

A BIT OF BASIC *by Thomas Dwyer and Margot Critchfield*

COMPUTER CHOICES *by H. Dominic Covvey and Neil Harding McAlister*
COMPUTER CONSCIOUSNESS

Surviving the Automated 80s *by H. Dominic Covvey and Neil Harding McAlister*

EXECUTIVE COMPUTING

How To Get It Done On Your Own *by John M. Nevison*

THE LITTLE BOOK OF BASIC STYLE

How to write a program you can read *by John M. Nevison*

PASCAL

A Problem Solving Approach *by Elliot B. Koffman*

PROGRAMMING A MICROCOMPUTER

6502 *by Caxton C. Foster*

REAL TIME PROGRAMMING

Neglected Topics *by Caxton C. Foster*

Library of Congress Cataloging in Publication Data

Thornburg, David D.

Picture this!

Includes index.

1. Computer graphics. 2. Pilot (Computer program)

I. Title.

T385.T497 001.64'43 81-20548

ISBN 0-201-07768-X AACR2

Copyright © 1982 by Innovision and David D. Thornburg. Philippines
copyright © 1982 by Innovision and David D. Thornburg.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada. Library of Congress Catalog Card No. 81-07768

Second Printing, November 1982

BCDEFGHIJ-HA-898765432

Preface

THIS BOOK is devoted to one basic idea: that everyone—children and adults alike—is capable of making a computer do what he or she wants it to do. Most people with access to a computer use programs written by others. While this is satisfying for many people, it is quite exciting to create programs of one's own.

To learn programming is to gain true mastery over the computer. There are no age limits, educational requirements, or special skills needed. If you can read these words and are willing to experiment with your computer, this book will help you to create beautiful pictures on your display screen.

The reason such bold claims can be made is due to the decision of Atari, Inc. to incorporate “turtle graphics” in their version of PILOT—a user-friendly computer language. PILOT is exceptional in that it is one of the easiest computer languages to learn, and yet allows the programmer to create extraordinary programs.

For the benefit of those readers who want to know about turtle graphics before reading this book, a brief explanation may suffice. In familiar coordinate geometry, the location of a point in a plane is specified by its coordinates (usually denoted by the letters x and y). The x coordinate measures the point's distance from a vertical reference line, and the y coordinate measures the point's distance above (or below) a horizontal reference line. This representation system has been in use for hundreds of years, and it works just fine.

Another way of describing the properties of a point is to specify its *orientation* as well as its x and y coordinates. There are several reasons why this additional piece of information is valuable. First, it allows simple representation of a graphic object through a procedure that, when followed, will generate the object. For example, if our point (which we will call the turtle) is pointing straight up, we can describe a 50-unit square by the set of instructions:

DRAW 50 (units)
TURN 90 (degrees)
DRAW 50

TURN 90
DRAW 50
TURN 90

DRAW 50
TURN 90

Aside from the utility of this type of representation in developing an intuition for analytical geometry, an even more compelling reason to be interested in this descriptive process is that it makes sense to kids.

Consider the following two responses to the question, “Where do you live?”

Response 1: “I live at 1234 Upsey Lane.”

Response 2: “You go down this street for two blocks, turn right, and go down three houses to the one with the blue door and the oak tree in front.”

The first response, an address measured against a fixed reference, is typical of adults. The second response, typical of many youngsters, describes the procedure by which you would get to the house, given your present position and orientation. If you were in a strange city, you probably would find the second answer more useful than the first. Because each instruction is given with respect to the position and orientation of the participant at the end of the previous instruction, this descriptive procedure is identical to that used in turtle graphics.

Just as descriptive procedures make sense to children, so does the exceptional power of turtle graphics make it most valuable for illustrating important properties of geometrical figures (for example, curvature). Its similarity to natural descriptive language has made turtle graphics a most powerful vehicle in allowing children to discover important geometrical principles on their own.

My first exposure to turtle graphics came when I worked at Xerox’s Palo Alto Research Center—the home of the language SMALLTALK. While SMALLTALK supports an exceptional turtle-graphics environment, it was implemented on computer systems that were far too expensive for the average consumer. LOGO, an earlier language that

incorporates turtle graphics, was also designed to operate on large computers, although versions of this language for small machines are now becoming available. It was the publication of the book *Mindstorms—Children, Computers, and Powerful Ideas* by Seymour Papert (of LOGO fame) that finally convinced me a practical book should be written focusing on the actual use of the turtle-graphics environment as it exists on affordable computers.

To be useful, such a book must be written with a specific machine in mind. This version of the book is written for those who have access to an Atari 400 or Atari 800 personal computer with the PILOT cartridge. As turtle-graphics environments become available on other computers, we will endeavor to support these machines with books tailored to their specific capabilities and commands.

I have used several turtle environments over the years, including computer languages such as WSFN (Which Stands For Nothing) and “toys” like Milton Bradley’s Big Trak. My experience has shown that whatever the form—whether it’s an actual robot or a graphic display—turtle environments are especially liked by children.

I have been encouraged to write this book by many members of the research, product design, and development community who are actively generating user-friendly languages. I am especially indebted to Harry Stewart, who implemented the Atari version of PILOT (with its excellent turtle-graphics environment), and to Ted Kahn and others at Atari who got me involved in their effort to make this language a reality.

I am also indebted to Seymour Papert, Alan Kay, John Starkweather, and their colleagues and coconspirators in the quest for user-friendly computer languages.

But most of all I am indebted to those many children who have worked with portions of this book and who have experienced the utility of this language themselves. Were it not for the joyful inquisitiveness of children, this book would not exist.

Contents

PREFACE / 3

1. INTRODUCTION—WHY A BOOK? / 8

Why teach programming? / 9

2. GETTING STARTED / 11

How to get a computer / 11

How to get a kid / 12

3. TURNING PILOT ON / 13

A little surprise / 14

4. LET'S DRAW A SQUARE / 15

Undoing our first mistake / 22

5. LET'S DRAW SOME MORE / 25

How clear is CLEAR / 27

A sudden storm / 31

The turtle's TURNT0 turn / 31

How big a number can PILOT handle / 34

And now for more on TURNT0 / 36

Some even better shorthand / 39

Going beyond squares / 39

Grand finale / 43

6. MODULES: BUILDING THE TURTLE'S DICTIONARY / 44

And now for something really different . . . / 45

Building a dictionary—our first experience with modules / 50
Bigger modules for fancy pictures / 55

7. MODULES USING MODULES / 67

8. MODULES USING VARIABLES / 75
The turtle's windmill / 82

9. THE VARIABLE VARIABLE / 89
The turtle's star performance / 93
The turtle asks a question / 104

10. SQUARES AND SPIRALS / 108
The growing square / 112
Spirals and spirals / 117

11. DRAWING CURVES / 131
The turtle draws a circle / 131
Parts of circles—the turtle's arc / 141
Spirals / 152

12. THE LAST ONE / 159
The turtle has its FILL / 159
A flower for our turtle / 168

APPENDIX: Module Listings / 179

1

introduction —why a book?

THIS IS A book about tools and people. Now that is a pretty broad topic, so, just to keep things manageable, I am restricting the tools to computers (specifically the Atari 400 or 800 running PILOT), and the people to kids.

The reason for picking the computer as the tool is that digital computers are fast becoming affordable items. They are showing up in schools and homes, with applications ranging from bookkeeping to games. But the most important property of the computer is that it can be made to do what *you* want it to do.

In order for you to make the computer perform the desired tasks, you must learn a special language that translates your desires into the machine's actions.

The computer language PILOT is a good one to start with because it is both powerful and easily mastered by a very important group of users—kids.

Now some folks have the wrong idea about what constitutes a kid. You might be a decade past retirement but still satisfy my definition of a kid by being curious about things and eager to learn and tinker and even do things wrong once in a while. Of course, there are lots of young people who are kids too; I think everything that goes on in this book should be understandable to them. If you are an adult and you don't understand something in this book, you could do worse than have a child explain it to you.

Why teach programming?

There comes a point in the life of most computer enthusiasts when the realization hits that there is more to this technology than escaping from mythical caves, blasting spaceships (or bricks), plotting biorhythms, or keeping holiday mailing lists. This is the point where the topic of “programming” comes up. There are probably more than one zillion books (give or take a few) on computer programming—most of which deal with a language called BASIC, and most of which try to teach you something without giving you the chance of seeing if you really want to know it.

PILOT is at least as powerful as BASIC, and in some areas it is much more powerful. But most important, PILOT programs are easily read and understood. This makes it possible for lengthy PILOT programs to be written by a team of authors who are virtually guaranteed of being able to read one another's “coding.”

Most of the people I know who want to learn about computers want to use the machine to solve problems, or to create pictures, or to invent games, or to do useful things like that. What they don't want to do is learn about variable types, loop structures, string arrays, and the like in full detail.

Now, if you already are a computer whiz who wants to learn about PILOT, you've accidentally picked up the wrong book. On the other

PICTURE THIS!

hand, if you want to learn something about making the computer into a useful tool, this just might be the right book for you.

Our approach to teaching PILOT is to have you pick it up as you use it. This is how young children learn language, and it works rather well. Most youngsters learn to express themselves just fine without ever knowing about verbs, subjects, clauses, and the like.

By now it is probably pretty clear that this is a learning-by-doing book. One problem with learning by doing is that you are likely to make some mistakes—the equivalent of, for example, “Throw my mother from the train a kiss.”

If you look at mistakes properly (that is, without being anxious about them), you can learn more from making some than you can from everything’s always working.

People who work with computers make mistakes all the time—they call the mistakes *bugs*. A program that has a few bugs in it can usually be fixed. The fixing process is called *debugging*. So don’t feel bad about making mistakes when you’re using the computer; it’s how you will learn a lot of handy things.

Q: What is a computer language, anyway?

A: Whoops! I guess I forgot to talk about that. Well, a computer language is a special piece of ‘software’ that acts like a translator between instructions that are easy for you to understand and instructions that the computer carries out. If we didn’t have computer languages, it would be hard for most of us to make computers do anything useful at all.

Q: If the language is “software,” that means it is a computer program itself, doesn’t it?

A: Yes—and that means it might have some bugs in it too!

getting started

TO GET STARTED with this book you will probably need two things:

1. An Atari 400 or 800 computer with the Atari PILOT language cartridge.
2. A kid.

How to get a computer . . .

If you don't have an Atari computer handy, you might check your local library or pizza parlor to see if they have one. If they don't, tell them to write to ComputerTown, USA! (c/o People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025) right away—your community is missing out on a lot of fun! You might want to call your neighbors to see if they can help out. As a last resort, you could always buy a system and help your local computer store stay in business.

PICTURE THIS!

How to get a kid . . .

Look around the house. If you see a bundle of pure energy wrapped in sand with some marbles and a frog in his or her pocket, congratulations, you've got yourself a kid.

You might also check yourself out to see if you are a kid—if you are, then you're all set.

A final note on kids and computers. You should be sure you're helping your kids keep their priorities straight. If your kid is playing in the sand or making mudpies with a friend, he or she is doing the kinds of things that are far more important than anything you are going to do with the computer. In general, kids shouldn't be interrupted—especially when they are busy with other kids.

On the other hand, you might sometimes hear:

“Gee, I'm bored! There's nothing to do around here.”

When this happens, you know the time has come for your Robin or Tracy or Johnny to discover (with you) the power of a truly neat tool—the computer.



turning pilot on

I AM GOING to assume that you already have your Atari computer hooked up and have been using it for a while. If you haven't used the computer before, you should use the Atari instruction books to check everything out.

Most of the things we will do don't require a tape recorder or a floppy disk system for saving programs, but both of these are handy accessories. You, very likely, will want to save some of your work later on.

First turn on your television. (I will assume you are using a color set, although you can use black and white just as well.) Make sure the Atari PILOT cartridge is plugged into the left cartridge slot on the Atari 800 or into the only cartridge slot on the Atari 400 computer. Close the cartridge lid until it "clicks" shut, and then turn the computer on. If everything is working all right, there will be a slight pause and then—*Ta Daa!*—the TV screen will show:

ATARI PILOT (C) COPYRIGHT ATARI 1980
READY

PICTURE THIS!

If you don't see this message, or if you see ATARI MEMO PAD instead, turn the computer off and make sure the cartridge is pressed properly into place before trying again.

As you look at the image on the TV screen, you might notice a white, solid rectangle just under the R in READY. This rectangle is the *cursor*. It always shows where the next letter will appear on the screen when something is typed into the computer. You can move the cursor around on the screen using the cursor control keys. By using this feature you can even correct incorrectly-typed words. To learn how to do this you should read the instructions on using the "Screen Editor" that came with your PILOT cartridge.

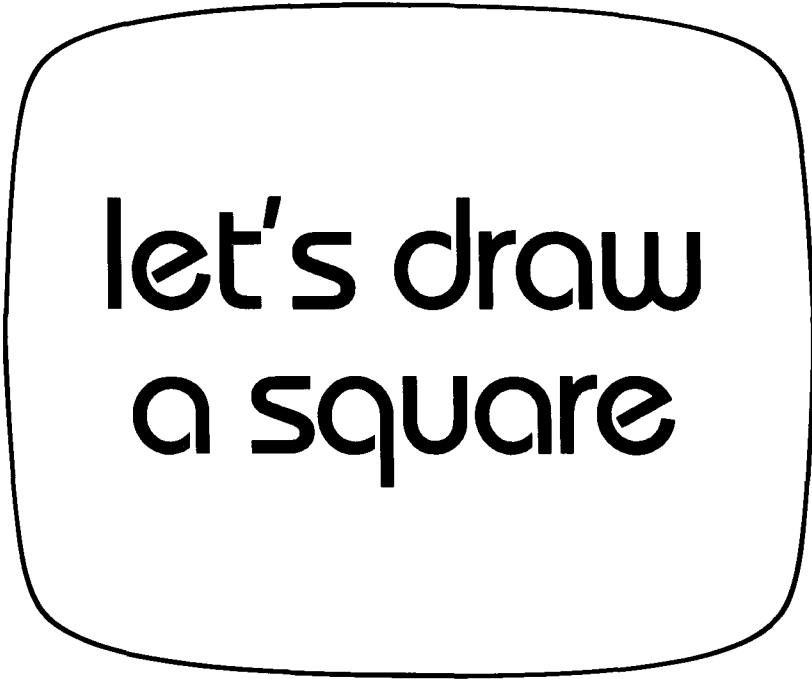
A little surprise . . .

Let's say everything has gone according to plan this far, and then the phone rings. It is your Aunt Mabel from Minneapolis, and she hasn't spoken with you for ages. If, when you return from your little talk, you have been away from your computer for more than a few minutes, you might be quite surprised to see what is happening on the TV screen. The screen is changing color every few seconds! Atari makes their computers do this so your TV screen won't be damaged if you, for instance, accidentally leave the computer on overnight.

To make the computer go back to displaying the nice, blue background, just press any key (the SPACE bar, for example).

Now that you know everything is working, we are ready to give the computer something to do. Of all the things we can do with Atari PILOT, I have picked graphics as the area of concentration for this book. The reasons for this are:

1. It is easy to see what is going on in a graphics program, and you learn a whole lot about programming in the process.
2. It looks pretty.



let's draw
a square

IN ORDER FOR the computer to draw some pictures, we first have to “clear” some space on the TV screen. To do this, you should type the following:

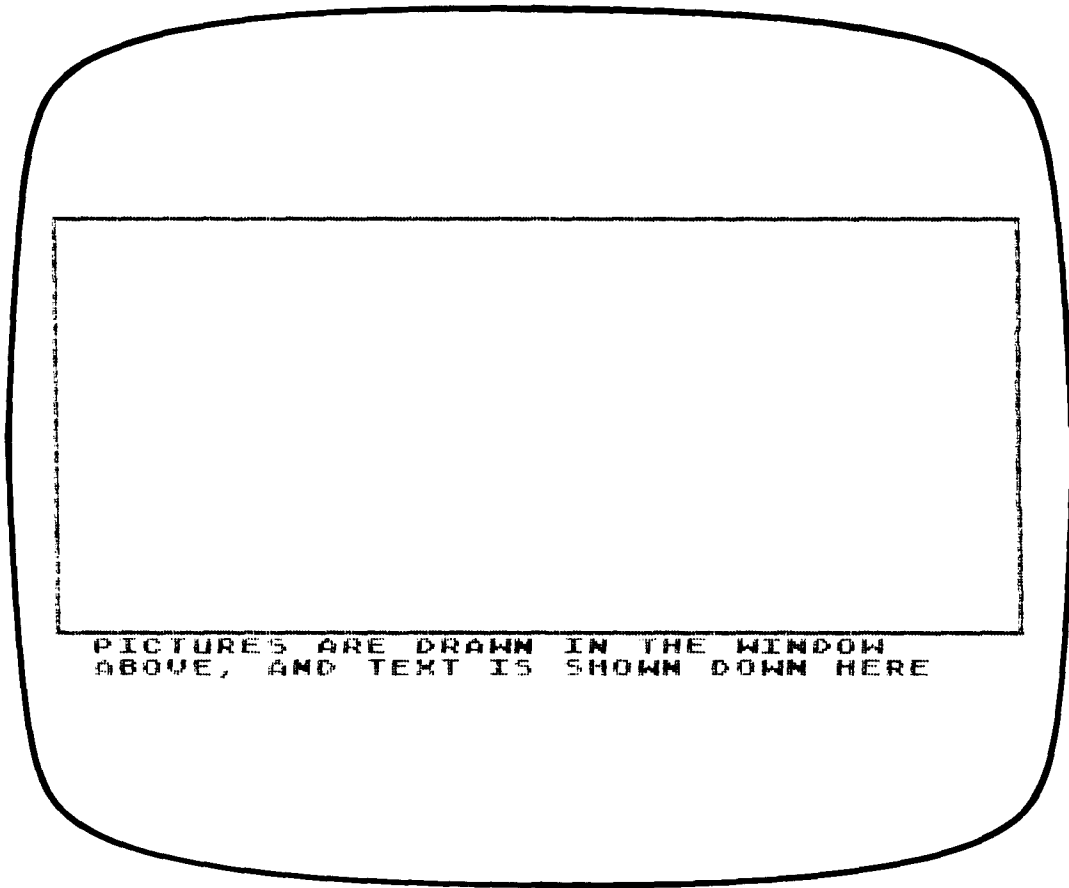
GR: CLEAR

Notice how the white, rectangular cursor moves along as you type. The cursor shows you where the letters are going to appear when you type something. If you make a mistake in typing, you can fix it by pressing the backspace key and retyping over the error (fancier ways of fixing typing mistakes are shown in the PILOT manual that came with your cartridge).

After you type GR: CLEAR, you should press the RETURN key. Actually, you should press the RETURN key after you type *any* command to the computer since this lets the machine know you are ready for it to do something.

PICTURE THIS!

Once you press RETURN, the display on the TV screen changes quite a bit. In fact, it should be divided into two areas like this:



Notice that the screen now has two “windows.” The big black area at the top of the screen is where our pictures will be drawn. The smaller blue area at the bottom is where our typed instructions to the computer can be seen.

Q: I think I know that CLEAR cleans up the screen, but what does GR: mean?

A: The PILOT language is set up so that the very first item on each line is an instruction telling the computer what kind of operation is going to be performed. GR, for instance, means we are using a *G*Raphics instruction. The colon (:) separates this operation from the specific task we want to have accomplished. (In this case, CLEAR the screen.)

So far all we have done is get a clear screen—not terribly exciting. Next we want to draw some lines on the screen. Every computer-graphics language has its own way of drawing lines. The method chosen for Atari PILOT uses something called a *turtle*.

You should think of the turtle as a computer creature that carries a pen with which it can draw lines as it moves along. When you tell the turtle where you want it to go, it goes there in a straight path. If you wish, the turtle can hold the pen down and leave a mark on the screen as it moves. Or, if you don't want it to draw a line, it can pick the pen up. Our turtle even has three different-colored pens to choose from as well as an erase pen.

(Please realize that even though we use phrases like “tell the turtle to draw a line,” we don't want you to think that the turtle, or the computer in which “it” resides, really understands anything. It is not our object to give human capabilities to the computer. Instead, we use this type of language as a convenient shorthand for more cumbersome ways of saying the same thing. The turtle does not really exist—it is just a useful model for describing how the PILOT graphics commands work.)

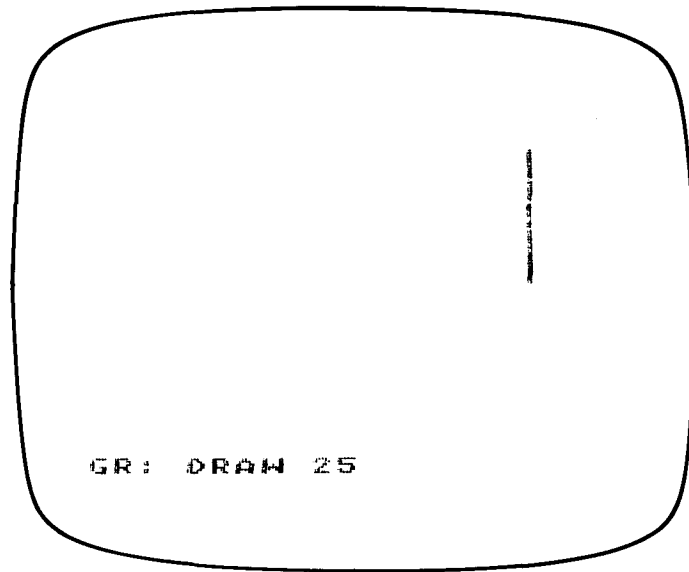
When you first clear the screen and set up the graphics window, the turtle is located near the middle of the screen, is pointing straight up, and is holding a yellow pen.

Now let's watch the turtle draw a line. Type the following (then press RETURN):

GR: DRAW 25

PICTURE THIS!

As soon as you press RETURN, you should see a yellow line appear on the screen like this.



You may not actually see the line being drawn because, for a turtle, our friend is very fast.

Here's what happens when you issue the command. GR: tells PILOT you are issuing a graphics command. DRAW tells the turtle to move with the pen down, and 25 tells the turtle to move 25 screen units.

The Atari screen is set up to be 78 units high and 158 units wide. You can have the turtle walk off the edge of the screen, but it's easy to lose it this way, so keep the screen boundaries in mind when sending the turtle off on a trip.

Now let's make the line a little longer. Type the following:

```
GR: DRAW 5
```

Let's Draw a Square

Did the yellow line get a little longer? (Did you remember to press RETURN?)

The turtle always stays where you last left it—it doesn't walk off on its own.

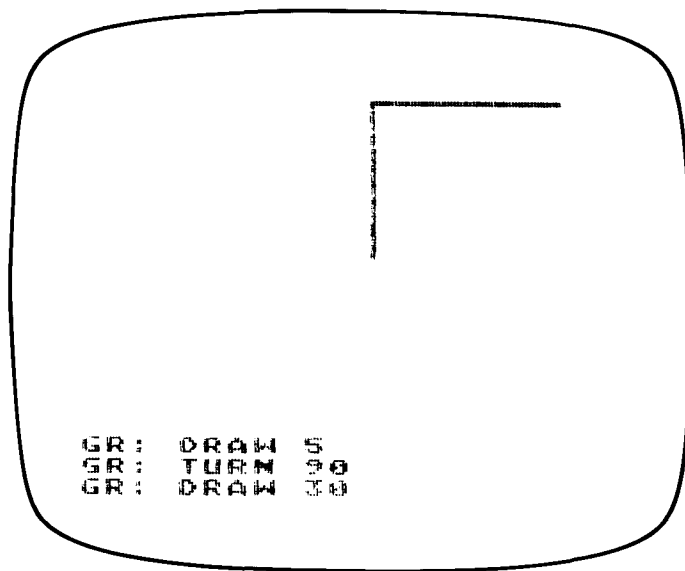
Perhaps you would like to change the turtle's direction. If you now type:

```
GR: TURN 90
```

the turtle turns 90 degrees to the right, as soon as you press RETURN. (The amount you turn when you go around a corner is 90 degrees.) You don't see anything happen on the screen because the turtle is invisible. You can see that it has turned, however, by having it move some more:

```
GR: DRAW 30
```

Now your screen should look like this:



PICTURE THIS!

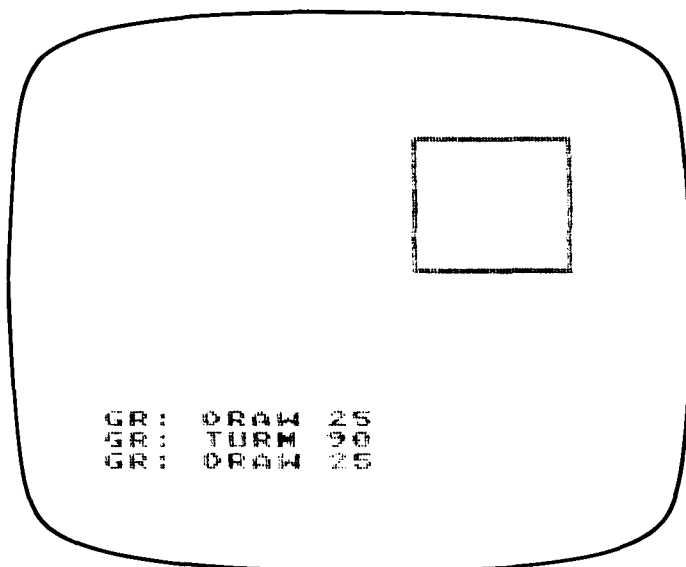
As you can see, by turning the turtle 90 degrees, we made it ready to go around a corner.

If you know that a square has four sides of the same length and has four 90-degree corners on it, you probably have already figured out how to make your turtle draw a square. Here is one way. First, press the SYSTEM RESET button (in the upper right corner of the keyboard) to reset PILOT. This also takes us out of the graphics "mode," so you should see an all blue screen with the word READY in the upper left corner.

Next, type:

```
GR: CLEAR
GR: DRAW 25
GR: TURN 90
GR: DRAW 25
GR: TURN 90
GR: DRAW 25
GR: TURN 90
GR: DRAW 25
```

If you have typed all these things into the computer correctly, on the screen you should now see a yellow square like this:



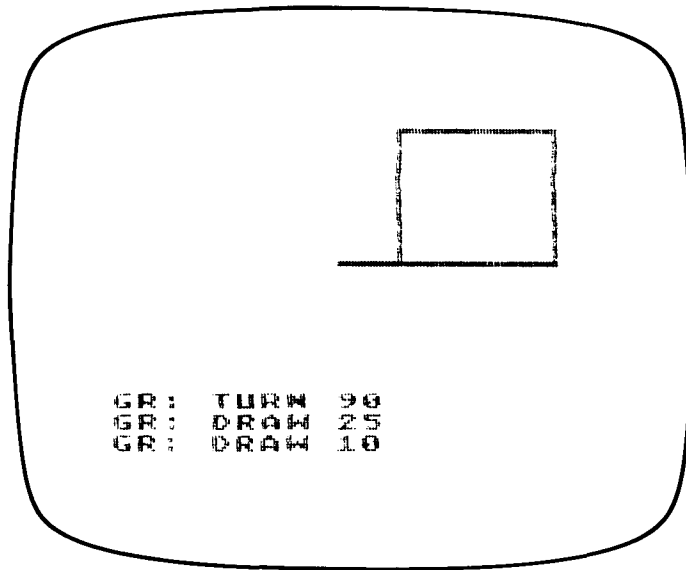
Let's Draw a Square

Congratulations! You just made the turtle draw a square.

Next, without erasing the screen, draw another square (with sides 10 units long) on top of this one. For our first command let's type:

```
GR: DRAW 10
```

Oh-oh! What happened? Does your screen look like this?



Hmmm, it looks as if this new square is starting off in the wrong direction. Why do you suppose this happens?

Well, if you look at the instructions for our first square, it appears that we became so excited with finishing our square that we forgot to TURN the last corner!

An easy way to see this is what happened is to pretend you are the turtle. Try walking in a square following the same instructions we gave to the turtle (but you might want to walk fewer than twenty-five steps on each side). Did you notice that, although you ended up in the same place you started, you were pointing in the wrong direction? This was our mistake.

PICTURE THIS!

Undoing our first mistake . . .

At this point we could start all over, but as you will soon see, we don't need to do that. If you were drawing this square on paper with a pencil and you made a mistake, you could erase the wrong line. Well, our versatile turtle has an eraser too! To get the turtle to use it we just type:

GR: PEN ERASE

and then press RETURN. This instruction tells the turtle to change its present pen to the special ERASE pen it keeps handy for emergencies.

Next we need to make the turtle go back 10 units, erasing its line as it goes. Do you think the turtle understands BACKWARDS? Let's find out:

GR: BACKWARDS

Oh boy, now what? Your screen probably shows:

GR: BACKWARDS

WHAT'S THAT

with the B in BACKWARDS shown in reverse field (blue on a white background). This is PILOT's way of telling you it doesn't understand the command you gave it. As usual, no harm is done, so we can proceed with our task.

One way of going backwards is to turn a 90-degree corner twice. So if we type:

GR: TURN 180

this should work since 180 is twice as large as 90.

Next, let's type:

GR: DRAW 10

Since this is how far we went in the wrong direction, and —*Ta Daa!*— the wrong line is (almost) erased. There seems to be one dot left to remind us of our original error. The reason for this is that when we turned the turtle around and started it moving, it missed the last point. We needn't worry about it now, though; we will fix this problem later.

If you are keeping notes (or playing turtle), you probably notice we still aren't pointing the right way. To do this we can type:

GR: TURN - 90

The minus sign in front of the 90 means to turn left instead of right. If everything has gone according to plan, the turtle should be pointing straight up, just as it was when we first started.

To draw our smaller square, we must remember that the turtle is still holding the eraser. Since this isn't too useful to us, let's change the pen again:

GR: PEN RED

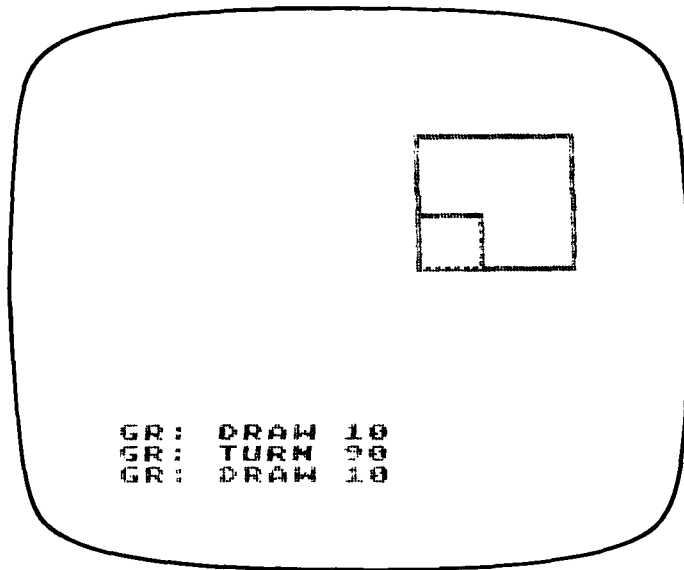
Now the turtle has a red pen. Let's see if you can make the turtle draw a square with sides 10 units long.

If you typed this:

GR: DRAW 10
GR: TURN 90
GR: DRAW 10
GR: TURN 90
GR: DRAW 10
GR: TURN 90
GR: DRAW 10

PICTURE THIS!

you should have a picture that looks like this:



Now let's clean up that last trace of the mistake:

```
GR: PEN ERASE  
GR: DRAW 10
```

And we're all done. Not bad for your first session with the turtle!



let's draw
some
more

SO FAR WE have learned quite a bit about our graphics turtle. We know how to make it DRAW, TURN, and change pens. We even found out that it has a special pen that works like an eraser.

Now I don't know about you, but I'm getting a little tired of having the turtle do only one thing for each line we type. Of course we, too, sometimes do only one thing per sentence ("Go home"), but we also use sentences that have us do more than one thing ("Go home and eat dinner"). Well maybe we should see if the turtle knows how to understand longer "sentences." Let's try one out.

Press SYSTEM RESET and type:

GR: CLEAR

GR: DRAW 20 AND DRAW 5

Whoops! That didn't do any good, did it?

PICTURE THIS!

Since your screen probably shows

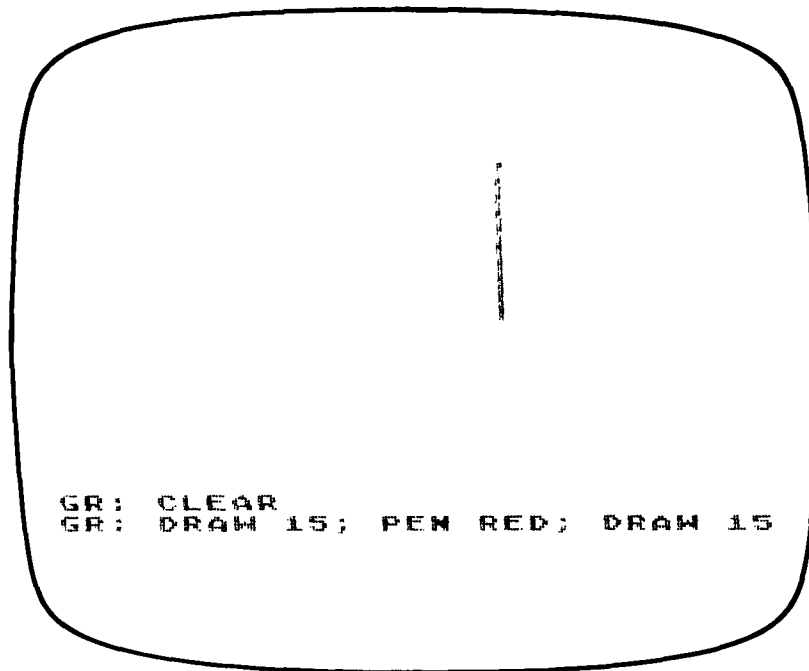
```
GR: DRAW 20 AND DRAW 5
*** WHAT'S THAT ***
```

with the A in AND reversed, I guess the turtle doesn't understand the word AND. Hmmm, there *must* be a solution to this.

In English there is another way of connecting several ideas in the same sentence, and that is with the semicolon (;). Maybe our turtle friend will understand this symbol. Let's try it!

```
GR: CLEAR
GR: DRAW 15; PEN RED; DRAW 15
```

Wow! Did you see that? Your screen should show a line that changes colors halfway along it like this:



See, the turtle is really quite accommodating after all—once we figure out how its language works.

How clear is CLEAR . . .

Let's do a little experiment. Repeat the previous two commands:

```
GR: CLEAR  
GR: DRAW 15; PEN RED; DRAW 15
```

Are you surprised at the result? The first thing you probably notice is that, instead of getting a two-colored line, the line is entirely red. If you look a little closer you might notice that this line starts in a different place than the previous line. It seems that

```
GR: CLEAR
```

erases everything on the screen without changing the turtle's pen color or location at all.

You already know that

```
GR: PEN YELLOW
```

will “reset” the turtle's pen to the original color. It would be equally nice to move the turtle into its original home near the middle of the screen and to make sure it is pointing straight up without our having to push SYSTEM RESET (which accomplishes the same thing).

Now that you are thoroughly expert at DRAW and TURN, you probably have already figured out how to accomplish the turtle's homecoming using these two commands—so long as you remember how far the turtle is away from its home, you should have no trouble. I can never remember these things myself, so it's lucky for me (and perhaps for you) that there is a command that lets us “pick up the turtle” to move it and another command that lets us “point” it in some specific direction.

If you type

```
GR: GOTO 0,0
```

PICTURE THIS!

you are instructing PILOT to pick the turtle up from wherever it is and to move it into its home. (When you type the command GOTO, be certain there is *no* space between the GO and the TO.) The numbers 0,0, which follow the command, tell PILOT where the turtle should be moved. The first number determines its horizontal location, and the second number determines its vertical location.

Q: Why is home at 0,0. What do these numbers really mean?

A: In the last chapter we said the graphics screen is 158 units wide and 78 units high. By putting “home” near the middle of the screen and calling these home positions 0 (for the horizontal location) and 0 (for the vertical location), we can move the turtle to any place on the screen simply by using numbers larger than 0 for locations to the right of, or higher than, home and by using negative numbers for locations to the left of, or lower than, home.

Let’s do some experimenting to see how this works. First type the following:

```
GR: CLEAR
GR: PEN YELLOW
GR: GOTO 0,0
```

You should now see a yellow dot near the middle of the screen. This is where the turtle is located. Next type:

```
GR: GOTO 10,20
```

Where did the turtle move to? Do you see a new yellow dot a little to the right and a little higher than the first one? We have picked the turtle up and moved it to this new location.

Now let’s try some more locations:

```
GR: GOTO -20,0
```

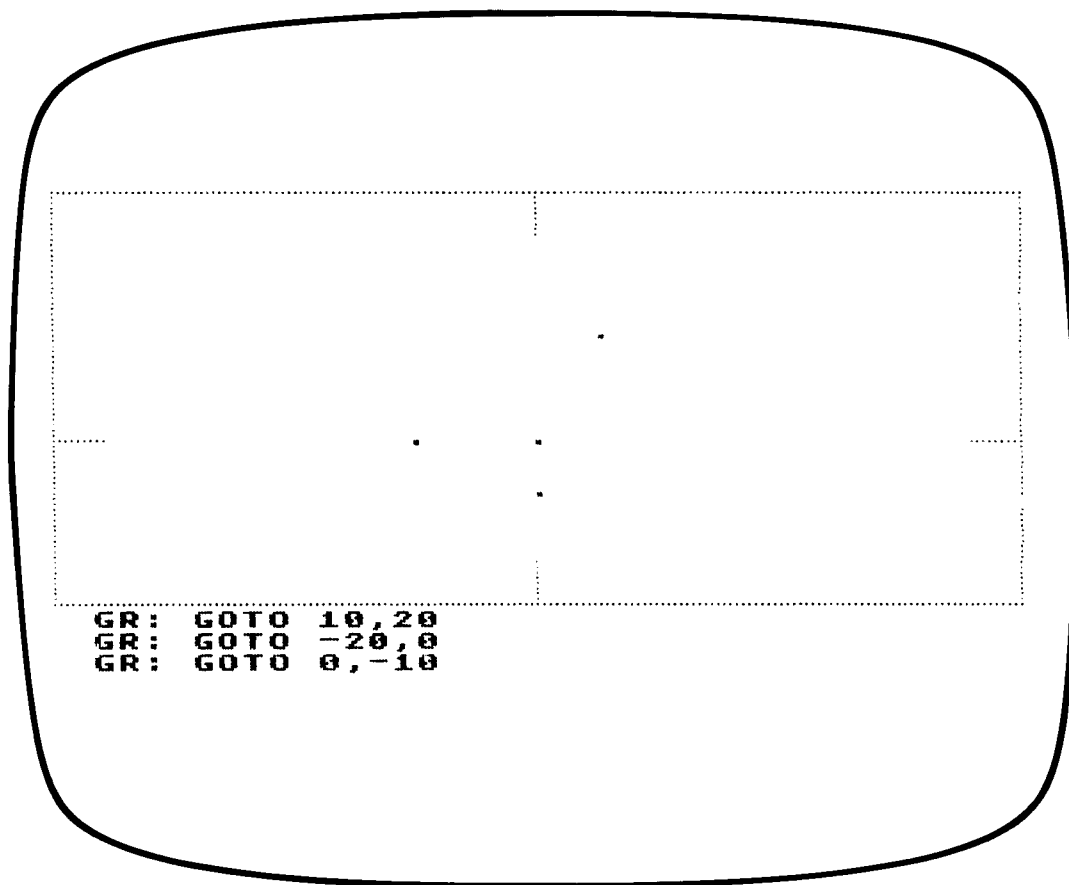
The turtle is in the home “row,” but it is 20 units to the left of home.
Try this:

```
GR: GOTO 0, -10
```

Another yellow dot now shows that we are in the home column, but 10 units below the home row. (Remember that columns go up and down and rows go sideways.)

The figure below shows the dots you should have on the screen along with the limits to which you can use to move the turtle in each direction and still have it on the screen.

Conduct an experiment to find the limits for the row and column coordinate values which still create visible points.



PICTURE THIS!

Now that you know how GOTO works, why don't you use this command to give your screen a case of the yellow polka dots! Here are a few dots to get you started:

```
GR: GOTO 5,5; GOTO -20,10; GOTO 17,20
```

Well, now you are an expert on GOTO, right? Maybe? Hmmm, let's see. Try the following:

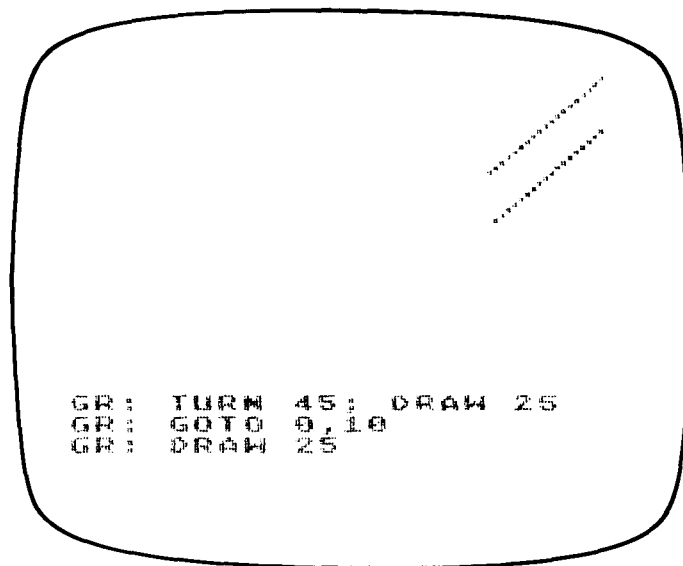
```
GR: CLEAR  
GR: GOTO 0,0  
GR: TURN 45; DRAW 25
```

You should now have a yellow line going off to the right at a diagonal. (Don't worry about the "jaggies"; all lines at angles other than vertical or horizontal will have this bumpiness to some extent.) Next type:

```
GR: GOTO 0,10
```

Now, before you type anything, try to guess what will happen if you give the command

```
GR: DRAW 25
```



What is your guess? Will you get a vertical line, a diagonal line, or what? Is the suspense getting to you? Give up? OK, try it:

```
GR: DRAW 25
```

Well, well—it appears that when PILOT moves the turtle in response to a GOTO command, the “mover” (the PILOT software) is very careful not to turn the turtle around at all. Unless we TURN the turtle, each line we draw will run in the same direction as the previous line, no matter where we locate the turtle first.

