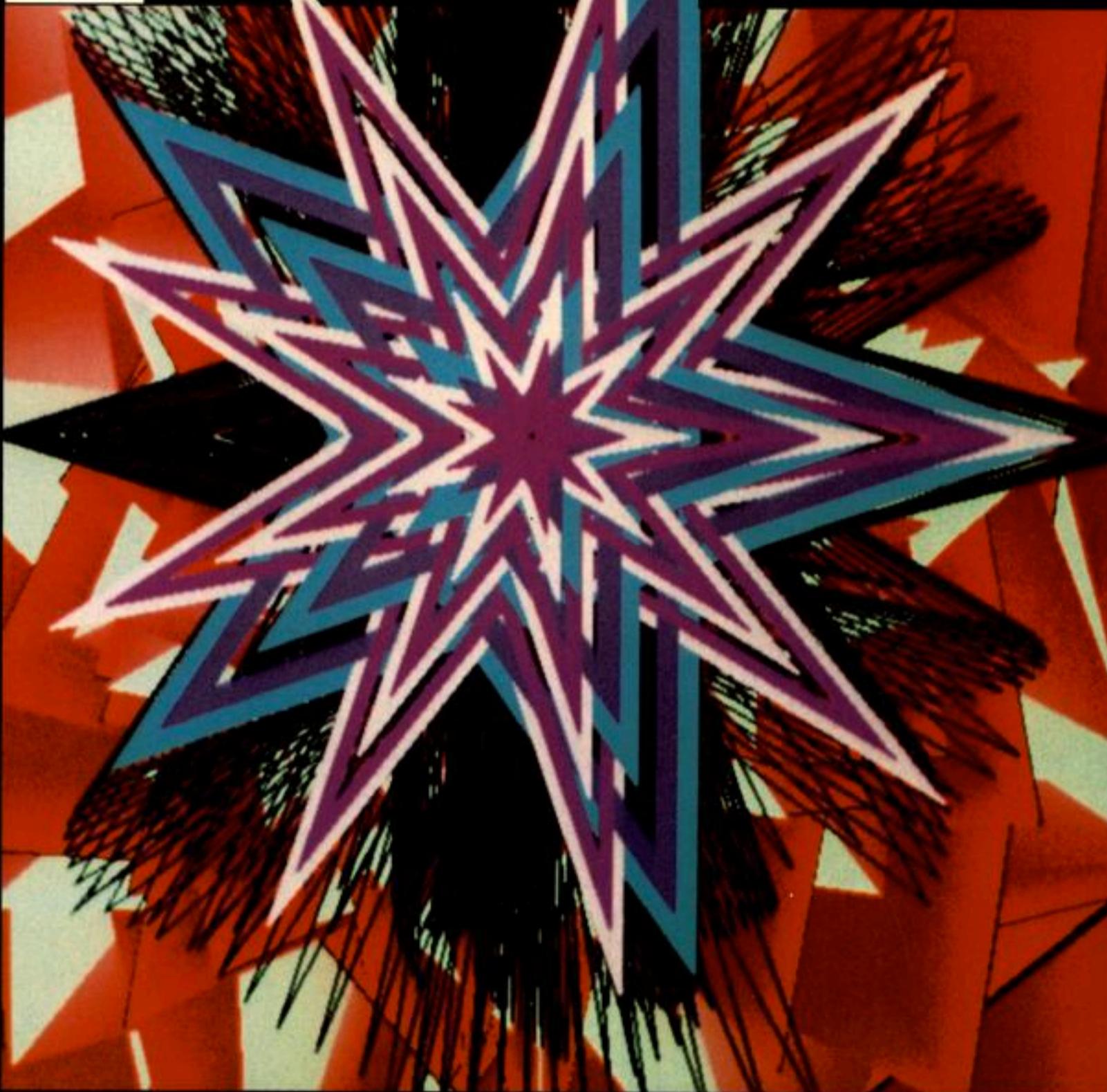


REVUE EDITION

Cynthia Solomon
Margaret Minsky
Brian Harvey

LOGOWORKS



Challenging Programs in Logo

LogoWorks

Challenging Programs in Logo

Edited by

Cynthia Solomon, Margaret Minsky, and Brian Harvey

McGRAW-HILL BOOK COMPANY

New York, St. Louis, San Francisco, Auckland

Bogotá, Hamburg, Johannesburg, London, Madrid

Mexico, Montreal, New Delhi, Panama, Paris

São Paulo, Singapore, Sydney, Tokyo, Toronto

Disclaimer of warranties and limitation of liabilities

The authors have taken due care in preparing this book and the programs in it, including research, development, and testing, to ascertain their effectiveness. The authors and publishers make no expressed or implied warranty of any kind with regard to these programs nor the supplementary documentation in this book. In no event shall the authors or publishers be liable for incidental or consequential damages in connection with or arising out of the furnishing, performance, or use of any of these programs.

Designed by C. Linda Dingler
Interior art by Kim Llewellyn

The names of all computer programs and computers included herein are registered trademarks of their makers.

Copyright © 1986 by McGraw-Hill, Inc.

All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

1234567890 EDW/EDW 89876

ISBN 0-07-042425-X

Library of Congress Cataloging in Publication Data
Main entry under title:

LogoWorks : challenging programs in Logo.

Includes index.

1. LOGO (Computer program language) 2. Computer programs. I. Solomon, Cynthia. II. Minsky, Margaret. III. Harvey, Brian, date. IV. Title: Logo Works.
QA76.73.L63L635 1986 005.36'2 85-14976

ISBN 0-07-042425-X

Two disks with all of the programs in this book, ready to run in Atari Logo, are available. Any Atari Computer that has a disk drive can run these disks, as long as you have Atari Logo. For information write

Computer Science Editor
Professional & Reference Division – 26
McGraw-Hill Book Company
1221 Avenue of the Americas
New York, NY 10020

Contents

Preface	vii	Turtle Race 206
Contributors	xiii	Four-Corner Problem 210
Introduction	xv	Towards and Arctan 212
Acknowledgments	xvii	Gongram: Making Complex Polygon Designs 214
1. Wordplay	1	Polycirc 222
Sengen: A Sentence Generator	1	Animating Line Drawings 227
Argue	6	
Animal Game	11	
Dictionary	21	
Hangman	27	
Math: A Sentence Generator	39	
Number Speller	46	
Drawing Letters	50	
Mail	58	
Wordscram	67	
Madlibs [™]	74	
2. Stories	80	
Exercise	80	
Cartoon	87	
Jack and Jill	104	
Rocket	124	
3. Games	133	
Boxgame	133	
Pacgame	140	
Blaster	151	
Alien	160	
Adventure	172	
Dungeon	191	
4. Turtle Geometry	206	
Index		381
5. Music	230	
Melodies	230	
Ear Training	239	
Sound Effects	242	
Naming Notes	251	
6. Programming Ideas	258	
Adding Numbers	258	
Fill	267	
Savepict and Loadpict	282	
Display Workspace Manager	292	
A Logo Interpreter	302	
Map	322	
Mergesort	331	
Bestline	337	
Lines and Mirrors	347	
Appendix: Special Features of Atari Logo	362	
Turtle Graphics	362	
Turtles and Their Shapes	366	
Sounds and Music	371	
Demons, Turtle Collisions, and Other Events	376	

List of Contributors

Max Behensky

Jeanry Chandler

Susan Cotten

James Davis

Lisa Delpit

Annette Dula

Gregory Gargarian

Michael Grandfield

Brian Harvey

Edward Hardebeck

W. Daniel Hillis

Julie Minsky

Margaret Minsky

Marvin Minsky

Toby Mintz

Keith Sharman

Cynthia Solomon

Erric Solomon

William Weinreb

Lauren Young

Introduction

LogoWorks: Challenging Programs in Logo is for beginning and advanced Logo programmers who want suggestions of things to do that go beyond an introductory level. The projects touch on diverse areas of interest: from graphics and video games to word games, language extensions, and development of new languages. Each project is intended for your exploration and to suggest other worlds you can build yourself.