A sudden storm . . .

Have you ever looked out a window on a very rainy day and seen the rain coming down in streaks? If the wind is blowing, the rain streaks down at an angle. Now that you know about GOTO, DRAW, TURN, and parallel lines, I'll bet you can make a nice picture of a yellow (or red or maybe even blue) rainstorm on your TV screen.

The turtle's TURNTO turn . . .

So far we have learned how to pick the turtle up and set it down anywhere on (or off) the screen. As you've seen though, moving the turtle doesn't cause it to turn one bit. If it is facing northeast before the move, it stays pointing in that direction afterwards as well.

Fortunately, PILOT gives us another tool to help us get the turtle headed in any direction we want. Try the following:

```
GR: GOTO 0,0; TURNTO 0; CLEAR  
GR: DRAW 25
```

PICTURE THIS!

No surprises, right? Next, type:

```
GR: GOTO 0,0; TURNT0 90
GR: DRAW 25
```

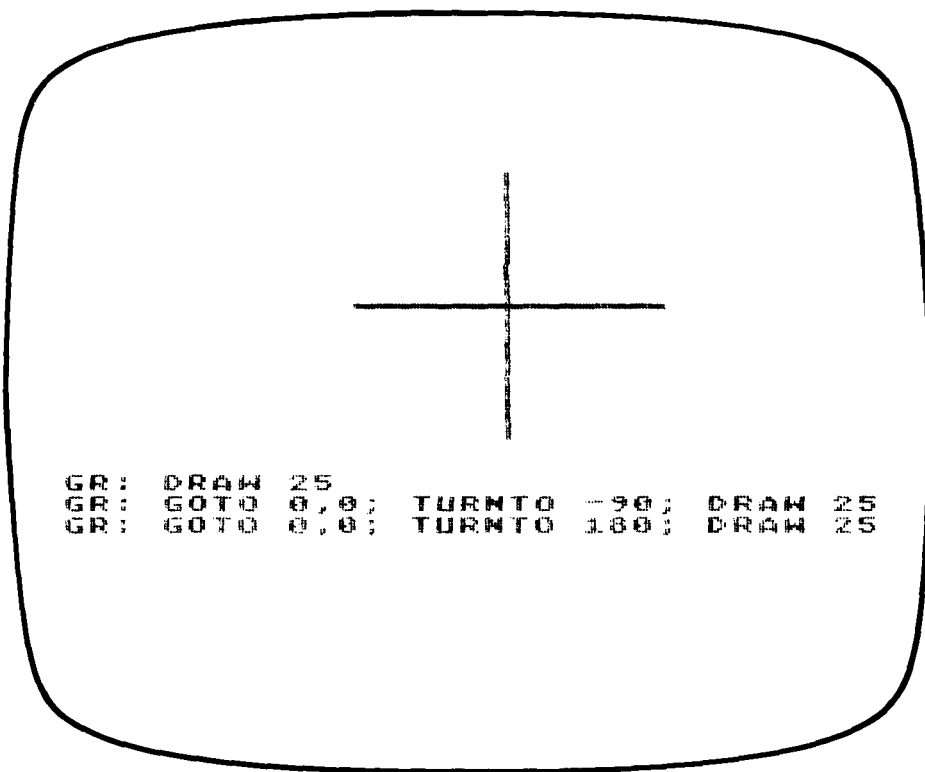
Do you understand what happens? When you draw the first line, you put the turtle in its home and make certain it is pointing straight up. The TURNT0 command turns the turtle to any angle you want, with the reference point (0 degrees) being straight up. When you type TURNT0 90, you tell the turtle to face the right side of the screen. This is why the second line goes to the right. Now you know what this command will do:

```
GR: GOTO 0,0; TURNT0 -90; DRAW 25
```

How about this one:

```
GR: GOTO 0,0; TURNT0 180; DRAW 25
```

By now you probably have a big yellow plus sign on your screen that looks something like what you see in the figure below.



Let's Draw Some More

What do you suppose happens when you use bigger numbers in the TURNTO command? Let's try some to see.

First we should change the pen to RED (to show we are about to embark on a dangerous mission):

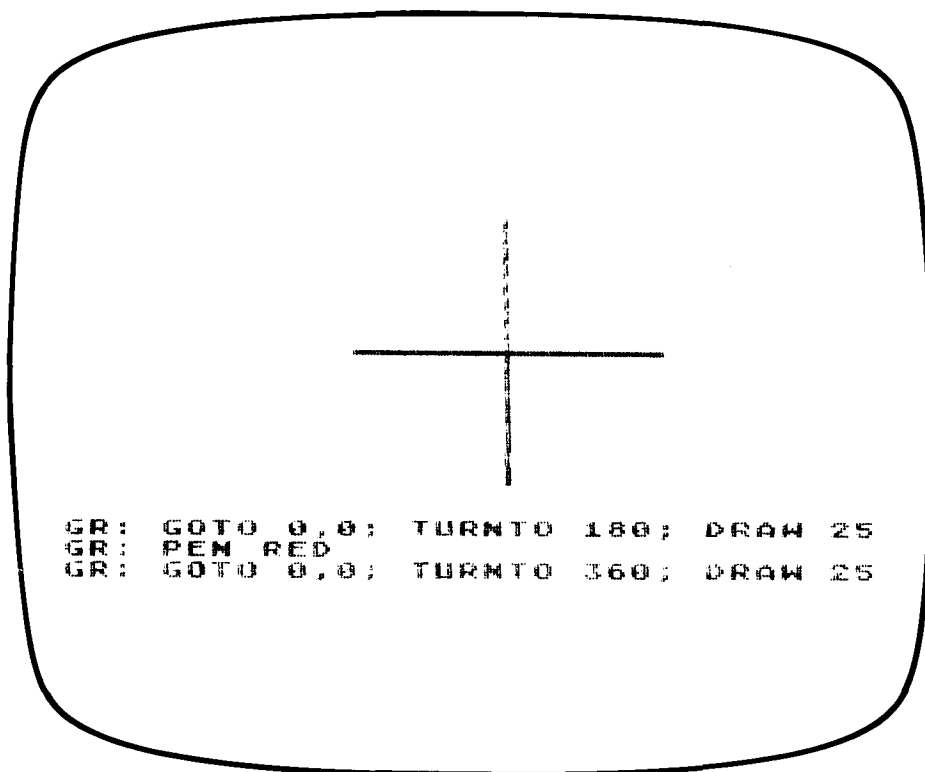
```
GR: PEN RED
```

Now type (but *don't* press RETURN):

```
GR: GOTO 0,0; TURNTO 360; DRAW 25
```

Before you press RETURN, can you guess what is going to happen? Now press RETURN to see if you were right.

If everything went according to plan, your screen should show the image below.



The first yellow line we drew (when we said TURNTO 0) has been replaced by a red line. This must mean that when we TURNTO 360 we 33

PICTURE THIS!

have gone in a complete circle and arrived at the same place we would get to with TURNTO 0. Now that is a pretty useful discovery (even if you already knew it).

We can check this out some more. Since $360 + 90$ equals 450, you probably have already figured out what will happen when you type:

```
GR: GOTO 0,0; TURNTO 450; DRAW 25
```

No surprises? Good! Next let's try some even bigger numbers:

```
GR: PEN BLUE
```

```
GR: GOTO 0,0; TURNTO 720; DRAW 25
```

Wow! Our vertical red line is replaced by a blue one. If you guessed this is because 720 equals $360 + 360 + 0$, you are right. Each time we add (or subtract) 360 to the turn, we end up where we started from. This property of TURNTO is *quite* important, as we will see later.

How big a number can PILOT handle . . .

If you have played around with PILOT very much, you might have tried using some *really* big numbers and come up with a surprise rather than the expected result. For example, let's suppose you want to make the turtle spin *many* times before settling down. Try this:

```
GR: CLEAR
```

```
GR: GOTO 0,0; TURNTO 0; PEN YELLOW
```

```
GR: DRAW 25
```

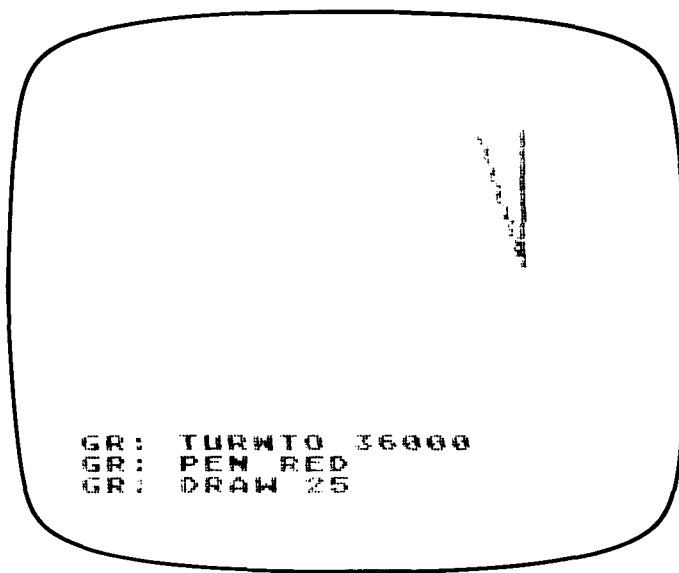
```
GR: GOTO 0,0
```

```
GR: TURNTO 36000
```

This last command should make the turtle turn around one hundred times. Next type:

```
GR: PEN RED
GR: DRAW 25
```

Oh boy! Something sure went wrong. We should have seen a red line pointing straight up, but instead we saw a line going off at an angle to the left.



The reason for this error is that we asked PILOT to handle a number that's too large. How large is too large? The Atari PILOT can't handle single numbers larger than 32767 or smaller than -32768.

Q: What is so magical about these numbers?

A: This is going to be a fairly technical answer, so feel free to skip it if you wish. Still here? OK—Here goes.

Unlike human beings, computers cannot work with numbers containing an arbitrary number of digits. Imagine how confused you would become if someone asked you to calculate

PICTURE THIS!

something with the number 86436 and you didn't know how to count past 100. While we work with decimal numbers (and each of our digits can have any value from 0 through 9), the Atari computer—when running the PILOT language—only works with numbers of fewer than sixteen *binary digits*, (and its digits can have only two values, 0 or 1). We humans can count as high as we wish, but PILOT can only work with numbers of sixteen or less binary digits. Each place in the computer's numbering scheme is called a *bit*, so we can talk of sixteen-bit numbers. Now, it so happens that if you decide you're going to have a range of numbers balanced around zero, the largest decimal number you can make with sixteen bits is 32767. When PILOT comes across a number larger than 32767, it subtracts 65536 from the number as many times as it needs to in order to bring the number within the -32768 to 32767 range. Sorry to have to be so technical, but at least you know why you should keep your numbers within the range from -32768 to 32767 .

If you skipped the explanation (or didn't understand it), don't worry. Just remember to keep your numbers in the correct range and everything will be just fine.

And now for more on TURNT0 . . .

Q: What is the difference between TURN and TURNT0?

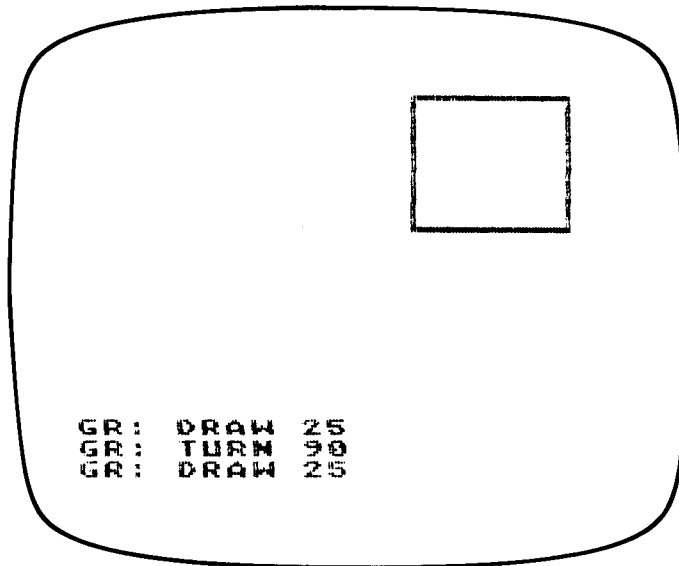
A: The command TURNT0 always turns the turtle *to* a specific angle, while the command TURN always turns the turtle *by* a specific angle. The TURNT0 command is *absolute*—always turning against a fixed or absolute reference. In our case the reference is that 0 degrees corresponds to straight up. The TURN command, on the other hand, is *relative*—always turning by some amount relative to or dependent on the turtle's present location.

I have an idea, let's have the turtle show us the difference.

First try this:

```
GR: CLEAR
GR: GOTO 0,0; TURNT0 0; PEN YELLOW
GR: DRAW 25; TURN 90
GR: DRAW 25; TURN 90
GR: DRAW 25; TURN 90
GR: DRAW 25; TURN 90
```

I don't know about you, but I just saw a yellow square that looks like this:



Next let's try:

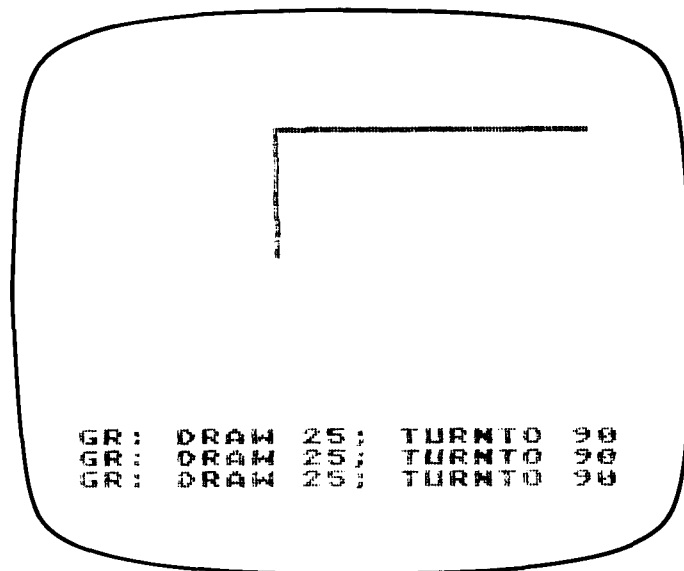
```
GR: CLEAR
GR: DRAW 25; TURNT0 90
GR: DRAW 25; TURNT0 90
```

So far so good?

```
GR: DRAW 25; TURNT0 90
```

PICTURE THIS!

What happened? Instead of turning, the turtle continued to move to the right like this:



Now you are probably thinking that if you added 90 to each TURNT0, you would get a square. This would make each TURNT0 command different (the first one would be 90, the second 180, and so on). So as you can see, both TURN and TURNT0 are quite useful, although they behave quite differently.

Q: If the turtle understands TURN and TURNT0, does that mean it understands GO (since it understands GOTO)

A: Yes. (My but you are learning fast!) Just as TURNT0 and GOTO are absolute, TURN and GO are relative. This means that GO will make the turtle move (without drawing anything) by a certain amount in whichever direction it was originally pointing. The only difference between GO and DRAW is that the turtle lifts its pen up before GOing and puts it down before DRAWing.

Q: Aha!

A: Now what?

Q: Does this mean that the turtle understands DRAWTO?

A: Why don't you try it and find out?

Some even better shorthand. . .

Are you getting tired of typing the same things over and over again just to draw a square? Fortunately, PILOT allows another bit of shorthand that will help us here. To figure it out, think about how you would tell a turtle to draw a square.

Would you say—

“Draw a line 25 units long, turn right, draw another line 25 units long, turn right again, draw yet another line 25 units long, turn right, draw still another line 25 units long, turn right and stop.”

I'll bet you're far more likely to say something like—“OK, turtle, I want you to do this four times: draw a line 25 units long and turn right.”

Now I must admit that neither of these approaches worked very well when I tried them on a live turtle, but we should see how the PILOT turtle likes them before passing judgment. First clear the graphics screen and make sure the turtle is home and pointing up. Next type:

```
GR: 4(DRAW 25; TURN 90)
```

Hey! How about that! The turtle drew a square. Can you see what happened by looking at the line you typed? This command tells the turtle to do something four times. Whatever you want the turtle to do is typed between the two parentheses. In this case, it was told (with only one command) to repeat DRAW 25 and TURN 90 four times. That sure is a handy way to get the turtle to do something a whole bunch of times.

Going beyond squares . . .

Squares are useful and pretty, but there are other closed figures that you might want to draw. A closed figure whose sides are each of

PICTURE THIS!

equal length is called a regular polygon. A square is an example of a regular polygon. Let's find some more.

Clear the screen, and put the turtle home and facing up. Now let's do an experiment. Type:

```
GR: 4(DRAW 15; TURN 90)
```

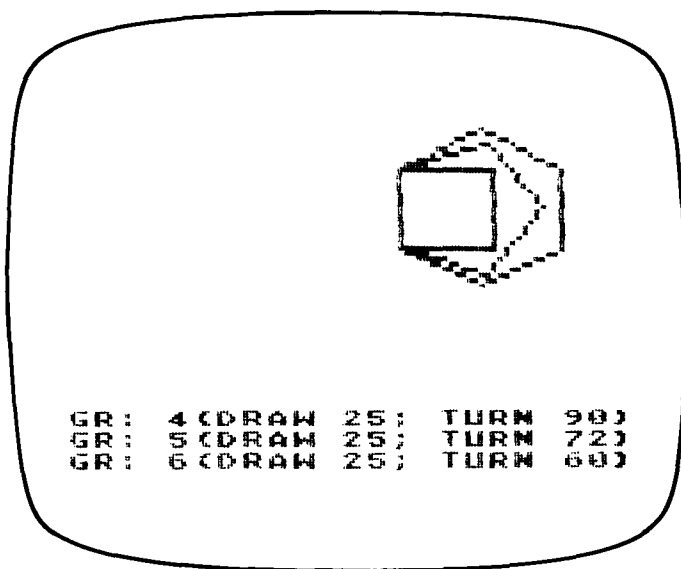
You now have a square. Next try this:

```
GR: 5(DRAW 15; TURN 72)
```

Wow! You just added a five-sided polygon to your picture. (A five-sided polygon, as you may know, is called a pentagon since *penta* is the Greek root for "five.") Next type:

```
GR: 6(DRAW 15; TURN 60)
```

The new figure on your screen is called a hexagon.



Q: If you are going to put confusing words in this book, why don't you just give us a table of names so we can find the right word when we need it?

A: OK, here goes:

<i>Number of sides</i>	<i>Name of the figure</i>
3	triangle
4	square
5	pentagon
6	hexagon
7	heptagon
8	octagon
9	nonagon
10	decagon

Q: Not bad for starters, but what do you call a polygon with 137 sides?

A: Freddy.

Now if you look closely at the three sets of instructions we gave, you may see two things. First, the difference between drawing a square, drawing a pentagon, and drawing a hexagon, is the number of times you draw lines and turn (four for the square, five for the pentagon, and six for the hexagon). Secondly, the square is made by turning 90 degrees at each turn, the pentagon by turning 72 degrees, and the hexagon by turning only 60 degrees.

From this we may gather that the more sides there are to a polygon, the fewer the degrees of each turn.

Whenever you come across an observation like that, you should try to discover if there is a rule that could tell us how to make any regular polygon.

Let's try adding the angles as we go around each of these three polygons to see what that might tell us.

Square: $90 + 90 + 90 + 90 = 360$

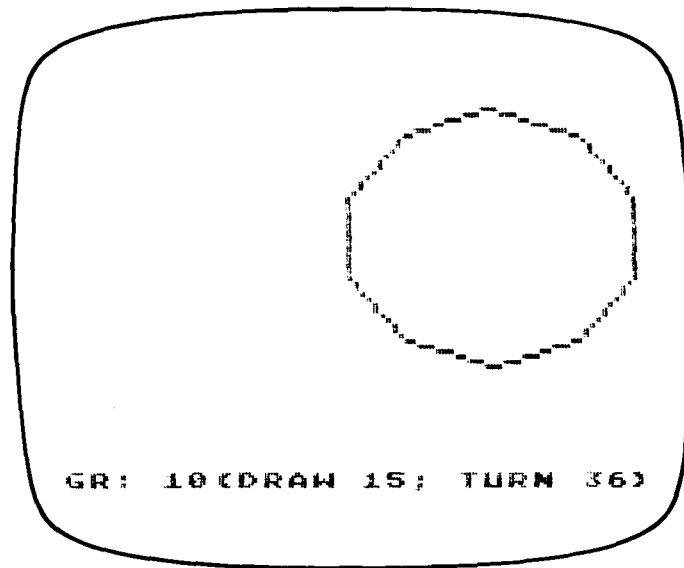
Pentagon: $72 + 72 + 72 + 72 + 72 = 360$

Hexagon: $60 + 60 + 60 + 60 + 60 + 60 = 360$

PICTURE THIS!

This looks promising. It looks as though we are on the road to discovering a rule that says the sum of all the turns in a regular polygon is equal to 360 degrees. Now *that* is an easy rule to test out. Let's pick a number (Did you say ten?) and try the rule out. If we want ten turns to equal 360 degrees, each should be 36 degrees. Clear the screen and try this:

GR: 10(DRAW 15; TURN 36)



Now that's encouraging! We seem to have found a rule. Let's call it "Turtle Rule #1."

TURTLE RULE #1: The sum of the turns made while drawing a regular polygon equals 360 degrees.

Later on we might see if this rule holds true for other polygons as well, but for now let's be content with what we have.

Q: I have a problem. If I try to draw a polygon with seven sides, I need an angle of 360 divided by 7, or $51\frac{3}{7}$ degrees. How do I do that in PILOT?

A: Well, this is one area where Atari PILOT isn't going to be of much help. Unfortunately, you can tell PILOT to use only whole numbers. If you turn 51 degrees each time, you will fall a little short, and 52 degrees will be a little too much. If you are careful, you should be able to get the turtle to come pretty close to drawing anything you want, though.

Grand finale . . .

Before ending this chapter, let's take Rule #1 and use it to make a general-purpose, regular-polygon plotter (or GPRPP for short). Each time we draw a polygon, we have to figure out the needed angle by dividing 360 by the number of turns we want to make—unless, that is, we can persuade PILOT to do the dividing for us. And so we can, simply by using the PILOT symbol for divide, the slash (/). (I could tell you that this mark is called a solidus, but would you really care?) To draw a square we could type:

GR: 4(DRAW 15; TURN 360/4)

To draw an octagon we could type:

GR: 8(DRAW 15; TURN 360/8)

and so on.

I think you have the idea, so treat yourself to a picture with a triangle, square, pentagon, hexagon, octagon, nonagon, and a decagon in it. Change the colors once in a while too!

6

modules: building the turtle's dictionary

WE HAVE DISCOVERED quite a few interesting things about the turtle. We know how to make it move, draw, and turn, how to pick it up, orient it, and make it change pen colors. Now these are all pretty terrific things, but we aren't done yet! How would you like the turtle to look up procedures in a dictionary? Think about the number of possibilities this might allow. Suppose you wanted the turtle to draw lots of pentagons in different places on the screen. You already know how to do this by typing:

```
GR: 5(DRAW 25; TURN 72)
```

Suppose we could tell the turtle to use a command procedure that we have defined by the *name* PENTAGON. This could save a lot of typing.

As it turns out, we can build a dictionary for the turtle to use. Each entry in the dictionary is called a *module*. Each module has three parts: its name, the things we want the module to perform, and a command to let PILOT know when the module is finished.

When you first turn on PILOT, there are no modules in your dictionary. You have to create the modules, and you can create as many as you like. However, they will disappear when you turn off the computer unless you save them on a tape or disk memory. The Atari PILOT manual shows you how to save your programs, so you might want to get familiar with this feature soon.

And now for something really different . . .

In order to create dictionary entries, we first need to learn some new PILOT commands.

So far, our experience with PILOT has been that everything we typed caused something to happen right away. Because PILOT has been obeying each command immediately, we can call our usual way of doing things the *immediate mode*.

To build a dictionary, we need to find a way of postponing commands until we need them later; this we can call the *deferred mode*. To see how this works, turn on your computer and type:

AUTO

As soon as you press RETURN, you see some astounding changes, right? The pretty blue background is replaced by a yellow background, and the white letters turn to dark brown. This isn't all that happens, though, as we shall soon see.

Next type:

GR: CLEAR

Hmmm—nothing seems to happen. Well, let's try something else:

GR: 4(DRAW 25; TURN 90)

PICTURE THIS!

Hmmm—still nothing. Is PILOT broken? Where did these instructions end up?

To see what happens, press RETURN without typing anything else. Aha! we are now back in our familiar blue immediate mode. To check this out, type:

```
GR: CLEAR
GR: DRAW 25
```

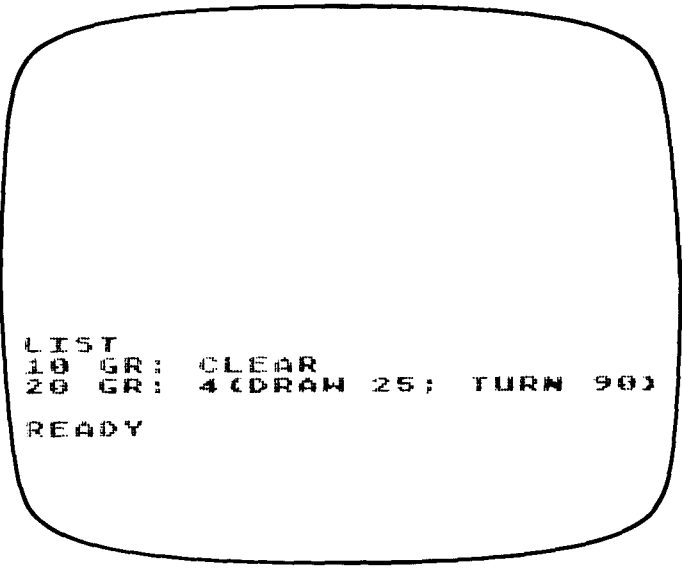
Yup! We are in our familiar mode again. Now let's find out what happened to those other lines we typed. To go back to the big blue screen type:

```
GR: QUIT
```

(I'm sure you can figure out what *this* does) Next type:

```
LIST
```

Here are the missing lines!



```
LIST
10 GR: CLEAR
20 GR: 4 (DRAW 25; TURN 90)
READY
```

What do the numbers in front of each command mean? First things first. The command LIST gives us a way of seeing our deferred instructions. This is quite useful since we'd probably have a hard time remembering them ourselves. The difference between a deferred and an immediate instruction is the presence of numbers in front of each line of PILOT commands. We can test this out by typing:

```
30 GR: DRAW 7
```

Nothing happens, right? Now type LIST again and notice that line 30 is added to the deferred commands.

Q: Does this mean that AUTO AUTOMATICALLY places numbers before each line of deferred instructions?

A: Yes. In fact, AUTO will assign the number 10 to the first line you type and will make each line number larger by 10 until you press the RETURN key without typing anything else.

The numbers on each line serve two purposes. First, they tell PILOT to store these commands for later use, rather than obey them right away. Second, they determine the sequence in which the commands are to be stored by the computer. To see the value of this second function, try the following.

First type:

```
NEW
```

This command erases *all* deferred instructions from the computer. If you type LIST at this point, you will see only the word READY—all the numbered commands have been erased. (This means you must be very careful when using NEW!) Next type AUTO (and then press RETURN) and enter the following lines:

```
GR: TURN 45; DRAW 10
GR: TURN -90; DRAW 10
GR: TURNTO 180; DRAW 10
```

PICTURE THIS!

(These lines aren't part of anything of interest right now, I just wanted to create some sample commands so I picked these at random.) Press RETURN a second time in order to leave the AUTO mode. Now if you type LIST, you will see these commands on the screen with the number 10 in front of the first one, 20 in front of the second one, and 30 in front of the third one.

Suppose you want to change the pen color to red between statements 10 and 20. How do you suppose you would do this? One way would be to type:

```
15 GR: PEN RED
```

since 15 is between 10 and 20. Type LIST to see if this works. It does? Great! You see how line numbers can help us put commands in the right order.

Before we show you how to get these deferred commands to do something, we have one last bookkeeping chore to handle.

Suppose you have already written some deferred instructions using AUTO, and you now want to write some more. If you type AUTO again, your new commands will cause old ones (those that have line numbers identical to the newer ones) to be erased. The first new line will AUTOMatically be assigned the number 10, the second line, 20, and so forth.

There are several ways to fix this. For our application, my favorite is to RENumber the existing lines with the REN command. Here is how it works.

Type.

```
REN 1000
```

then type

```
LIST
```

and see what happens.

```

LIST
10 GR: TURN 45; DRAW 10
15 GR: PEN RED
20 GR: TURN -90; DRAW 10
30 GR: TURNT0 180; DRAW 10

READY
REN 1000

READY
LIST
1000 GR: TURN 45; DRAW 10
1010 GR: PEN RED
1020 GR: TURN -90; DRAW 10
1030 GR: TURNT0 180; DRAW 10

READY

```

Our deferred instructions have been given new numbers, starting with 1000, and increasing by 10 for each line of commands. The sequence of instructions (which is all that PILOT uses the numbers for) is left unchanged. Now if we type AUTO and then type:

GR: PEN BLUE

and leave the AUTO mode, you'll see (when you type LIST) that the new line has been given the number 10 and that none of the old lines has been erased.

Q: What would happen if I typed REN 250?

A: All the deferred statements would be renumbered in their original order, but the first line number would be 250.

Q: What if I just type REN by itself?

A: Try it and type LIST to see what happens.

PICTURE THIS!

Q: Another question—I know how to add new lines, but how do I erase a line I don't want?

A: Just type the number of the line you want erased and press RETURN. Try this and type LIST to see how it works.

The reason we have spent so much time on AUTO, NEW, LIST, and REN is that these PILOT commands will make it easy for us to build dictionary entries.

Building a dictionary—our first experience with modules . . .

As mentioned before, the turtle can be made to “look up” procedures in a “dictionary.” We have complete control over the dictionary entries. This means that we can have as many or as few of them as we wish.

Please understand that there is no magic in the name of a dictionary entry. If we create a module called *SQUARE that tells the turtle how to draw a triangle, then a triangle is what we will get every time we use *SQUARE. The turtle cannot think; it can do only what we tell it to do. I guess this means the turtle isn't terribly smart after all!

Let's make our first dictionary entry and then try it out to see how it works. For starters, let's make an entry called *SQUARE.

The asterisk (*), by the way, *must* precede the name of every dictionary entry. Labeling all of our entries this way makes it easier for PILOT to find them. Just to help *us* understand what is going on, we will make this module draw a square each time we use it.

Type AUTO and then enter:

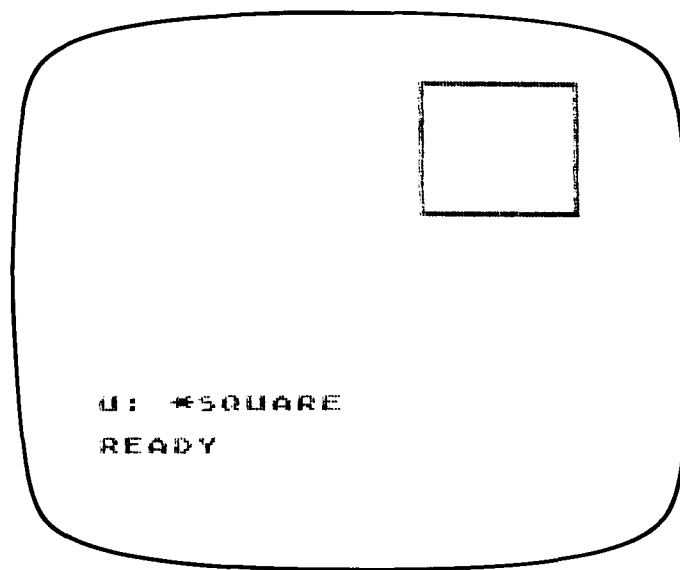
```
*SQUARE
GR: 4(DRAW 25; TURN 90)
E:
```

Press RETURN again to leave the AUTO mode. As mentioned before, each dictionary entry has a beginning, a middle, and an end. The beginning is always the name of the module, preceded by an asterisk (*SQUARE, in our case). The middle is one or more lines of commands that tell PILOT (or the turtle) what to do each time this particular module is used. The end of the module is represented by the *End command* (E:).

Now let's see what it means to have created the module *SQUARE. Make sure your screen is blue (which indicates you're in the immediate mode) and type the following:

```
GR: CLEAR
U: *SQUARE
```

If everything works, you should see a yellow square on your screen.

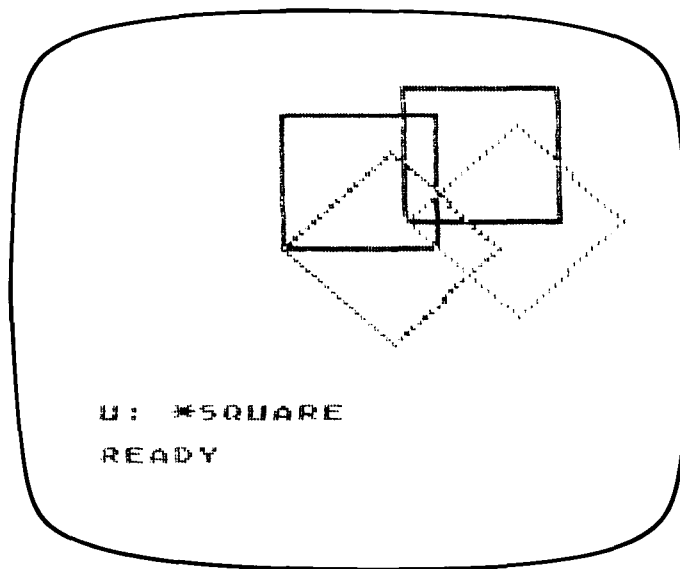


PICTURE THIS!

Here's what happened. When you typed U:, you gave PILOT a command to use a module. The name of the module to be used (*SQUARE) is to the right of the colon. The *Use command* (U:) is one of the ways we can have the turtle find procedures in our dictionary. We will learn other ways later.

Let's try some more:

```
GR: GOTO -20, -5
U: *SQUARE
GR: PEN RED; TURN 45
U: *SQUARE
GR: PEN BLUE; GOTO 0,0
U: *SQUARE
```



Now we have a picture with both squares and (somewhat jaggy) diamonds in it. Do you see how we used *SQUARE to make the red and blue diamonds? By turning a square on its corner (45 degrees) we get a diamond shape—now that's worth remembering!

Next let's try some other figures. Type the following:

```
U: *PENTAGON
```

Oh-oh! It appears that we have made a mistake. When I typed this line I got the message:

```
U: *PENTAGON
```

```
*** WHERE? ***
```

with the P in PENTAGON reversed. Our poor turtle tried to find the module *PENTAGON in the dictionary but couldn't locate it. Do you suppose this is because we haven't yet defined it? Let's fix this oversight. First type:

```
GR: QUIT
```

to bring us back to the full text window (the blue screen). To create a new module, you might think we should first type AUTO. But wait a minute! Let's see what kind of trouble we would get into if we did that. Type LIST to see what we already have in the dictionary. As you can see, we have the entry *SQUARE, which starts at line number 10. We need to move this existing entry somewhere to make room for a new module. Do this by typing:

```
REN 1000
```

This will move all existing entries so that the lowest line number now reads 1000. This means that when we type AUTO, we can define a new procedure with as many as one hundred lines in it before we need to worry about its running into an existing definition.

You should always remember to clear some workspace with REN before typing AUTO. This procedure will save you from many surprises later on.

PICTURE THIS!

To define our new module, enter the AUTO numbering mode and type:

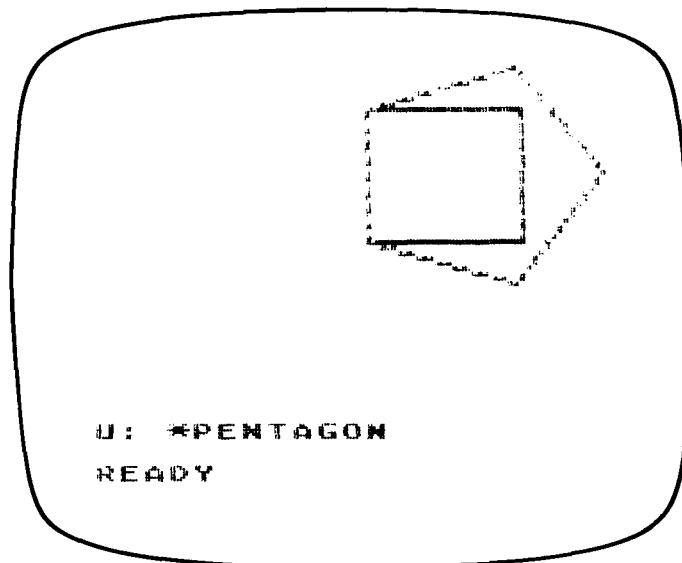
```
*PENTAGON
GR: 5(DRAW 25; TURN 72)
E:
```

and press RETURN again to exit from the AUTO mode and to enter the immediate mode.

Now type:

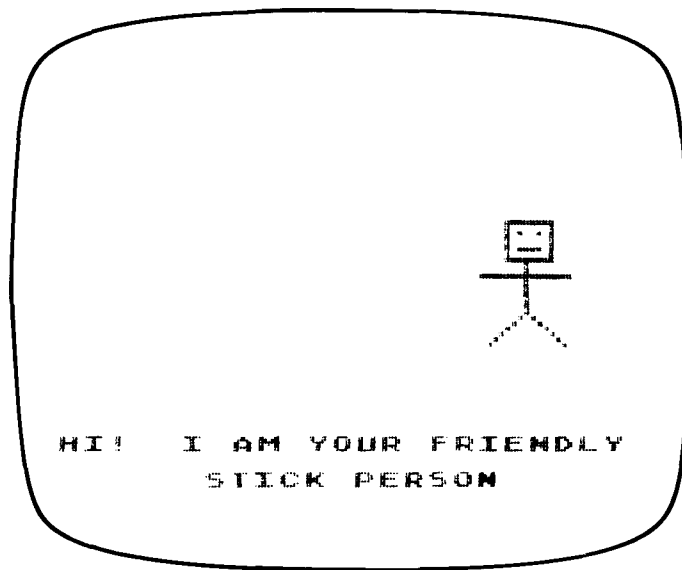
```
GR: CLEAR
U: *SQUARE
GR: PEN RED
U: *PENTAGON
```

Hooray! The turtle now is able to look up two procedures in its dictionary. When I type these commands I get both a yellow square and a red pentagon. They look like this:



Bigger modules for fancy pictures . . .

So far we have made very small modules, so it is not yet obvious that modules can save a lot of work. In our next example, we will do something quite ambitious just to show how useful the turtle's dictionary can be. Let's create a module to draw a stick figure of a person.



In creating this module we will go through three stages. First, we will decide what a figure of a person ought to have on it. Second, we will try having the turtle draw each part of the figure in the immediate mode. Third, we will create the module `*PERSON` and use it to draw some nice pictures.

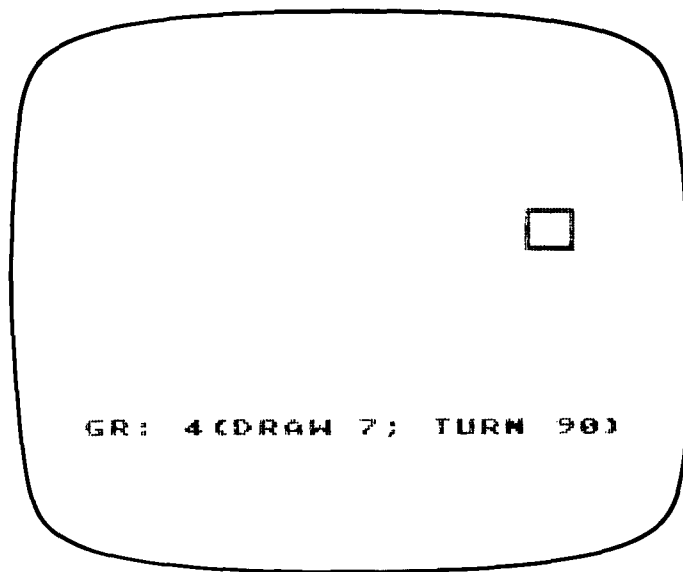
Look at the figure above and decide just what sorts of things we need to have the turtle do to draw a convincing stick person. Our friend seems to have a head, a body, two arms, and two legs.

Let's determine how to draw the head first.

PICTURE THIS!