We think this book will draw attention to Logo as a general-purpose programming language as well as a powerful tool for thinking. *LogoWorks* demonstrates that Logo is not only good for young children, but also provides people of all ages with compelling and challenging worlds to explore.

We have tried to show a diversity of projects and programming styles as well as ways of talking about the projects. The descriptions of the various projects vary in their details and their points of view. To help in meeting our goal, we encouraged many people to contribute to the book. This eventually presented us with a problem: we did not want to obscure the individual personalities represented in each of the projects. We see this as an important and necessary element in the rich development of Logo computer cultures. On the other hand, we wanted to maintain a consistency in the quality of each project. To do this, a group of us met to discuss each project that was submitted to us for inclusion in the book. The core group consisted of Margaret Minsky, Cynthia Solomon, Brian Harvey, Michael Grandfield, Lauren Young, and Susan Cotten. Again, within this group there was a wide range of interests and expertise as well as personal preferences. We think this diversity has enriched the book.

Many of the projects reflect the personal interests of their creators. For example, Michael Grandfield is a dancer and has a fascination with body movements that influenced the leaping figures in Jack and Jill. Brian Harvey, an educator and systems programmer, has contributed several projects, for instance, Drawing Letters, which draws upon interesting ideas in computer science and shows clever ways of expressing them. Susan Cotten, Lauren Young, and Annette Dula are recent Logo enthusiasts, and their projects reflect both their enthusiasms and their more recent experiments with Logo.

Some Advice About Using LogoWorks

The chapter divisions are intended as a rough guide to the projects. In each chapter there is a common theme. Nevertheless, many projects overlap chapter boundaries thematically. For example, Animal Game is in the

INTRODUCTION

"Wordplay" chapter, but it is also a game. Jack and Jill is in the "Stories" chapter, but it is also an example of turtle geometry. Furthermore, although we have not grouped the material within a chapter into levels of difficulty, there is a natural tendency to put content of particular interest for beginners toward the start of each chapter.

We assume that you are already comfortable with the elements of Logo and are looking for new challenges. Thus we expect that you have gone through the *Introduction to Programming Through Turtle Graphics*, which is part of the Atari Logo package.

For those of you familiar with Logo, but not aware of the special features Atari computers bring to the language, a chapter at the end of the book highlights special features of Atari Logo. These include four dynamic turtles, sound generation, demons, and detection of events like one turtle bumping into another or a turtle colliding with a line drawn on the screen. These features, unique to the Atari computers, are fully explored in many projects throughout this book.

We anticipate that you will want to use many of the projects without looking at the detailed explanations of how they were made. For this reason we have included complete program listings at the end of each project.

Using the Projects

There are several ways you can use the projects in this book.

- You can use a project as it appears. For example, some of the projects are games that you can play without having to do any Logo programming yourself.
- You can start with one of these projects and add to it. Some sections have explicit suggestions for ways the project might be extended; others do not. In either case, we expect you will think of your own improvements.
- A project in this book may spark an idea for a completely new project of your own.
- Some of the projects in this book are *utility* procedures, which can be used as part of a larger project. (See, for example, Towards and Arctan). You may find these procedures useful in your own projects, even if you don't understand how the procedures themselves work.

The projects in this book are written in Atari Logo. If you have another version of Logo, you will find that most of the projects are easy to adapt to your system; others depend on special features of Atari Logo and will need more effort to adapt.

Acknowledgments

Although a book of this sort has been in the planning stages for several years, this particular book owes its flavor to the fact that we were part of Atari Cambridge Research, and so Atari Logo became a focus for us. Some of the programs were adapted for the Atari computer from previous work, and others were developed to help debug Atari Logo. Some were developed while working with kids, and others were developed by kids for their own pleasure.

One of the joys of creating this book was that new and old friends contributed their programming projects. Their contributions reflect not only different programming styles, but also different ways of talking about the process of translating ideas into working programs. The list of these contributors can be found in a separate section of the book. At the beginning of each project, credit is given to the people who worked on it. We thank them collectively and individually.

We thank Atari Cambridge researchers Max Behensky, Susan Cotten, Jim Davis, Lisa Delpit, Annette Dula, Greg Gargarian, Michael Grandfield, Ed Hardebeck, Henry Minsky, Julie Minsky, and Lauren Young. We thank Jeanry Chandler and Toby Mintz, who were high school students; Danny Hillis, whose involvement with Logo dates from his undergraduate years at the MIT Logo Laboratory, and who is now a researcher at Thinking Machines Corporation; Keith Sharman, who is a programmer and Logo teacher in Alberta, Canada; Erric Solomon, who teaches Logo in the San Francisco Bay area; and Billy Weinreb, who was an undergraduate at Wesleyan.

We would also like to thank Pam Davis, who cheerfully organized and kept track of our various versions of the manuscript, and Susan Cotten, who was fantastic at keeping our diskettes up to date. Michael Grandfield and Erric Solomon not only generated beautiful pictures, but worked hard at capturing those images on film. We are particularly grateful to Peter Cann for his special ability to connect Logo to any outside device we needed. Special thanks to Greg Gargarian for his continual support in the life of this book and in our research in developing Logo worlds.

We thank the other members of Atari Cambridge Research for their support: James Russell Davis, Gary Drescher, Mark Gross, Ken Haase, Steven Hain, Jay Jones, Susan Kroon, David Levitt, Dan Melnechuk, Bill St. Clair, Nancy Smith, and Tom Trobaugh.

To those whose contributions are not apparent, like Dan Suttin and his

ACKNOWLEDGMENTS

kids at the Cambridge Montessori School and Paul Goldenberg and others from Lincoln-Sudbury Regional High School, thank you for your energy and activities in developing Logo environments.

We have been very fortunate in receiving comments on early drafts from enthusiasts like Gary Dreyfoos, Mary Jo Moore, and Jon Solomon.

This book has benefited greatly from the editorial guidance of Jane Isay, formerly of Harper & Row, Publishers. We are also indebted to Steve Guty at McGraw-Hill.

We give special thanks to Michael Grandfield for taking all the photographs used in this book and for his rendering of Gongram, which appears on the cover. He prepared this design in Object-LISP using the computing facility of LISP Machines, Inc.

We thank Brian Silverman of Logo Computer Systems for his clever implementation of Atari Logo. We also thank Hal Abelson and members of the MIT Logo community for their continuing conversations.

We are grateful to many people who were part of Atari Sunnyvale Research. We especially thank Alan Kay for his enthusiastic support.

Preface

Adults worry a lot these days. They worry especially about how to make other people learn more about computers. They want to make us all “computer-literate.” “Literacy” means both reading and writing, but most books and courses about computers only tell you about writing programs. Worse, they only tell about commands and instructions and programming-language grammar rules. They hardly ever give examples. But real languages are more than words and grammar rules. There’s also literature—what people use the language for. No one ever learns a language from being told its grammar rules. We always start with stories about things that interest us. This book tells some good stories—in Logo.

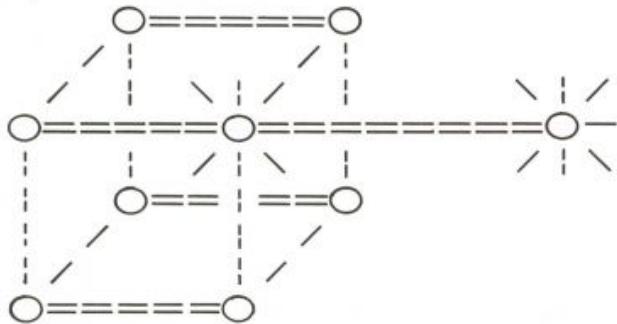
The trouble is, people often try to explain computers the same ways they explain ordinary things—the way they teach arithmetic by making you learn “tables” for adding and multiplying. So they start explaining computers by telling you how to make them add two numbers. Then they tell you how to make the computer add up a lot of numbers. The trouble is, that’s boring. For one thing, most of us already hate adding up numbers. Besides, it’s not a very interesting story.

You can’t blame teachers for trying to make numbers interesting. But—let’s face it—numbers by themselves don’t have much character. In fact, that’s the real reason mathematicians like them. They find something magical about things that have no interesting qualities at all. That sounds like a paradox. Yet, when you think about it, that’s exactly why we can use numbers in so many different ways! Why is it that we get the same kind of result when we count different kinds of things—whether we’re counting flowers or trees or cars or dinosaurs? Why do we always end up the same—with a number? That’s the magic of arithmetic. It wipes away all fine details. It strips things of their character. The qualities of what you count disappear without a trace.

Programs do the opposite. They make things come to be, where nothing ever was before. Some people find a new experience in this, a feeling of freedom, a power to do anything you want. Not just a lot—but anything. I don’t mean like getting what you want by just wishing. I don’t mean like having a faster-than-light spaceship, or a time machine. I mean like giving a child enough kindergarten blocks to build a full-sized city without ever running out of them. You still have to decide what to do with the blocks. But there aren’t any outside obstacles. The only limits are within yourself.

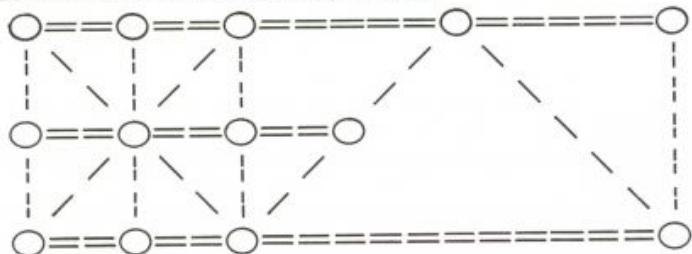
Myself, I first had that experience before I went to school. There weren’t any Logos yet, but we had toy construction sets. One was called TinkerToy™. To build with TinkerToy you only need two kinds of parts—just sticks and spools. Spools are little wooden wheels. Each has one hole

through the middle and eight holes drilled into the rim. Sticks are just round sticks of various lengths, which you can push into the spool holes. They have little slits cut in their ends, which make them hold tight when they're pressed into the holes.



What's strange is that those spools and sticks are enough to make anything. Some spools are drilled with larger holes, so sticks pushed through those holes can turn. You can make towers, bridges, cars, bulldozers. Windmills. Giant animals. You can put wheels on your cars and make bearings for pulleys and gears to make them do more interesting things. You have to make the gears yourself: just stick eight sticks into a spool. They work, though not too well, and always go *click-click-click* when they turn.

The sticks are cut to several lengths. One series of lengths come in the ratios one, two, four, and eight. The other lengths are cut so that they fit across the diagonals of squares made from the first series of sticks. The amazing thing is that you can also use the first kind as diagonals for squares made with the other kind of sticks, like this:



The secret is in finding out how much can come from so few kinds of parts. Once, when still a small child, I got quite a reputation. My family was visiting somewhere and I built a TinkerToy tower in the hotel lobby. I can't recall how high it was, but it must have been very high. To me it was just making triangles and cubes, and putting them together. But the grownups were terribly impressed that anyone so small could build anything so big. And I learned something too—that some adults just didn't understand how you can build whatever you want, so long as you don't run out of sticks and spools. And only just this minute while I'm writing this, I realize what all that meant. Those adults simply weren't spool-stick-literate!

When my friend Seymour Papert first invented Logo, I had the same experience again. Logo has some things like sticks: we call them FORWARD :LENGTH. And Logo also has its spools: we call them RIGHT :ANGLE. I recognized old building friends at once. Making Logo programs is a lot like

building with construction toys—but it's even better. You can make drawings of things and structures, but you can make procedures too. You can make them use words. You can make things change their forms. And you can make them interact: just give them each procedures which can change the values of the other ones. As toys, those programs have their faults: you can't take Logo cars outside and roll them down a real hill—but, in exchange, their parts don't get loose and fall out and get lost. And the basic experience is still there: to see how simple things can interact to make more wonderful things.

Logo started many years ago; several writers of this book were children, and among the first to find new things to do with it. I'm very pleased to write this introduction now, recalling what a great adventure this has been and knowing, too, that it has just begun.

There were other good construction toys, like ErectorTM sets and Meccano. They had many kinds of parts, but the basic ones are metal strips with many holes and different kinds of angle brackets. You got a million little screws and nuts to put them together with, and long steel shafts, which fit through the holes just loose enough to turn. And there are gears and pulleys to attach to the metal shafts, so you can make complicated things that really work.

When I was older, I built one of the very first modern, remote-controlled robots, using parts of a Number 10 Meccano set and ideas invented at MIT's first computer research laboratory in the 1940s. And, speaking of building computers, some of the people in this book once built a real, honest-to-goodness computer out of nothing but TinkerToy parts. A group including Danny Hillis, Brian Silverman, and Ed Hardebeck built a machine of TinkerToy parts to play the game tic-tac-toe. It actually worked and is now in a museum in Arkansas. It was made of spools and sticks. They also used some string and, since the truth must be told, they hammered in some little brass nails to keep the sticks from falling out. It took about 100 sets and was too big to fit in your room.

The golden age of construction sets came to its end in the 1960s. Most newer sets have changed to using gross, shabby, plastic parts, too bulky to make fine machinery. Meccano went out of business. That made me very sad. You can still buy Erector, but insist on the metal versions. Today the most popular construction set seems to be LEGOTM—a set of little plastic bricks that snap together. LEGO, too, is like Logo—except that you only get `RIGHT 90`. It is probably easier for children, at first, but it spans a less interesting universe and doesn't quite give that sense of being able to build “anything.” Another new construction toy is FischerTechnikTM, which has good strong parts and fasteners. It is so well made that engineers can use it. But because it has so many different kinds of parts, it doesn't quite give you that Logolike sense of being able to build your own imaginary world.

About the time that building toys went out of style, so did many other things that clever kids could do. Cars got too hard to take apart—and radios, impossible. No one learned to build much any more, except to snap together useless plastic toys. And no one seemed to notice this, since sports and drugs and television crime came just in time. Perhaps computers can help bring us back.

PREFACE

After you've built something with your construction set, you have to take it all apart again—or you won't have enough parts for the next project. With programs, you can keep them on your disk and later get them out and build them into bigger ones. This year, you might run out of memory—but that won't be a problem for *your* children, because memory will soon be very cheap. What's more, you can share your programs with your friends—and still have them yourself! No emperor of ancient times could even dream of that much wealth. Still, many adults just don't have words to talk about such things—and maybe, no procedures in their heads to help them think of them. They just do not know what to think when little kids converse about "representations" and "simulations" and "recursive procedures." Be tolerant. Adults have enough problems of their own.