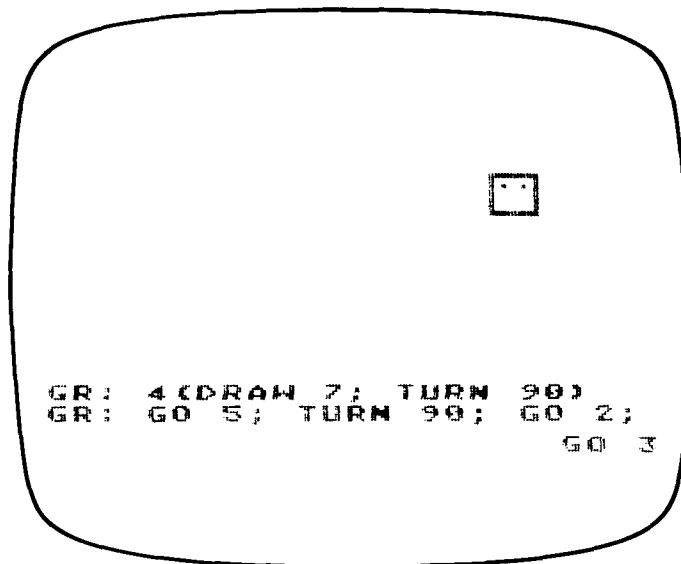
Clear the graphics screen and make sure the turtle is in its home and pointing up. Let's next make the square we will use for the person's head;

GR: 4(DRAW 7; TURN 90)



Next, add two eyes to the head by moving the turtle 5 units up the left side of the square, turning it to the right, and making two dots:

GR: GO 5; TURN 90; GO 2; GO 3

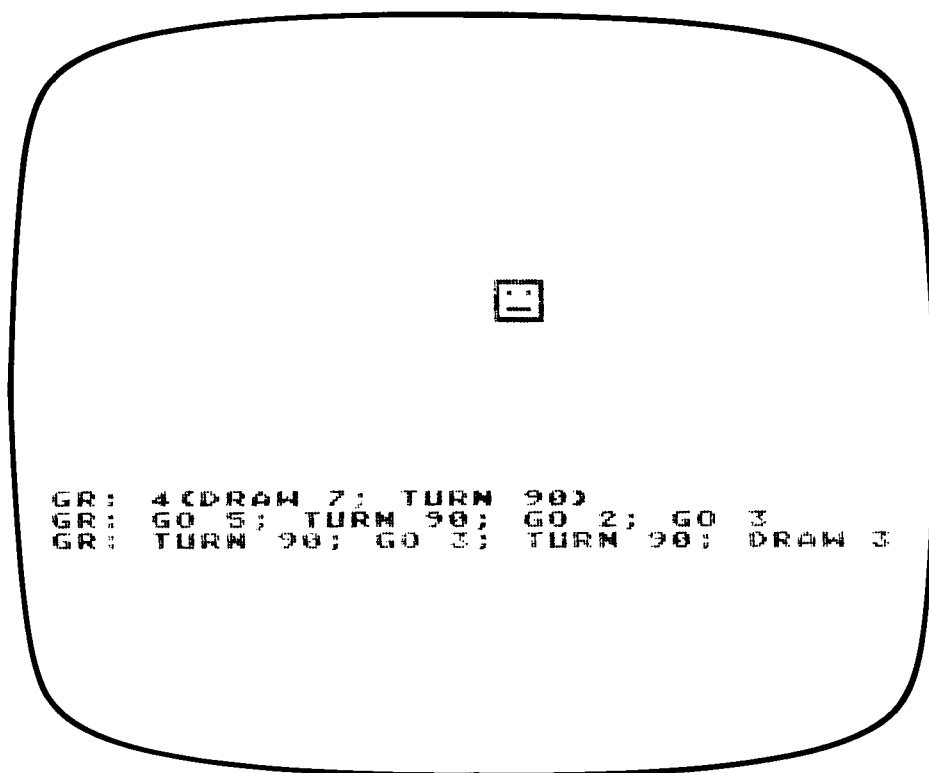


Do you see how we get the turtle to make the dots for the eyes?

Now—a very important question—Do you know where the turtle is and in which direction it is pointing? If you have any doubts, you should play turtle yourself and draw as much of the head as we have drawn so far.

Did you say that the turtle is at the right eye and is pointing to the right? Very good! Now we have to draw the person's mouth. Since each side of the square is 7 units long, and since we have moved 5 units up from the bottom to draw the eyes, we should move back down 3 units to draw the mouth. Let's make the mouth this way:

GR: TURN 90; GO 3; TURN 90; DRAW 3



Hooray! we now have a "blockhead" for our person.

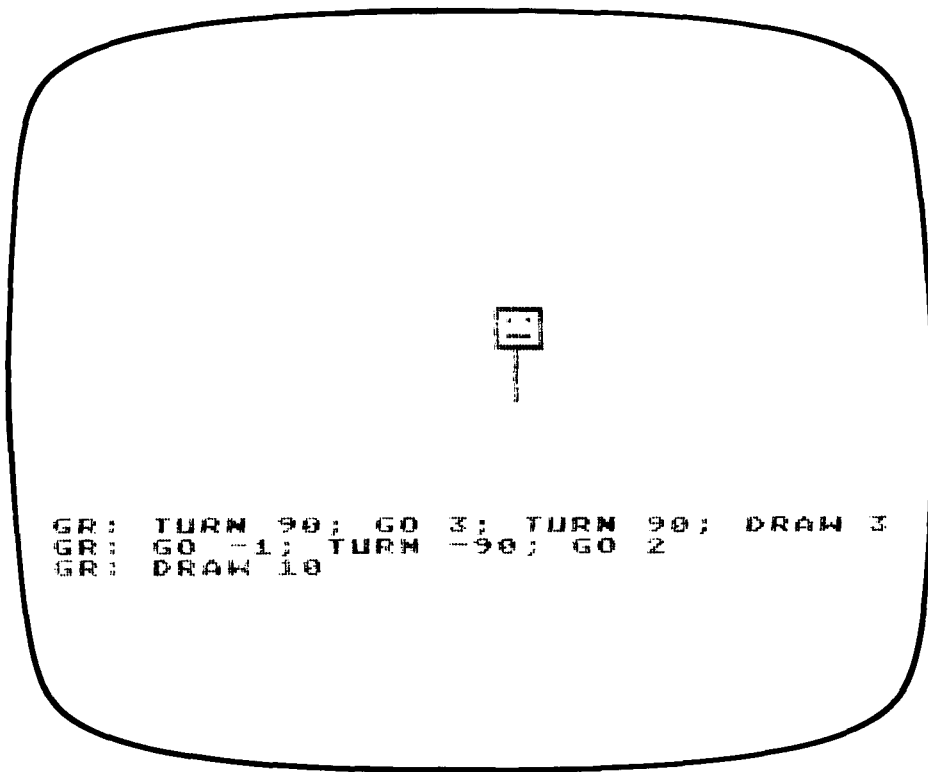
PICTURE THIS!

Next we need to draw the body. I don't know about you, but I sure would like the neck to come out the middle of the bottom of the square. Since we are at the left edge of the mouth pointing to the left, we need to back the turtle up a little and move it down a bit. Suppose we try this:

```
GR: GO -1; TURN -90; GO 2
```

Do you think the turtle can manage all of its movements backwards? Let's see how well it did drawing the body:

```
GR: DRAW 10
```



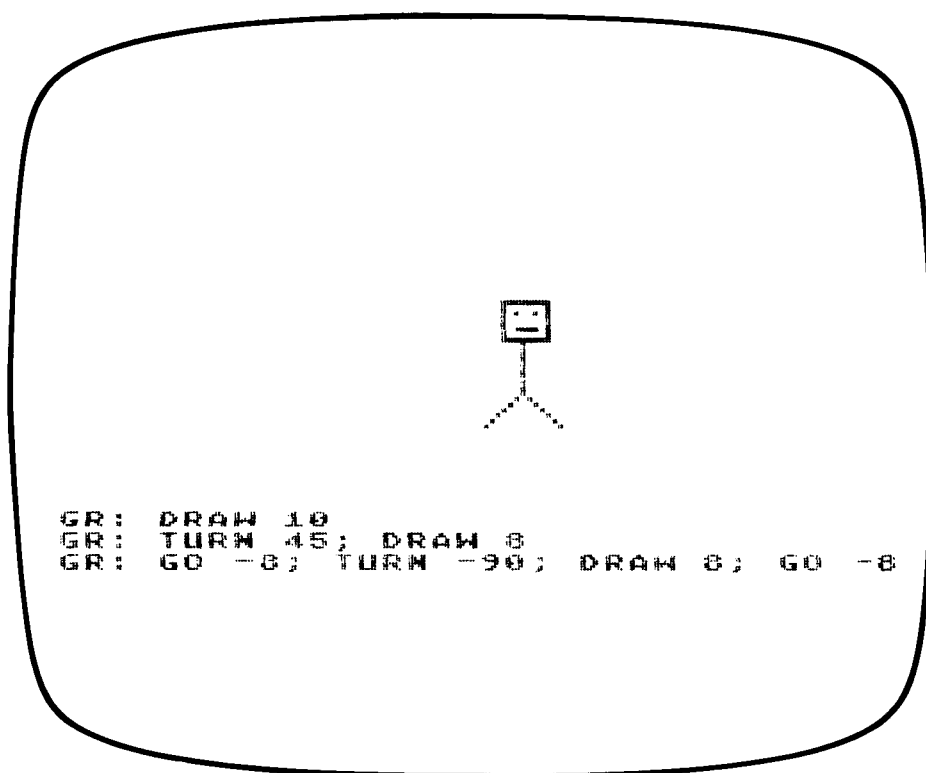
Well, well! It isn't fancy, but at least the turtle drew the body in the right place. Since the turtle is at the bottom of the body, we should probably draw the legs next.

We can draw the left leg this way:

```
GR: TURN 45; DRAW 8
```

To draw the right leg, we need to go back to the body, turn the turtle by 90 degrees (so the right leg will point in the correct direction) and then draw this leg. Let's see how this works:

```
GR: GO -8; TURN -90; DRAW 8; GO -8
```



We are getting close to completing our person. Unless we want to call our friend *VENUS-DE-MILO, we should probably add some arms. If you have been paying attention (or playing turtle, which is much more fun than paying attention), you might have noticed that the turtle is back at the bottom of the body (Did you see the GO -8 command at the end of the last set of instructions?) and we now have to

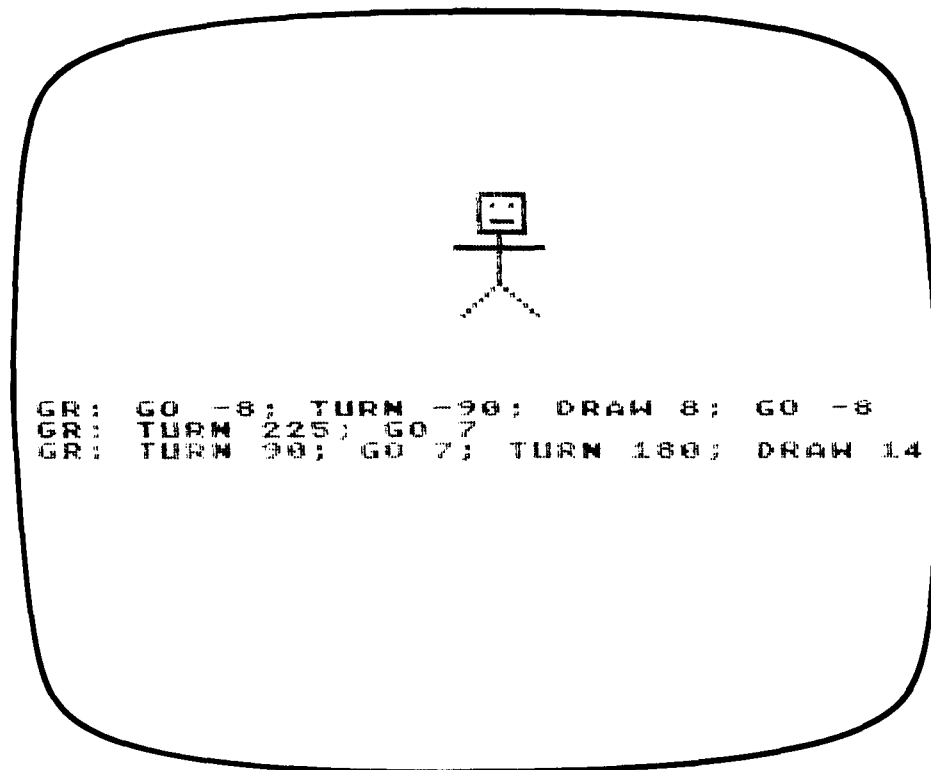
PICTURE THIS!

turn it around to go up the body. Since the body is 10 units long, we will put the arms 7 units up from the bottom of the body so that our person has a neck:

GR: TURN 225; GO 7

Do you know why we had to turn 225 degrees? When you typed this line of commands, the turtle was pointing down the right leg. If we turned it only 45 degrees it would be pointing straight down, but by turning it an additional 180 degrees it is pointing straight up the body. Since $45 + 180$ is equal to 225, I combined both turns in one command. Now for the arms.

GR: TURN 90; GO 7; TURN 180; DRAW 14



Ta Daa! Our friendly stick person is finished. The only problem we have now is that each time we want to draw this figure, we have to type these commands all over again. Now you see why modules can be so handy. Let's repeat our effort, but this time do it by adding *PERSON to the turtle's dictionary. Type:

```
REN 1000
AUTO
```

and, when you see the yellow screen, enter the following:

```
*PERSON
GR: 4(DRAW 7; TURN 90)           [draw head
GR: GO 5; TURN 90; GO 2; GO 3    [draw eyes
GR: TURN 90; GO 3; TURN 90; DRAW 3 [draw mouth
GR: GO -1; TURN -90; GO 2        [move to neck
GR: DRAW 10                      [draw body
GR: TURN 45; DRAW 8              [draw left leg
GR: GO -8; TURN -90; DRAW 8; GO -8 [draw right leg
GR: TURN 225; GO 7               [move to arms
GR: TURN 90; GO 7; TURN 180; DRAW 14 [draw arms
E:
```

Press the RETURN key again to go back to the blue screen. If you are planning to save your work on tape or magnetic disk, this is a good time to do it.

Q: What is the meaning of the words after the square bracket ([)? Am I supposed to type that too?

A: The left square bracket is a signal to PILOT that you are making some comments to yourself, and that PILOT shouldn't try to figure out what they mean. It is purely up to you if you want to use them or not. I put them in so that we wouldn't forget what each line of commands was doing.

Q: When I turned the turtle around to move him up the body, I typed TURN 225. Couldn't I have done this another way? What is the correct way to have the turtle do things?

PICTURE THIS!

A: You will find that there are many many ways of getting the turtle to accomplish the same result. There is no one right answer. As long as the turtle ended up drawing what you wanted it to, you did things the right way.

Q: Another question—Why didn't we use TURNTO 0 whenever we wanted the turtle to point straight up?

A: Aha—you noticed that *PERSON was created by using only relative commands like GO, DRAW, and TURN, instead of absolute commands like GOTO, DRAWTO, and TURNTO. There is a very good reason for making most modules this way. It allows you to place the figure described by the module anywhere on the display screen—not just where you defined it.

Now let's reward ourselves for our heroic effort by using *PERSON to draw some pictures on the screen.

First, set up a clear graphics screen and type:

U: *PERSON

If everything works according to plan, you should now have a picture of a yellow person standing in the middle of the screen. If you have something else (or if the arms are on crooked, or a leg is missing, or something else is wrong), you should type

GR: QUIT

and then type

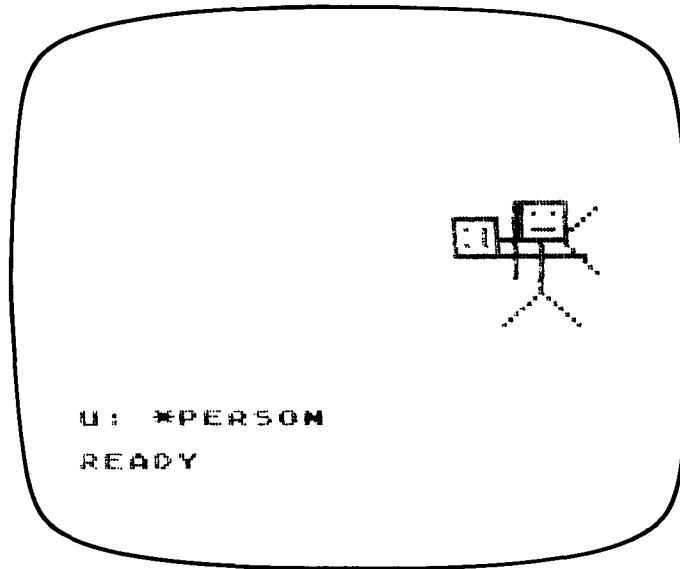
LIST

to see what you typed earlier. Compare each line of the module *PERSON with the lines shown on page 60, and repair any errors you might discover.

Because we have defined *PERSON, we can use this module any time

we want to draw a person on the screen. Let's try using this module again:

U: *PERSON



Hmmm—now that's interesting! The second time we used *PERSON, the turtle laid the person on its side. Do you suppose this is because it is tired of being used? No? Just thought I'd ask.

Actually, you probably realize that the reason for *PERSON's new orientation is that we finished the module with the turtle in a direction and location different from the one in which it was at the beginning of the module. While we won't bother to fix this now, we can make a new rule (let's call it a suggestion) to keep this from happening in the future:

A TURTLE SUGGESTION: When making a module, make sure the turtle ends up in the same place and direction that it was in at the beginning.

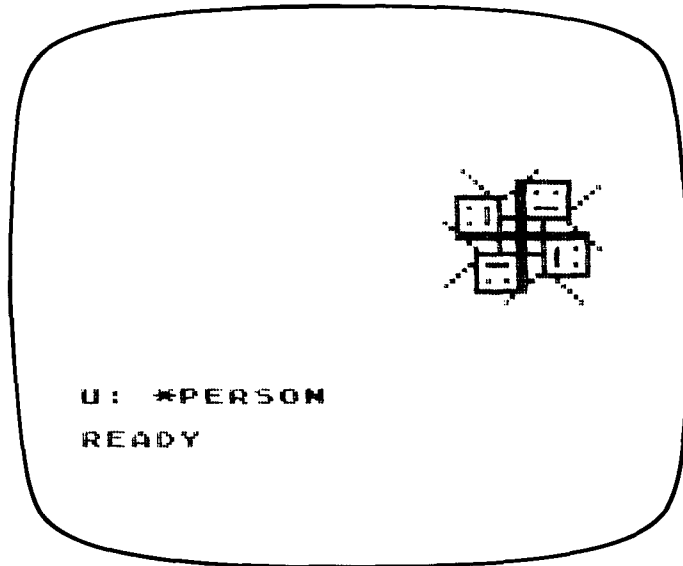
This should make your modules much easier to use.

PICTURE THIS!

In any event, we can draw some nice designs with *PERSON—for example:

U: *PERSON

U: *PERSON



Now let's make a row of persons. Clear the screen, move the turtle to its home and point it straight up. Next type:

GR: GOTO -50,10

U: *PERSON

GR: GOTO -30,10; TURNT0 0; PEN RED

U: *PERSON

GR: GOTO -10,10; TURNT0 0; PEN BLUE

U: *PERSON

GR: GOTO 10,10; TURNT0 0; PEN YELLOW

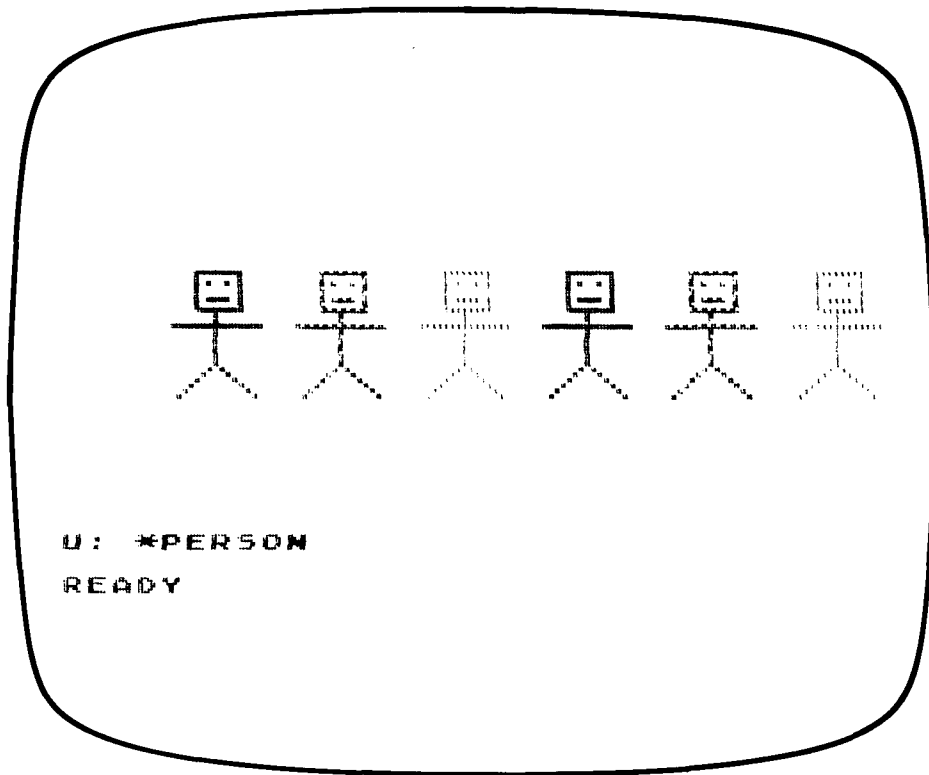
U: *PERSON

GR: GOTO 30,10; TURNT0 0; PEN RED

U: *PERSON

GR: GOTO 50,10; TURNT0 0; PEN BLUE

U: *PERSON

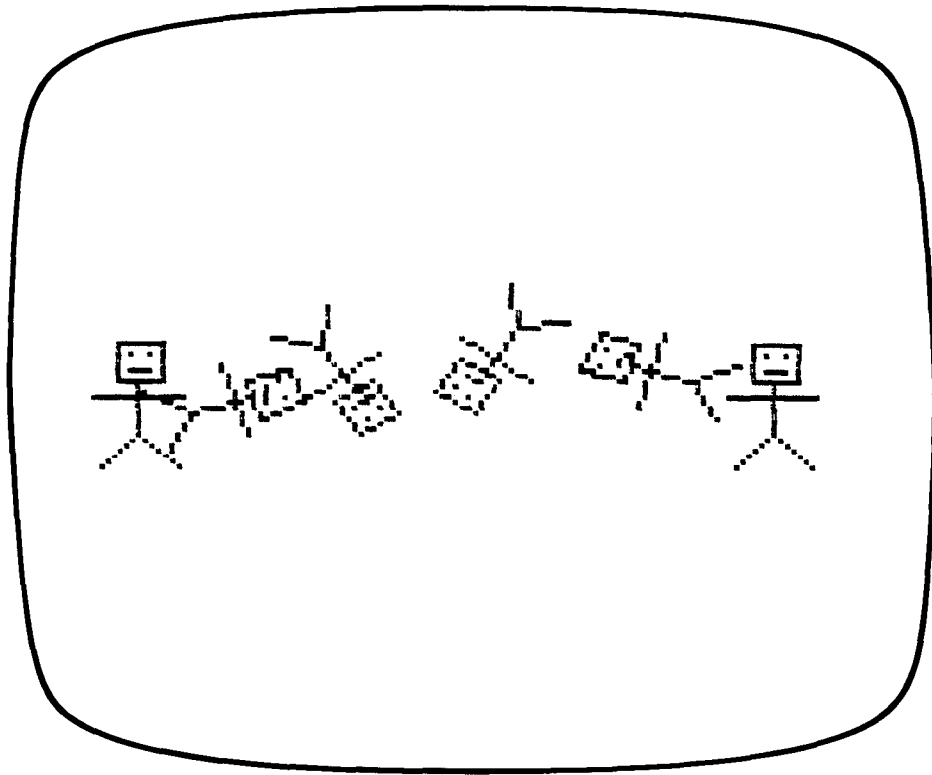


Isn't it neat to be able to draw figures as complicated as *PERSON using just one command? I think so!

Next let's make *PERSON do a cartwheel:

```
GR: CLEAR
GR: PEN YELLOW; GOTO -50,10; TURNT0 0
U: *PERSON
GR: GOTO -30,10; TURNT0 72
U: *PERSON
GR: GOTO -10,10; TURNT0 144
U: *PERSON
GR: GOTO 10,10; TURNT0 216
U: *PERSON
GR: GOTO 30,10; TURNT0 288
U: *PERSON
GR: GOTO 50,10; TURNT0 360
U: *PERSON
```

PICTURE THIS!



I think that *PERSON really looks funny at some angles, don't you?

This would be a good time for you to modify *PERSON so that it obeys our suggestion of having the turtle end up in the same place and direction it had at the beginning of the module.

Why don't you spend some time building modules of your own. You might want to start with simple ones and then progress to more complex designs. When you use a new module, try it out at different angles to see how strange it can look. Remember that we discovered how to make a diamond from a square in this way! Have fun!

modules using modules

AND THEN THERE was Rose.

Rose was her name and would she have been Rose if her name had not been Rose. She used to think and then she used to think again.

Would she have been Rose if her name had not been Rose and would she have been Rose if she had been a twin.

(from *The World is Round* by Gertrude Stein)

We have now seen that our graphics turtle can look up procedures in a dictionary and that it isn't hard to get some pretty pictures on the display screen. One of the things we will do in this chapter is find out just how much the turtle can look up.

Is it, for example, possible for one dictionary entry to use another one? Can *ROSE be *ROSE if it has another name? We sometimes find this sort of thing in the dictionaries you and I use, where one word is defined by referring you to another word (e.g., *labor union*—

PICTURE THIS!

see trade union). These kinds of definitions are just ways of giving another name to something that is defined somewhere else. Let's see how the turtle handles this. Enter:

```
NEW  
AUTO
```

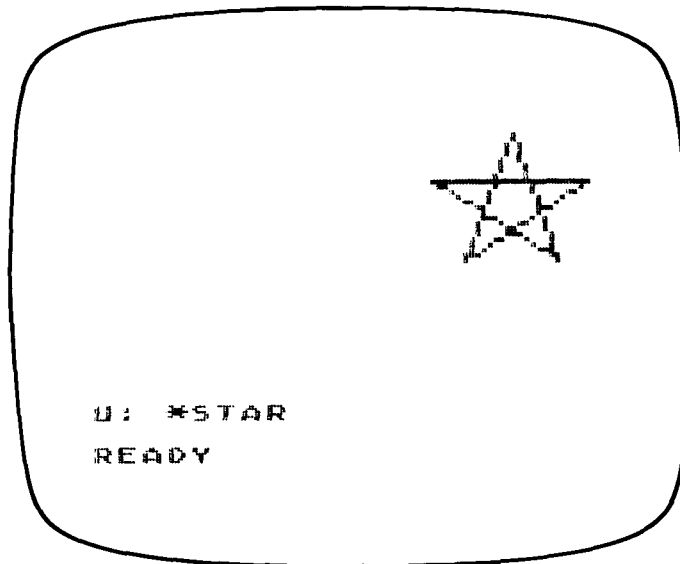
and then type:

```
*STAR  
GR: TURN 90  
GR: 5(DRAW 25; TURN 144)  
GR: TURN -90  
E:
```

Now press RETURN again in order to leave the AUTO mode. (Did you notice that the statement GR: TURN -90 puts us back in our starting direction before leaving the module? The Turtle Suggestion (p. 63) is quite helpful.)

Next let's try out this module:

```
GR: CLEAR  
U: *STAR
```



Are you surprised at the result? We just drew a five-pointed star. In a later chapter we will try to figure out *why* this module gave us a five-pointed star, but we have other things to try out first.

Next let's define a "new" picture module. Type:

```
REN 1000  
AUTO
```

and then enter:

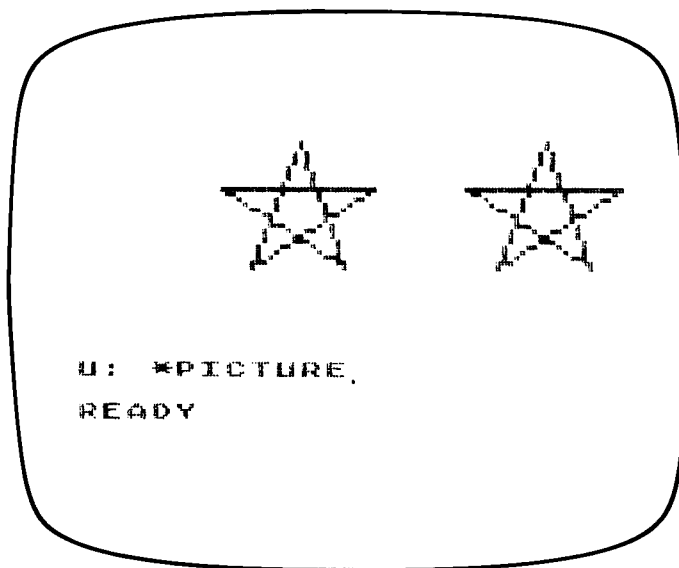
```
*PICTURE  
U: *STAR  
E:
```

Press RETURN to leave the AUTO mode. Let's now see what happens when we use *PICTURE by trying the following:

```
GR: CLEAR  
GR: GOTO -30,0  
U: *STAR
```

This should put a nice, yellow star on the left side of your screen. Next type:

```
GR: GOTO 10,0  
U: *PICTURE
```



PICTURE THIS!

Well, what do you know! When we asked the turtle to use *PICTURE, all it saw there was an instruction to use *STAR, so that means *PICTURE and *STAR give the same result. We can check on this by changing *STAR and seeing what then happens to *PICTURE. To do this type:

```
GR: QUIT
LIST
```

Your screen should now show:

```
10 *PICTURE
20 U: *STAR
30 E:
1000 *STAR
1010 GR: TURN 90
1020 GR: 5(DRAW 25; TURN 144)
1030 GR: TURN - 90
1040 E:
```

As you can see, the module *STAR starts at line 1000 and runs to line 1040. To make a noticeable change, let's alter line 1020 to read:

```
1020 GR: 9(DRAW 25; TURN 160)
```

Now type:

```
GR: CLEAR
GR: GOTO - 30,0
U: *STAR
```

We now have a nine-pointed star. Next type:

```
GR: GOTO 10,0
U: *PICTURE
```

By now you should have two nine-pointed stars on your screen. *PICTURE is a perfect copycat—whatever we do to *STAR shows up each time we use *PICTURE as well!

Q: Aha—we can have all kinds of definitions that refer to each other, right?

A: Almost. You could have a module called *COPY that uses *PICTURE, and each time you use *COPY you will get a star. (Try it out.) As we shall soon see, there is a limit to this process.

Let's see what limits the turtle has on looking things up.

First type:

```
GR: QUIT
REN 1000
AUTO
```

and enter:

```
*PING
U: *PONG
E:
```

This defines the module *PING. Now, to define *PONG, type:

```
*PONG
U: *PING
E:
```

and press RETURN to leave the AUTO mode. Do you see anything strange about these definitions? Do you think the turtle might find it strange to see *PING defined in terms of *PONG and *PONG defined in terms of *PING? How would you handle this if you were the turtle?

Let's see what the turtle does with these definitions:

```
GR: CLEAR
U: *PING
```

Oh-oh, something went wrong.

PICTURE THIS!

When I tried this my screen showed:

```
20 U: *PONG
```

```
*** U: TOO DEEP ***
```

Now that sure looks like a pretty cryptic message to me; maybe we can do some experiments to find out what it means.

To start with, neither *PING nor *PONG drew anything on the screen (which I find rather boring). Let's start over again by having the turtle make some "footprints" each time it uses *PING or *PONG. Type:

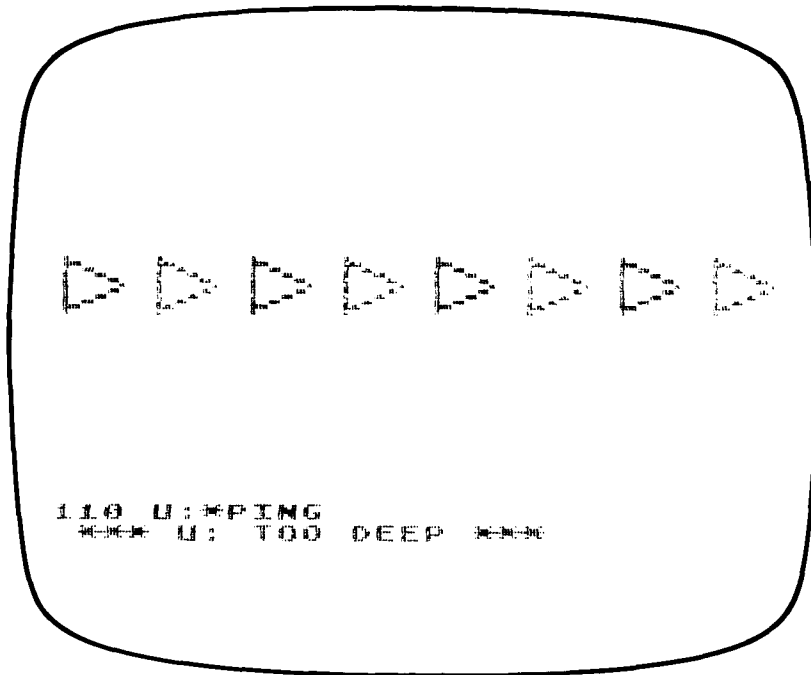
```
GR: QUIT  
NEW  
AUTO
```

and then enter:

*PING	
GR: PEN YELLOW	[*PING has a yellow pen
GR: 3(DRAW 10; TURN 120)	[the turtle has triangular feet
GR: TURN 90; GO 15; TURN - 90	[it steps 15 units to the right
U: *PONG	[now we use *PONG
E:	
*PONG	
GR: PEN RED	[*PONG has a red pen
GR: 3(DRAW 10; TURN 120)	[it still has triangular feet
GR: TURN 90; GO 15; TURN - 90	[another step for turtle
U: *PING	[now we use *PING
E:	

After typing these things, press RETURN again to leave the AUTO mode and enter:

```
GR: CLEAR
GR: GOTO -70,0
U: *PING
```



Now we have some footprints on the screen to help us see how far the turtle got before PILOT gave us the error message (** TOO DEEP **). By counting the marks the turtle drew on the screen, it appears that it took nine steps before running into trouble.

Hmmm, nine small steps for turtle, one large crash for PILOT.

If you look at *PING and *PONG closely, you'll see that the turtle was trying to use a module for the ninth time when it ran into trouble. There is a limit to the number of references that the turtle can keep track of at one time, and that limit is eight.

The reason for this limit is that neither *PING nor *PONG was ever able to finish; they always were interrupted before getting to the end (E:) statement.

PICTURE THIS!

When PILOT uses a module, it makes a reference to it in something called a *stack*. If another module is used before the first one is finished, then PILOT saves a reference to both the first and second modules. This is so that when the second one finishes, PILOT knows that it should go back to the first module to see if it needs to do anything else before stopping. This process continues every time another module is used, with each new reference being put on top of the former one. Computer folk say the references are being “pushed down the stack” (have you noticed that computer folk talk funny sometimes?). Our problem came when we tried to “push the stack too deep.” Now you know the origin of the words TOO DEEP in the error message you received.

Don’t worry if you don’t understand all the fine points mentioned above; just remember that you’re going to get in trouble if you try to use modules more than eight times before letting any of them finish.

Later on we will find ways of getting around this limitation.

modules using variables

THERE ARE LOTS of terrific things you can do once you start having modules use each other. Next we will learn how to create a module that uses another module a certain number of times. This not only will give us some pretty pictures but also will let us understand more about computer programs.

We already know one way of getting the turtle to do something several times. For example, when we type

```
GR: 4(DRAW 25; TURN 90)
```

we are instructing the turtle to do something four times—namely, to draw a line 25 units long and to turn right by 90 degrees. This shorthand is quite useful. Let's see if we can use it with modules in the same way. Press SYSTEM RESET and type:

```
NEW  
AUTO
```


PICTURE THIS!

Now enter the following:

```
*STAR
GR: 5(DRAW 25; TURN 144)
E:
```

Press RETURN again to leave the AUTO mode and type:

```
GR: CLEAR
U: *STAR
```

And you'll see we have created a module that draws a five-pointed star.

Next let's see if we can convince the turtle to use *STAR a few times. Enter the following and watch what happens:

```
U: 4(*STAR)
```

Oh-oh! That didn't work at all, did it? Does your screen show

```
U: 4(*STAR)
```

```
*** WHAT'S THAT ***
```

with the 4 shown in reverse field? Apparently we can't use modules the same way we can use graphics commands. We have to find another way of doing this.

Q: Wait a minute. Even if the command had worked, how would you know? The module *STAR finishes where it started, so each time you use it you will get the same thing unless you move the turtle somewhere else in between uses, correct?

A: Hey! That's right! In fact it *wouldn't* be very useful for our command to work if we obeyed the Turtle Suggestion (p. 63) and always had the module end where it started. That must be why we need to find another way of doing this task.

Now, if I wanted to do a series of things several times, I could count the number of times I had done it so far, compare this number with the total I wanted, and either go on or stop depending on the condition of the comparison.

You may be pleased to learn that PILOT lets us *count* things, allows us to *compare* numbers with each other, and allows us to do things depending on a *condition*. Aren't we lucky!

First, let's learn about the counter. If you are counting to five, you might use the fingers of one hand. Your hand becomes the place where you temporarily keep track of how high you have so far counted. Since this number will vary during the counting process, we can call the hand a *variable*. Just remember that a variable is a place where you can keep a number stored for safekeeping. Each time you count, you extend one more finger on your hand. This means you look at the present value of the number in the variable and you add 1 to it. Finally, when you reach 5, you notice that the number in the variable is the same as the limit you had picked, so you stop.

Now let's look at this process in a way that your computer might do it. We already know how to do some things, such as create labels (these are words with asterisks before them that we use to name modules like *SQUARE). Here is a rough outline of a counter that counts to five:

(Don't type this in—we aren't quite ready for it yet)

```
*COUNTER
set the counter to zero
*ADD1
add one to the number that is in the counter
jump (if the counter is less than five) to *ADD1
E:
```

You may have noticed that there are three lines in this module for which we don't have PILOT statements. Let's take them one by one:

"set the counter to zero"

PICTURE THIS!

OK, first we have to know what to call our counter. Atari PILOT allows you to use up to twenty-six numeric variables, each of which is identified by a letter of the alphabet placed after the number sign (#). So #A is the name of one PILOT variable, #B is another one, and so on up to #Z. The number sign in front of each letter tells PILOT (and us) that the variable contains a number (rather than some letters, for example). Each numerical variable can hold any whole number between -32768 and 32767. You probably aren't surprised by this since we already found out that PILOT has a hard time dealing with numbers outside this range.

By now you are probably wondering how we get to pick the value of the number stored in a variable. This is done with the *Compute command* (C:). Let us take the variable #A as our counter. To set the counter to zero we would enter

```
C: #A = 0
```

The command C: is an instruction that tells PILOT to do whatever computations are called for on the right side of the colon. In this case, the instruction is to replace whatever number was already in the variable #A with the number 0. It is very important to realize that in PILOT (as in BASIC and many other computer languages) the equals sign (=) means "take whatever is on the *right* side of the equals sign and place it in whatever variable is on the *left* side of the equals sign." When the equals sign is used in this manner, it is called a *replacement operation*.

We now have a PILOT command that is the equivalent of "set the counter to zero," namely,

```
C: #A = 0
```

Not bad for starters.

Q: What number is in each variable when we first begin?

A: When PILOT is first started, all the numerical variables contain zero (0).

Q: If that is the case, why do we have to set #A to zero in our counter?

A: Well, the main reason is that we might find ourselves using the counter module several times, and the counter will start at zero only the first time we use it. The second time it will start with whatever number happens to be in it when it reached the end of its last use. It is always a good idea to set variables to zero in order to be sure they calculate the right value each time you use them. This prevents many unwelcome surprises later on.

Next we need to figure out how to perform the second undefined line of our counter program:

“add one to the number that is in the counter”

Since you know that the replacement operation works in a special way, can you figure out how to write the correct command? Here is the way I do it:

C: #A = #A + 1

Of course, if you think about the equals sign in the usual way, this statement will look like total nonsense. After all, nothing can be equal to itself plus one. But once we think of the equals sign as the replacement operation, everything becomes much clearer. What this command says to PILOT is “replace the present value of the variable #A with the present value of #A plus 1.” This command will increase the number in #A by one, just as you might extend one more finger when counting on your hand. If you think about it, you might recognize this command to be the heart of our counter.

Our counter is set up to add by ones. Can you figure out how to change the counter so that it adds by twos?)

Finally, we come to the last undefined line of our counter program:

“jump (if the counter is less than five) to *ADD1”

PICTURE THIS!