To understand what computers are, and what they do, you shouldn't listen to what people say about those "bits" and "bytes" and binary decisions. I don't mean that it isn't true. Computers are indeed mostly made of little two-way switches. But everyone who tells you this is what you need to understand them is simply wrong. It's just as true that houses can be made from sticks and stones—but that won't tell you much about architecture. It's just as true that animals are mostly made of hydrogen, carbon, oxygen, and nitrogen—but that won't tell you much about biology.

A Martian szneech once mindlinked me; it wanted to know what literature was. (It seems they're far behind the times and haven't even got to that.) I told it how we make sentences by putting words together, and words by putting letters together, and how we put bigger spaces between words so that you can tell where they start and stop. "Aha," it said, "but what about the letters?" I explained that all you need are little dots since, if you have enough of them, you can make anything.

The next time, it called to ask what tigers were. (Apparently, they haven't even got to vertebrates.) I explained that tigers were mostly made of hydrogen and oxygen. "Aha," it said, "I wondered why they burned so bright."

The last time it called, it had to know about computers. I told it all about bits and binary decisions. "Aha," it said, "I understand."

You really need two other facts to understand what computations do. Here's the first one.

Computer programs are societies. Making a big computer program is putting together little programs.

To make a good program, you build a larger process out of smaller ones. I suppose you could truthfully say that sculptors make large shapes from stuck-together grains of clay. But that shows what's wrong with the bits-and-bytes approach. No sculptor or scientist or programmer ever thinks that way. An architect first thinks of shapes and forms, then walls and floors, and only last about how those will be made.

Here is the other thing most people don't know. It doesn't matter very much what you start with! Even if we start with different kinds of computers, with different kinds of parts inside—still, they mostly can be made to do the same things, when seen from the outside. You can build a windmill

with either wooden sticks or metal beams. When you look closely, each part will be quite different. But windmills will have a base, a tower, and a propeller. The same with computers:

Any computer can be programmed to do anything that any other computer can do—or that any other kind of “society of processes” can do.

Most people find this unbelievable. But later in this book you'll find a hard project called “A Logo Interpreter” that shows how to make one type of Logo computer act like a different kind of Logo computer. The way to do this was first discovered by an English scientist named Alan Turing. First he asked what a computer is—and realized that the only important thing about a computer X is the set of laws that make its parts change their states. Except for that, it doesn't matter how parts are made. Then Turing asked what programs are, and realized: a program is a certain way to fix some set of X's states. This, then, will prearrange the way that X's other states will later change.

Next, Turing thought, suppose you wanted a different kind of computer, Y. Then make a program for X that will make the rest of X's states act just like Y's. Once that's done, X's behavior will look exactly like Y's—to anyone watching from outside. A programmer might say that X is “simulating Y.” Of course, you have to pay a price for this: it won't work at all unless X has enough memory to hold a description of Y. And if X and Y are very different, then the simulated programs will run very slowly. But aside from that, Turing showed, any kind of computer can be programmed to simulate any other kind! That is why we can write special programs to make the same Logo programs run on all the different computers in the world.

In fact, every Logo system uses just such a program, to make the underlying computer act like a Logo computer. The one for Atari Logo was written by Brian Silverman and his friends. I'm sorry you can't examine it; the trouble is that it's not written in Logo but in a different language buried deep inside your machine. But it's there, hiding out of sight, making your computer simulate a Logo computer. The strange thing is that Alan Turing figured out how to do such things fifty years ago, in 1936, long before computers were even invented! How could he do that? He simulated them inside his head.

There's something “universal” about how big things don't depend so much on what's inside their little parts. This must be the secret of those magical experiences I had, first with those construction sets and, later, with languages like Logo. What matters is how the parts affect each other—not what they are themselves. That's why it doesn't matter much if one makes houses out of boards or bricks. Similarly, it probably won't matter if aliens from outer space have bones of gold instead of bones of stone, like ours. People who don't appreciate how simple things can grow into entire worlds are missing something important. They find it hard to understand science, because they find it hard to see how all the different things we know could be made of just a few kinds of atoms. They find it hard to understand evolution because they find it hard to see how different things like birds and bees and bears could come from boring, lifeless chemicals—by testing trillions of procedures. The trick, of course, is that it's done by many steps, each using procedures that have been debugged already, in the same way, but on smaller scales.

PREFACE

Why don't our teachers tell us that computers have such glorious concerns? Because most adults still believe the only things computers do are big, fast, stupid calculations of arithmetic. Besides, our teachers have too many other things to learn and teach: how to build computers, how to make languages for programming them, and how to train programmers to use those languages. And so those dreary practicalities of billion-dollar industries crowd out our dreams and fantasies of building giant mind-machines.

Marvin Minsky

1

Wordplay

Sengen: A Sentence Generator

SENGEN makes up English sentences similar to the following ones:

PECULIAR BIRDS HATE JUMPING DOGS
FAT WORMS HATE PECULIAR WORMS
RED GUINEA PIGS TRIP FUZZY WUZZY DONKEYS
FAT GEESE BITE JUMPING CATS

One of the questions you might ask is this: Does SENGEn make up sentences the way we do or the way we did when we first learned to talk or write? Another question you might ask is: What relationship does SENGEn have to understanding grammar? The first question is open to research and speculation. The second might be an easier one to answer. Often when I first discuss this project with children, they do not relate the programming process to the learning of grammar. Later as they use their programs, the children frequently exclaim: "So this is why they call words nouns and verbs!" They also begin to appreciate formal systems. Studying grammar by generating sentences that obey certain rules requires the programmer to become aware of rules as well as of their exceptions.

Since this program seems to make sensible sentences without knowing very much about grammar, children often develop an appreciation for cleverness. For example, SENGEn doesn't know that some words are singular and some are plural or that singular subjects should be matched with singular verbs; it does not know about verb tenses or pronomial relations. Its apparent intelligence comes from the programmer's choice of words and categories.

In the following examples, the nouns and verbs are all plurals and the verbs are all in the present tense.

SENGEn builds sentences from vocabulary lists of nouns, verbs, adjectives, connectives, and so on. It then assembles its selections according to some rule of grammar.

Making the Program

One strategy in making a program might be to concentrate on developing a random sentence generator that outputs only a verb. For example:

Go.
Run.

By Cynthia Solomon.

WORDPLAY

To do this a procedure is needed to blindly (randomly) pick out a selection from a list of possibilities.

Let's make up a list of verbs and then make a procedure to select a word from the list. In this example, the procedure VERBS outputs the vocabulary list.

```
TO VERBS
OP [EAT SCARE LOVE HATE [LAUGH AT] TRIP BITE]
END
```

Whenever VERBS is called, it outputs that list.

```
PR VERBS
EAT SCARE LOVE HATE [LAUGH AT] TRIP BITE
```

```
PR FIRST VERBS
EAT
```

```
PR LAST BL VERBS
TRIP
```

What we now want is a procedure that will randomly choose one of the items in this list. Here is the plan for this task: use a number obtained from RANDOM to point to an item in the given list of choices. Then get that item from the list. PICK does this and outputs the selection.

```
TO PICK :LIST
OP SELECT RANDOM COUNT :LIST :LIST
END
```

PICK's input is a vocabulary list. PICK calls SELECT, giving it the list and a number indicating which item in the list SELECT is to output.

There is a slight problem. RANDOM outputs a number from 0 up to but not including its input number. Thus its output in PICK is always one less than the length of the list. We can fix that by adding 1 to RANDOM's output.

```
TO PICK :LIST
OP SELECT 1 + RANDOM COUNT :LIST :LIST
END
```

SELECT carries out its job recursively. When its input number is one, it outputs the first item from its input list. Otherwise, SELECT subtracts one from its input number and takes away the first item from its input list and continues the process until the item is found.