To write this in PILOT, we need a command that will “jump” to a label (in other words, to a name with an asterisk in front of it) and that will do this only if the value of the number in #A is less than five. The PILOT command that does this looks like this:

```
J (#A<5): *ADD1
```

J: is the PILOT *Jump command*. If we had written

```
J: *ADD1
```

PILOT would jump to the label that is to the right of the colon (*ADD1) every time the program came to this command. By placing a condition between the J and the colon, PILOT will obey the jump command only if the condition is true. For the kinds of things we are likely to be doing, the condition is always placed inside parentheses. Our condition is that the value of #A is less than five. The symbol for “less than” is <, so we write this condition as #A<5. If we wanted the jump to take place when the counter was less than ten, we would write:

```
J (#A<10): *ADD1
```

Now let’s put everything together and see what the counter looks like. Type:

```
GR: QUIT  
REN 1000  
AUTO
```

and then enter:

```
*COUNTER  
C: #A = 0           [set the counter to zero  
*ADD1              [label the place to jump to  
C: #A = #A + 1     [make #A larger by one  
J (#A<5) : *ADD1   [jump to *ADD1 if #A is less than five  
E:                 [end of the module
```

Press RETURN again to leave the AUTO mode and then type:

```
LIST 0,60
```

This command will list only those instructions with line numbers between 0 and 60.

If everything is typed correctly, you will see the following lines on the screen:

```
10 *COUNTER
20 C: #A = 0
30 *ADD1
40 C: #A = #A + 1
50 J (#A<5) : *ADD1
60 E:
```

Let's pretend we are using this module and see what happens. First, we see that #A is set to zero in line 20. Next #A is increased by 1, so that #A is now equal to 1. In line 50, #A is checked to see if it is less than 5. If it is less than 5 (which it is), the PILOT jumps back to *ADD1 in line 30 and the process is repeated. Each time the process is repeated, 1 is added to the value in #A until that value equals 5. Once that happens, the condition (#A<5) in line 50 won't be true any more, and PILOT will skip to the next command, which, in this module, is end (E:).

We can try out this module to see what happens by typing:

```
U: *COUNTER
```

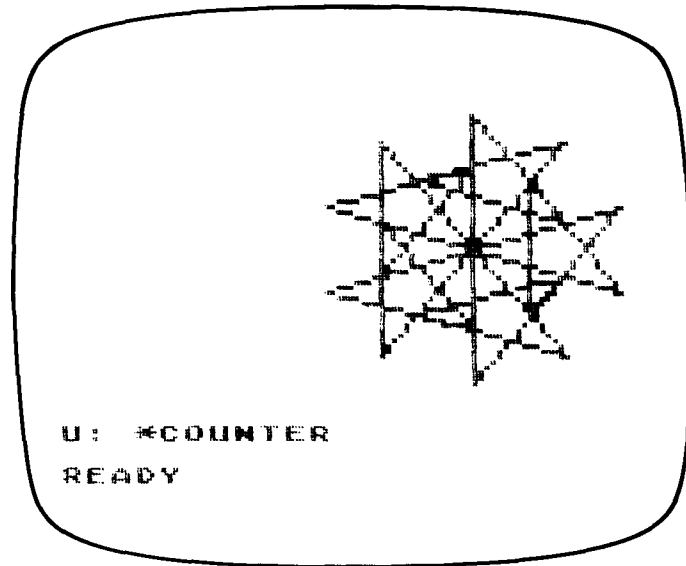
Hmmm—When I try it all that happens is that the screen displays the word READY. I guess everything works all right; after all, we've simply told PILOT to count very quietly to itself. To make things more interesting, let's add two commands to *COUNTER. Type the following:

```
42 U: *STAR
44 GR: TURN 72
```

PICTURE THIS!

Now, each time the counter passes between the label *ADD1 and the jump command, it will use the module *STAR and turn the turtle by 72 degrees. Let's see what this looks like:

```
GR: CLEAR  
U: *COUNTER
```



How pretty! We just drew five stars in a circle! By changing the instructions in *STAR we can draw five copies of anything we want by just using the module *COUNTER.

The turtle's windmill . . .

Now that we know all about counters, we are ready to make some really pretty pictures that I call the "turtle's windmill." These pictures are made by taking a simple object and repeating it at eight angles around the center of the screen. The main part of this picture-drawing module is called *WINDMILL. A second part of the program

will use a module called *PICTURE, which will be set up to use any of several simple figures we have defined elsewhere (such as *STAR).

To get started, type

```
REN 1000
AUTO
```

and then enter

```
*WINDMILL
GR: GOTO 0,10; CLEAR    [move the turtle up a little and clear
                        the screen
C: #A = 0               [set the counter to zero
*JUMPHERE               [label for the jump command
C: #A = #A + 1          [increase the counter by one
U: *PICTURE             [use the picture definition
GR: TURN 360/8          [turn one-eighth of a circle
J (#A<8) : *JUMPHERE    [do it all again if #A is less than eight
E:                      [stop
```

Then type:

```
*PICTURE
U: *STAR                [use *STAR since we already defined it
E:
```

When you have typed in these lines, press RETURN again in order to leave the AUTO mode.

Q: Just a quick question—why did you use the label *JUMPHERE instead of *ADD1 as we did before?

A: While there are many ways of making PILOT programs do things you wouldn't expect them to do, there are few ways worse than using a label more than once. When we entered the *WINDMILL module, we already had used the label *ADD1, which we did not erase in the module *COUNTER. If

PICTURE THIS!

we were to use *ADD1 again, PILOT would have to figure out which *ADD1 we wanted. PILOT's rule in cases like this is to look through all the statements in numerical order until it finds the first appearance of the label. Since *WINDMILL has lower line numbers than *COUNTER (as you can see by typing LIST), PILOT would use the correct version of *ADD1 whenever *WINDMILL is used. However, if we later decide to use *COUNTER, we would be quite surprised at the result since that module's jump command would send us to the *ADD1 label in *WINDMILL rather than send us to the *ADD1 label in *COUNTER.

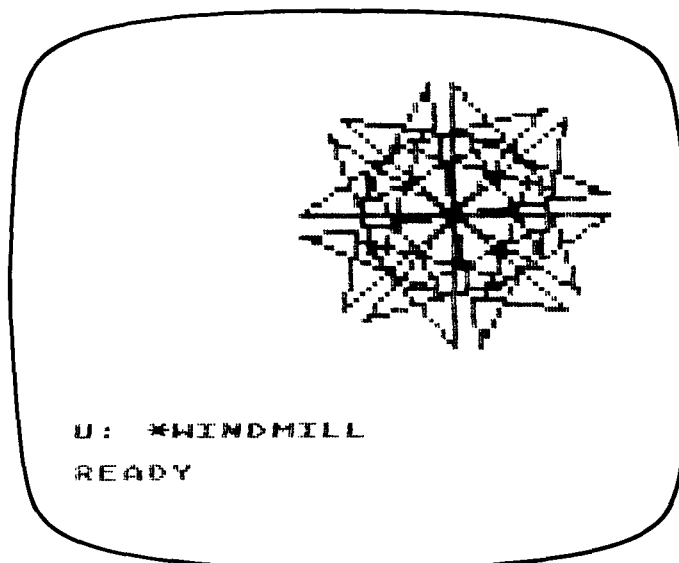
I think it is time for a PILOT rule:

A PILOT RULE When several modules are in the computer at the same time, make sure all the labels are different.

You might want to break this rule a few times to see just how confusing the results can be. If you don't mind, I will probably try to keep this rule in mind for the rest of this book!

Now let's see what we have accomplished. Type:

```
GR: CLEAR
U: *WINDMILL
```



Considering the small amount of effort it took to draw this picture, I think it looks kind of nice. To try the windmill out on other figures, we first need to build other modules. Let's enter the following:

```
GR: QUIT
REN 1000
AUTO
```

Next type:

```
*SQUARE
GR: 4(DRAW 25; TURN 90)
E:
*PENTAGON
GR: 5(DRAW 20; TURN 72)
E:
*HEXAGON
GR: 6(DRAW 15; TURN 60)
E:
```

and then change the module *PICTURE to try out each of these figures. In order to make the right changes in *PICTURE, we first have to press the RETURN key so that we can leave the AUTO mode and then we have to type LIST. After you press RETURN you will see all the deferred instructions displayed on the screen. As the screen fills up, lines with lower line numbers move off the top of the screen and lines with higher numbers are added at the bottom. When the listing is finished, you should see this on the screen:

```
1020 C: #A = 0
1030 *JUMPHERE
1040 C: #A = #A + 1
1050 U: *PICTURE
1060 GR: TURN 360/8
1070 J (#A < 8) : *JUMPHERE
1080 E:
1090 *PICTURE
1100 U: *STAR
```

PICTURE THIS!

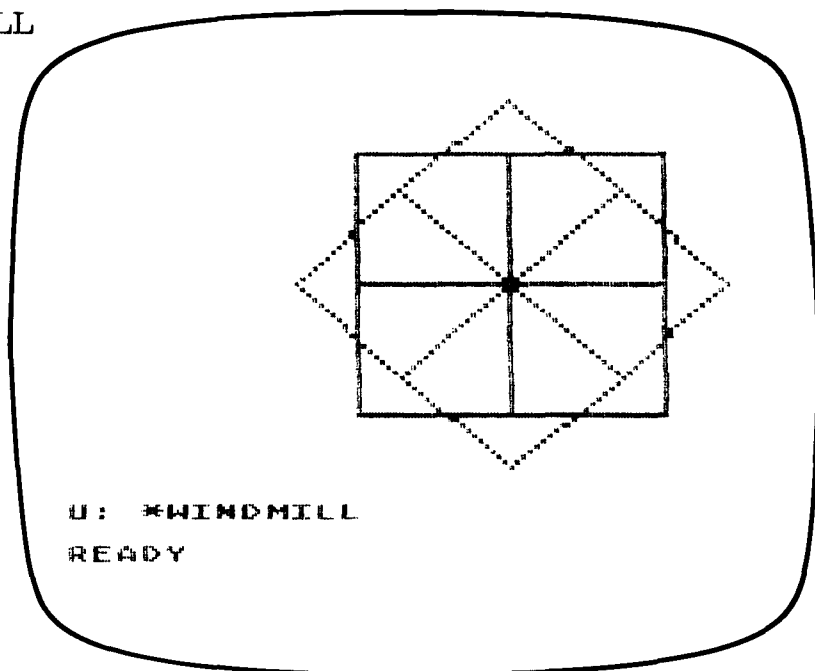
```
1110 E:
1120 *COUNTER
1130 C: #A=0
1140 *ADD1
1150 C: #A=#A+1
1160 U: *STAR
1170 GR: TURN 72
1180 J (#A<5): *ADD1
1190 E:
1200 *STAR
1210 GR: 5(DRAW 25; TURN 144)
1220 E:
```

To make the windmill draw a different picture, all we have to do is change the label in line 1090 to one of the new figures we just defined (*SQUARE, *PENTAGON, or *HEXAGON). To change *PICTURE so that it depicts the square, for example, we would type:

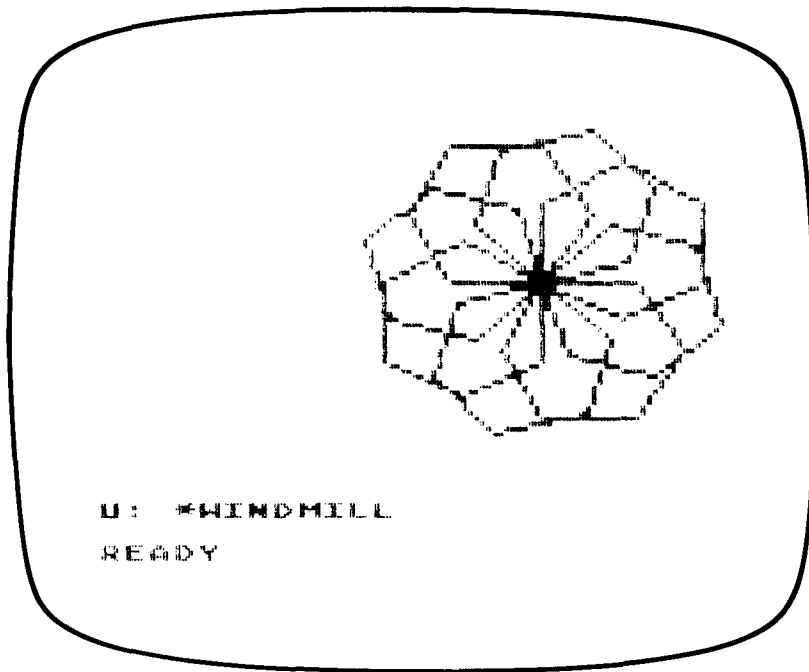
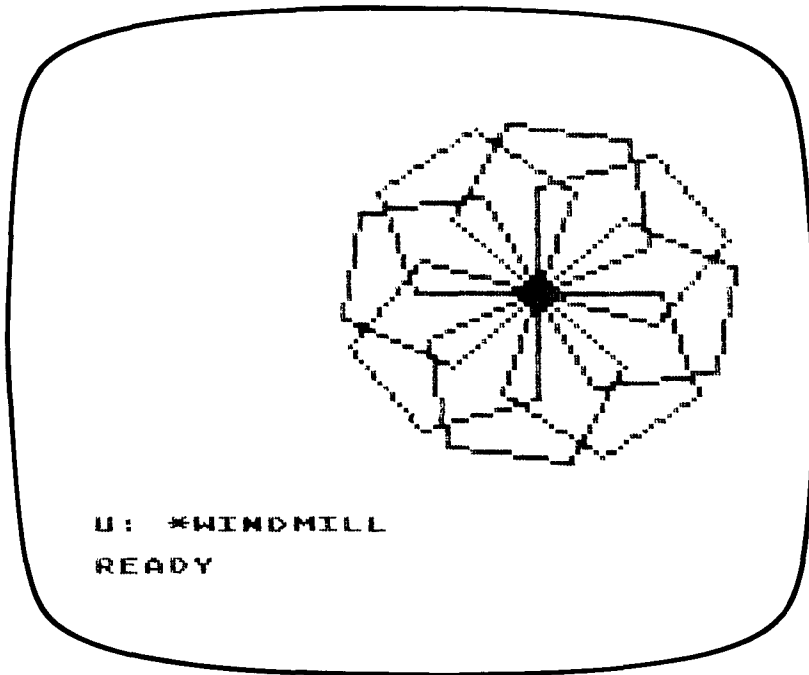
```
1100 U: *SQUARE
```

and then type:

```
GR: CLEAR
U: *WINDMILL
```



to see the result. The following three figures show the pictures we get when we use the square, the pentagon, and the hexagon in this way.



PICTURE THIS!

Can you think of any other shapes that might make interesting starter patterns for *WINDMILL? Add some of your own patterns, be sure you type REN 1000 before you type AUTO. If you neglect to do this, you might accidentally erase some of the modules you have already defined.

If you have a cassette recorder or a floppy disk drive, this would be a good time to save these modules so you don't have to enter them again some other time.

the variable variable

IN THE LAST chapter, we explored the use of variables to help us build a counter. Another powerful use of variables is in place of numbers we aren't ready to specify when we write a module.

To illustrate this, let's take a look at our general-purpose, regular-polygon plotter (GPRPP), which we discovered back in Chapter Five. You may recall that we discovered an important property of regular polygons—the sum of the angles turned while drawing a polygon always equals 360 degrees. We expressed this for the turtle by saying that a polygon of, for example, four sides would result from the command

```
GR: 4(DRAW 15; TURN 360/4)
```

a polygon with eight sides would result from the command

```
GR: 8(DRAW 15; TURN 360/8)
```

and so on.

PICTURE THIS!

Suppose we want to make a polygon module that we could use to draw many different polygons, the kinds of which depend on the value of a number we place in a variable. Let's do some experimenting to see how we would do this. First type:

NEW

to clear the computer's memory of old modules, loose bits, and old bottle caps. Then type:

AUTO

to get us ready to enter a new module. Since our variable for this module will be the number of *sides* in the polygon, let's use the variable #S. As far as the computer is concerned, we could have picked any letter, but the letter S might help us remember that it refers to the number of sides in the polygon.

Next enter:

```
*POLYGON
GR: GOTO 0,0; TURNT0 0           [start with the turtle at
                                home
GR: #S(DRAW 25; TURN 360/#S)    [try this out to see how it
                                works
E:
```

and remember to press RETURN again to leave the AUTO mode.

What we are about to learn is whether we can replace two numbers—the number of times we are to repeat something and the number to be divided into 360—by variables.

To find out, we should first type:

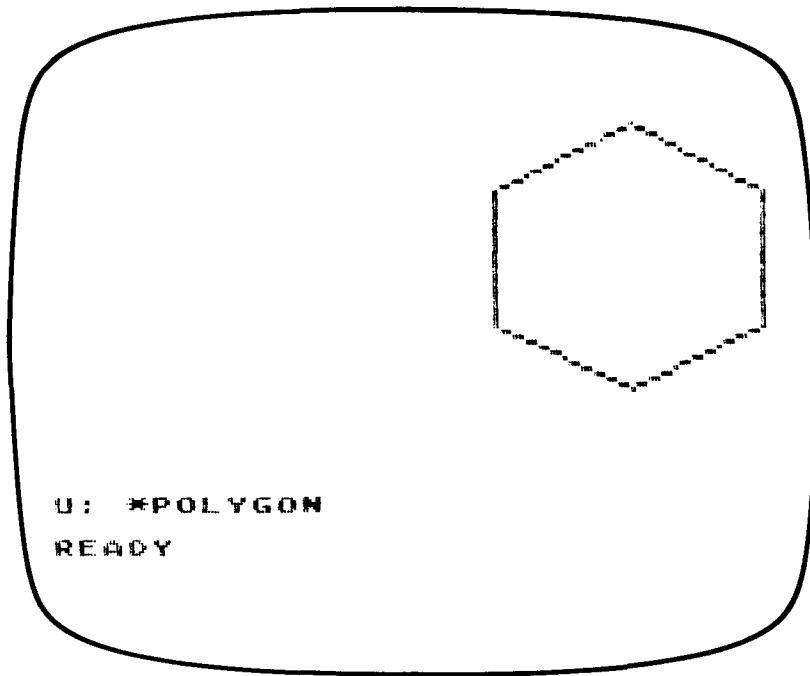
GR: CLEAR

and then make sure that #S contains the number of sides we want.

Let's start with a hexagon:

```
C: #S=6
U: *POLYGON
```

Wow! That works perfectly!



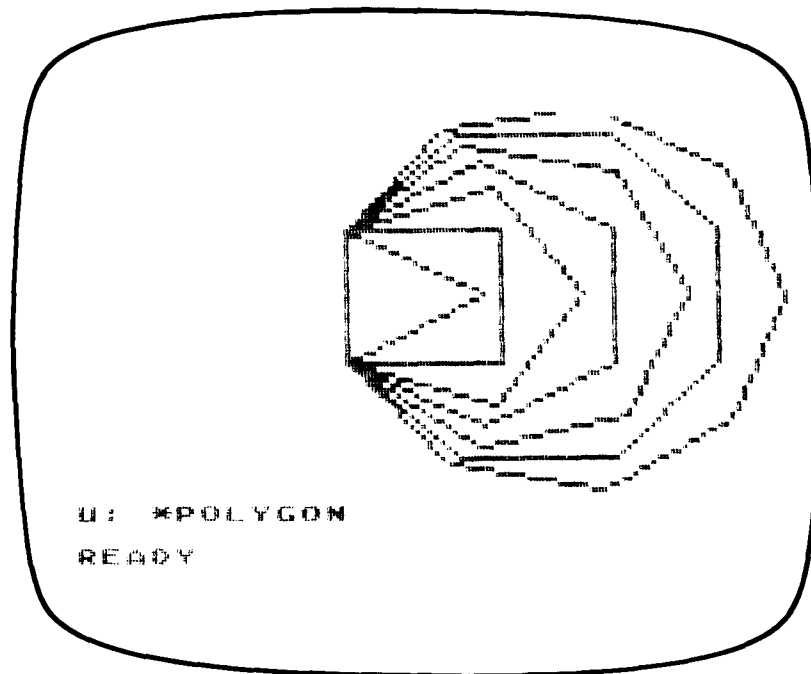
Next let's add some more polygons to the picture:

```
C: #S=3
U: *POLYGON
C: #S=4
U: *POLYGON
C: #S=5
U: *POLYGON
C: #S=7
U: *POLYGON
```


PICTURE THIS!

(Notice that 360 divided by 7 has a fractional part, so we return the turtle to its home at the start of each polygon in order to take care of round-off errors.)

```
C: #S=8
U: *POLYGON
C: #S=9
U: *POLYGON
```



You probably notice that the last figure we drew (a nonagon) has a small piece missing at its top. This is because we sent the turtle a little beyond its range for drawing pictures. We could fix this either by making the sides of the polygons shorter or by changing the turtle's starting location.

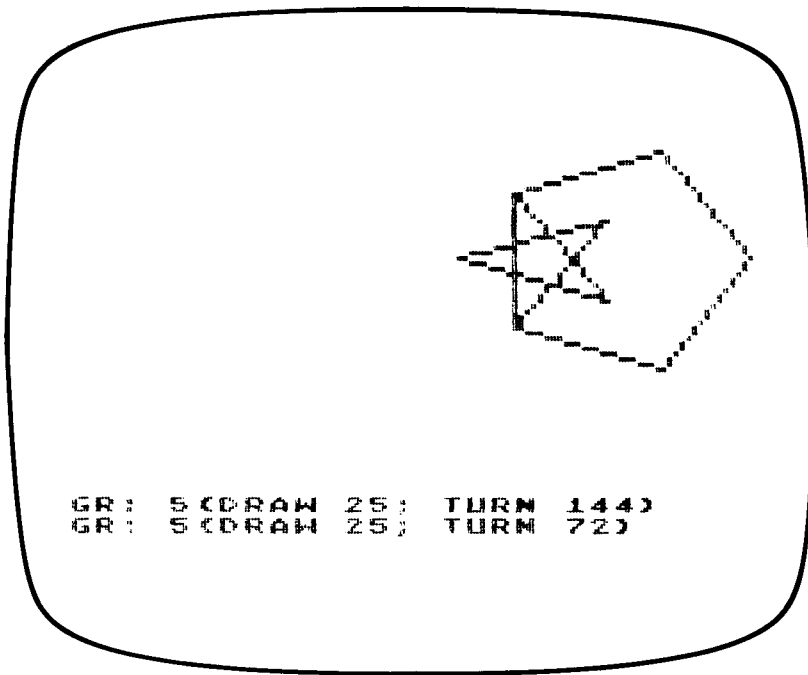
The turtle's star performance. . .

In Chapter Seven we learned how to draw a five-pointed star. Just to refresh our memory, type:

```
GR: GOTO 0,0; TURNT0 0; CLEAR  
GR: 5(DRAW 25; TURN 144)
```

You should now see a five-pointed star on your screen. Next let's draw another figure closely associated with the number 5—a pentagon. Enter:

```
GR: 5(DRAW 25; TURN 72)
```



PICTURE THIS!

Do you see **any** interesting differences between our instructions for a star and our instructions for a pentagon? The only difference I see is that we turn 144 degrees each time we draw a line on the star and only 72 degrees each time we draw a line on the pentagon. Hmmm—144 for a star and 72 for a pentagon—what can this mean? Do you know that 144 is $72 + 72$? Now *that* looks interesting. It means there is a very simple difference between a five-pointed star and a pentagon. To draw a five-pointed star, you *double* the angle you would use to draw a pentagon.

Multiplying a number or a variable by 2 is something PILOT can handle for us. PILOT uses the asterisk (*) as the symbol for multiplication. While you probably use an \times or a dot for this purpose, most computer languages use the asterisk. If we wanted to tell PILOT to multiply “two times three”, we would write $2 * 3$. So remember that there are *two* uses for the asterisk—as the first character in a label and as a symbol for multiplication.

If we write the instruction for a pentagon this way:

GR: 5(DRAW 25; TURN 360/5)

then we can write the instruction for a five-pointed star this way:

GR: 5(DRAW 25; TURN 360*2/5)

Clear the screen and try both of these commands to see how this works.

Q: Do we now have a rule telling us that we double the angle turned to convert a polygon into a star?

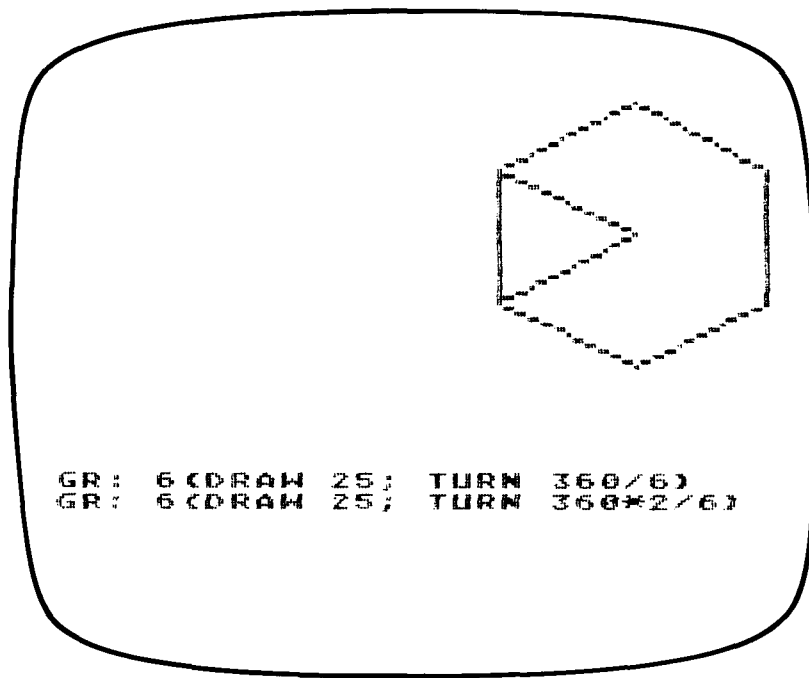
A: I don't know about that. Maybe we should try another figure to see how it works.

Let's see if we can change the instruction for a hexagon into that for a six-pointed star. Type:

```
GR: GOTO 0,0; TURNT0 0; CLEAR
GR: 6(DRAW 25; TURN 360/6)
```

which is the instruction for drawing a hexagon. Next let's see if this is the way to draw a six-pointed star:

```
GR: 6(DRAW 25; TURN 360*2/6)
```



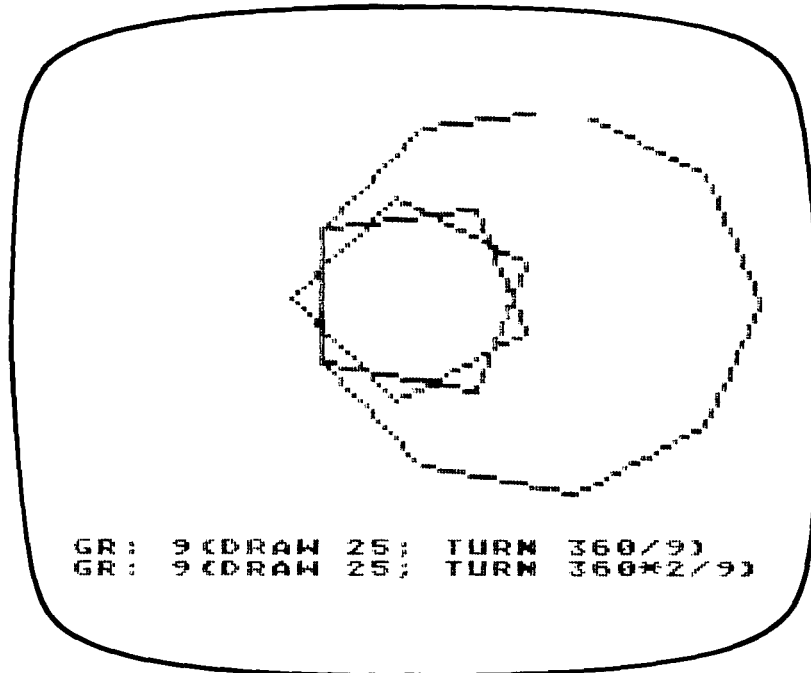
Hmmm. This second command doesn't give us a star at all—it draws a triangle instead! Are you confused by this? Let's try a nonagon next to see what that gives us. Type:

```
GR: GOTO 0,0; TURNT0 0; CLEAR
GR: 9(DRAW 25; TURN 360/9)
```

to see our nine-sided polygon. Now try this:

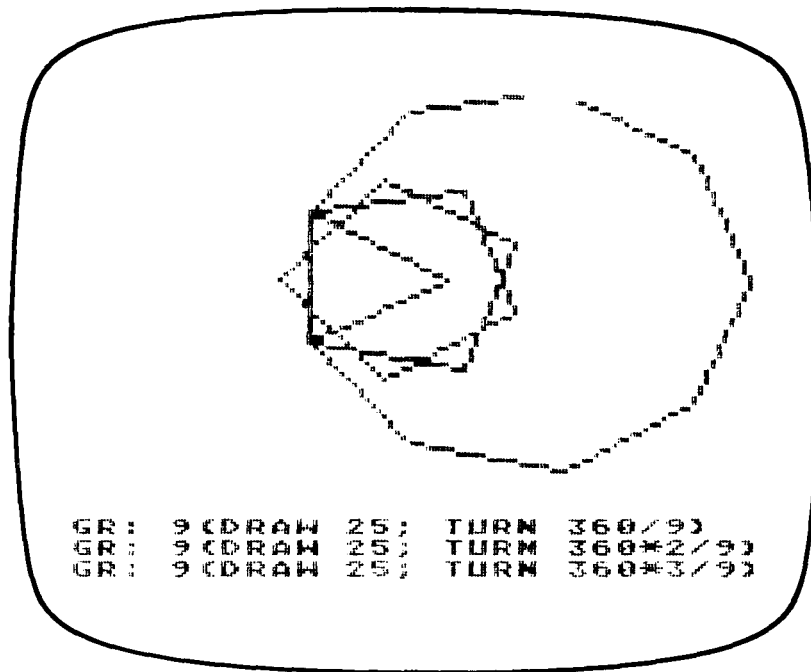
```
GR: 9(DRAW 25; TURN 360*2/9)
```

PICTURE THIS!



Aha! This gives us a nine-pointed star. Just for fun, let's try using a multiplier of 3 instead of 2 and see what happens.

GR: 9(DRAW 25; TURN 360*3/9)



Hmmm. There's that triangle again.

If we are ever going to figure out this star business, I think we need to be able to draw lots of figures based on lots of different polygons. Let's make a star module that will be as general purpose as possible. To do this we will make a module that looks a lot like *POLYGON, but that has one additional variable. This extra variable is the *angle multiplier*. We can call this variable #M to remind us that it is a multiplier. As we said before, PILOT will accept any letter for the variable name, but M reminds us that it is a multiplier. Just to be consistent, again we will use #S for the number of sides. Enter the following:

```
GR: QUIT
REN 1000
AUTO
```

and then type

```
*TRYSTAR
GR: GOTO 0,0; TURNTO 0; CLEAR
GR: #S(DRAW 25; TURN 360*#M/#S)
E:
```

and press RETURN again to leave the AUTO mode.

To use this module, we must put numbers into variables #M and #S. Let's set #M equal to 1 and see if *TRYSTAR then gives us regular polygons:

```
GR: CLEAR
C: #M = 1
C: #S = 4
U: *TRYSTAR
```

So far so good—I have a square on my screen, how about you? Next enter:

```
C: #S = 5
U: *TRYSTAR
```

PICTURE THIS!

This gives us a pentagon. Next try:

C: #M=2

U: *TRYSTAR

Before pressing RETURN, guess what this will give us. If you remembered that #S was set to 5, you probably guessed that it will draw a five-pointed star. Press RETURN to see if you were right.

This process of trying out new modules on things we already know about is a most important part of programming. This is how we can find out if we made any mistakes in writing the module. Computer folk call this process debugging; but, as I said before, computer folk talk funny sometimes.

When we were experimenting with different polygons and multipliers, we found that sometimes we got stars and sometimes we didn't. It would be nice if we could know if there are certain combinations of multipliers and sides (values of #M and #S) that give stars and other combinations that do not. One way of keeping track of this is to make, on a sheet of paper, a table that looks like this:

TRYSTAR TABLE		
#M	#S	Star? (Yes or No)
1	4	NO
2	4	

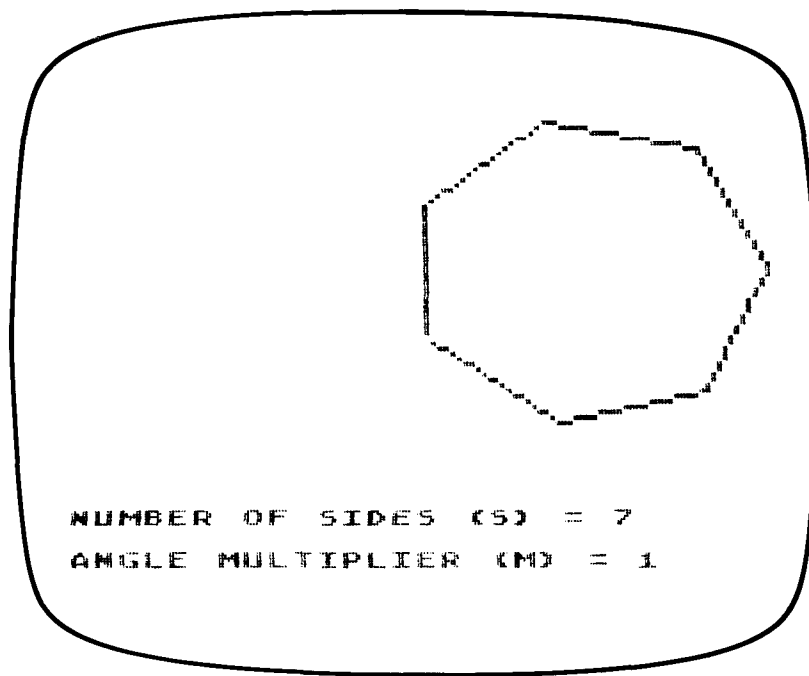
In each column we will list different values of #M and #S, and then see what happens when we use them. For example, if #M=1 and #S=4, we get a square. Try the second line of values (#M=2 and #S=4) to see what you get. Put your answer in the third column.

Q: How high can we go in choosing values for #M?

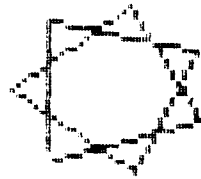
A: Well, if #M is equal to #S, then the turtle turns 360 degrees each time, and we never get a closed figure at all. So I recommend that #M always be less than the value of #S.

As it turns out, we don't even have to use values of #M that are more than half as large as #S. To see why, let's do the following experiment. Type:

```
C: #S=7      [set up for seven sides
C: #M=1      [set up for heptagon
U: *TRYSTAR  [heptagon is drawn on the screen
C: #M=2      [try for a star
U: *TRYSTAR  [hooray! we got one
C: #M=3      [try for another star
U: *TRYSTAR  [we just got another one
C: #M=4      [try another pattern
U: *TRYSTAR  [this is the same star as #M=3
C: #M=5      [try again
U: *TRYSTAR  [this is the same star as #M=2
C: #M=6      [try one more
U: *TRYSTAR  [heptagon (#M=1) is drawn on the screen
```



PICTURE THIS!



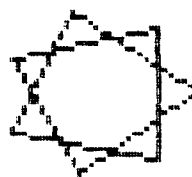
NUMBER OF SIDES (S) = 7
ANGLE MULTIPLIER (M) = 2



NUMBER OF SIDES (S) = 7
ANGLE MULTIPLIER (M) = 3

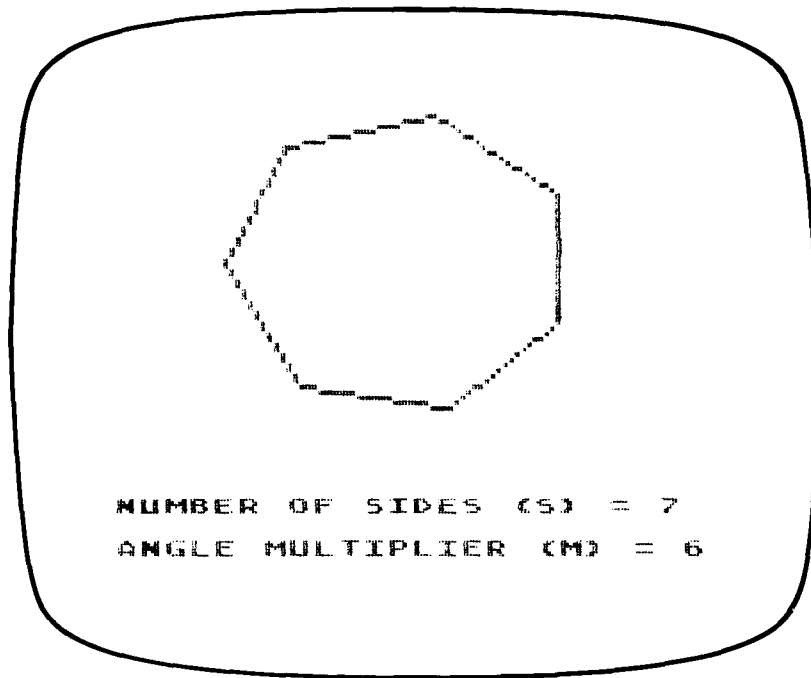


NUMBER OF SIDES (S) = 7
ANGLE MULTIPLIER (M) = 4



NUMBER OF SIDES (S) = 7
ANGLE MULTIPLIER (M) = 5

PICTURE THIS!



Do you notice anything interesting when you draw these figures? Once we reach values of #M that are greater than 3, each new figure becomes a mirror image of a figure we drew before. The new figure is drawn to the left of the home position, and the old one is drawn to the right of this position. We could spend many hours working with the properties of mirror images, but I want us to concentrate on drawing stars for a while longer.

If you try other combinations of #M and #S, you'll soon convince yourself that once the number stored in #M is more than half as large as that in #S, you'll be repeating an earlier figure.

We are now ready (at last!) to fill in our table of #M and #S values to see if we can discern why some patterns give stars and others don't. You should make a table of your own and see if your results agree with mine on page 103.

(We did not include #M = 1 in this table because we already know that this value will give us a polygon.)

TRYSTAR TABLE		
#M	#S	Star? (Yes or No)
2	4	NO
2	5	YES
2	6	NO
3	6	NO
2	7	YES
3	7	YES
2	8	NO
3	8	YES
4	8	NO
2	9	YES
3	9	NO
4	9	YES
2	10	NO
3	10	YES
4	10	NO
		(5 pointed)
5	10	NO
2	11	YES
3	11	YES
4	11	YES
5	11	YES

As we look at this table we see responses in the “yes” column and many in the “no” column. Let’s look more closely to see if there is a pattern to these results. First let’s see if there are any values of #S for which *all* values of #M (which are greater than 1) give a star. You can see that when #S is equal to 5, 7, or 11, all values of #M produce stars! It turns out that 5, 7, and 11 belong to a special set of numbers called *prime numbers*.

Prime numbers have the property of not being able to be formed by the multiplication of two whole numbers, both of which must be greater than 1. The number six is not a prime number because it can be written as $2 * 3$. We call 2 and 3 the *factors* of 6. The number seven, however, cannot be written as the product of two whole numbers, so it is a prime number.

PICTURE THIS!

So far we have discovered that polygons, the sum of whose sides equals a prime number, can be turned into stars without any problems whatsoever. But what about numbers like eight? When we set #S equal to 8, we find that some values of #M give stars and others don't. Let's look at this more closely. When #M is equal to 2 or 4, we don't get a star; and when #M is equal to 3, we do get a star. Now it so happens that 8 is $2 * 4$, so both 2 and 4 are factors of 8. We do get a star when we use 3, but 3 is not a factor of 8. If you look at the case when #S is equal to 9, you'll see the same sort of thing. If, in any particular case, #M and #S have a common factor, we will not get a star with #S points.

Now, at long last, we can create a rule for stars:

TURTLE RULE #2: A polygon with #S sides can be turned into a star with #S points by increasing each turning angle by #M times when #M is a whole number that shares no factors with the value of #S.

Wow, did *that* get technical!

At this point you might want to:

1. Go over this last section again;
2. Decide that stars are made by magic and that you don't need any other explanation;
3. Decide that you don't really care about stars anyway.

For a polygon with #S sides, what values of the turning multiplier #M give stars with *less* than #S points?

The turtle asks a question . . .

Now that we know how to get the turtle to use variables for all sorts of things, it might be nice to find out if we can have the turtle stop

in the middle of a module and ask us for some information before proceeding. Up till now we have pretty much set things up so that when the turtle does something, we just sit back and watch. We can make our modules *interactive*.

The interactive mode occurs when you are in direct communication with the computer and are able to get immediate responses to your messages. This is possible through the use of two PILOT commands, T: and A:.

Let's do an experiment to see what the T: command does. Press the SYSTEM RESET key and type

NEW

to erase old modules. Next type:

T: HELLO FRIEND, HOW ARE YOU?

When you press RETURN you should see

HELLO FRIEND, HOW ARE YOU?

neatly printed on the screen. Next type

T: I THINK YOU ARE NICE.

and press RETURN to see this sentence typed on the screen.

T: is the *Type command*. Any words that appear to the right of the colon will be displayed on the screen when this command is executed. The type command has lots of useful features. To see one of them, enter the sentence below. As you approach the edge of the display screen, just keep on typing and the computer will automatically shift you to the next line. Do *not* press RETURN.

T: NOW IS THE TIME FOR ALL GOOD PROGRAMMERS TO TRY
OUT THEIR SKILLS WITH PILOT.

PICTURE THIS!

As you were typing PROGRAMMERS, you probably noticed that the PROGRA ended up on the first line and the MMERS showed up on the next line. At the end of the second line, the word PILOT was broken between the L and the O.

Now press RETURN. Your screen should show:

```
NOW IS THE TIME FOR ALL GOOD  
PROGRAMMERS TO TRY OUT THEIR SKILLS  
WITH PILOT.
```

As you can see, one of the nice features of PILOT is that it avoids breaking words in the middle, which keeps things easy to read.