Here is what SELECT looks like.

```
TO SELECT :ITEM :LIST
IF :ITEM = 1 [OP FIRST :LIST]
OP SELECT :ITEM - 1 BF :LIST
END
```

Now we can try PICK.

```
PR PICK VERBS
SCARE
```

```
PR PICK VERBS
EAT
```

We could try it on different lists:

PR PICK [1 2 3 4]

3

PR PICK [A B C D]

A

PICK seems to work.

Let's make a procedure that outputs just a verb.

```
TO VERB
OP PICK VERBS
END
```

```
PR VERB
BITE
```

Now we can move on to building a sentence by first making a one-word sentence.

```
TO SEN
PR VERB
SEN
END
```

```
SEN
LAUGH AT
SCARE
LAUGH AT
EAT
```

and so on.

Our attempt at making a one-word sentence fails because of the verbs in the verb list. Only EAT can be used without an object. So if we want to make grammatical one-word sentences, we have to restrict our choice of verbs.

Now let's make a sentence with a subject and an object. Let's follow the pattern already set up for verbs and make two operations NOUNS and NOUN. NOUNS outputs a list of nouns.

```
TO NOUNS
OP [BOYS [DOGS AND CATS] PUPPIES [SIAMESE FIGHTING FISH]
GEESE BIRDS GIRLS [GUINEA PIGS][MICE AND GERBILS] WORMS
TEACHERS DONKEYS CLOWNS [BASEBALL PLAYERS]]
END
```

NOUN outputs one of the items from NOUNS.

```
TO NOUN
OP PICK NOUNS
END
```

```
PR NOUN
CLOWNS
```

WORDPLAY

All we need to do to make a sentence is the following:

```
PR (SE NOUN VERB NOUN)
SIAMESE FIGHTING FISH SCARE BOYS
```

Imagine we had miscategorized the vocabulary and NOUNS could output a list like

```
RED LAUGHING TORTOISE BOY
```

We might then get sentences like

```
RED SCARE LAUGHING
BOY EAT TORTOISE
```

This kind of bug is typical of the kind people run into when they first do this project. Usually, when people confront their bugs, they begin to appreciate rules of grammar and the fantastic power we derive from categorizing words.

We can now make a procedure that outputs a sentence.

```
TO SEN
OP (SE NOUN VERB NOUN)
END
```

```
PR SEN
BASEBALL PLAYERS EAT DONKEYS
```

SENGEN can print this output and continue the process.

```
TO SENG
PR SEN
PR []
SENGEN
END
```

Extensions

One extension is to add adjectives to the sentences.

```
TO ADJECTIVES
OP [RED FAT [FUZZY WUZZY] PECULIAR JUMPING]
END
```

```
TO ADJECTIVE
OP PICK ADJECTIVES
END
```

Edit SEN.

```
TO SEN
PR (SE ADJECTIVE NOUN VERB NOUN)
END
```

SENGEN: A SENTENCE GENERATOR

5

The sentences are getting more complicated, so it is time to introduce additional categories like NOUNPHRASE and VERBPHRASE. For example:

```
TO NOUNPHRASE  
OP (SE ADJECTIVE NOUN)  
END
```

```
TO VERBPHRASE  
OP (SE VERB NOUN)  
END
```

```
TO SEN  
OP (SE NOUNPHRASE VERBPHRASE)  
END
```

Another possibility is to link two simple sentences by using connectives:

```
TO CONNECTS  
OP [BUT AND [EVEN THOUGH]]  
END
```

```
TO CONNECT  
OP PICK CONNECTS  
END
```

Finally, you change SENG to include the new sentence:

```
TO SENG  
PR (SE SEN CONNECT SEN)  
PR []  
SENG  
END
```

```
SENG  
FAT DOGS HATE DONKEYS EVEN THOUGH  
JUMPING BASEBALL PLAYERS SCARE BOYS
```

```
RED CATS EAT WORMS EVEN THOUGH  
FUNNY BUNNY BASEBALL PLAYERS LOVE BOYS
```

```
FAT MICE AND GERBILS SCARE TEACHERS  
AND FAT CLOWNS BITE MICE AND GERBILS
```

PROGRAM LISTING

```
TO SENG  
PR (SE SEN CONNECT SEN)  
PR []  
SENG  
END  
  
TO SEN  
OP (SE NOUNPHRASE VERBPHRASE)  
END
```

```
TO CONNECTS  
OP [BUT AND [EVEN THOUGH]]  
END  
  
TO CONNECT  
OP PICK CONNECTS  
END  
  
TO NOUNPHRASE  
OP (SE ADJECTIVE NOUN)  
END
```

WORDPLAY

```

TO NOUN
OP PICK NOUNS
END

TO NOUNS
OP [BOYS [DOGS AND CATS] PUPPIES ▶
    [SIAMESE FIGHTING FISH] GEESE ▶
    BIRDS GIRLS [GUINEA PIGS][MICE ▶
    AND GERBILS] WORMS TEACHERS ▶
    DONKEYS CLOWNS [BASEBALL ▶
    PLAYERS]]
END

TO VERB
OP PICK VERBS
END

TO VERBS
OP [EAT SCARE LOVE HATE [LAUGH AT] ▶
    TRIP BITE]
END

TO ADJECTIVE
OP PICK ADJECTIVES
END

TO ADJECTIVES
OP [RED FAT [FUZZY WUZZY] PECULIAR ▶
    JUMPING]
END

TO VERBPHRASE
OP (SE VERB NOUN)
END

TO PICK :LIST
OP SELECT 1 + RANDOM COUNT :LIST :LIST
END

TO SELECT :ITEM :LIST
IF :ITEM = 1 [OP FIRST :LIST]
OP SELECT :ITEM - 1 BF :LIST
END

```

Argue

ARGUE carries on a dialogue with you. When you run ARGUE, it expects you to type a statement in the form I LOVE LEMONS or I HATE DOGS. ARGUE comes back with contrary statements. For example, if you make the statement I HATE DOGS, the program types

I LOVE DOGS
I HATE CATS

If it doesn't already know the opposite of a word, it asks you. For example, if you type I LOVE LEMONS and ARGUE does not know the opposite of LEMONS, it types

I HATE LEMONS
WHAT IS THE OPPOSITE OF LEMONS?

If you tell it ORANGES, it will type

I LOVE ORANGES

Here is a sample dialogue.

```
ARGUE
-->I LOVE SALT
I HATE SALT
I LOVE PEPPER
```

```
-->I HATE CATS
I LOVE CATS
I HATE DOGS
```

```
-->I HATE DOGS
I LOVE DOGS
I HATE CATS
```

```
-->I LOVE LEMONS
I HATE LEMONS
WHAT IS THE OPPOSITE OF LEMONS?ORANGES
I LOVE ORANGES
```

```
-->
```

ARGUE Can Reply to Your Statements

When you run ARGUE, it types an arrow to let you know that it is ready for you to type your statement, then calls ARGUEWITH. ARGUEWITH is given the statement you type as its input. ARGUE is recursive so this process continues.

```
TO ARGUE
TYPE [\\-\\->]
ARGUEWITH RL
ARGUE
END
```

ARGUEWITH prints two responses to your statement. First, it turns around your statement; if you say that you *love* something, ARGUEWITH says that it *hates* it, and if you say you *hate* something, ARGUEWITH says that it *loves* it. Second, it makes a statement about the opposite of the object you mentioned.

```
TO ARGUEWITH :STATEMENT
PRINT ( SE "I LOVE.HATE SECOND :STATEMENT LAST :STATEMENT )
PRINT ( SE "I SECOND :STATEMENT OPPOSITE LAST :STATEMENT )
END
```

The procedure LOVE.HATE sees whether its input is "LOVE or "HATE and outputs the other one.

```
TO LOVE.HATE :WORD
IF :WORD = "LOVE [OP "HATE]
IF :WORD = "HATE [OP "LOVE]
END
```

WORDPLAY

The ARGUEWITH procedure works only with statements in the form I LOVE *something* or I HATE *something* because it assumes that the second word in your statement is LOVE or HATE and that the last word in your statement is something whose opposite it can find.

ARGUEWITH uses SECOND to grab the second word in a sentence.

```
TO SECOND :LIST
OP FIRST BF :LIST
END
```

The OPPOSITE procedure is the real guts of the ARGUE program. It takes a word as its input and outputs the opposite of that word.

The Program Keeps Track of Opposites

How does the program know that *pepper* is the opposite of *salt*? Somehow, the ARGUE program has to have this information stored. We use variables to hold this information. For example, :SALT is PEPPER, :CATS is DOGS. This is how we have chosen to store the facts the program "knows." We call this a *data base*. You can look at the data base for the ARGUE program by looking at all the variables in the workspace. Try:

```
PONS
MAKE "PEPPER "SALT
MAKE "SALT "PEPPER
MAKE "DOGS "CATS
MAKE "CATS "DOGS
MAKE "LIFE "MARRIAGE
MAKE "MARRIAGE "LIFE
MAKE "DARK "LIGHT
MAKE "LIGHT "DARK
```

These variables are loaded into the workspace with the ARGUE program.*

To find out the opposite of something, for example DARK, we can say

```
PR :DARK
LIGHT
or
```

```
PR THING "DARK
LIGHT
```

What if we want to find out the opposite of LIGHT? There is no easy way to find out it is DARK unless we have another variable named LIGHT, with value DARK. So we can say

```
PR THING "LIGHT
DARK
```

We have set up a convention in our data base that we always put in both parts of a pair. That way, we don't end up in the funny situation where it

*If you type in the procedures and there are no variables in the workspace, ARGUE will create these variables when it asks you for the opposites of things.

is easy to find out that the opposite of ROUGH is SMOOTH, but impossible to find out what the opposite of SMOOTH is. Our mental concept of opposite is that it "goes both ways," so we make our data base reflect that.

How the OPPOSITE Procedure Works

With this kind of data base we can write a procedure to output the opposite of something. Here is a possible first version of the OPPOSITE procedure:

```
TO OPPOSITE :OBJECT
OP THING :OBJECT
END
```

This is a good example of needing to use THING rather than dots(:). The word of which OPPOSITE is trying to find the value is whatever :OBJECT is. For example, if :OBJECT is the word SALT, then the program is trying to find :SALT. It must do this indirectly by using THING :OBJECT.

This first version of OPPOSITE has a problem. It only works for words that are already in the data base. If you make a statement like I LOVE SUNSETS and there is no variable named SUNSETS, then this OPPOSITE procedure will get an error. To solve this problem, we use NAMEP to check for the existence of a variable named by :OBJECT. In this example :OBJECT is the word SUNSETS; the program checks whether there is already a variable named SUNSETS. If there isn't, you'd like the program to learn the opposite of SUNSETS and put it in the data base. Then it can go ahead and argue with you about sunsets. The procedure LEARNOPP does this. OPPOSITE calls LEARNOPP when it needs to.

```
TO OPPOSITE :OBJECT
IF NAMEP :OBJECT [OP THING :OBJECT]
PRINT ( SE [WHAT IS THE OPPOSITE OF] :OBJECT "? ")
LEARNOPP :OBJECT FIRST RL
OP THING :OBJECT
END

TO LEARNOPP :OBJECT :OPP
MAKE :OBJECT :OPP
MAKE :OPP :OBJECT
END
```

When OPPOSITE tries to find the opposite of a word that is not in the data base, it asks the user for the opposite. After the user types the opposite, OPPOSITE passes both the problem word and its opposite to LEARNOPP. LEARNOPP puts that pair of words in the data base.

Now ARGUE Can Argue Pretty Well

So ARGUE can keep going as it adds new words to its data base.

WORDPLAY**ARGUE**

```
-->I HATE PEPPER
I LOVE PEPPER
I HATE SALT
-->I LOVE SUNSETS
I HATE SUNSETS
WHAT IS THE OPPOSITE OF SUNSETS?SUNRISES
I LOVE SUNRISES
-->I LOVE SUNRISES
I HATE SUNRISES
I LOVE SUNSETS
```

and so on.

If we look at the data base after this, we can see what has been added.

PONS

```
MAKE "SUNRISES" "SUNSETS"
MAKE "SUNSETS" "SUNRISES"
MAKE "PEPPER" "SALT"
```

and so on.

In order for the program to "remember" this data base, these variables must be saved by **SAVEing** this workspace on a diskette.

SUGGESTIONS

The ARGUE program assumes that the sentences you type in are going to be exactly in the form

I LOVE *something*

or

I HATE *something*

If they are not, an error occurs and the program stops. You could improve the program so that it checks for the right kinds of sentences and asks you to retype them if there are problems.

Maybe it could know about more emotion words such as DESIRE, LIKE, DISLIKE, DESPISE, DETEST.

If you try:

I LOVE GREEN PEAS

the program will say:

I HATE PEAS

and ask you for the opposite of PEAS. It will ignore the GREEN. You might make a better arguing program that tries to figure out if there is an adjective and finds its opposite, so it would do something sensible like

```
ARGUE
-->I LOVE GREEN PEAS
I HATE GREEN PEAS
I LOVE RED PEAS
```

ARGUE doesn't have any mechanism for dealing with single objects described by more than one word, like ICE CREAM. Perhaps a special way to type these in might be added.

You might want to look at the Madlibs and Sengen projects for more ideas that have to do with taking apart and putting together sentences. You might want to look at the Animal Game project for an example of a program with a different kind of data base that also appears to learn some simple things.

PROGRAM LISTING

```

TO ARGUE
TYPE [ \-\-\-\>]
ARGUEWITH RL
ARGUE
END

TO ARGUEWITH :STATEMENT
PRINT ( SE "I LOVE.HATE SECOND ▶
      :STATEMENT LAST :STATEMENT )
PRINT ( SE "I SECOND :STATEMENT ▶
      OPPOSITE LAST :STATEMENT )
END

TO LOVE.HATE :WORD
IF :WORD = "LOVE [OP "HATE]
IF :WORD = "HATE [OP "LOVE]
END

TO SECOND :LIST
OP FIRST BF :LIST
END

TO OPPOSITE :OBJECT
IF NAMEP :OBJECT [OP THING :OBJECT]
PRINT ( SE [WHAT IS THE OPPOSITE OF] ▶
      :OBJECT "? ")
LEARNOPP :OBJECT FIRST RL
OP THING :OBJECT
END

TO LEARNOPP :OBJECT :OPP
MAKE :OBJECT :OPP
MAKE :OPP :OBJECT
END

MAKE "PEPPER "SALT
MAKE "SALT "PEPPER
MAKE "DOGS "CATS
MAKE "CATS "DOGS
MAKE "LIFE "MARRIAGE
MAKE "MARRIAGE "LIFE
MAKE "DARK "LIGHT
MAKE "LIGHT "DARK
MAKE "SUNRISES "SUNSETS
MAKE "SUNSETS "SUNRISES

```

Animal Game

The animal game is a little like twenty questions: you think of an animal, and the game tries to guess it by asking yes-or-no questions.*

What makes the game interesting is that it learns new animals. When it can't guess your animal, it asks you to teach it the animal and its distinguishing characteristic. By learning new questions and new animals, the game gets "smarter."