So far we have figured out how to have the computer put words on the screen. Next we have to figure out how to have the computer accept information from us while it is running a module. PILOT accomplishes this task with the *Accept command* (A:).

Let's say that we created a module in which we want to change the value of the number stored in the variable #C. To do this we might write (in the AUTO mode),

```
T: PLEASE ENTER A NEW VALUE  
A: #C
```

What do you think happens when the second command is executed? When PILOT comes across the A: command, it stops everything and waits for something to be typed from the keyboard. Whatever you type is stored in something called the *accept buffer*. The accept buffer is a special memory location where the computer keeps track of anything you have typed from the keyboard. When you press the RETURN key, the things you typed are placed in any variable (in our example, the numeric variable #C) appearing to the right of the colon in the Accept command.

Let's try out a simple example to see how this works. Type:

```
AUTO
```

and then enter

```
*POLYGON
T: HOW MANY SIDES DO YOU WANT?
A: #S
GR: GOTO 0,0; TURNTO 0; CLEAR
GR: #S(DRAW 25; TURN 360/#S)
J: *POLYGON
E:
```

Next, let's try out this module. Type:

```
GR: CLEAR
U: *POLYGON
```

Do you see the question in the bottom window?

HOW MANY SIDES DO YOU WANT?

Type the numeral 4 and press RETURN. Do you see a square on your screen? Good. You also probably notice that the lower screen repeats the question. This time type the word FOUR and press RETURN. Hmm. That doesn't accomplish anything at all. Now type 5 and press RETURN to get a pentagon on the screen.

This little exercise showed us two useful things:

1. We now know how to have PILOT accept information from us in the middle of running a module.
2. We know that the accept buffer can put only numbers inside numeric variables; it can't put words inside numeric variables.

Later on we may explore T: and A: some more, but now we are ready to draw some more pretty pictures!

10

squares and spirals

SO FAR WE have done a lot of experimenting with modules that draw things and that make lots of copies of other pictures at different angles on the screen. In this chapter we will explore ways of making objects that “grow.” To do this we will use many of the things we learned in the last chapter: using variables, typing questions on the screen, and accepting information from the user.

But before we get started, we are going to learn a way of finding the turtle’s location on the screen.

Have you ever wondered what the screen position of the turtle was after you moved it somewhere? PILOT has two special variables that give us this information. These are the variables %X and %Y. The percent sign (%) in front of these letters lets PILOT know that the values stored in these variables are put there by the turtle and not by you.

Contrast this with the properties of those variables that start with the number sign. You can put any number you want (within the limits we described before) into the variables #X and #Y, and you can use these variables for anything you want.

The turtle, however, has exclusive use of the variables %X and %Y. The variable %X always contains the horizontal location of the turtle on the screen. If the turtle is in its home position, the value of %X is 0. If the turtle is to the right of the home location, the number in %X will be larger than 0 but less than 80. If the turtle is to the left of the home position, the number in %X will be smaller than 0 but greater than -80. The variable %Y always contains the vertical location of the turtle on the screen. If the turtle is in its home position, the value of %Y is 0. If the turtle is above the home location, the number in %Y will be larger than 0 but less than 48. If the turtle is below the home position, the number in %Y will be smaller than 0, but greater than -32.

Let's build a module that we can use any time we want to know where the turtle is located. First be sure you are *not* in the graphics mode, and then type:

NEW

to clear out any clutter we left behind. Next type:

AUTO

to enter the deferred instruction mode. Now enter:

*WHERE

T:

T: THE HORIZONTAL LOCATION IS %X

T: THE VERTICAL LOCATION IS %Y \

E:

PICTURE THIS!

Press RETURN again to leave the AUTO mode.

Let me explain a couple of things about this module before we go on. First of all, if you type T: by itself, you get a blank line. (I tell you this now because it may make the display look prettier later on.) Also, you might not understand that the reversed slash (\) in the third line has a very special purpose. It keeps PILOT from advancing to the next line after finishing the type (T:) command. The reason we need this is that when a module comes to an end, PILOT types a blank line, the word READY on a second line, and puts the cursor on the next line. This uses three of the four lines that the text display can show in the graphics mode. Unless we do something special, we would see only the turtle's vertical location displayed on the screen since the line displaying the horizontal location would have "scrolled" off the top. By putting the reverse slash at the end of the line, we see one less blank line, so both the horizontal and the vertical turtle positions can be seen on the screen.

If you are confused by any of this, you should try the module *WHERE both with and without the reversed slash after %Y.

Now let's see how all of this works. Type:

```
GR: CLEAR
```

You and I know that the turtle is at its home location. Let's see how our new module works. Type:

```
U: *WHERE
```

When I did this I got the following two lines on the screen:

```
THE HORIZONTAL LOCATION IS 0  
THE VERTICAL LOCATION IS 0
```

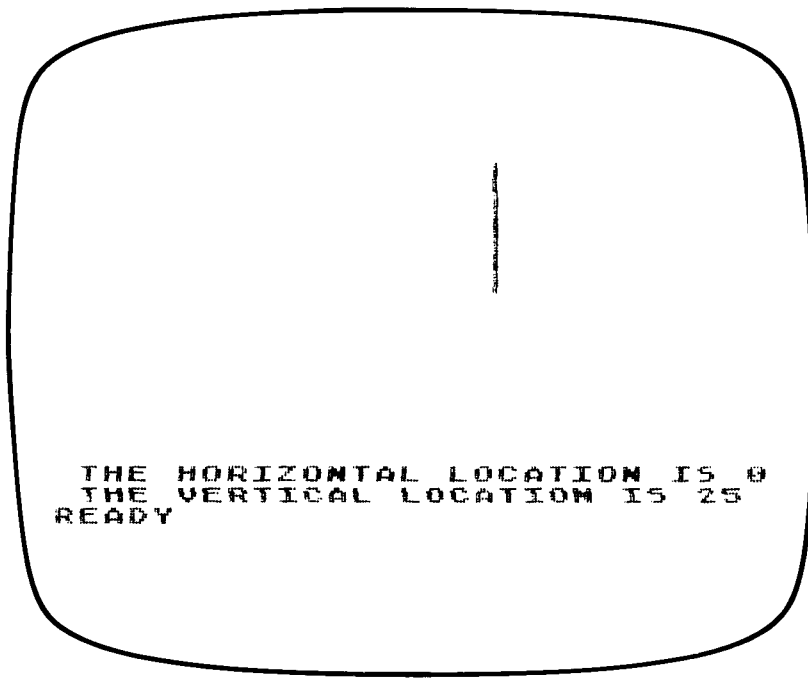
Q: Excuse me, but I am confused. When I created the module *WHERE, I used the *name* %X and the *name* %Y in the type commands. When I use this module, the screen shows the *numbers* contained in the variables %X and %Y. Why?

A: When a T: command is being used, PILOT looks at each character to see if it finds any special symbols. If it comes across a letter with a “#” or a “%” in front of it, PILOT recognizes it as a variable and prints its content rather than its name.

Let's do some more experiments. First type:

```
GR: DRAW 25
```

```
U: *WHERE
```



Since the turtle is pointing straight up after it draws this line, we see that the turtle is now at a new vertical location (25), but at the same horizontal one. You already knew that, of course, so this example probably doesn't make it any more obvious why %X and %Y are useful. Let's try something else:

```
GR: TURN 155; DRAW 38
```

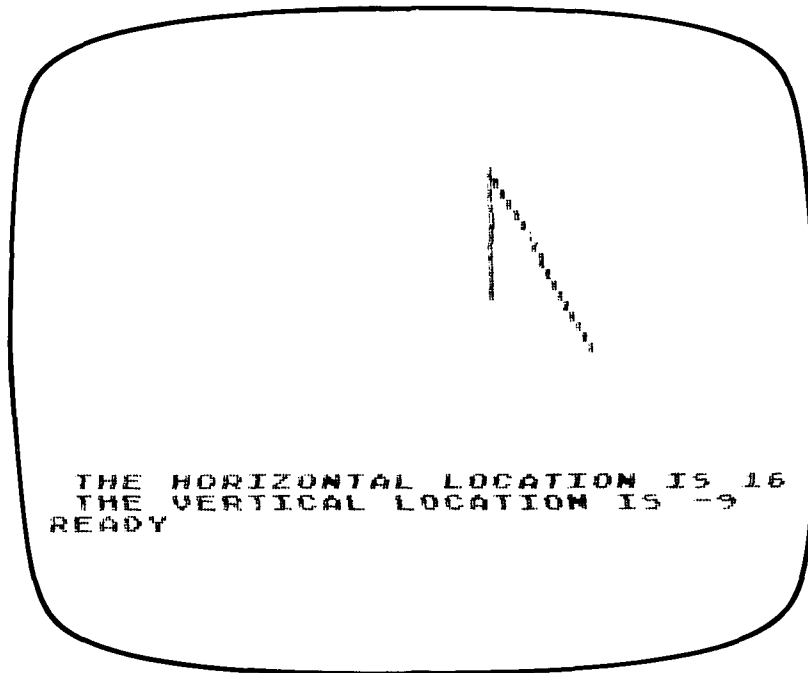
Now where do you think the turtle is located? You can see its location on the screen, but do you think you could use the GOTO statement

PICTURE THIS!

right now to put a different-colored dot on the end of this new line? Write down your guess for the present values of %X and %Y and then type:

U: *WHERE

How close was your guess?



As you can see, %X and %Y are really handy for finding the turtle's location when we have drawn lines that are neither altogether horizontal nor altogether vertical. Now let's get on with the main topic of this chapter!

The growing square . . .

In the last chapter we used variables to change the number of times we did something or to change the amount the turtle turned. Now let's see what happens when we use them to change how far the tur-

tle draws a line. To start out, we will use our recipe for drawing a square, but instead of drawing a line of fixed length (such as 25 screen units), we will use a variable for this value. Type:

```
GR: QUIT
REN 1000
AUTO
```

and then enter:

```
*SQUARE
GR: 4(DRAW #A; TURN 90)
E:
```

and press RETURN again to leave the AUTO mode.

If you now type:

```
GR: CLEAR
C: #A = 25
U: *SQUARE
```

you should see a familiar, yellow square on your display. To refresh your memory, we can build an interactive square-drawing module in the following way. Type:

```
GR: QUIT
REN 1000
AUTO
```

and then enter:

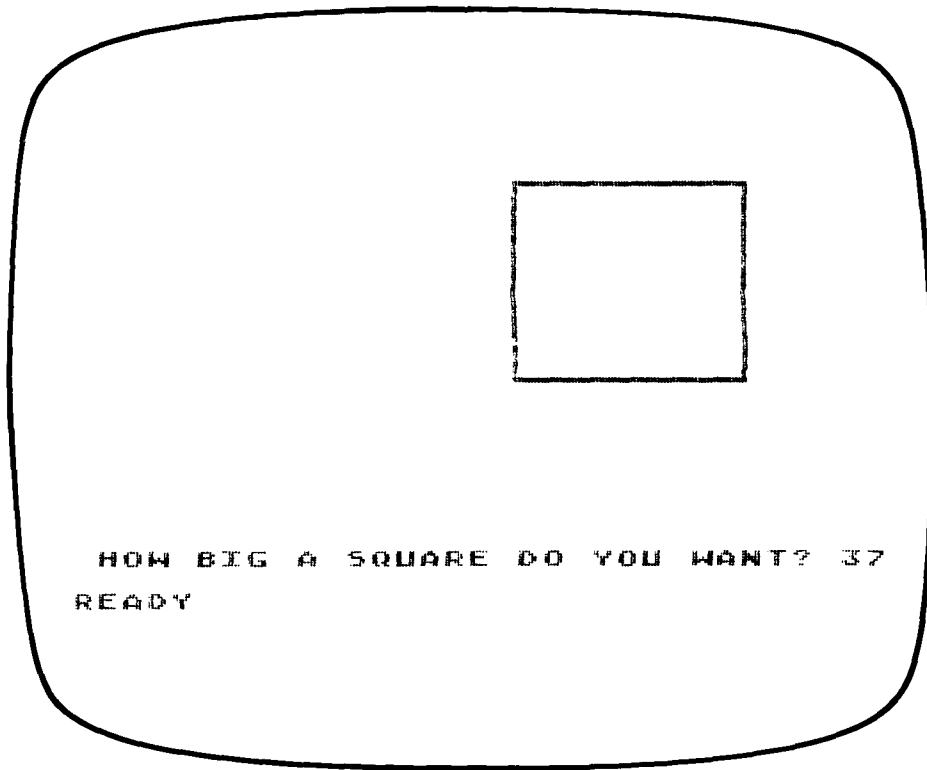
```
*TELLSQUARE
T: HOW BIG A SQUARE DO YOU WANT? \
A: #A
U: *SQUARE
E:
```

PICTURE THIS!

and press RETURN again to leave the AUTO mode. Notice that we used the reverse slash in this module. This will let you enter the answer to the question on the same line. Let's see how it works:

```
GR: CLEAR
U: *TELLSQUARE
```

Type a number (I used 37) and press RETURN. *Voilà!* A nice square is drawn on the screen.



Now we will make another module that will do something really different. We will learn how to make a square grow in front of your very eyes! Type:

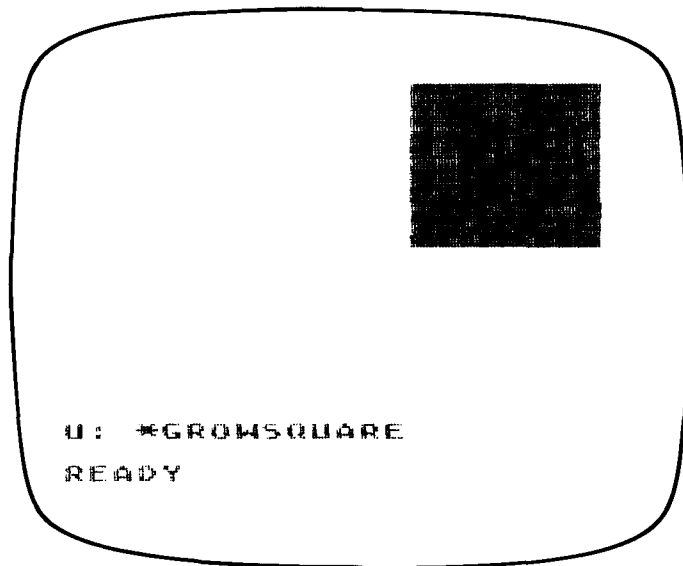
```
GR: QUIT
REN 1000
AUTO
```

and then type:

```
*GROWSQUARE
C: #A = 0           [start with zero length
*JUMPHERE           [come here for the next round
U: *SQUARE           [draw a square with length #A
C: #A = #A + 1       [make the sides longer by one
J (#A < 31) : *JUMPHERE [draw another square if #A is
                        less than thirty one
E:
```

Now, let's try it out. Again press RETURN to leave the AUTO mode and then type:

```
GR: CLEAR
U: *GROWSQUARE
```



Wow! That's interesting. We seem to have a square blob instead of a growing square!

In order to make a growing square, we need to erase the square that we previously drew and then draw a new one that is larger. If this happens quickly enough, the square will appear to grow on the screen.

PICTURE THIS!

We will now make a new module, *GROW, which should do this for us. First type:

```
GR: CLEAR
REN 1000
AUTO
```

and then enter:

```
*GROW
C: #A = 0
*HERE
GR: PEN ERASE
U: *SQUARE
C: #A = #A + 1
GR: PEN YELLOW
U: *SQUARE
J (#A < 31) : *HERE
E:
```

Now, after you press RETURN again to leave the AUTO mode and type:

```
GR: CLEAR
and then
```

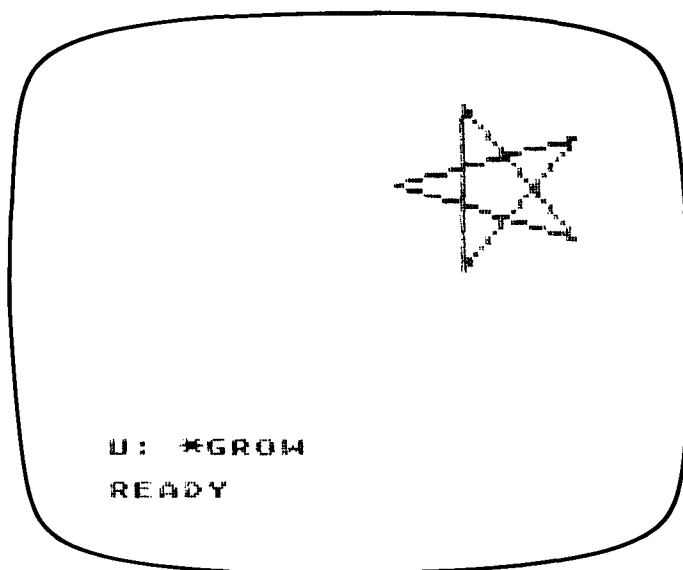
```
U: *GROW
```

you should see a yellow square growing on the screen. You probably noticed some flickering while this was going on. This is because our poor turtle can't do everything quite fast enough to prevent the flicker, but I think the result looks pretty good anyway.

We can make other objects grow as well. If you next make a new module that looks like this:

```
*STAR
GR: 5(DRAW #A; TURN 144)
E:
```

and change the references from *SQUARE to *STAR in the module *GROW, you will have a star growing instead of a square. This star starts from nothing in the center of the screen and ends up like this:



Squirals and spirals . . .

The objects we have experimented with so far pretty much have grown without leaving a record of their growth on the screen. If we stop the growth process at any time and look at the image, we don't really have a way of knowing that the object grew to its present shape. This is rather similar to the way we view our own growth as persons. When we are children, we get a little bit bigger every day, but we don't grow rings as trees do.

There is another way that some plants and animals grow—a way that leaves a record of their growth in the form of spirals. Have you ever looked closely at snail shells? When the snail is a tiny baby, its shell is very small. As it grows, its shell gets a little bigger and a spiral starts to form. By the time a snail is fully grown, its shell will have spiraled around itself many times. If you look at the arrange-

PICTURE THIS!

ment of the petals on many flowers (daisies, for instance), you can also see spirals that are related to growth. Still another good place to look for spirals is on the top of pine cones.

In fact, most of the spirals found in nature come from living things. Two exceptions are the spirals in tornadoes and whirlpools. I can't think of any other exceptions, can you?

In the remainder of this chapter we are going to explore some extremely interesting spirals that we can have the turtle draw for us. Let's start out with a really simple pattern—a "square spiral", which we will call a squiral. Suppose we want to draw a picture that looks like a square, except that each side is longer than the previous side. To do this we can create a module called *SQUIRAL. First type:

```
GR: QUIT
REN 1000
AUTO
```

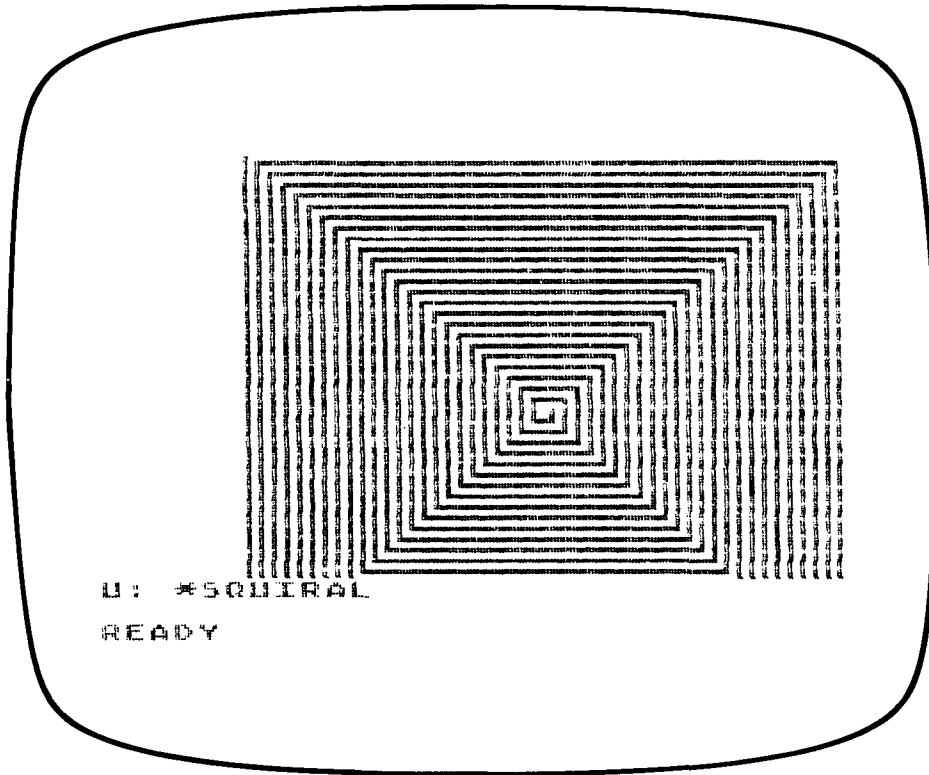
and then enter:

```
*SQUIRAL
GR: CLEAR; GOTO 0,0; TURNT0 0    [clear the screen
C: #A = 0                        [set starting length to zero
*DRAWLINe                        [come here on next round
GR: DRAW #A; TURN 90             [draw one line and turn
C: #A = #A + 1                   [make the side longer
J (%Y<48) : *DRAWLINe            [if picture isn't at the top
E:                                of the screen, draw
                                another line
```

Press RETURN again to leave the AUTO mode. Notice that we're using the number stored in the variable %Y to tell us when the squiral has grown too big. Now let's try this out. Type:

```
GR: CLEAR
U: *SQUIRAL
```

Wow! That is a rather impressive spiral. Because we turned 90 degrees each time, each corner is nicely nested with the one preceding



it. As it turns out, we can draw some even more fantastic pictures by changing this angle somewhat.

We need to make some changes in *SQUIRAL that will let us specify our own angle each time we use the module. To do this, type:

```
GR: QUIT
LIST 0, 100
```

Your screen should now show the following listing:

```
10 *SQUIRAL
20 GR: CLEAR; GOTO 0,0; TURNT0 0
30 C: #A=0
40 *DRAWLINE
50 GR: DRAW #A; TURN 90
60 C: #A=#A+1
70 J (%Y<48) : *DRAWLINE
80 E:
```

PICTURE THIS!

To change this program so that it's more interactive, use the variable #B as the angle you want to specify. Then type:

```
32 T: WHAT ANGLE WOULD YOU LIKE? \  
34 A: #B  
50 GR: DRAW #A; TURN #B
```

Now we are ready to draw some more pictures. Type:

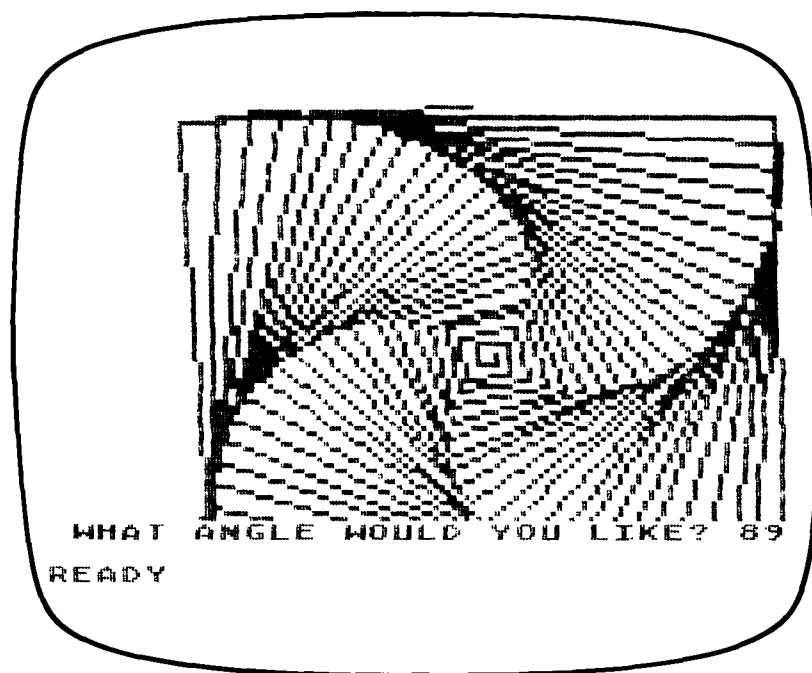
```
GR: CLEAR
```

and then enter:

```
U: *SQUIRAL
```

In response to the question "WHAT ANGLE WOULD YOU LIKE?" enter 90 and press RETURN. If your screen shows the same square spiral we saw before, you know everything is still working properly.

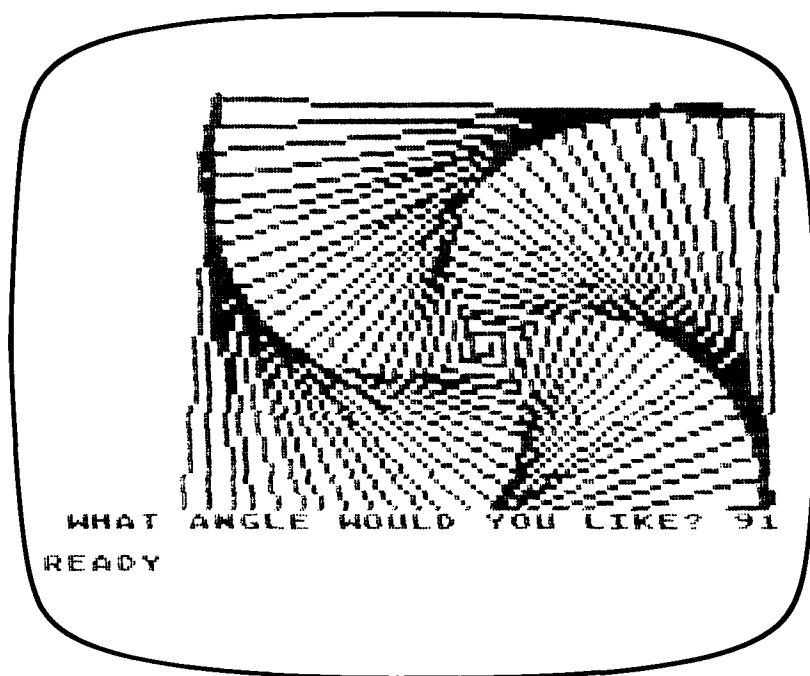
Use *SQUIRAL again, but this time enter 89 instead of 90. This is what I saw when I did it:



120 Now *this* is an interesting picture. By making the turning angle a little different from 90 degrees, we drew a picture that shows four

spiral arms radiating out from the center. The arms are turning to the left. Do you know why this happens? We do not quite make a square on the screen because we turn 89 degrees each time. By the time we have turned four times, we are off by 4 degrees. And by the time the turtle has gone around again, we are off by 8 degrees. As a result of this shift in angles, the corners of our almost-squares do not nest as nicely as they did when we used 90 degrees for the angle. The four spiral arms that we see are formed by the corners bumping into one another as each turn of the squiral deviates farther and farther from the 90-degree angles of our first squiral.

As it turns out, this same type of thing will happen if we turn a little too much. Can you guess what will happen if we use *SQUIRAL with an angle of 91 degrees? Try it and see if your guess is right!



This time we got four spiral arms that went off to the right instead of to the left. Do you find it amazing that such a big difference in the picture can be formed by such a small change in the angle?

You probably have seen examples similar to this if you have ever seen two window screens stacked together. There is no extra pattern produced if both screens are held at the same angle. If one screen is angled ever so slightly, you begin to see light patches and dark

PICTURE THIS!

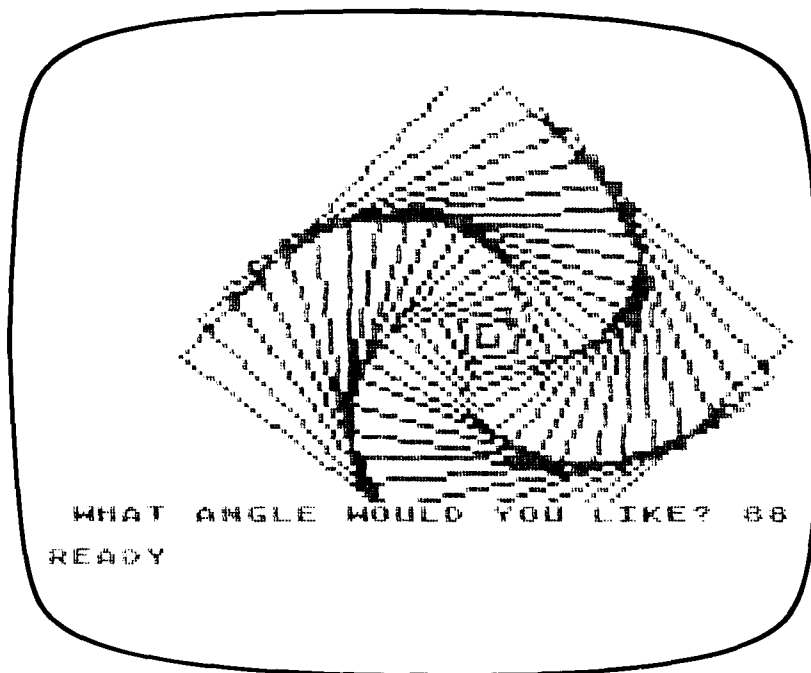
squares. As you increase the angle between the two screens, the patches get smaller and closer together. If you don't have two screens handy, you can see this same sort of effect by holding two identical combs together and looking through the teeth. When one comb is rotated at an angle different from the other one, you should see light and dark lines.

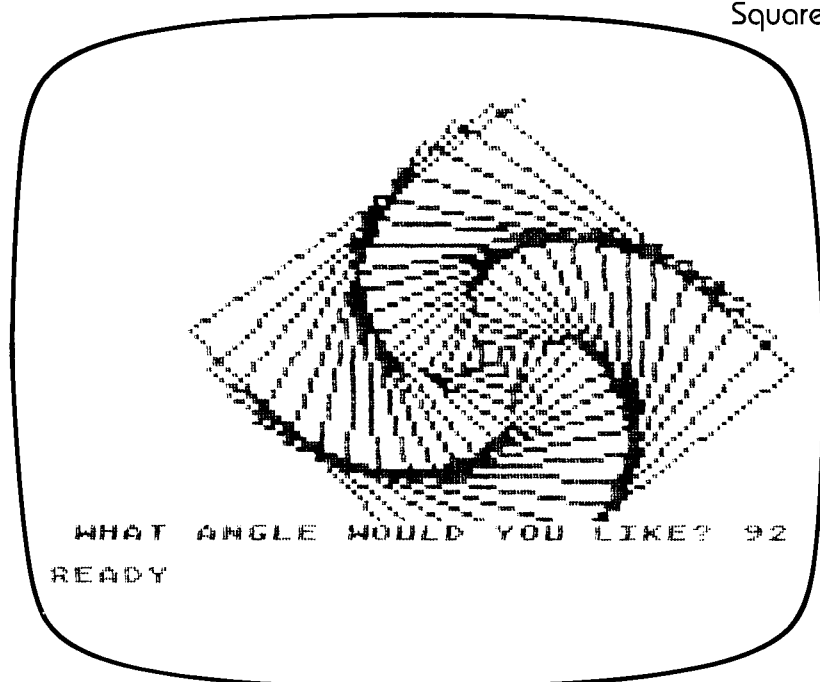
These extra patterns that are superimposed on the finer pattern of two interfering images are called *moire patterns*. You might want to be on the lookout for them.

Q: You said that if I held two screens together the light and dark patches would get closer together as I turned the screen more. Does that mean the spiral arms will turn faster in *SQUIRAL if we use angles that are farther away from 90 degrees?

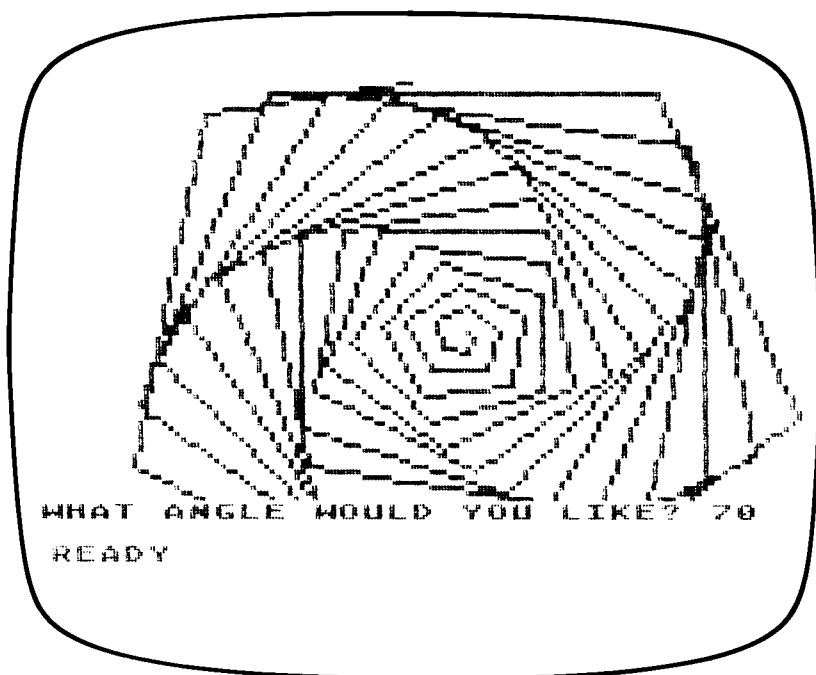
A: I know of only one good way to find out. Let's try it!

The following two figures were produced by using *SQUIRAL with angles of 88 and 92 degrees, respectively. It looks as though everything went according to plan.

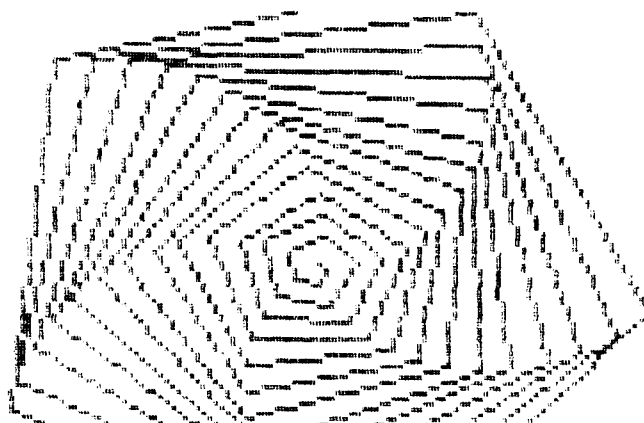




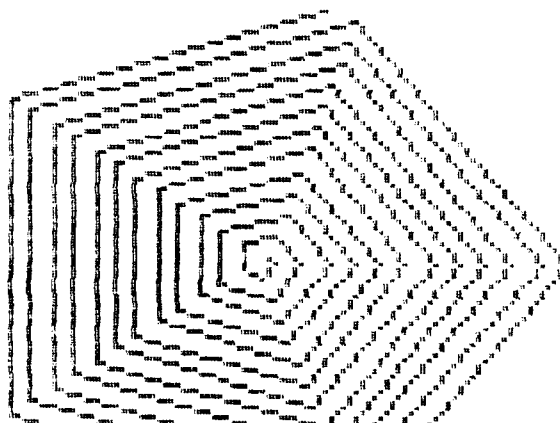
Now let's try some other angles that we know about. For example, 72 degrees should give us a pentagonal spiral. The following set of figures came from using angle values of 70, 71, 72, 73, and 74 degrees, respectively.



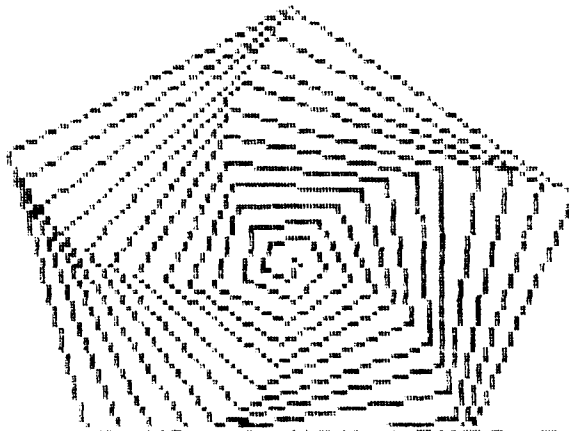
PICTURE THIS!



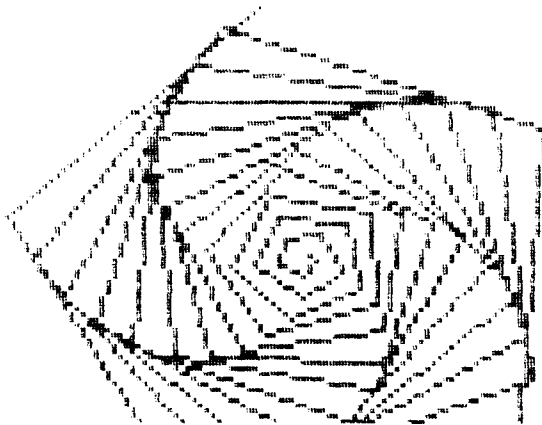
WHAT ANGLE WOULD YOU LIKE? 71
READY



WHAT ANGLE WOULD YOU LIKE? 72
READY



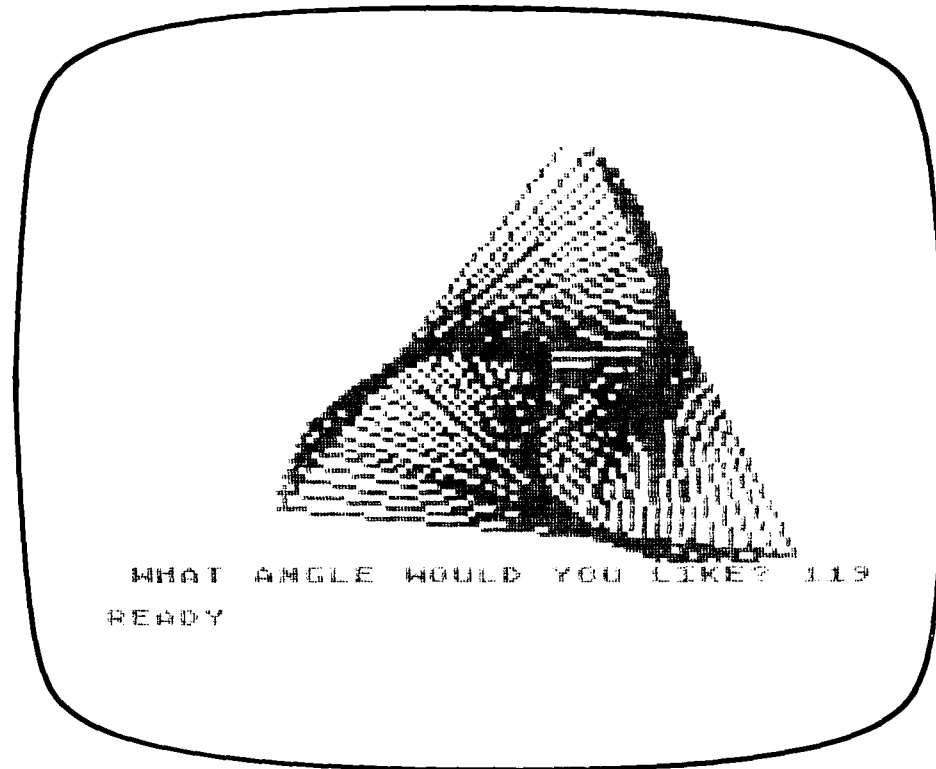
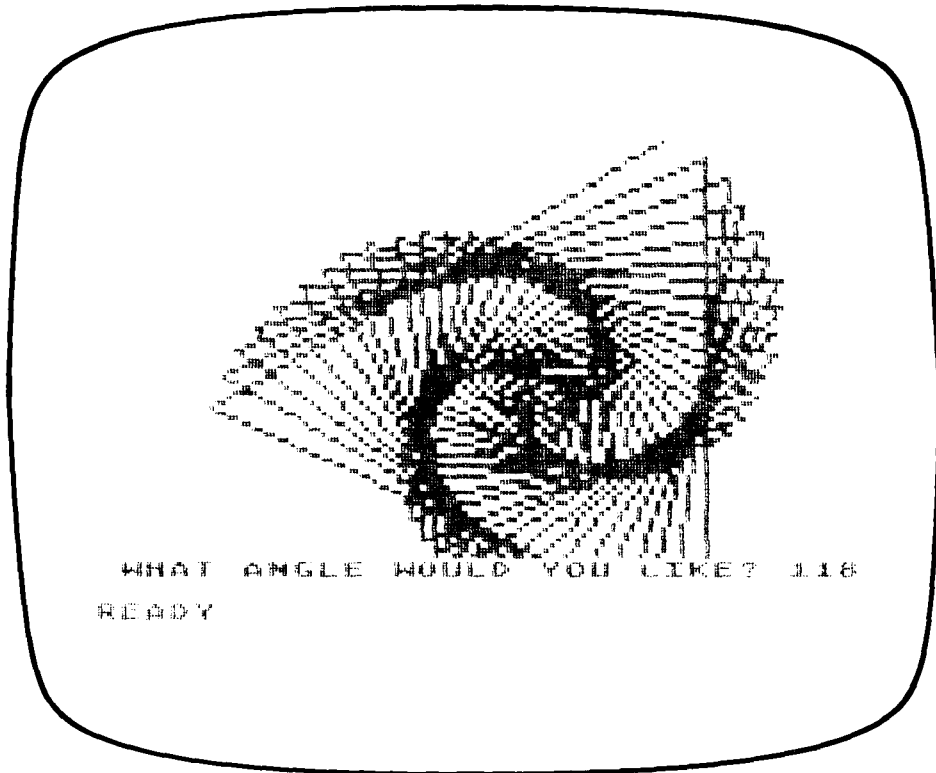
WHAT ANGLE WOULD YOU LIKE? 73
READY

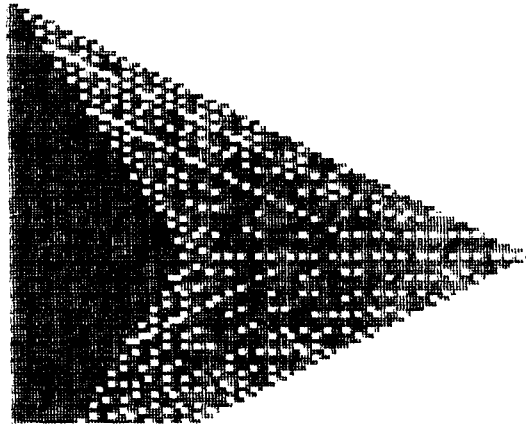


WHAT ANGLE WOULD YOU LIKE? 74
READY

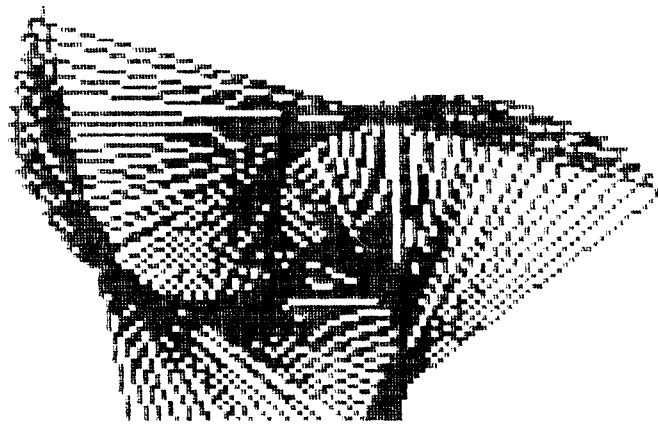
PICTURE THIS!