*This animal game is a popular computer game. It first appeared about ten years ago. Since then many people have implemented it in various computer languages. This Logo program was inspired by Bernard Greenberg's unpublished LISP textbook.

WORDPLAY

Here's a sample dialogue between the computer and a person playing the animal game. Everything the user types is boldface.

```
?ANIMALGAME
PICK AN ANIMAL, ANY ANIMAL
IS IT FURRY?
YES
HERE'S MY GUESS: IS IT A CAT?
NO
I GIVE UP. WHAT IS IT?
A DOG
PLEASE TYPE IN A QUESTION
WHOSE ANSWER IS 'YES' FOR A DOG
AND 'NO' FOR A CAT
DOES IT BARK?
DO YOU WANT TO PLAY AGAIN?
YES
PICK AN ANIMAL, ANY ANIMAL
IS IT FURRY?
MAYBE
PLEASE ANSWER YES OR NO
IS IT FURRY?
YES
DOES IT BARK?
YES
HERE'S MY GUESS: IS IT A DOG?
YES
I WIN!
I WIN!
DO YOU WANT TO PLAY AGAIN?
NO
?
```

The player's secret animal is "dog."

Here's where the game gets smarter.

The player's secret animal is "dog" again.

Here's where the game asks the question it just learned!

Knowledge Grows on Trees

Below is a diagram of the knowledge the game might have after someone has played it a few times. We call the diagram a *tree*, because it looks something like an upside-down tree.



The tree is made of questions and animal names. Each question has a "yes branch" and a "no branch." Each branch either leads to a question or ends at an animal name.

By drawing what the game knows in the form of a tree, we can get a more vivid picture of how the game works. For example, we can think of the game as exploring the tree from its top. It always starts at the IS

IS IT FURRY? question. Its goal is to climb down the branches to an animal name. The animal it finally reaches is the one it guesses.

Let's play an imaginary game and trace the game's progress on the tree. Our secret animal is "mouse."

The game's first question is always the question at the tree's top: **IS IT FURRY?** Since a mouse is furry, we answer yes.

The game follows **IS IT FURRY?**'s yes branch to the **DOES IT BARK?** question. From **DOES IT BARK?**, the game can descend to either of the furry animals, **DOG** or **CAT**, but it can no longer reach the unfurry animal, **FROG**. By descending **IS IT FURRY?**'s yes branch, the game has narrowed down its possible guesses to furry animals.

The game now asks the question **DOES IT BARK?**. A mouse does not bark, so we answer no.

The game follows **DOES IT BARK?**'s no branch to the animal name **CAT**. When the game reaches an animal name, it guesses that animal. Here, of course, the game's guess is wrong. To improve its chances of guessing right the next time, the game learns the player's secret animal. Before we look at the learning process, let's examine how the game represents its knowledge as lists.

Making Trees with Logo Lists

Consider the very simple tree below. Here we represent it as a list.

```
[[IS IT FURRY?] CAT FROG]
```

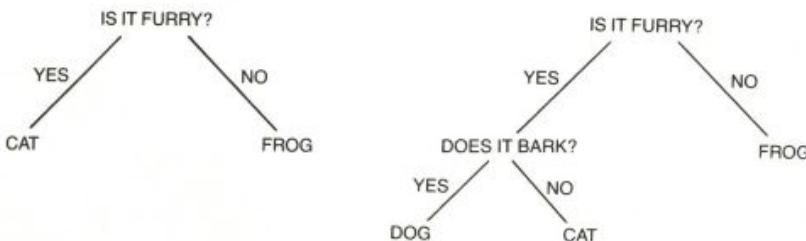
The tree is a list of three elements: a question, the question's yes branch, and the question's no branch. In this case, the question is **[IS IT FURRY?]**, its yes branch is **CAT**, and its no branch is **FROG**.

Both branches of the left tree below are animal names. Sometimes, as we've seen, a branch does not lead directly to an animal name but to another question that has its own two branches; it leads, that is, to another tree or *subtree*.

For example, look now at the slightly more complicated tree. Here it is represented as a list.

```
[[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
```

This slightly more complicated tree is also a list of three elements: a question, its yes branch, and its no branch. The question is **[IS IT FURRY?]**; its yes branch is the subtree **[[DOES IT BARK?] CAT DOG]**; its no branch is the animal name **FROG**.



WORDPLAY**Examining Trees**

We can write procedures that look at each of a tree's three parts. Sometimes we want to look at a subtree of a tree. Since a subtree is itself a tree, these procedures work on subtrees too. The procedures all expect a list of three elements as input.

```
TO QUESTION :TREE
OP FIRST :TREE
END

TO YES.BRANCH :TREE
OP FIRST BF :TREE
END

TO NO.BRANCH :TREE
OP FIRST BF BF :TREE
END
```

Here's an example of how they work.

```
?MAKE 'SAMPLE [[IS IT FURRY?]] [[DOES
IT BARK?]] DOG CAT] FROG]
?SHOW QUESTION :SAMPLE
[IS IT FURRY?]
?SHOW YES.BRANCH :SAMPLE
[[DOES IT BARK?]] DOG CAT]
?SHOW NO.BRANCH :SAMPLE
FROG
?SHOW NO.BRANCH YES.BRANCH :SAMPLE
CAT
?PR COUNT :SAMPLE
3
?PR COUNT YES.BRANCH :SAMPLE
3
```

Exploring the Game's Knowledge

The animal game's first task is to begin at the tree's top and follow branches to a guess. The procedure that does this is called EXPLORE.

```
TO EXPLORE :TREE
IF WORDP :TREE [FINISH.UP :TREE STOP]
IF YESP QUESTION :TREE
  [EXPLORE YES.BRANCH :TREE]
  [EXPLORE NO.BRANCH :TREE]
END
```

The first line, `IF WORDP :TREE [FINISH.UP :TREE STOP]`, means that if `:TREE` is a word—that is, an animal name—`EXPLORE` calls `FINISH.UP` with the animal name as input and `STOPs`. If `:TREE` is not a word, it's a subtree, so `EXPLORE` follows either its yes branch or its no branch.

Here are two paths EXPLORE can take if its first input is [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]:
 EXPLORE [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
 The player answers “yes” to IS IT FURRY?
 EXPLORE [[DOES IT BARK?] DOG CAT]
 The player answers “no” to DOES IT BARK?
 EXPLORE “CAT
 EXPLORE calls FINISH.UP with CAT as input and STOPS
 EXPLORE STOPS
 EXPLORE STOPS
 EXPLORE [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
 The player answers “no” to IS IT FURRY?
 EXPLORE “FROG
 EXPLORE calls FINISH.UP with FROG as input and STOPS
 EXPLORE STOPS

No matter what path EXPLORE takes, it always ends at an animal name, which it passes to FINISH.UP.