Next let's try using angles near 120 degrees and see what happens:



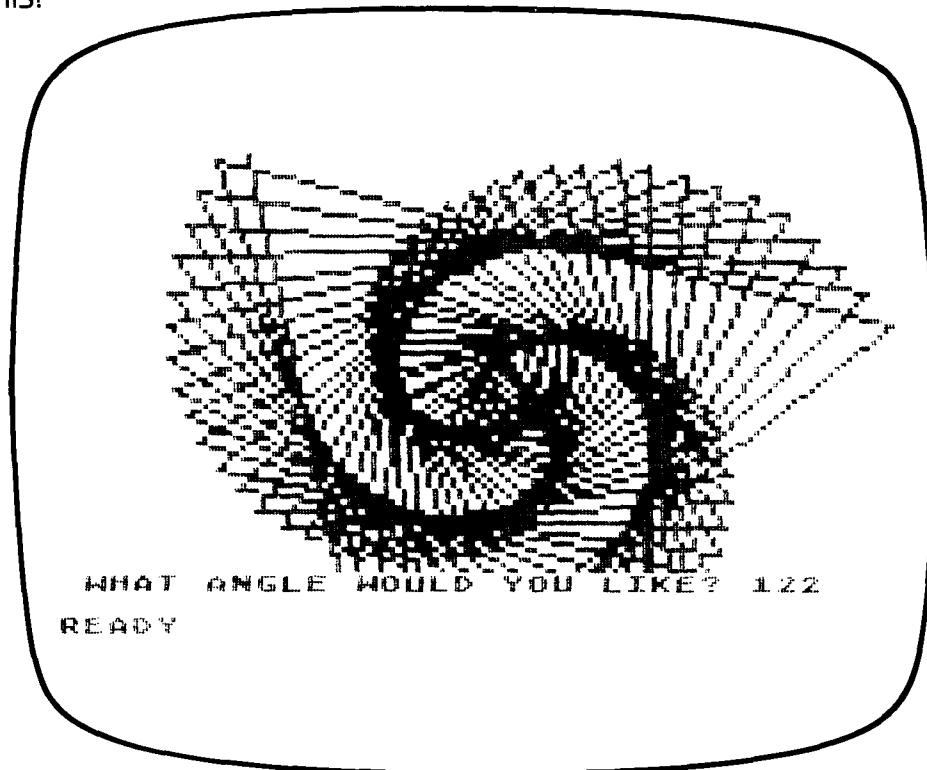


WHAT ANGLE WOULD YOU LIKE? 120
READY

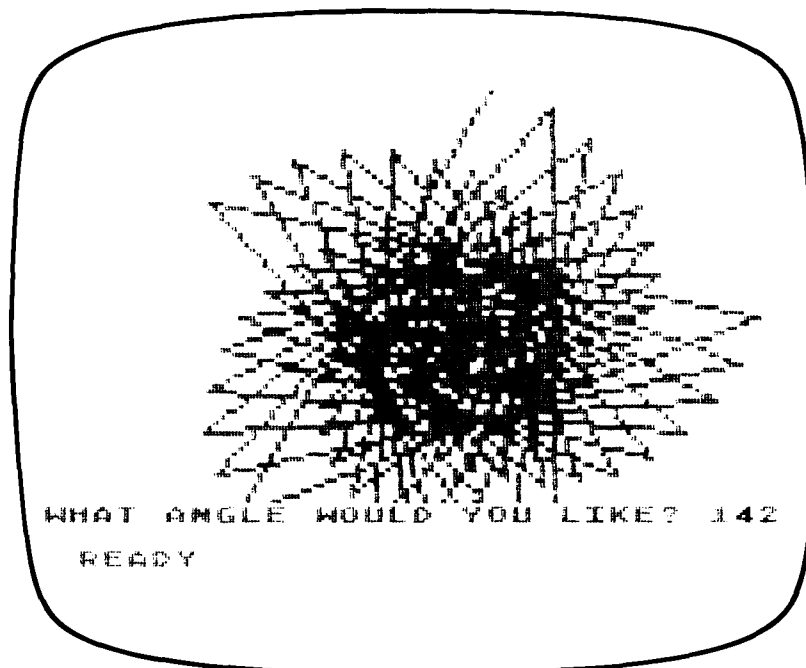


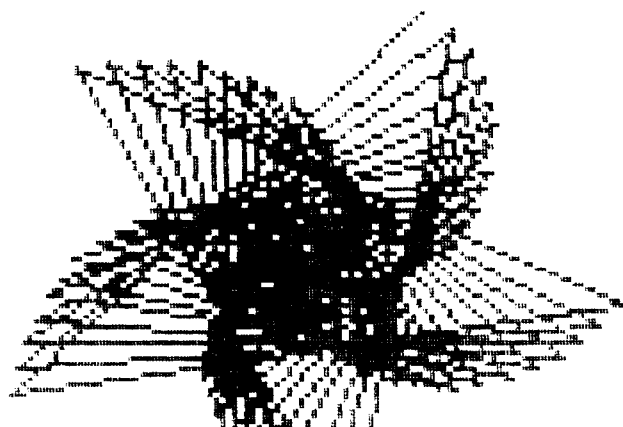
WHAT ANGLE WOULD YOU LIKE? 121
READY

PICTURE THIS!

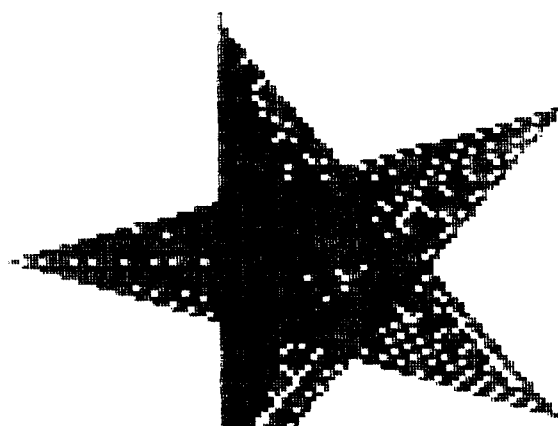


Finally, let's try a pattern with which we are already familiar—the five-pointed star. Since you know that an angle of 144 degrees will give us a star shape, we will try using *SQUIRAL with angles from 142 to 146 degrees:



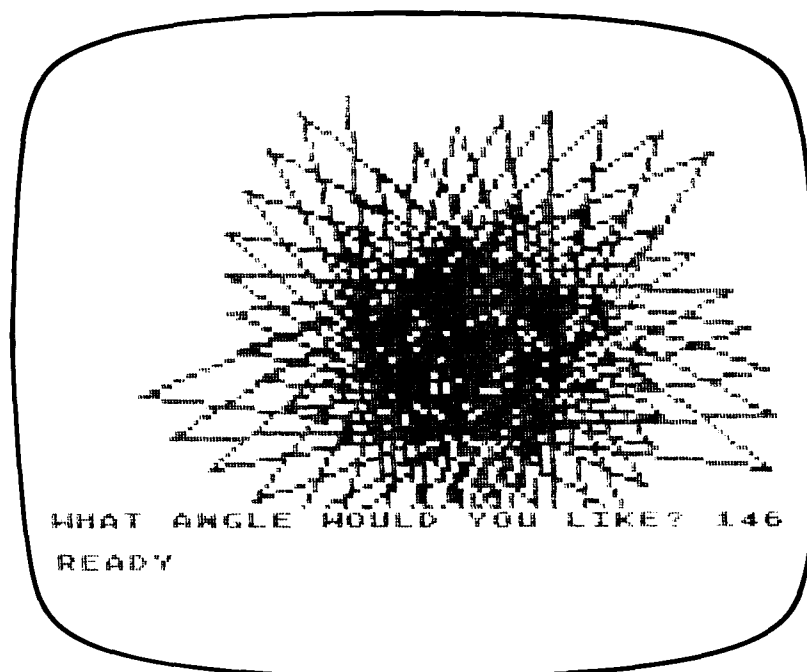
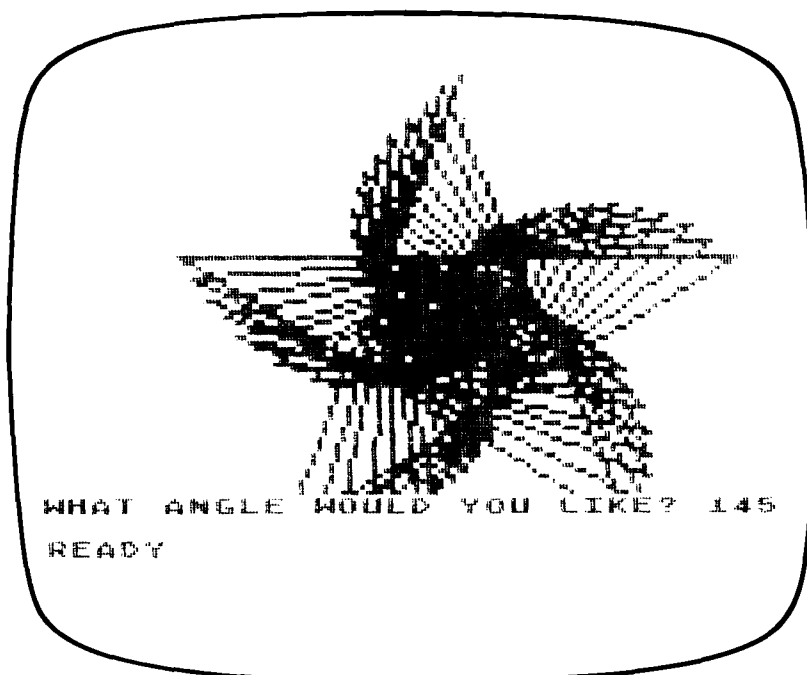


WHAT ANGLE WOULD YOU LIKE? 143
READY



WHAT ANGLE WOULD YOU LIKE? 144
READY

PICTURE THIS!



You should look closely at all of these turtle drawings to be sure you understand both the number of spiral arms you see and the direction in which the arms are turning.

Try some angles of your own. You might be very pleasantly surprised by the results!



drawing curves

SO FAR ALL of our pictures have been made with straight lines even though some of the squirrels appeared to have curved arms. One of the things we will do in this chapter is learn how to make the turtle draw curved lines.

The turtle draws a circle . . .

Let's start with a very important curved figure—the circle. Do you know how to draw a circle? I know at least three ways:

1. We could trace around the edge of a circular object, such as a coin. Since this requires that we have a circle to start with, I don't find this a terribly interesting way of drawing a circle.
2. We could use a special tool called a compass that will help us draw a circle centered around a point of our choosing. This is the usual method for many of us when drawing circles, and it works just fine if we have a compass.

PICTURE THIS!

Quite often we don't have any special tools handy, and we still want to draw a circle. So at these times we just draw a circle without any tools. The approach we use then is:

3. Draw a little bit, turn a little bit, and repeat this process until we get a circle.

Try out this approach by walking in a circle. If you take little steps and turn just a little bit after each step, your path will trace out a pretty big circle. If you turn a little more after each step, the circle will be smaller. You should try this yourself until you are convinced that the more you turn with each step, the smaller your circle will be.

If you walked in such a way that you turned by 1 degree with each step you took, how many steps would it take you to draw a circle?

Do you remember Turtle Rule #1? If your answer is 360 degrees you are to be congratulated because this is exactly right! To get back where it started, the turtle has to turn a total of 360 degrees. If the turtle turns only 1 degree with each step, it needs to take 360 steps to get back to its starting place.

Now let's try out this process with the turtle and see how it does. We will make a module that lets the turtle take a little step (1 is the smallest step it can take), and then turn by 1 degree after each step. Be sure you are not in the graphics mode and then type:

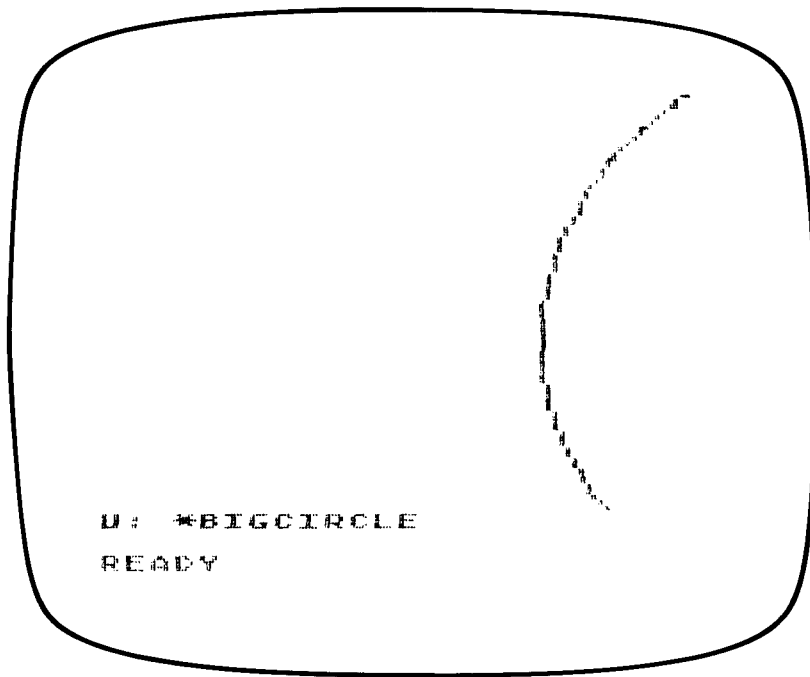
```
NEW  
AUTO
```

and then enter:

```
*BIGCIRCLE  
GR: 360(DRAW 1; TURN 1)  
E:
```

and press RETURN again to leave the AUTO mode. Turning by 1 degree each time is turning by the smallest amount we can. Let's see what this module gives us. Type:

```
GR: CLEAR
U: *BIGCIRCLE
```



Hmmm. This ends up drawing a circle so big that only part of it shows up on the screen. Maybe we should make a new module that lets us change the amount we turn after each step. Type:

```
GR: QUIT
REN 1000
AUTO
```

and then enter:

```
*DRAWCIRCLE
T: HOW MUCH WOULD YOU LIKE TO TURN? \
```

PICTURE THIS!

```
A: #A
GR: GOTO 0, -30; TURNT0 -90 [this starts the circle in a con-
                             venient place and starts the
                             turtle drawing at the left of
                             the screen instead of toward
                             the top
U: *CIRCLE
E:
```

This interactive module uses another module called *CIRCLE. In defining *CIRCLE, we want to have the turtle draw one step and then turn by the amount we enter from the keyboard. This amount is stored in the variable #A. The number of times we need to repeat this step-and-turn activity is given by dividing 360 by the number stored in #A. For example, if that number is 1, we need to repeat the process 360 times to get a closed circle. If the number is 2, then we need to take $360/2$, or only 180 steps to draw our circle.

Without leaving the AUTO mode, let's try typing this:

```
*CIRCLE
GR: 360/#A(DRAW 1; TURN #A)
```

Whoops! That didn't work did it! My screen shows

```
GR: 360/#A(DRAW 1; TURN #A)
```

```
*** WHAT'S THAT ***
```

with the number sign shown in reverse field. My guess is that this means the quantity in front of the left parenthesis needs to be either a number or a variable (we already know that these work) but cannot be anything that uses an operation (such as dividing a number by a variable). We can fix this problem by defining a new variable, #B, this way. Enter

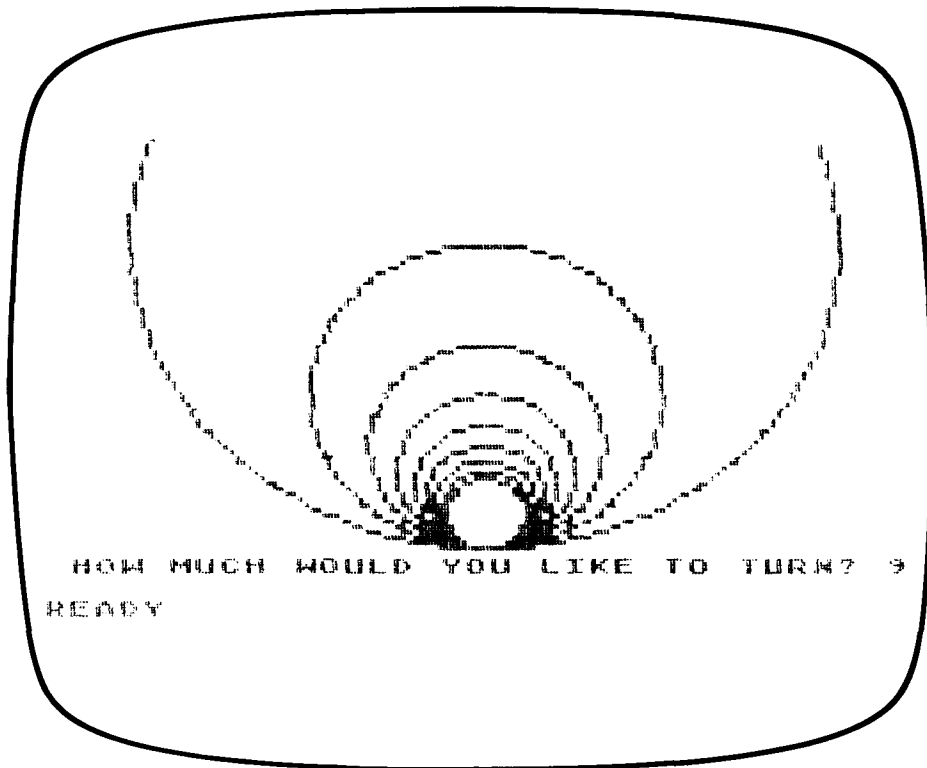
```
C: #B = 360/#A
GR: #B(DRAW 1; TURN #A)
E:
```

Well, at least PILOT accepted everything we typed; now let's try it all out to see how it works. Press RETURN again to leave the AUTO mode and then type:

```
GR: CLEAR
U: *DRAWCIRCLE
```

When you see the question HOW MUCH WOULD YOU LIKE TO TURN? you might first enter 1 and then press RETURN to see if you get a circle the same size as the one you got with *BIGCIRCLE. You can see more of the circle now since we moved the turtle down to give it more room. But we still don't have the circle completely on the screen.

I used *DRAWCIRCLE several times, with angle values of 1, 2, 3, 4, 5, 6, 7, 8, and 9 degrees. You should do this too. Does your picture look like this:



PICTURE THIS!

As you can see from this picture, it isn't until you use angles of 2 or more degrees that the circle fits entirely on the screen.

We now know that the larger the turning-angle, the smaller the circle. You already know that if we take larger steps each time, we will get larger circles. Next, let's do some experiments with both step size and turning-angle so we can determine if there is any special relationship between these two numbers.

Let's type the following:

```
GR: QUIT
```

and then enter:

```
LIST 0,100
```

to see the modules *DRAWCIRCLE and *CIRCLE. The following lines should be displayed on the screen:

```
10 *DRAWCIRCLE
20 T: HOW MUCH WOULD YOU LIKE TO TURN? \
30 A: #A
40 GR: GOTO 0, - 30; TURNTO - 90
50 U: *CIRCLE
60 E:
70 *CIRCLE
80 C: #B = 360/#A
90 GR: #B(DRAW 1; TURN #A)
100 E:
```

In order to control both the step size and the angle in these modules we have to make three changes in the program. Can you figure out what they are? Here is one way to make the changes. First, let's pick the variable #S to be the place where we will store the step size. We have to set up a way of entering a number into this variable from the keyboard. We can do this by adding the following two lines to *DRAWCIRCLE:

```
34 T: WHAT STEP SIZE WOULD YOU LIKE? \  
36 A: #S
```

Finally, we have to fix line 90 so that the distance the turtle moves with each step is given by the number stored in #S. To do this we just type:

```
90 GR: #B(DRAW #S; TURN #A)
```

and we are done.

Now we are ready for some more experiments.

We already know a good deal about what happens to circles made with a step size of 1 and turning-angles between 1 and 9 degrees. Let's see what happens when we make circles with steps equal to 2 units. Type:

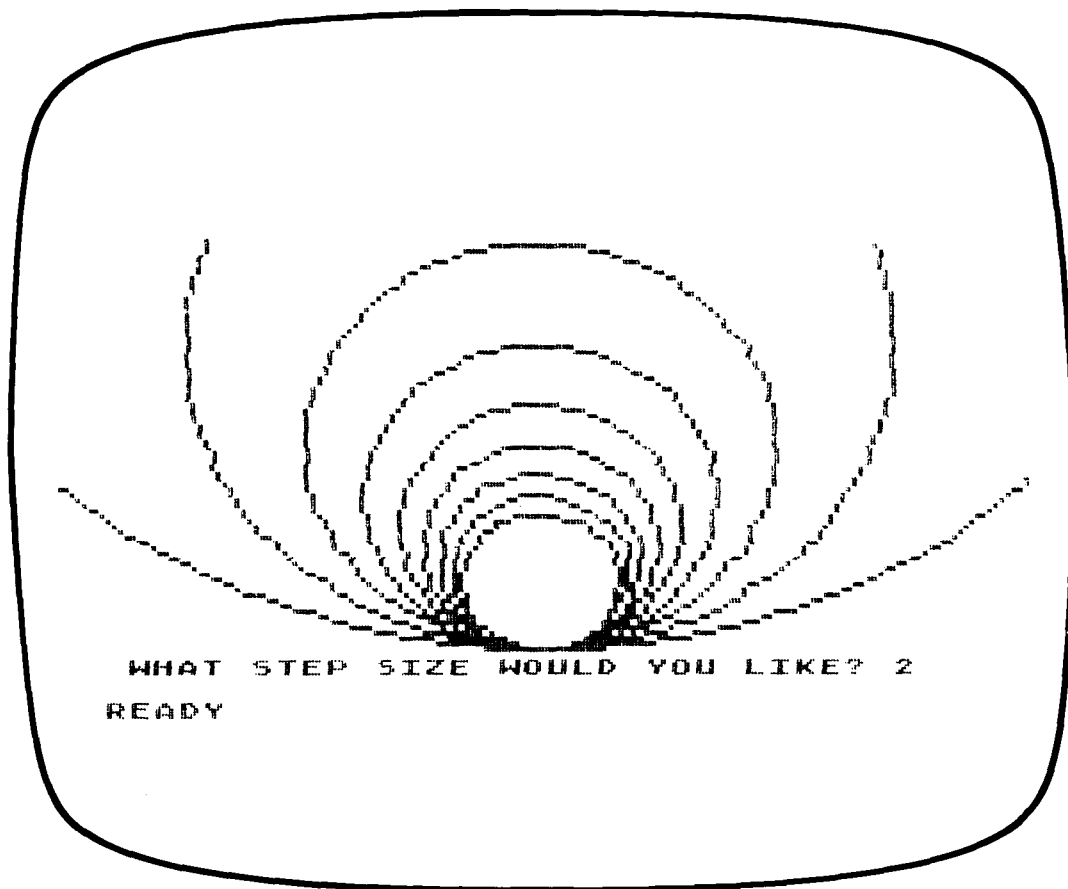
```
GR: CLEAR  
U: *DRAWCIRCLE
```

When the display shows "HOW MUCH WOULD YOU LIKE TO TURN?" enter 1 and then press the RETURN key. Next the display will show "WHAT STEP SIZE WOULD YOU LIKE?" Now enter 2 and press RETURN. Wow! That circle is so big that very little of it fits on the display. Let's use *DRAWCIRCLE a few more times with angles of different degrees and with a step size of 2. The next figure shows what I saw when I did this for angles between 1 and 9 degrees.

This figure looks similar to the one we made with different turning-angles and a step size of 1, but this time each circle is larger than those we made with that step size.

Since we know that larger turning-angles make *smaller* circles, and that larger step sizes also make *larger* circles, let's do an experiment to see if these numbers balance each other.

PICTURE THIS!



Let's start with a circle made with a turning-angle of 2 degrees and a step size of 1. Then double the step size as well as the angle and repeat the process. First type:

GR: CLEAR

and then enter:

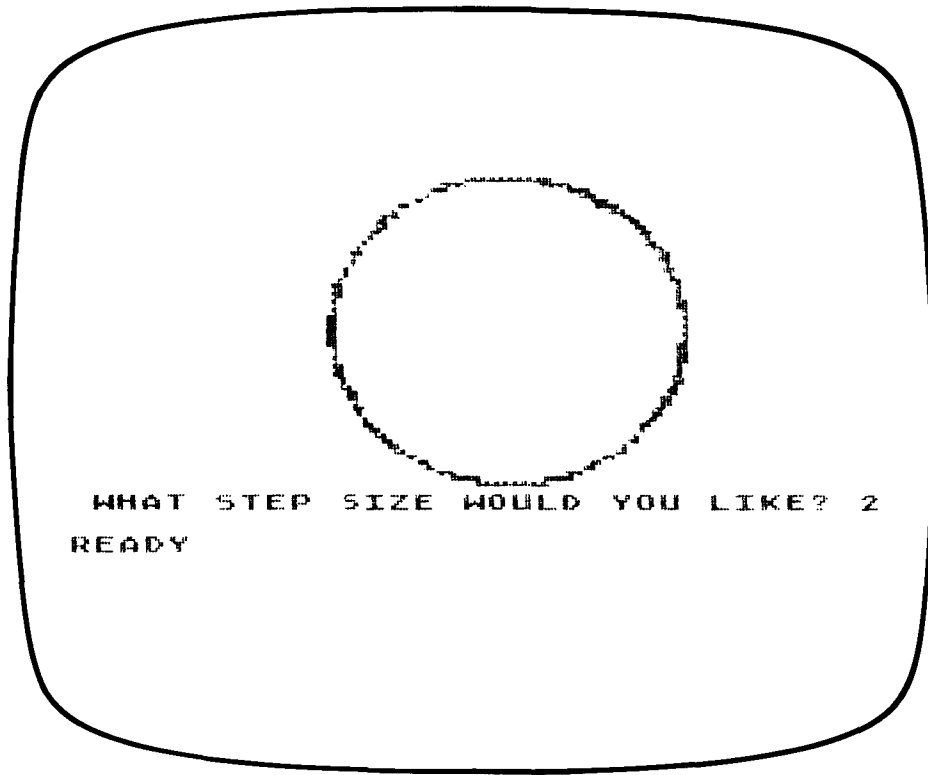
U: *DRAWCIRCLE

For our first circle, enter 2 for the angle, press RETURN, and enter 1 for the step size. When you press RETURN again, a familiar, yellow circle should appear on the screen. Next, let's double both the turning-angle and the step size. Type:

GR: PEN RED

138 U: *DRAWCIRCLE

This time enter 4 for the angle and 2 for the step size. When I did this I saw a red circle sitting on top of the yellow one!



Next let's double things again and see what happens. Type:

```
GR: PEN BLUE
U: *DRAWCIRCLE
```

and enter 8 for the angle and 4 for the step size. Now we have *another* circle of the same size. The only difference between this circle and the earlier ones is that it is shifted to the left a little bit. It is also a little bumpier than our other circles because of the larger step size.

It may have occurred to you that none of these pictures show a true circle, since these figures were each drawn by taking straight steps and then turning. A “circle” made by taking 1 step and turning 1 degree each time is really only a polygon with 360 tiny sides. A true circle isn't made with any straight sides—no matter how small they are.

PICTURE THIS!

We seem to be on the road to another of our famous discoveries. As long as we double *both* the angle and the step size, all the figures we get seem to be the same size overall. It is almost as if the size of the figure is determined by the ratio of the turning-angle to the step size, and the detailed shape of the figure, by the value we choose for the step size (or angle). So far we have looked at different figures made with an angle-to-step-size ratio of 2. (We turned 2 degrees for a step size of 1, 4 degrees for a step size of 2, and so on.) Let's make some big changes in angle and step size (keeping the ratio the same) and see where this leads.

First, let's clear the screen and draw our smoothest circle. Type:

```
GR: CLEAR
GR: PEN YELLOW
U: *DRAWCIRCLE
```

Enter 2 for the angle and 1 for the step size. Next type:

```
GR: PEN RED
U: *DRAWCIRCLE
```

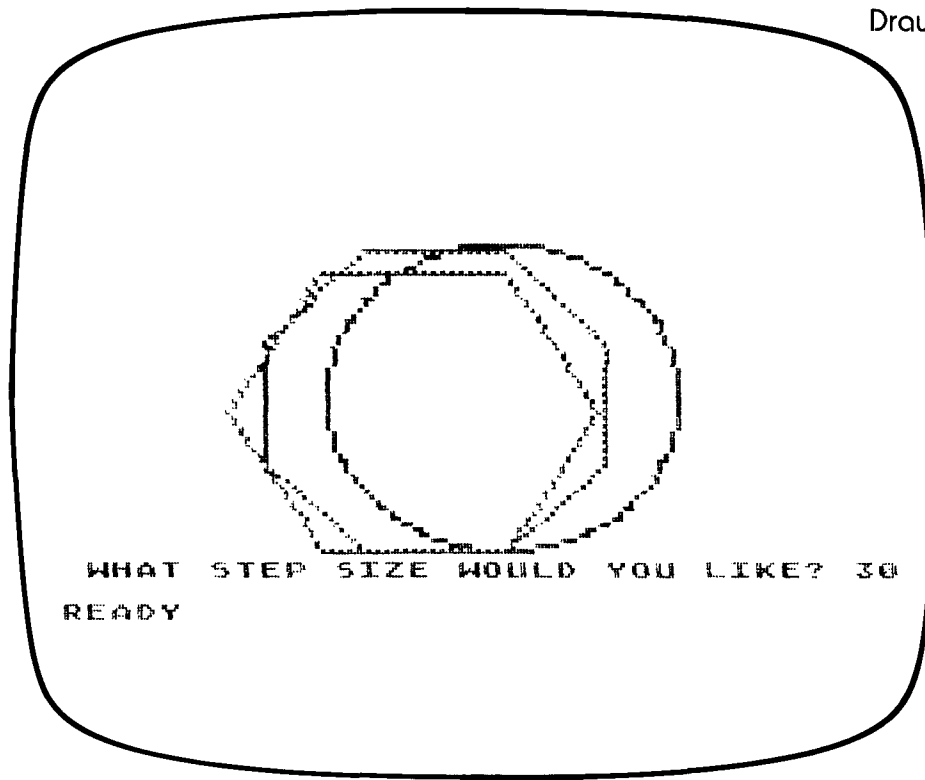
This time, enter 45 for the angle and 23 for the step size (we should use a step size of 22.5, but, as you know, we must use whole numbers, so we picked 23 instead). This newest figure is an octagon of approximately the same size as our first circle. Next type:

```
GR: PEN BLUE
U: *DRAWCIRCLE
```

and enter 60 for the angle and 30 for the step size. This will add a blue hexagon to the picture.

Each of these three figures is quite different from the others in shape, yet they all have the same overall size.

You might want to try other shapes to see what happens. Generally, the sizes will all be similar if the ratio of turning-angle to step size is kept the same.



Parts of circles—the turtle's arc

While circles are useful figures for us to know how to draw, sometimes we need only part of a circle. A curve that is made from part of a circle is called an *arc*. We know that to draw a circle we have to turn a total of 360 degrees. And it now should be clear that if we stop drawing before turning a full circle, we will get an arc. Let's make a module for drawing arcs that turn 2 degrees for each step we take. First exit from the graphics mode and type:

```
NEW
AUTO
```

and then enter:

```
*ARC
T: HOW BIG AN ARC DO YOU WANT? \ [prompt for angle
```

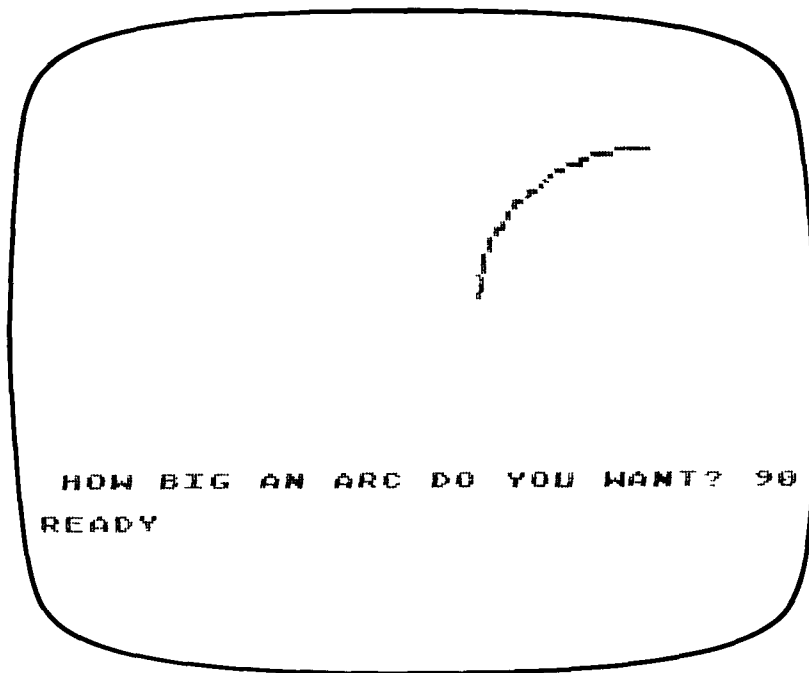
PICTURE THIS!

A: #A	[accept angle
C: #A = #A/2	[divide this number
	by two and put the
	result back into #A
GR: #A(DRAW 1; TURN 2)	[draw the arc
E:	

Press RETURN again to leave the AUTO mode and then type:

GR: CLEAR
U: *ARC

What number should we type in response to the question "HOW BIG AN ARC DO YOU WANT?" Let's try making an arc that is one-fourth of a circle in size. Since one circle is 360 degrees, we should enter one-fourth of that, which is 90, and press RETURN.



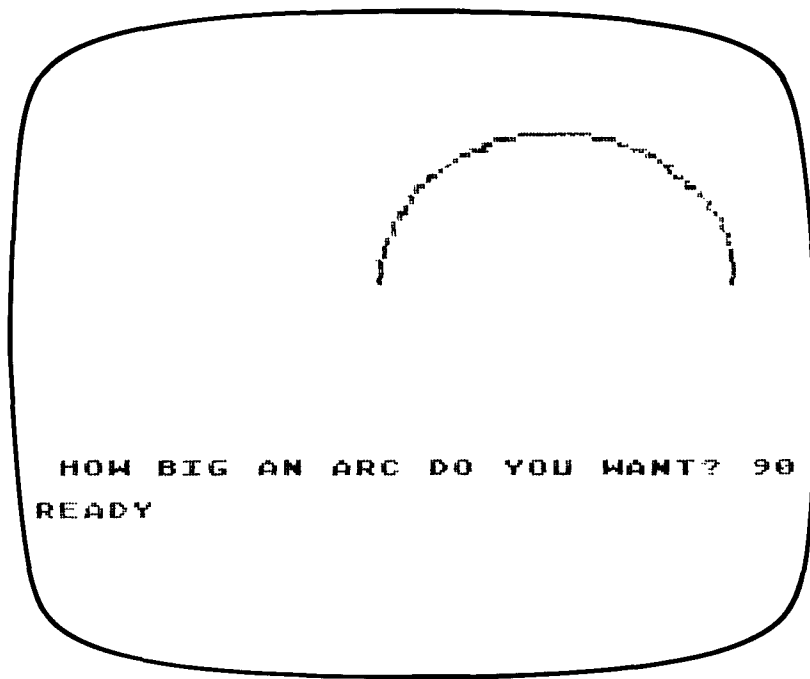
We now have a quarter-circle arc on the display screen.

Let's now extend the arc in a new color. Type:

GR: PEN RED

U: *ARC

Again enter 90 for the angle, and press RETURN.



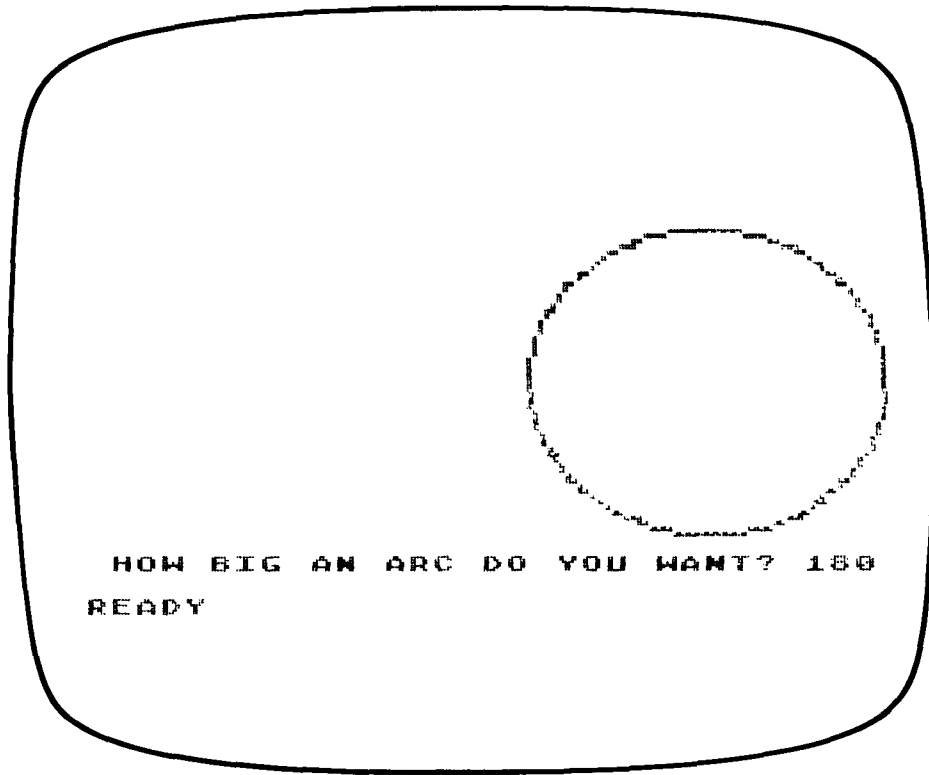
Now we have two 90-degree arcs—one yellow and one red. Together these arcs make a half circle. This makes sense since $90 + 90$ is 180, and 180 is half of 360—the number of degrees we need to turn a full circle. Let's check this out. Type:

GR: PEN BLUE

U: *ARC

This time enter 180 for the angle. When you press RETURN, you will get a blue half circle that closes the figure to give us a full, multi-colored circle. Not bad work for starters!

PICTURE THIS!



Next let's use our arcs to draw some pictures. Can you think of any pictures you could draw with arcs? How about a bird? Have you noticed, when a bird is flying a good distance from you, how its wings form two arcs? Let's try to make a bird that will fly across the screen!

To start with, we should plan our course of action. We need a module to draw a wing, a module that *uses* the wing to draw a bird, a module to move the bird, and a module (called *CARTOON) to make everything happen properly.

Let's start with the wing. I think we know how to handle that! Exit from the graphics mode and type:

NEW
AUTO

and enter:

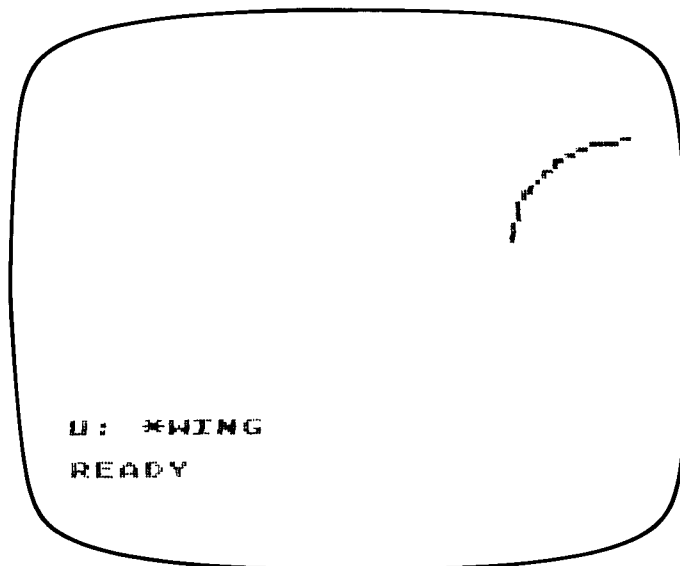
```
*WING
GR: 30(DRAW 1; TURN 3)
E:
```

This module constructs our wing out of a 90-degree arc. Press RETURN again to leave the AUTO mode, and let's try to determine how to use this module to build our bird. First type:

```
GR: CLEAR
```

and then type:

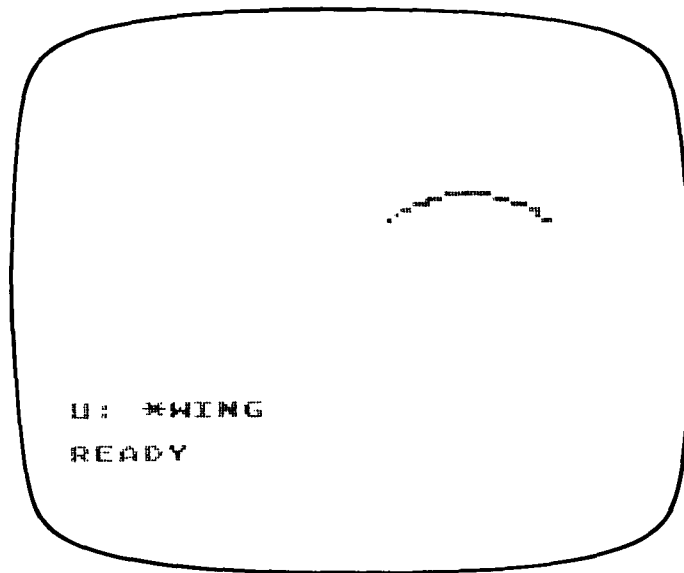
```
U: *WING
```



Hmmm, this looks like a good wing, but I think it ought to be more horizontal. Let's try this:

```
GR: CLEAR
GR: TURNT0 45
U: *WING
```

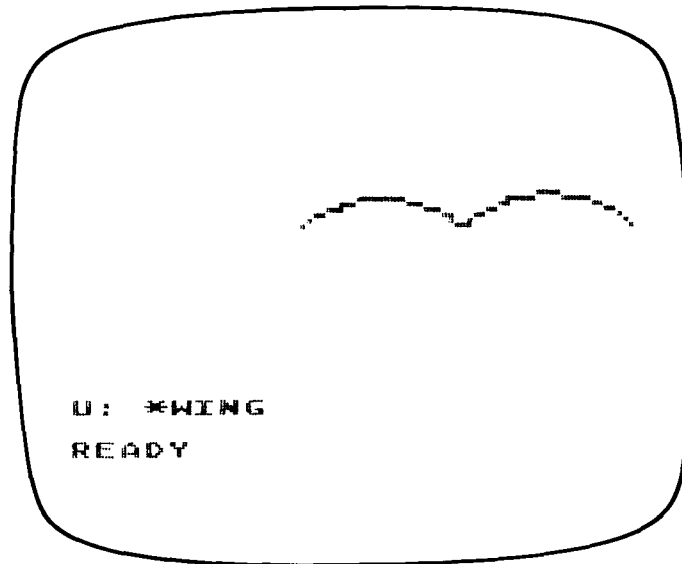
PICTURE THIS!



Now that's much better! All we need to do is repeat this and we will have a bird. Type:

GR: TURNT0 45
U: *WING

Now we have a bird on the display screen.



I don't know about you, but I would like another picture of a bird with its wings high in the air so they look as though they were flapping. To draw this bird, let's try the following:

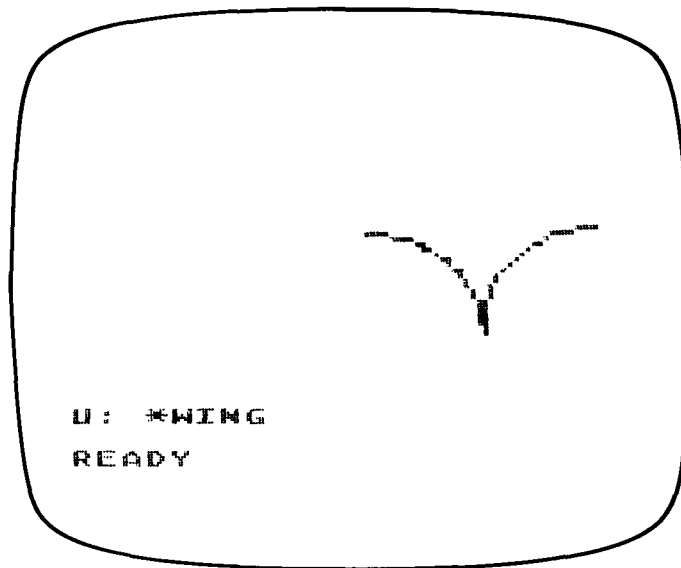
GR: GOTO 0,0; CLEAR

GR: TURNT0 90 [this starts the first wing out horizontally

U: *WING

GR: TURNT0 0 [this starts the second wing straight up

U: *WING



Wow! Now we can make two pictures of a bird. The only problem is that to show the bird flapping its wings we need to erase one picture before drawing the second, and then erase that one before redrawing the first. We know that changing the PEN from YELLOW to ERASE can help us solve this problem, but we may not know exactly how it will help us. There are lots of ways of dealing with this, and one of them is to make a new module, *ANTIWING, which traces backwards over our wing and (if we have made the pen change first) erases it.

Well, let's give this a try. Type:

GR: QUIT

REN 1000

AUTO

PICTURE THIS!

and enter:

```
*ANTIWING
GR: 30(TURN - 3; DRAW 1)
E:
```

Press RETURN again to leave the AUTO mode. Do you notice anything interesting about *ANTIWING? Not only is the direction reversed, but so is the order of our graphics command. Perhaps you should play turtle to see if you can convince yourself that this is the correct way of doing this.

Let's try things out to see how this all works. First type:

```
GR: CLEAR
U: *WING
```

and you see that we have one yellow wing on the screen. Next we want to erase this wing. To do this we have to get the turtle pointing in the correct direction. Since the turtle ended up pointing to the right, we have to get it pointing back to the left. Type:

```
GR: TURNTO - 90
GR: PEN ERASE
U: *ANTIWING
```

Presto! The wing disappeared (except for one dot that we'll have to take care of later).

Now we are ready for a real challenge—drawing a bird with flapping wings! To do this requires our drawing the bird in one wing position, erasing that picture, then drawing the bird in the second wing position, and erasing that picture of the bird as well. Type:

```
GR: QUIT
REN 1000
AUTO
```

and enter:

*FLYBIRD	
GR: PEN YELLOW	[set pen to yellow
GR: TURNT0 45	[turn for first position
U: *WING	[draw first wing
GR: TURNT0 45	[position turtle
U: *WING	[draw second wing
PA: #P	[pause before erasing
GR: PEN ERASE	[set pen to erase
GR: TURNT0 90; DRAW 1;	[take care of extra dot
DRAW - 1	
GR: TURNT0 - 45	[position turtle
U: *ANTIWING	[erase second wing
GR: TURNT0 - 45	[position turtle
U: *ANTIWING	[erase first wing
GR: TURNT0 0; GO 20;	[move turtle for second bird
TURNT0 90; GO 8	
GR: PEN YELLOW	[set pen to yellow
GR: TURNT0 90	[position turtle
U: *WING	[draw first wing
GR: TURNT0 0	[position turtle
U: *WING	[draw second wing
PA: #P	[pause before erasing
GR: PEN ERASE	[set pen to erase
GR: TURNT0 90; DRAW 1;	[take care of extra dot
DRAW - 1	
GR: TURNT0 - 90	[position turtle
U: *ANTIWING	[erase second wing
GR: TURNT0 0	[position turtle
U: *ANTIWING	[erase first wing
GR: TURNT0 0; GO - 20;	[move turtle back for first
TURNT0 90; GO - 8	bird
E:	

Press RETURN again to leave the AUTO mode.

PICTURE THIS!

You probably noticed that you used a new command in this module, the *Pause command* (PA:), which will stop PILOT from executing any subsequent commands for as long as the number stored in variable #P indicates. (We could, of course, use any variable we care to, but #P reminds me of pause.) The length of time that PA: waits is measured in "jiffies." One jiffy is the same as one-sixtieth ($1/60$) of a second. If we pause for 60 jiffies (#P=60), each picture will stay on the screen for one second before being erased.

Now let's get on with the show! First, let's see the bird all by itself. Type:

```
GR: CLEAR
C: #P=60  [set the delay for one second
U: *FLYBIRD
```

If everything worked correctly, you should see each picture of the bird drawn and on the screen for one second before being erased. Good work so far!

If you didn't have this happen, you should first try to find the error. To make things easier, you might want to temporarily change PEN ERASE to PEN BLUE so you can see what the erased lines look like. If guessing doesn't help and you are still stuck, go back over the program listing carefully and see if you can isolate your error.

Next let's build a module to move our bird across the screen. Type:

```
GR: QUIT
REN 1000
AUTO
```

and enter:

```
*MOVE
GR: TURNTO 90; GO 10
E:
```

and press RETURN to leave the AUTO mode. Each time this module is used it will move our bird to the right by ten units. Now we are ready to make a cartoon! Type:

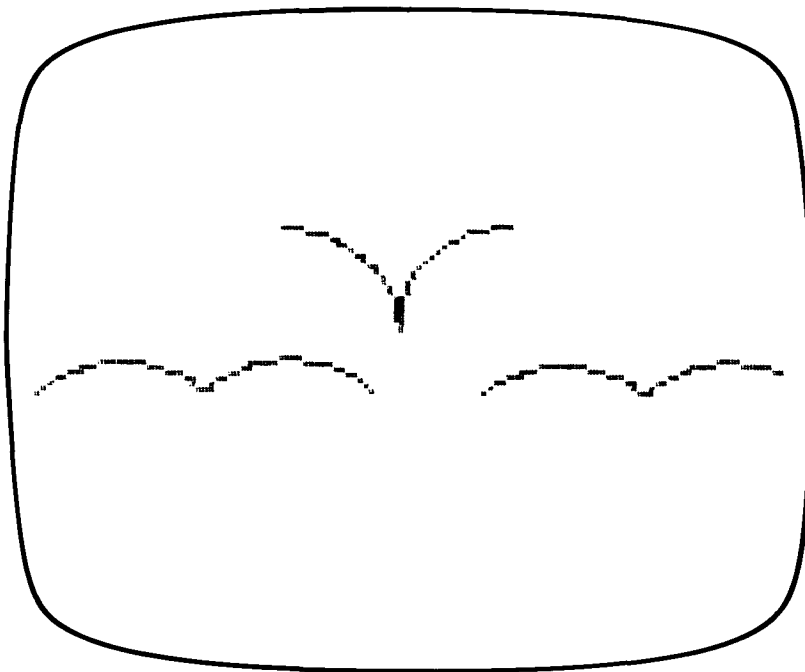
```
REN 1000
AUTO
```

and enter:

```
*CARTOON
GR: GOTO -70,0 [start bird to left of screen
C: #P=30       [set up for 30-jiffy delay
*FLY
U: *FLYBIRD    [flap-flap!
U: *MOVE       [move
J (%X<60) : *FLY [keep going if bird is still on the screen
E:
```

And now we are finally ready for the flight of the century. Type:

```
GR: CLEAR
U: *CARTOON
```



PICTURE THIS!

There it goes! Our bird is slowly flapping its way across the screen! The previous figure shows a few “snapshots” of the bird in flight.

You should do some experiments with *MOVE to see if you can make the bird fly all over the screen. Can you make the bird fly in a circle? Try it and see what happens!

Spirals . . .

In Chapter Ten we learned how to make “squirls.” When we made a squiral, we kept the turning-angle the same and increased the length of the line with each step. Because we increased the step size each time, our squiral became larger as time went on.

Now what do you suppose would happen if we kept our step size the same and increased the turning-angle with each step. Why don’t you try to find out by playing turtle. Take a step, turn a little bit, take another step, turn by a little bit more, and so on. Do you see what happens? Do you find yourself curving into a spiral? Well, let’s let the turtle do some work by seeing how it handles our attempt to draw a spiral.

I will make two modules. One that resets everything, and another that draws the figure we want to study. In this manner we can draw a few steps, think about what is happening, and then keep on going without having to start again.

Be sure you are not in the graphics mode and type:

```
NEW  
AUTO
```

Once the screen changes color, enter:

```
*RESET  
GR: PEN YELLOW
```

```
GR: GOTO 0,0; TURNT0 0; CLEAR [put the turtle home and
                                clear the screen
VNEW: [reset all variables
E:
```

Next enter:

```
*SPIRAL
T: HOW MANY STEPS DO YOU WANT? \
A: #S
C: #C=0 [reset step counter
*DRAWSPIRAL
GR: DRAW 3; TURN #A [draw line and turn
C: #A=#A+1 [increase turning-
              angle by 1 degree
C: #C=#C+1 [increase counter by
              one
J (#C<#S): *DRAWSPIRAL
E:
```

Press RETURN again to leave the AUTO mode.

Q: Before we get too much further, can you tell me what VNEW:
is all about?

A: Of course. I was just about to explain that VNEW: can be
used any time you want to reset all the variables in PILOT.
This means that all twenty-six numeric variables will be reset
to zero (0).

Now, to move right along, let's try these modules out. First type:

```
U: *RESET
```

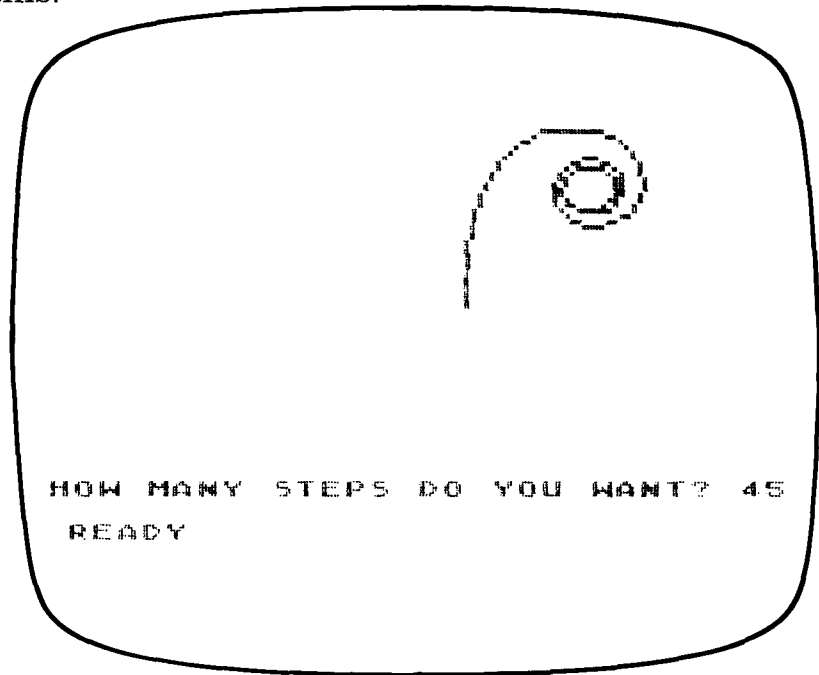
and then

```
U: *SPIRAL
```

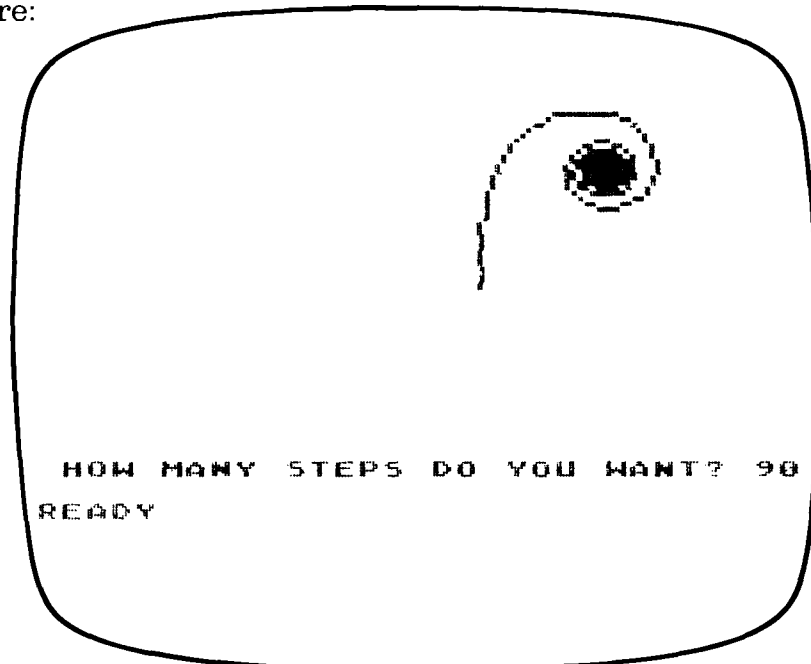
When the question, "HOW MANY STEPS DO YOU WANT?" appears
on the screen, enter 45 and press RETURN.

PICTURE THIS!

When I did this, I saw a line start out vertically and then bend over to the right like this:



Let's keep this process up to understand what is happening. Use *SPIRAL again, but this time let it draw an additional 45 steps. Now the spiral is starting to form quite clearly. What do you think will happen if we go an additional 90 steps? Do you think the turtle will spiral itself right into the center of its arc? Try it and see if you get this figure:



Wow! There is sure a lot of congestion in this spiral. Do you know where the turtle is? Is it in the center of the blob? We can find out by typing the following lines:

```
GR: PEN RED
GR: GO 0
```

We didn't actually move (the command GO 0 doesn't take us anywhere), but we did succeed in getting the turtle to show us where it is. Do you see the red dot inside the blob? That is the present location of the turtle.

Before typing anything else, let's try to figure out how the turtle got where it is and what will happen when we continue.

First, we know that when we started, the turtle was pointing straight up. As we kept going, the turtle turned by 1 degree, then by 2 degrees, and so on. By the time we reached 90 steps, the turtle was turning by 90 degrees with each step. If we look at the first four steps the turtle took, we see that the total turning-angle was $1 + 2 + 3 + 4$, which is only 10 degrees. This isn't much of a turn. Now let's look at how much turning the turtle did during the four steps starting at step 89. The turning that took place during these four steps was $89 + 90 + 91 + 92$, which is 362 degrees. In other words, by the time we reached step 90, the turtle was turning more than one complete revolution for every four steps. This is quite a change from the way things started out.

As we kept turning and drawing, we found ourselves homing in on an area that we see as a circular blob. What happens as we draw step 180? At this point the turtle is simply going back and forth, which means that our spiral has pretty much stopped growing.

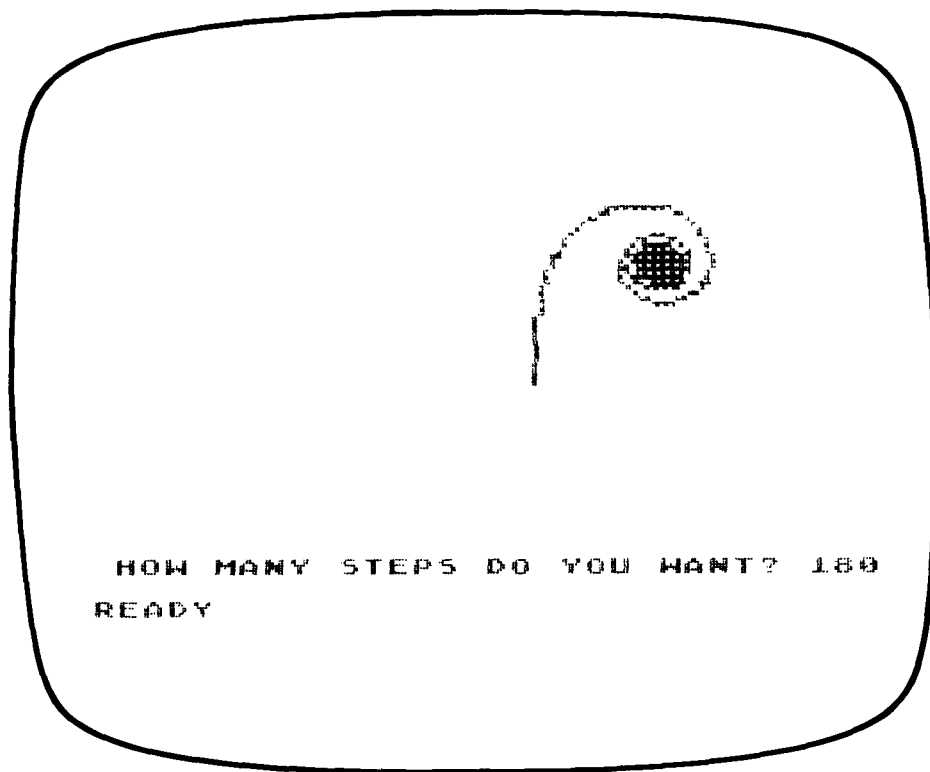
What do you suppose happens for steps larger than 180? Which are accompanied by angles greater than 180 degrees. Can you guess what will happen? Why don't you draw a picture of what you think the turtle would do.

PICTURE THIS!

Now that you have done that, let's allow the turtle to keep going so we can see what it really does. Since we have already changed the pen to red, we will be able to see the new line no matter where it is drawn. Type:

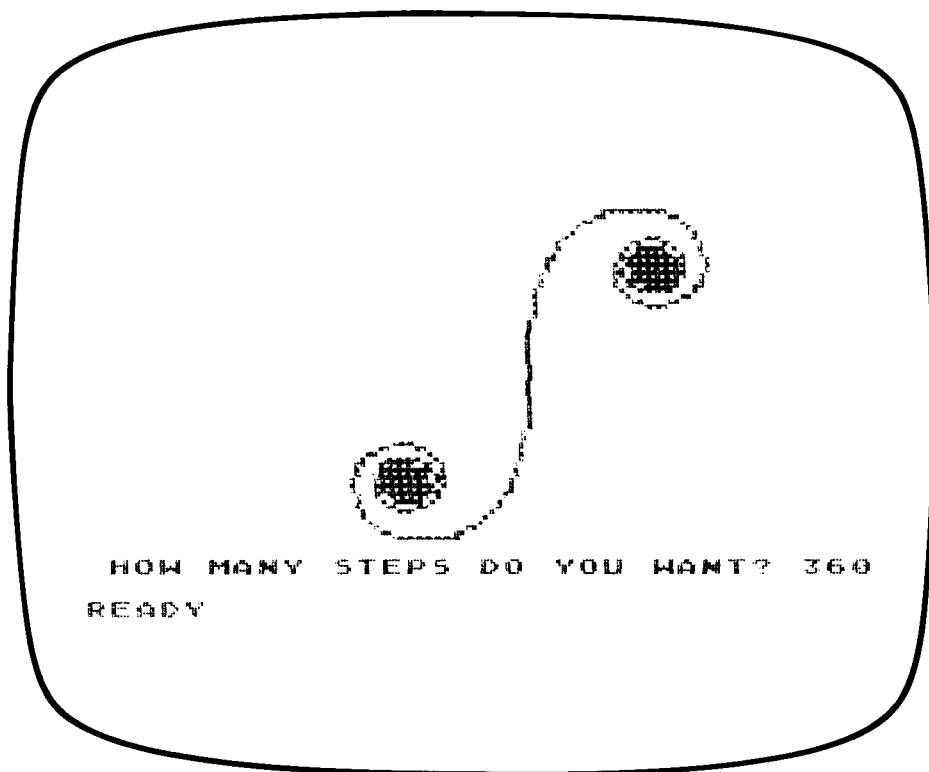
U: *SPIRAL

and enter 180 for the number of steps. Initially, we see the yellow blob turn to red, and then the turtle traces its path back to the center of the screen. Now, instead of yellow, we have a red spiral on the screen.



If you have been keeping track, you'll know that we now have taken 360 steps and, except for one major difference, have ended where we started. Do you know what is different about the turtle now compared to when it started out? Yes, it is holding a red pen instead of a yellow one. It is also pointing down instead of up.

Now let's use *SPIRAL again to see what happens. This time enter 360 for the number of steps.



PICTURE THIS!

Our figure is now complete since we are back at the starting point and facing straight up. You should test this by typing:

GR: PEN YELLOW

U: *SPIRAL

and entering 720 for the number of steps taken. Since 720 is the same as $360 + 360$, this should draw both arms of the spiral in yellow.

By now you should be an expert at getting the turtle to draw curves. In the next chapter we will bring together many of the things we have learned so far, as well as introduce a few more tricks before finishing.



the last one

AS YOU MAY have noticed, we are just about finished with this book. By this time you have acquired most of the skills you need for drawing all sorts of designs and pictures with PILOT. We will draw one last picture in this chapter as a celebration of our progress through this book—after all, you have much to be proud of!

First we will learn about another graphics command.

The turtle has its FILL . . .

So far the pictures we have drawn were all made with open figures. When we drew a square, we drew the outline of a square. Sometimes it is more pleasing to draw objects that are filled with color than it is to draw outlines.

PICTURE THIS!

One way of filling an object with color is to move to the bottom left edge of the object and draw a line to the right until we hit a boundary. We repeat this process with the lines above until we reach the top of the object that we're filling with color. We could build a module to do this for us, but we don't have to. The turtle has two commands that do this job for us. Let's see how they work.

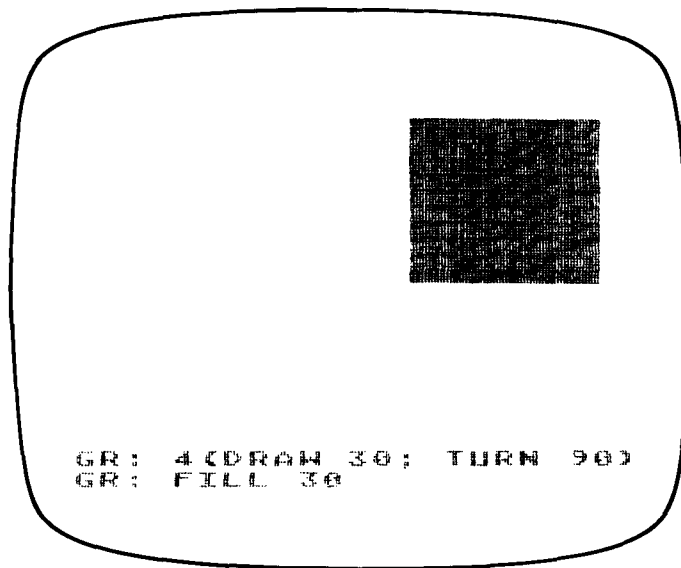
First you should enter the graphics mode making sure the screen is clear and the turtle is pointing straight up. Next type:

```
GR: 4(DRAW 30; TURN 90)
```

This (as you most certainly have noticed by now) will draw a square on the screen. At this time the turtle is in its home location and pointing straight up. Now type the following instruction:

```
GR: FILL 30
```

Wow! The last command completely filled in our yellow square!



The FILL command (as well as its counterpart, FILLTO) is an easy way to fill in open figures. There are a few simple properties of FILL that you need to know in order to use it effectively. First, FILL al-

ways works from left to right. When a FILL command is given, a number of horizontal lines are drawn to the right from each point along the turtle's path. Each horizontal line runs until it bumps into another line or the right edge of the screen. If it hits the right edge of the screen, it reappears at the left edge and keeps moving to the right until it bumps into a line or itself.

Let's try out some more uses of FILL to get a better idea of how it works. Type:

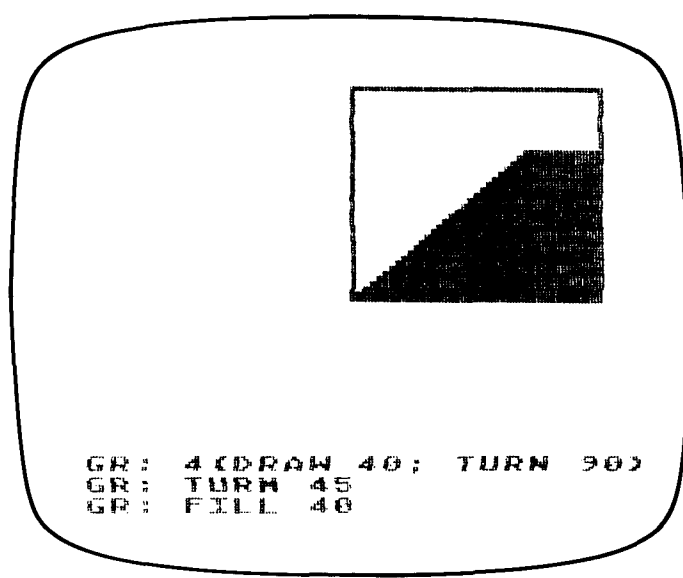
```
GR: GOTO 0,0; CLEAR  
GR: 4(DRAW 40; TURN 90)
```

We now have a yellow square on the screen. Next we will fill in this square along one diagonal. Type:

```
GR: TURN 45
```

to turn the turtle toward the direction of the diagonal. Next, let's fill in the square from its lower left corner (where we are starting) to its upper right corner (where we are ending). How far do you think we should go to get from the bottom left corner to the upper right corner? Should we go 40 units? More? Let's find out by trying! Enter the following:

```
GR: FILL 40
```



PICTURE THIS!

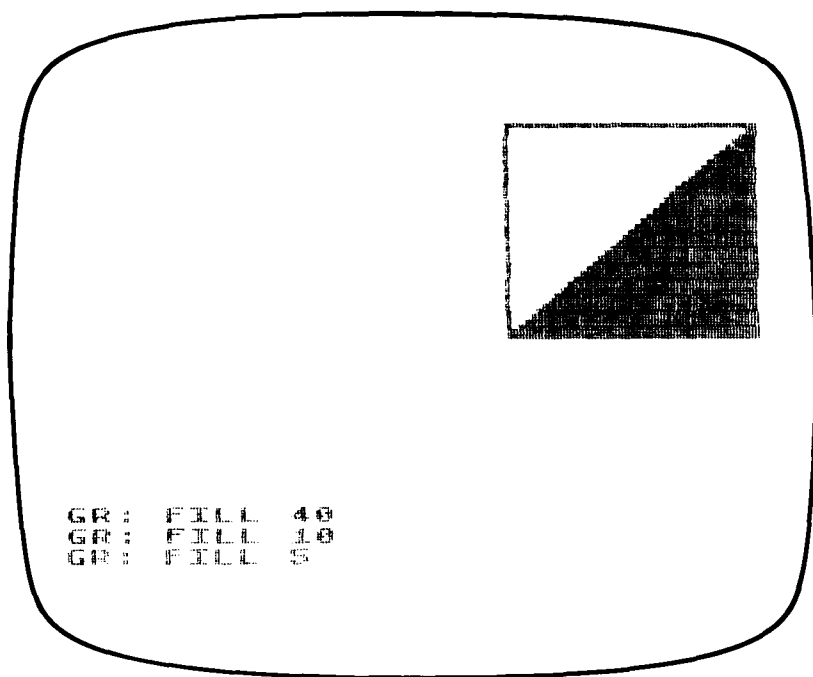
Well, that got us going in the correct direction, but it didn't take us far enough, did it? Next, enter:

GR: FILL 10

This brings us closer, but not quite close enough. If we type:

GR: FILL 5

we see that the job is finally complete.



We just discovered that to fill a diagonal in a square with 40 units on a side, we must FILL for a distance of $40 + 10 + 5$, or 55 units. Actually, if you are very observant, you might notice that we could go one more unit so the turtle sits on top of the upper right corner. However, if we used the FILL command to move us to that position,

we would have a most unpleasant surprise. FILL would look to the right of the turtle's position and start drawing. Since there is no boundary to the right of this point, we would get a horizontal line running completely across the screen. (You should type GR: FILL 1 if you want to see this.)

Q: What is so special about the number 55 (or 56)? How can I tell in advance how far I need to go without using trial-and-error methods?

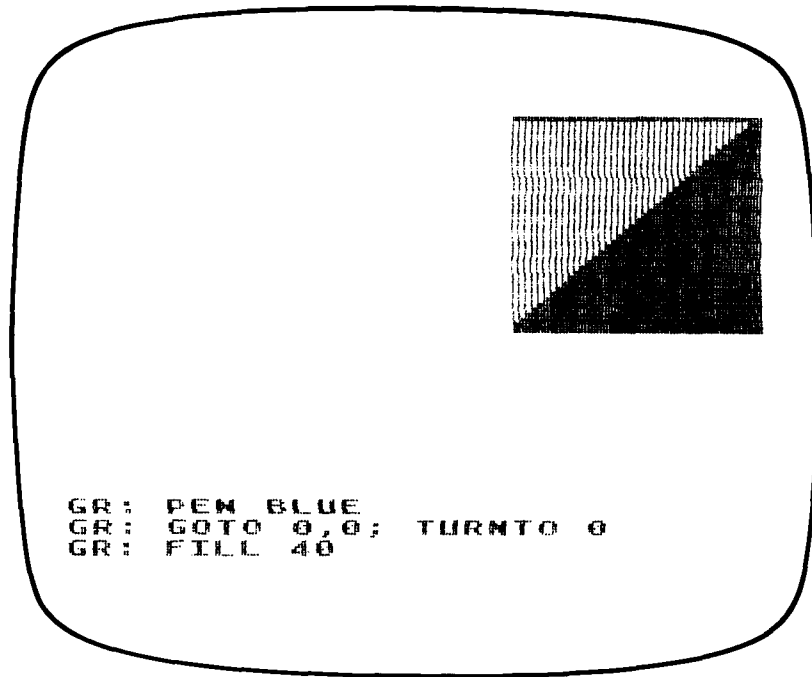
A: The answer to this question is buried in something called the Pythagorean theorem. Basically, it says that in any triangle formed with one 90-degree angle, the square of the length of the side opposite this angle is equal to the sum of the squares of the other two sides. In our case, the triangle we formed had two equal sides. This means that the square of the long side is twice as long as the square of either of the shorter sides. Since our short side is 40 units long, the square of our long side should be exactly $2 \times 40 \times 40$, which is 3200. It so happens that there is no whole number that, when multiplied by itself, gives 3200. However, 56×56 is equal to 3136, which is as close as we can get. Because of the special way that FILL works, we need to stop one position short of this. As you see, it worked out perfectly.

Now that we have filled in half of the square along the diagonal, we should fill in the other half with another color. Before doing this, let's think for a minute. Where is the turtle? In which direction is it pointing? If you know that the turtle is in the upper right corner of the square and that it is pointing along the diagonal, then you are to be congratulated. All we need to do now is move the turtle back to its home, get it to point straight up, and instruct it to FILL the square to the top. To do this just type:

```
GR: PEN BLUE
GR: GOTO 0,0; TURNT0 0
GR: FILL 40
```


PICTURE THIS!

Now we have a fancy, filled square on the screen!



As you can see, FILL is a very useful command.

However, FILL does have some limitations. Let's draw a big square.
Type:

```
GR: QUIT
NEW
AUTO
```

and then type:

```
*FIRSTSQUARE
GR: GOTO - 15, - 15; TURNT0 0; PEN YELLOW; CLEAR
GR: 4(DRAW 60; TURN 90)
GR: FILL 60
E:
```

Press RETURN to leave the AUTO mode and type:

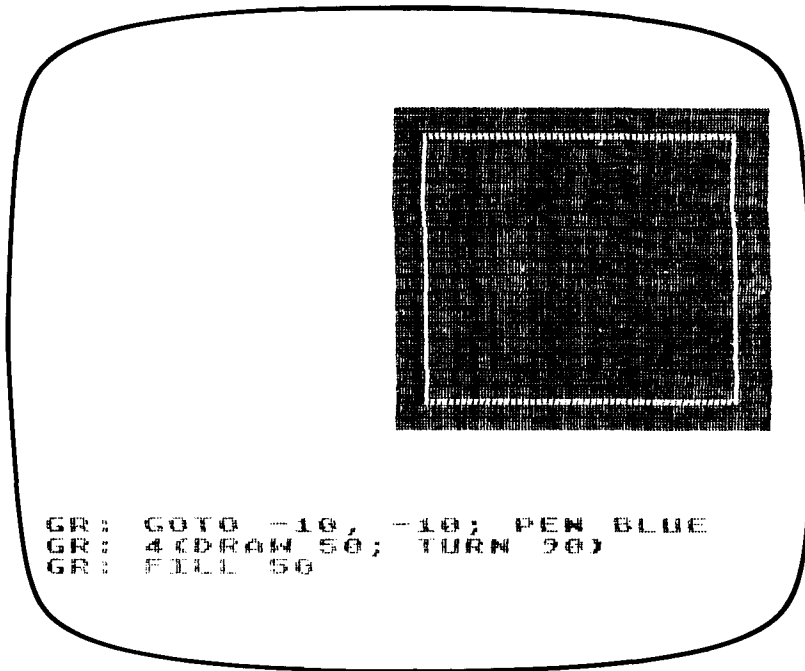
```
U: *FIRSTSQUARE
```

This gives us a solid, yellow square on the screen. Next, let's draw a solid, blue square in the middle of this square. First type:

```
GR: GOTO - 10, - 10; PEN BLUE  
GR: 4(DRAW 50; TURN 90)
```

So far there have been no surprises, right? Next enter:

```
GR: FILL 50
```



Hmmm—nothing seemed to happen! The reason for this is that FILL works only when the open space has the black background color. So, if we are going to have a solid square inside a second solid square, we will have to make it ourselves.

PICTURE THIS!

To do this, type:

```
GR: QUIT
REN 1000
AUTO
```

and then enter:

*SOLIDSQUARE	[this module draws a solid square whose length is stored in the variable #A
C: #B = 0	[set counter to zero
*KEEPPGOING	
GR: TURNT0 90; DRAW #A; DRAW - #A	[draw out and back
C: #B = #B + 1	[increase counter by one
J (#B > #A) : *EXIT	[stop if square is done
GR: TURNT0 0; DRAW 1	[move up one line
J: *KEEPPGOING	[keep going
*EXIT	
E:	

Now let's use this module. Type:

```
GR: GOTO - 10, - 10
```

and

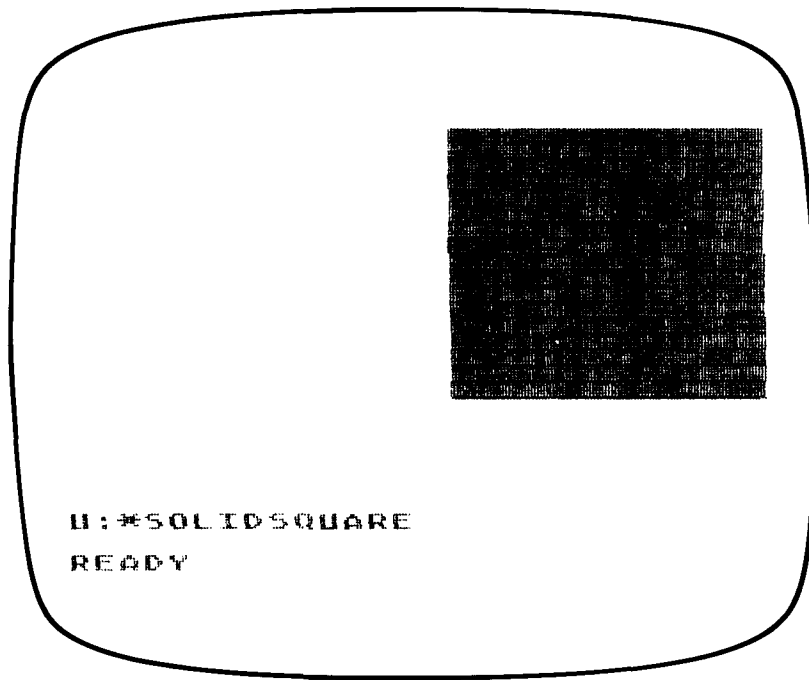
```
C: #A = 50
```

to set the size of our solid square, and then type:

```
U: *SOLIDSQUARE
```

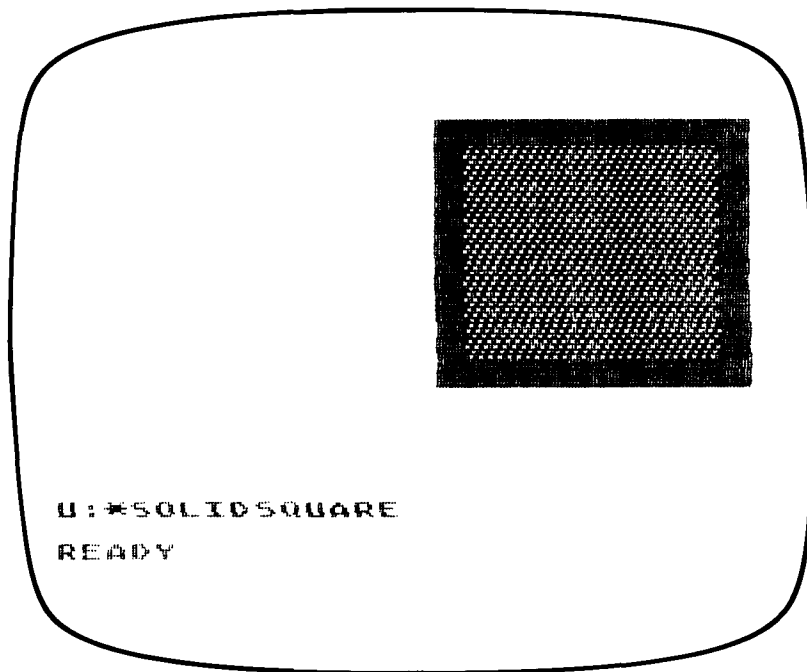
That worked perfectly didn't it!

The last One



Next, let's use this module again to draw a smaller red square. Type:

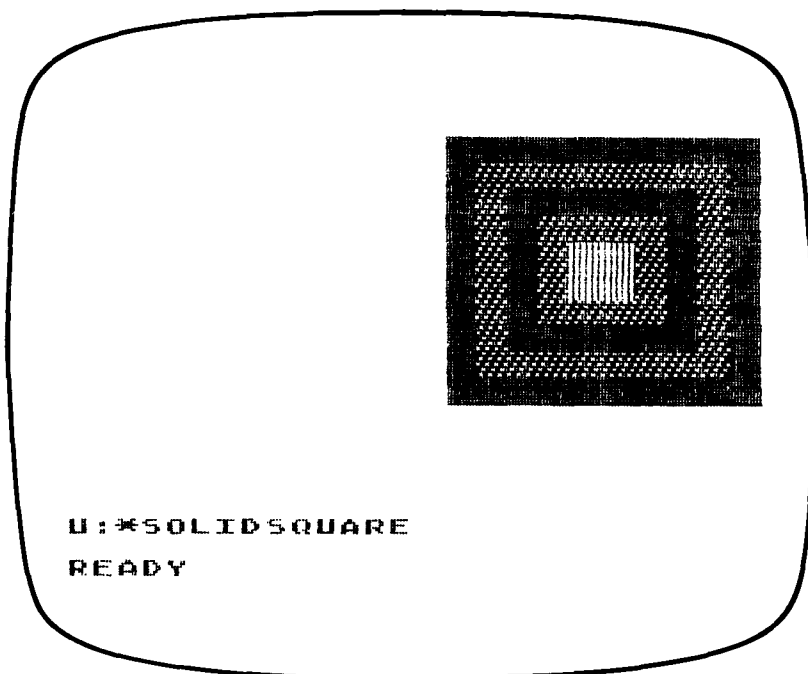
```
GR: GOTO -5, -5; PEN RED  
C: #A=40  
U: *SOLIDSQUARE
```



PICTURE THIS!

Do you see what we are doing with each square? We are moving the turtle by 5 units along each axis and then drawing a square that is smaller than the previous one by 10 units.

Using this method, draw three more solid squares with the colors yellow, blue, and red. This will give you a square bull's-eye.



Each time you constructed a new square it covered up the previous colors. The Atari computer system builds color pictures from the back to the front and has no prebuilt mechanism for keeping track of which color was under the one you just used. This means that you must be careful when building your pictures. Colors are always laid on top of other colors; they are never mixed with or inserted behind existing colors. This is a useful thing to remember.

A flower for our turtle . . .

Since we are going to make a pretty picture, we ought to learn how to draw a flower. In fact, if we had a flower-drawing module, we could draw many flowers.

The flower we will draw first has three parts to it: a stem, a leaf, and the blossom. In some ways drawing a flower will be a lot like drawing a person (remember back that far?). We will first have to solve how to build the modules we need.

Let's start with a leaf. One way of drawing a leaf's shape is to use two arcs. Do you remember how to draw an arc? For a small leaf-sized arc, try this. Be sure to exit from the graphics screen and then type:

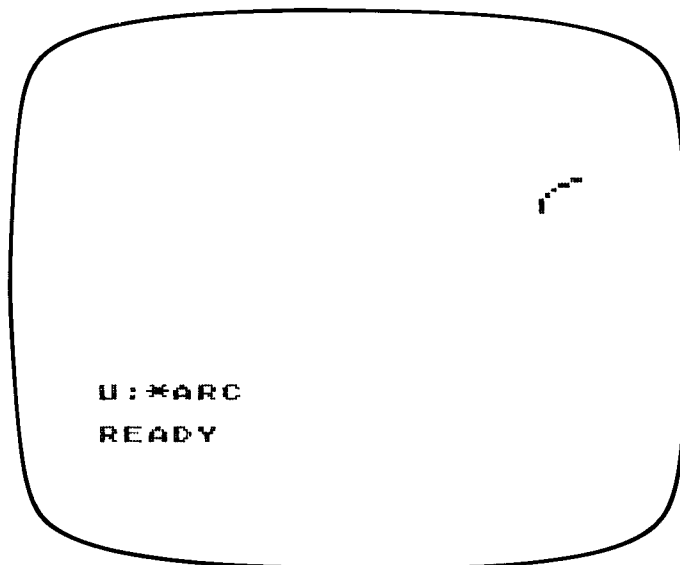
```
NEW  
AUTO
```

Next enter:

```
*ARC  
GR: 10(DRAW 1; TURN 9)  
E:
```

Press RETURN again to leave the AUTO mode and type:

```
GR: CLEAR  
U: *ARC
```



This draws a nice, little arc on the screen. Now we have to draw another arc to finish the figure of the leaf. We already know (from our bird-drawing experience) that we must turn the turtle before using

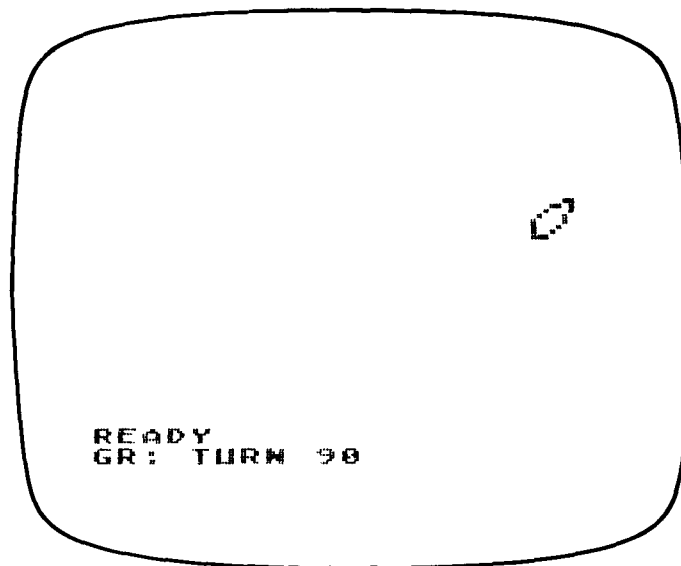
PICTURE THIS!

*ARC again, or we will get simply a semicircle. So how much should we turn to create a closed leaf? Do you remember any turtle rule that might help us? Does the fact that a leaf is a closed figure help?

Well, let's think about this for a minute. There is a turtle rule that says a closed figure is made by turning 360 degrees on our way around the figure. To draw our arc, we turned 9 degrees for each of the 10 steps we took. This gives us a 90-degree arc. If we use *ARC again, we will account for a total of 180 degrees. This leaves us with 180 degrees left to turn. Since we have to get back to exactly where we started, we should turn the turtle by 90 degrees at the end of each use of the *ARC module.

Let's see if this works. Type:

```
GR: TURN 90
U: *ARC
GR: TURN 90
```



Wow! We just used an old rule to make something new! Now let's use this information to make the *LEAF module. Type:

```
170 GR: QUIT
    REN 1000
    AUTO
```

and enter:

```
*LEAF
U: *ARC
GR: TURN 90
U: *ARC
GR: TURN 90
E:
```

Exit from the AUTO mode, and then use this new module to make sure it works properly.

Next, let's work on the flower itself. A long time back, we made a module called *WINDMILL that drew some nice, flowerlike patterns for us. We can use the same technique to draw the blossom of our flower. First, we define the petal shape, and then we repeat this shape eight times, by rotating the turtle one-eighth of a circle each time. To do this, type:

```
GR: QUIT
REN 1000
AUTO
```

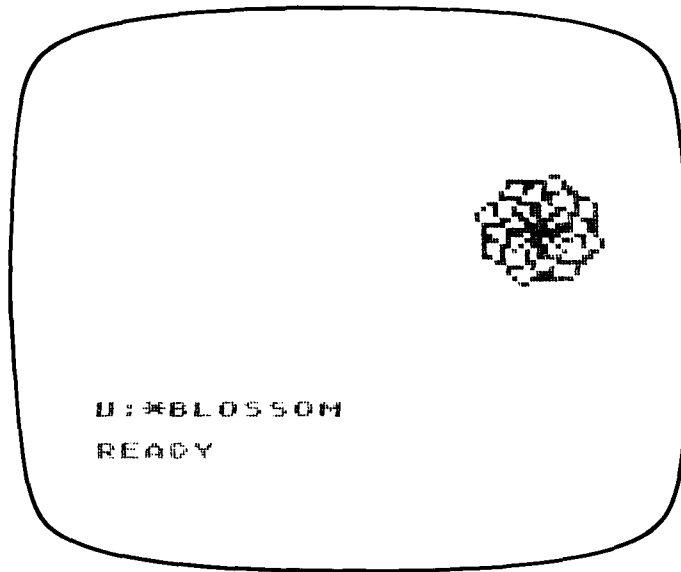
and enter:

```
*PETAL
GR: 6(DRAW 5; TURN 60) [this gives us hexagonal petals
E:
*BLOSSOM
C: #A = 0
*LOCALJUMP
C: #A = #A + 1
U: *PETAL
GR: TURN 360/8
J (#A < 8) : *LOCALJUMP
E:
```

Exit from the AUTO mode and type:

```
GR: CLEAR
U: *BLOSSOM
```


PICTURE THIS!



Now that we have a blossom and a leaf, we need to draw a stem before we will have all the parts needed to draw a flower. I think our stem should be a gentle arc, perhaps arching to the right a little bit. Type:

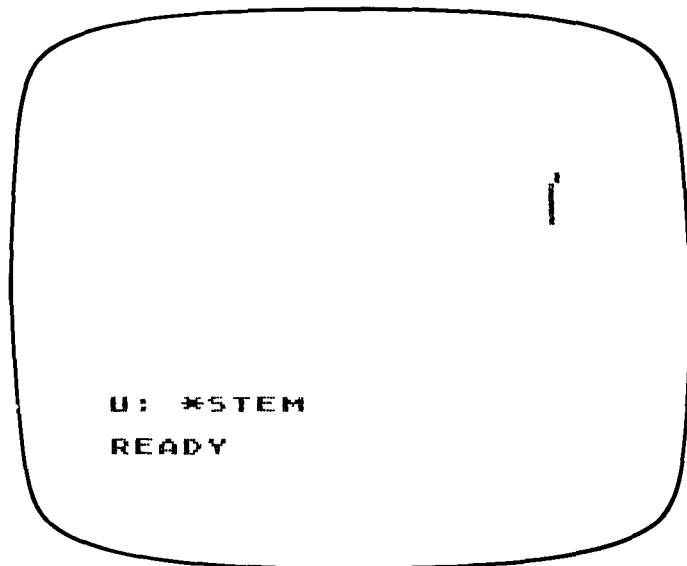
```
GR: QUIT
REN 1000
AUTO
```

and then enter:

```
*STEM
GR: 10(DRAW 1; TURN 1)
E:
```

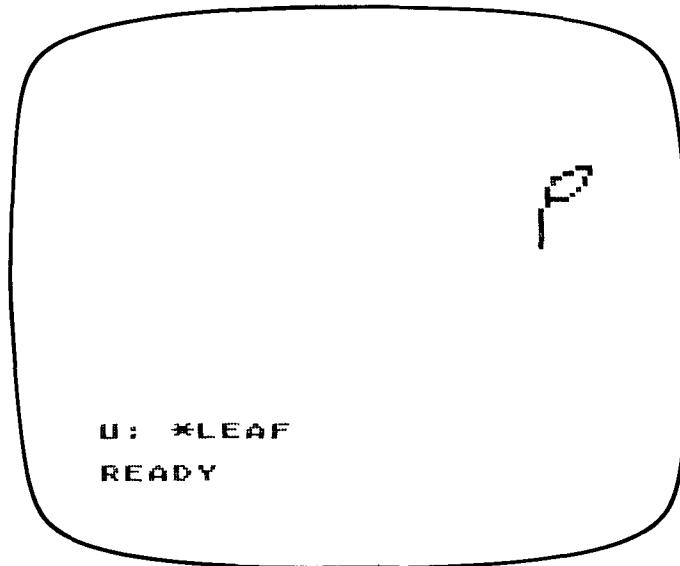
Leave the AUTO mode. Now we can experiment with our flower parts. Type:

```
GR: CLEAR
U: *STEM
```



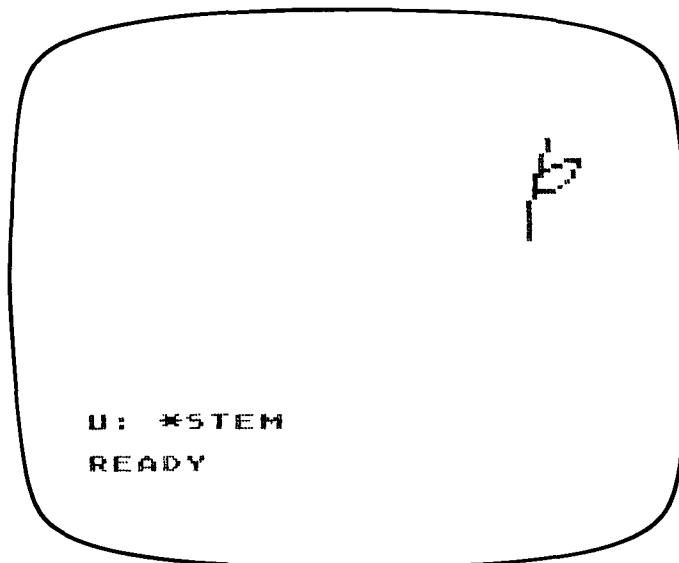
This, as you see, gave us a portion of the flower's stem. Now let's add a leaf. Type:

U: *LEAF



So far, so good. Now that we have a leaf on the stem, we should make the stem longer. To do this type:

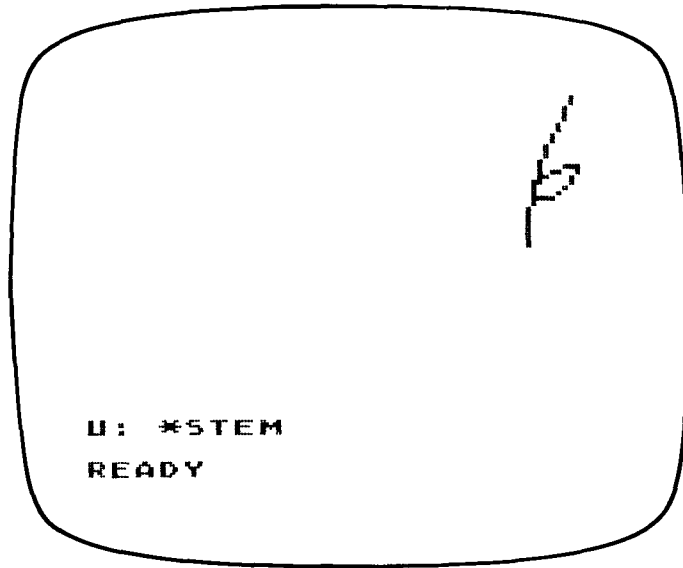
U: *STEM



PICTURE THIS!

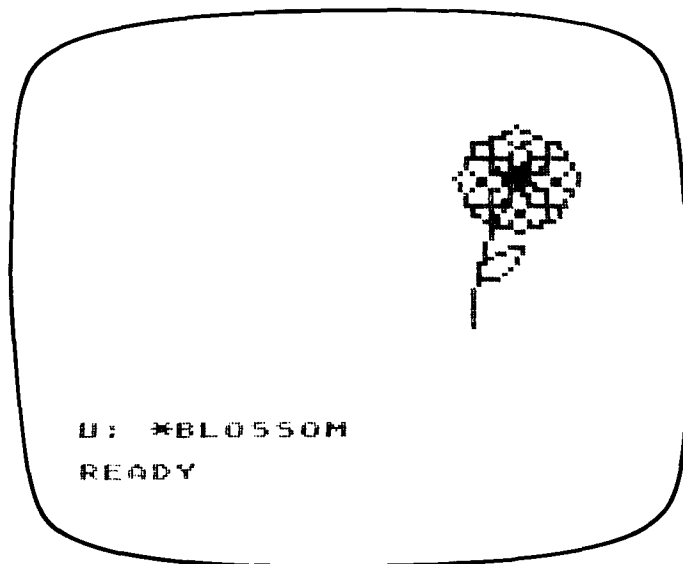
Hmmm, I don't think this will be quite long enough. Let's make the stem longer. Type:

U: *STEM



Now that is much better. Let's finish our flower with the blossom. Type:

U: *BLOSSOM



At last we have a complete flower on the screen. There is real merit in making a complex picture from smaller parts. We have the freedom to move parts around, to use them several times (as we did with the stem), and generally to check everything out before making the final module.

To make our final flower module, type:

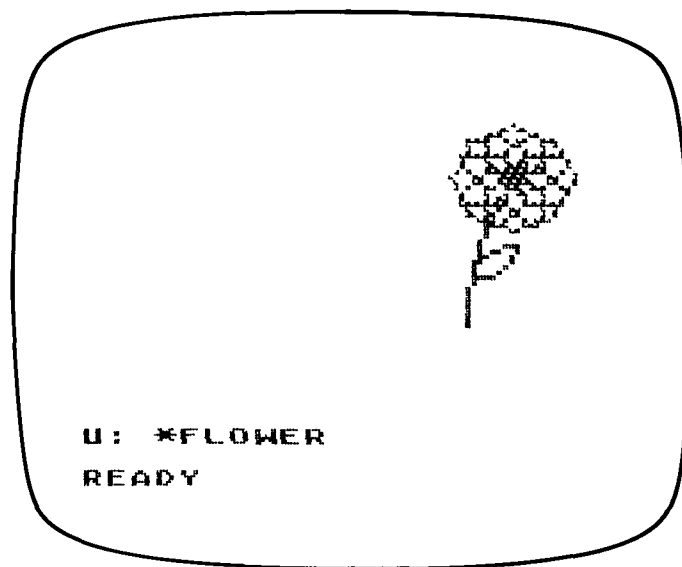
```
GR: QUIT
REN 1000
AUTO
```

and enter:

```
*FLOWER
GR: PEN YELLOW
U: *STEM
U: *LEAF
U: *STEM
U: *STEM
GR: PEN RED      [for a red blossom]
U: *BLOSSOM
E:
```

Leave the AUTO mode and type:

```
GR: CLEAR
U: *FLOWER
```



PICTURE THIS!

Ta Daa! We have finished our flower!

We could leave our flower hanging in thin air, or we could plant it in the ground or put it in a vase. I think we should put our flower in a pretty, blue vase. Of course, you can (and should!) design your own vase for your flower. I am just going to show you a vase I like so we can finish our picture.

The vase I am going to draw uses semicircles (180-degree arcs). We will first draw the right side, then the bottom, and then the left side, filling the vase with color as we go. Ready? Type:

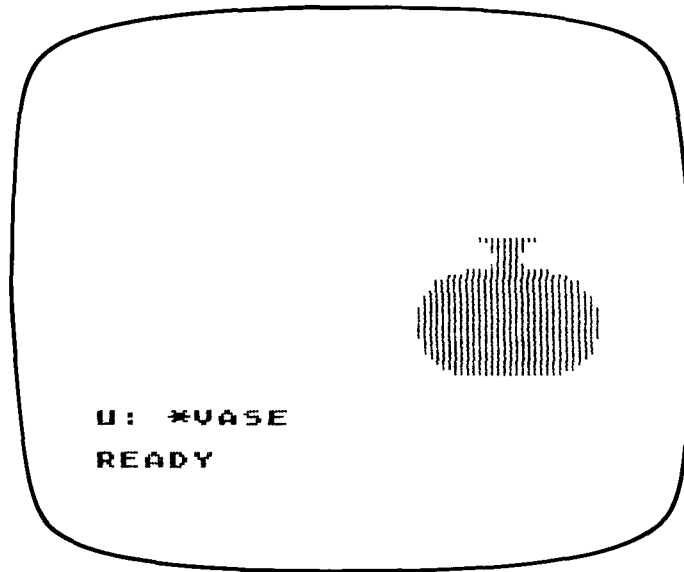
```
GR: QUIT
REN 1000
AUTO
```

and enter:

```
*VASE
GR: PEN BLUE; TURNTO -90
GR: 10(DRAW 1; TURN -18)    [upper lip of vase
GR: 30(DRAW 1; TURN 6)      [lower part of vase
GR: DRAW 10                  [bottom of vase
GR: 30(FILL 1; TURN 6)       [lower part draw and fill
GR: 10(FILL 1; TURN -18)     [upper lip draw and fill
E:
```

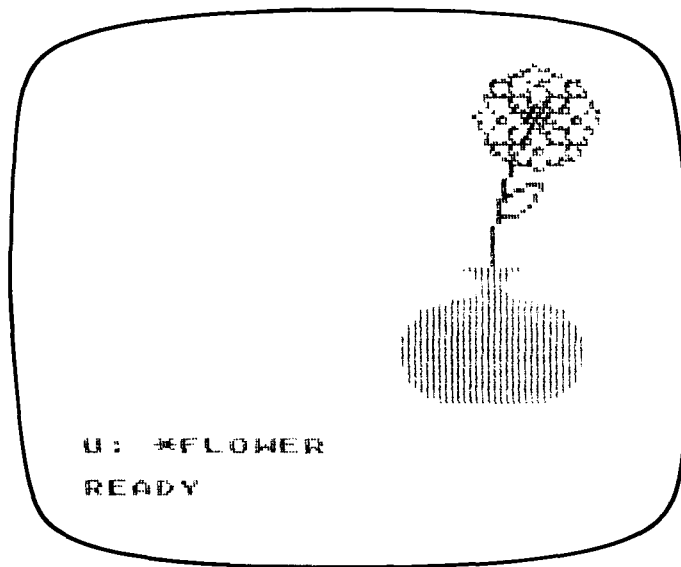
Exit from the AUTO mode, and then try using *VASE to see how it works.

Now that we have a pretty, blue vase on the screen, let's put our flower in it to finish our picture. Let's first figure out where the turtle is. If you have been keeping track of such things, you probably know that the turtle is at the upper left lip of the vase and is pointing to



the left. We should move the turtle to the middle of the vase and get it to point straight up before drawing our flower. Since the bottom of the vase is 10 units wide, we should back up 5 units in order to get to the middle. The following lines should give us what we want:

```
GR: GO -5; TURNT0 0
U: *FLOWER
```



PICTURE THIS!

And now, with this flower as our gift to each other, it gives me great pleasure to leave you to your own discoveries in the exciting world of turtle graphics!

Q: Does this mean that we are all done learning about PILOT: that we are done with our studies?

A: A woman once went into a courtyard where she happened to meet a wise old man. "Oh," she said, "I am so happy! My son just wrote me a letter from school to tell me that he has finished his studies!" The wise man said, "Well, I wouldn't worry about it too much, I'm sure life will find more for him to study soon."

APPENDIX

summary of modules used in this book

Modules Used in Chapter Six:

This module draws a square.

```
*SQUARE  
GR: 4(DRAW 25; TURN 90)  
E:
```

This module draws a pentagon.

```
*PENTAGON  
GR: 5(DRAW 25; TURN 72)  
E:
```

This module draws a stick figure of a person.

```
*PERSON  
GR: 4(DRAW 7; TURN 90)  
GR: GO 5; TURN 90; GO 2; GO 3  
GR: TURN 90; GO 3; TURN 90; DRAW 3  
GR: GO - 1; TURN - 90; GO 2  
GR: DRAW 10  
GR: TURN 45; DRAW 8  
GR: GO - 8; TURN - 90; DRAW 8; GO - 8  
GR: TURN 225; GO 7  
GR: TURN 90; GO 7; TURN 180; DRAW 14  
E:
```


PICTURE THIS!

Modules Used in Chapter Seven:

This module draws a picture of a star.

```
*STAR
GR: TURN 90
GR: 5(DRAW 25; TURN 144)
GR: TURN -90
E:
```

This module also draws a picture of a star. Do you remember why?

```
*PICTURE
U: *STAR
E:
```

This module uses *PONG. Remember the trouble we had?

```
*PING
GR: PEN YELLOW
GR: 3(DRAW 10; TURN 120)
GR: TURN 90; GO 15; TURN -90
U: *PONG
E:
```

This module uses *PING.

```
*PONG
GR: PEN RED
GR: 3(DRAW 10; TURN 120)
GR: TURN 90; GO 15; TURN -90
U: *PING
E:
```

Modules Used in Chapter Eight:

This module draws a star.

```
*STAR
GR: 5(DRAW 25; TURN 144)
E:
```

This module draws a pattern with five stars.

```
*COUNTER
C: #A = 0
*ADD1
C: #A = #A + 1
```

```

U: *STAR
GR: TURN 72
J (#A<5) : *ADD1
E:

```

This module draws patterns out of eight copies of *PICTURE.

```

*WINDMILL
GR: GOTO 0,10; CLEAR
C: #A = 0
*JUMPHERE
C: #A = #A + 1
U: *PICTURE
GR: TURN 360/8
J (#A<8) : *JUMPHERE
E:

```

This is the *PICTURE module, which uses *STAR.

```

*PICTURE
U: *STAR
E:

```

This is a *SQUARE module to use in place of *STAR in the *PICTURE MODULE.

```

*SQUARE
GR: 4(DRAW 25; TURN 90)
E:

```

This is a *PENTAGON module to use in place of *STAR in the *PICTURE module.

```

*PENTAGON
GR: 5(DRAW 20; TURN 72)
E:

```

This is a *HEXAGON module to use in place of *STAR in the *PICTURE module.

```

*HEXAGON
GR: 6(DRAW 15; TURN 60)
E:

```

Modules Used in Chapter Nine:

This module draws a polygon with #S sides. Remember to set #S to the value you want with the C: command.

```

*POLYGON
GR: GOTO 0,0; TURNT0 0
GR: #S(DRAW 25; TURN 360/#S)
E:

```

PICTURE THIS!

This module draws polygons and stars with #S points. The variable #M is the angle multiplier. Be sure to set both #S and #M with the C: command before using *TRYSTAR.

```
*TRYSTAR
GR: GOTO 0,0; TURNT0 0; CLEAR
GR: #S(DRAW 25; TURN 360*#M/#S)
E:
```

This module draws polygons with any number of sides. When the module needs a value for #S, it asks you to type one in on the keyboard.

```
*POLYGON
T: HOW MANY SIDES DO YOU WANT?
A: #S
GR: GOTO 0,0; TURNT0 0; CLEAR
GR: #S(DRAW 25; TURN 360/#S)
J: *POLYGON
E:
```

Modules Used in Chapter Ten:

This module prints the location of the turtle on the screen.

```
*WHERE
T:
T: THE HORIZONTAL LOCATION IS %X
T: THE VERTICAL LOCATION IS %Y \
E:
```

This module draws a square with the length of a side given by the value stored in #A.

```
*SQUARE
GR: 4(DRAW #A; TURN 90)
E:
```

This module uses the previous module to draw a square of any size.

```
*TELLSQUARE
T: HOW BIG A SQUARE DO YOU WANT? \
A: #A
U: *SQUARE
E:
```

This module draws a solid square that grows on the screen.

```
*GROWSQUARE
C: #A=0
*JUMPHERE
U: *SQUARE
C: #A=#A+1
J (#A<31): *JUMPHERE
E:
```

This module draws a growing square outline in which each previous square is erased from the screen before drawing the next square.

```
*GROW
C: #A=0
*HERE
GR: PEN ERASE
U: *SQUARE
C: #A=#A+1
GR: PEN YELLOW
U: *SQUARE
J (#A<31): *HERE
E:
```

This module draws a square spiral.

```
*SQUIRAL
GR: CLEAR; GOTO 0,0; TURNT0 0
C: #A=0
*DRAWLINE
GR: DRAW #A; TURN 90
C: #A=#A+1
J (%Y<48): *DRAWLINE
E:
```

This module draws a squiral with any angle.

```
*SQUIRAL
GR: CLEAR; GOTO 0,0; TURNT0 0
C: #A=0
T: WHAT ANGLE WOULD YOU LIKE? \
A: #B
*DRAWLINE
GR: DRAW #A; TURN # B
C: #A=#A+1
J (%Y<48): *DRAWLINE
E:
```

PICTURE THIS!

Modules Used in Chapter Eleven:

This module draws a big circle on the screen.

```
*BIGCIRCLE
GR: 360(DRAW 1; TURN 1)
E:
```

This module will draw different-sized circles by changing the turning-angle (stored in #A).

```
*DRAWCIRCLE
T: HOW MUCH WOULD YOU LIKE TO TURN? \
A: #A
GR: GOTO 0, -30; TURNT0 -90
U: *CIRCLE
E:
```

The previous interactive module uses another module called *CIRCLE.

```
*CIRCLE
C: #B = 360/#A
GR: #B(DRAW 1; TURN #A)
E:
```

This module draws circles for which you choose both the angle increment and the step size.

```
T: HOW MUCH WOULD YOU LIKE TO TURN? \
A: #A
GR: GOTO 0, -30; TURNT0 -90
T: WHAT STEP SIZE WOULD YOU LIKE? \
A: #S
GR: GOTO 0, -30; TURNT0 -90
U: *CIRCLE
E:
*CIRCLE
C: #B = 360/#A
GR: #B(DRAW #S; TURN #A)
E:
```

This module draws an arc on the screen.

```
*ARC
T: HOW BIG AN ARC DO YOU WANT? \
A: #A
C: #A = #A/2
GR: #A(DRAW 1; TURN 2)
E:
```

This module draws a bird's wing.

```
*WING
GR: 30(DRAW 1; TURN 3)
E:
```

This module erases a wing if the pen is set to ERASE first.

```
*ANTIWING
GR: 30(TURN - 3; DRAW 1)
E:
```

This module draws a picture of a flying bird. The value stored in #P determines the length of the pause between "flaps."

```
*FLYBIRD
GR: PEN YELLOW
GR: TURNTO 45
U: *WING
GR: TURNTO 45
U: *WING
PA: #P
GR: PEN ERASE
GR: TURNTO 90; DRAW 1; DRAW - 1
GR: TURNTO - 45
U: *ANTIWING
GR: TURNTO - 45
U: *ANTIWING
GR: TURNTO 0; GO 20; TURNTO 90; GO 8
GR: PEN YELLOW
GR: TURNTO 90
U: *WING
GR: TURNTO 0
U: *WING
PA: #P
GR: PEN ERASE
GR: TURNTO 90; DRAW 1; DRAW - 1
GR: TURNTO - 90
U: *ANTIWING
GR: TURNTO 0
U: *ANTIWING
GR: TURNTO 0; GO - 20; TURNTO 90; GO - 8
E:
```

This module moves the bird on the screen.

```
*MOVE
GR: TURNTO 90; GO 10
E:
```

PICTURE THIS!

This module puts everything together to make an animated scene of a bird flying across the screen.

```
*CARTOON
GR: GOTO -70,0
C: #P=30
*FLY
U: *FLYBIRD
U: *MOVE
J (%X<60) : *FLY
E:
```

This module resets the turtle and all variables.

```
*RESET
GR: PEN YELLOW
GR: GOTO 0,0; TURNT0 0; CLEAR
VNEW:
E:
```

This module draws a spiral whose size is given by the number stored in #S.

```
*SPIRAL
T: HOW MANY STEPS DO YOU WANT? \
A: #S
C: #C=0
*DRAWSPIRAL
GR: DRAW 3; TURN #A
C: #A=#A+1
C: #C=#C+1
J (#C<#S) : *DRAWSPIRAL
E:
```

Modules Used in Chapter Twelve:

This module draws a large, filled square.

```
*FIRSTSQUARE
GR: GOTO -15,-15; TURNT0 0; PEN YELLOW; CLEAR
GR: 4(DRAW 60; TURN 90)
GR: FILL 60
E:
```

This module draws a solid square whose length is stored in the variable #A.

```
*SOLIDSQUARE
C: #B=0
```

```

*KEEPPGOING
GR: TURNT0 90; DRAW #A; DRAW - #A
C: #B = #B + 1
J (#B > #A) : *EXIT
GR: TURNT0 0; DRAW 1
J: *KEEPPGOING
*EXIT
E:

```

This module draws a small arc used in drawing a leaf.

```

*ARC
GR: 10(DRAW 1; TURN 9)
E:

```

This module draws a leaf.

```

*LEAF
U: *ARC
GR: TURN 90
U: *ARC
GR: TURN 90
E:

```

The next two modules draw a flower blossom.

```

*PETAL
GR: 6(DRAW 5; TURN 60)
E:
*BLOSSOM
C: #A = 0
*LOCALJUMP
C: #A = #A + 1
U: *PETAL
GR: TURN 360/8
J (#A < 8) : *LOCALJUMP
E:

```

This module draws the stem.

```

*STEM
GR: 10(DRAW 1; TURN 1)
E:

```

This module draws a complete flower.

```

*FLOWER
GR: PEN YELLOW
U: *STEM
U: *LEAF
U: *STEM
U: *STEM

```


PICTURE THIS!

```
GR: PEN RED
U: *BLOSSOM
E:
```

This module draws a vase for the flower.

```
*VASE
GR: PEN BLUE; TURNTO -90
GR: 10(DRAW 1; TURN -18)
GR: 30(DRAW 1; TURN 6)
GR: DRAW 10
GR: 30(FILL 1; TURN 6)
GR: 10(FILL 1; TURN -18)
E:
```

index

Absolute commands, 62. *See also*

Commands

Accept buffer, 106–107

Accept command (A:), 106–107

Angle multipliers, 94–104

*ANTIWING, 148–149, 185

*ARC, 141–144, 169–171, 184, 187

Arcs, 141–152

Asterisks (*)

as multiplication sign, 94

before dictionary entries, 50

before modules, 77

AUTO command, 47

BASIC, 9

*BIGCIRCLE, 132–133, 184

Binary digits, 36

*BLOSSOM, 171, 174–175

Bracket, left square, 61

Cartesian geometry, 3

*CARTOON, 151, 186

*CIRCLE, 134, 184

Circles, drawing, 131–140

CLEAR command, 15–16, 27

Color

use with graphics, 159–178

Commands, 45–50

A: (Accept), 106–107

AUTO, 47

C: (Compute), 78

CLEAR, 15–16, 27

DRAW, 18, 25–43, 62

DRAWTO, 62

ERASE, 22–23, 115–116, 148

FILL, 160–165

GO, 38, 62

GOTO, 27–31, 38, 62

GR: (Graphics), 15–18

J: (Jump), 80–81

LIST, 46–50

NEW, 47

PA (Pause), 150

QUIT, 46

REN: (Renummer), 48

TURN, 19, 36–38, 62

TURNT0, 31–34, 36–38, 62

T: (Type), 105–106

U: (Use), 51–52

VNEW, 153

Compute command (C:), 78

Computer language, 10

Conditions, 77, 80–81

Coordinates, 3

*COUNTER, 80–83, 180–181

Counters, 77–82

Cursor, 14

Curves, 131–158

Debugging, 10, 98

Decagon, 41

Deferred mode, 45–47, 49

Dictionary

entries, 45, 50

limits of, 71

to define modules, 44, 55, 67, 68

DRAW command, 18, 25–43, 62

*DRAWCIRCLE, 133–140, 184

DRAWTO command, 62

Equals sign

in replacement operations, 78

ERASE command, 22–23, 115–116,
148

Factors, 103–104

Figures, closed, 170

FILL command, 160–165

*FIRSTSQUARE, 164–165, 186

*FLOWER, 175, 177, 187–188

Flowers, 168–176

*FLYBIRD, 149–150, 185

GO command, 38, 62

GOTO command, 27–31, 38, 62

GPRPP (general purpose regular
polygon plotter), 43, 89

Graphics command (GR:), 15–18

*GROW, 116, 183

*GROWSQUARE, 115, 183

Heptagons, 41, 99

*HEXAGON, 85–86, 181

Hexagons, 40–41, 94–95, 140

Immediate mode, 45–47, 51

Interactive mode, 105

Jiffies, 150

Jump command (J:), 80–81

*KEEPPGOING, 166, 187

Labels, 77, 84

*LEAF, 171, 173, 187

LIST command, 46–50

LOGO, 4

Mirror images, 102

Moire patterns, 112

Modules, 44–88, 179–188

*ANTIWING, 148–149, 185

*ARC, 141–144, 169–171, 184, 187

*BIGCIRCLE, 132–133, 184

*BLOSSOM, 171, 174–175

*CARTOON, 151, 186

*CIRCLE, 134, 184

*COUNTER, 80–83, 180–181

*DRAWCIRCLE, 133–140, 184

*FIRSTSQUARE, 164–165, 186

*FLOWER, 175, 177, 187–188

*FLYBIRD, 149–150, 185

*GROW, 116, 183

*GROWSQUARE, 115, 183

*HEXAGON, 85–86, 181

*KEEPPGOING, 166, 187

*LEAF, 171, 173, 187

*MOVE, 150–151, 185

*PENTAGON, 44, 53–54, 85–86,
179, 181

*PERSON, 55–66, 179

*PETAL, 171, 187

*PICTURE, 69–70, 83, 180, 181

*PING, 71–73, 180

*POLYGON, 90–92, 107, 181, 182

*PONG, 71–73, 180

*RESET, 152–153, 186

*SOLIDSQUARE, 166–168, 186

*SPIRAL, 153–158, 186

*SQUARE, 51–52, 85–86, 113, 179,
181, 182

*SQUIRAL, 118–130, 183

*STAR, 68–70, 76, 116–117, 180

*STEM, 172–174, 187

*TELLSQUARE, 113–114, 182

*TRYSTAR, 97–103, 182

*VASE, 176–177, 188

*WHERE, 109–112, 182

*WINDMILL, 82–84, 181

*WING, 145–149, 185

Multiplying, 94

NEW command, 47

Nonagon, 41, 95–96

Numbers, size limitation of, 34–36
Numbers, whole, 43

Octagon, 41, 43, 140

Pause command (PA:), 150

*PENTAGON, 44, 53–54, 85–86, 179,
181

Pentagons, 41, 93–94

Percent variables, 108–112

*PERSON, 55–66, 179

*PETAL, 171, 187

*PICTURE, 69–70, 83, 180, 181

PILOT, 3, 9–10

*PING, 71–73, 180

*POLYGON, 90–92, 107, 181, 182

Polygons, 39–43, 139. *See also*
GPRPP

*PONG, 71–73, 180

Prime numbers, 103–104

Programming, 9–10

Programs, *see* Modules

Pythagorean theorem, 163

QUIT command, 46

References, limits to, 73–4

Relative commands, 62. *See also*
Commands

Renumber command (REN), 48

Replacement operations, 78–79

*RESET, 152–153, 186

*SOLIDSQUARE, 166–168, 186

*SPIRAL, 153–158, 186

Spirals, 117–130, 152–158

*SQUARE, 51–52, 85–86, 113, 179,
181, 182

Squares, 20–21, 22–24, 41, 43,
108–117

*SQUIRAL, 118–130, 183

Squirals, 117–130

Stack, 74

TOO DEEP, 72–74

*STAR, 68–70, 76, 116–117, 180

*STEM, 172–174, 187

*TELLSQUARE, 113–114, 182

TOO DEEP, 72–74

Triangles, 41

*TRYSTAR, 97–103, 182

TURN command, 19, 36–38, 62

Turning-angles, 136–140

TURNTO command, 31–34, 36–38, 62

Type command (T:) 105–106

Use command (U:), 51–52

Variables, 75–107, 134

percent (%), 108–112

*VASE, 176–177, 188

VNEW command, 153

*WHERE, 109–112, 182

*WINDMILL, 82–84, 181

*WING, 145–149, 185



David D. Thornburg
PICTURE THIS!

An Introduction to
Computer Graphics for
Kids of All Ages

**Yes! Your Atari home computer can be
THE ULTIMATE TEACHING TOOL!**

PICTURE THIS! explores the wonderful possibilities of teaching kids from ages 6–60 how to use the home computer. A remarkable combination of two modern educational tools, PILOT and Turtle Geometry, makes it possible to solve problems, create pictures, and invent games on your Atari home computer. PILOT is a powerful computer language that is simpler than BASIC and allows kids to talk with the computer. Turtle Geometry makes it possible for kids to create pictures in full color with a myriad of designs. This imaginative book, with the help of your Atari home computer and a PILOT cartridge (incorporating Turtle Geometry), is the modern replacement for coloring books and crayons.

PICTURE THIS!:

- will let you use the Atari 400/800 computer as a valuable teaching tool at home or school
- shows you how to use the simple, but power-packed, language PILOT—without the detail found in other computer languages
- examines the possibilities for problem solving, creating pictures, or inventing games, all on the Atari home computer
- features a step-by-step approach which integrates projects throughout for constant learning reinforcement
- is the complete programming and graphics guide—from the flick of the Atari switch, to hands-on computing, to painting flowers and shapes of all types

If you are young, or young at heart, and want to discover what this computer craze is all about, this book designs the path to proper computer use. **PICTURE THIS!** offers the joy of learning-by-doing for everyone. What has emerged is a remarkable and rare contribution, a book that makes the conquest of computing the adventurous fun it ought to be.