Guessing and Learning

Guessing

When EXPLORE calls FINISH.UP, the game is ready to guess that FINISH.UP’s input (:BEAST) is your animal. FINISH.UP calls GUESS to do the actual guessing. If GUESS outputs TRUE, the game’s guess is right, and BRAG is called. If GUESS outputs FALSE, the game’s guess is wrong, and LEARN is called.

```
TO FINISH.UP :BEAST
IF GUESS :BEAST [BRAG] [LEARN :BEAST [] []]
END

TO GUESS :BEAST
OP YESP (SE [IS IT] A.OR.AN :BEAST [])
END

TO BRAG
PR [I WIN!]
PR [I WIN!]
END
```

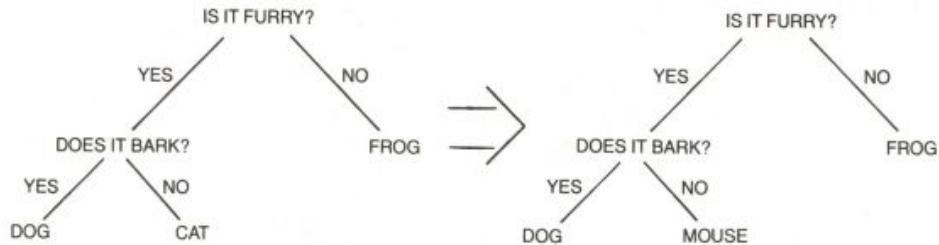
Learning

LEARN Adds to the Game’s Knowledge

How does the animal game get smarter? Let’s review the imaginary game we played earlier. Our secret animal was “mouse,” and the game guessed CAT. Obviously, if the game had guessed “mouse” instead of CAT, it would have won. We might want to change the game so that, from now on, it will guess MOUSE whenever it would have guessed CAT.

WORDPLAY

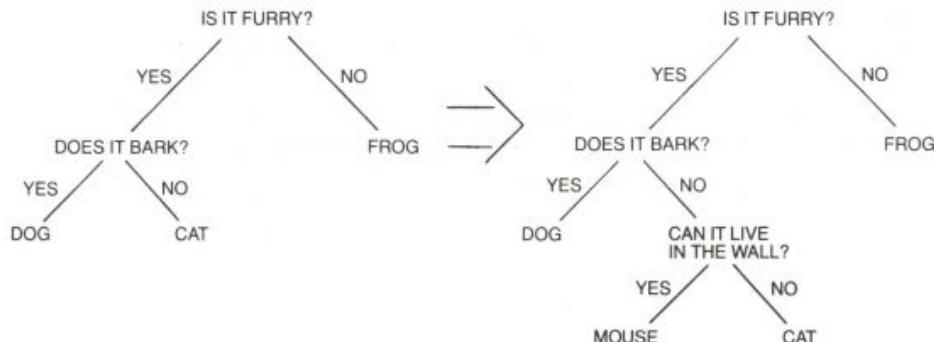
Look at the tree below. To make the game guess MOUSE instead of CAT, we could remove CAT (the wrong guess) from the tree and put MOUSE (the right guess) in its place.



Has the game learned? Not really. We've added a new animal to its knowledge, but we've also subtracted one.

If we want the game's knowledge to include both MOUSE and CAT, we must teach the game a new question, such as CAN IT LIVE IN THE WALL? We also teach it that if a player answers "yes" to the new question, it should guess MOUSE, and if a player answers "no," it should guess CAT.

The next tree shows the result of adding a new animal and a new question to the game's tree. Instead of replacing CAT with MOUSE, we replace CAT with a *new subtree*. The subtree—like all trees—consists of a question ([CAN IT LIVE IN THE WALL?]), a yes branch (MOUSE), and a no branch (CAT).



Building a New Subtree

GET.RIGHT.GUESS and GET.NEW.QUESTION get parts for a new subtree.

```

TO GET.RIGHT.GUESS
PR [I GIVE UP. WHAT IS IT?]
OP LAST RL
END

TO GET.NEW.QUESTION
PR [PLEASE TYPE IN A NEW QUESTION]
(PR [WHOSE ANSWER IS 'YES' FOR] :RIGHT.GUESS)
(PR [AND 'NO' FOR] :WRONG.GUESS)
OP RL
END
  
```

Adding to the Game's "Tree of Knowledge"

The game's entire "tree of knowledge" is stored in the global variable BIGTREE. For the game to get smarter, the new subtree must be added to :BIGTREE. LEARN and ALTER are the main procedures that do this. ALTER uses 3LIST, which outputs a list of its three inputs.

```

TO LEARN :WRONG.GUESS :RIGHT.GUESS :NEW.QUESTION
MAKE "RIGHT.GUESS GET.RIGHT.GUESS
MAKE "NEW.QUESTION GET.NEW.QUESTION
MAKE "BIGTREE ALTER
    :BIGTREE
    :NEW.QUESTION
    :RIGHT.GUESS
    :WRONG.GUESS
END

TO ALTER :TREE :NEW.QUESTION :RIGHT.GUESS :WRONG.GUESS
IF :TREE = :WRONG.GUESS
  [OP 3LIST :NEW.QUESTION :RIGHT.GUESS :WRONG.GUESS]
IF WORDP :TREE
  [OP :TREE]
OP 3LIST (QUESTION :TREE)
  (ALTER YES.BRANCH :TREE :NEW.QUESTION
   :RIGHT.GUESS :WRONG.GUESS)
  (ALTER NO.BRANCH :TREE :NEW.QUESTION
   :RIGHT.GUESS :WRONG.GUESS))
END

```

Let's recall how LEARN is called. EXPLORE climbs down to an animal name and passes the animal to FINISH.UP. FINISH.UP calls GUESS to guess the animal. If the guess is right, BRAG is called. If the guess is wrong, LEARN is called.

LEARN has three inputs. When it is called, :WRONG.GUESS is the animal the game guessed, and :RIGHT.GUESS and :NEW.QUESTION are empty lists.

LEARN calls GET.RIGHT.GUESS to get the player's secret animal and stores this animal in :RIGHT.GUESS. It calls GET.NEW.QUESTION to get the player's new yes-or-no question and stores it in :NEW.QUESTION. Then LEARN makes BIGTREE the output from ALTER.

ALTER's four inputs are the game's current "tree of knowledge" and the three parts for the new subtree. ALTER looks through the game's current tree, finds the animal the game guessed, and replaces this wrong guess with the new subtree. It then outputs a new, enlarged "tree of knowledge" to LEARN.

Here's a sample set of inputs to ALTER.

```

:TREE      [[IS IT FURRY?]
              DOES IT BARK?] DOG CAT] FROG]

:NEW.QUESTION  [CAN IT LIVE IN THE WALL?]

:RIGHT.GUESS   MOUSE

:WRONG.GUESS   CAT

```

WORDPLAY

The following display traces how ALTER works with the preceding inputs. The only input traced is :TREE, since the other inputs are the same each time ALTER is called recursively.

```

ALTER [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
    ALTER [[DOES IT BARK?] DOG CAT]
        ALTER "DOG"
        ALTER outputs "DOG"
        ALTER "CAT"
        ALTER outputs [[CAN IT LIVE IN THE WALL?] MOUSE CAT]
        ALTER outputs [[DOES IT BARK?] DOG [[CAN
                        IT LIVE...?] MOUSE CAT]]
        ALTER "FROG"
        ALTER outputs "FROG"
    ALTER outputs [[IS IT FURRY?] [[DOES IT BARK?] DOG [[CAN
                        IT LIVE...?] MOUSE CAT]] FROG]
```

Starting the Game

You begin each session with the animal game by typing ANIMALGAME. This procedure checks whether the game knows anything yet. If no variable named :BIGTREE exists in your workspace, the game knows no questions or animals, so MAKETREE creates a “tree of knowledge” and puts it in :BIGTREE.

PLAY prompts you to think of a secret animal; calls EXPLORE with :BIGTREE as input; and, when the game is over, asks if you’d like to play again.

```

TO ANIMALGAME
IF NOT NAMEP "BIGTREE [MAKETREE]
PLAY
END

TO MAKETREE
MAKE "BIGTREE [[IS IT FURRY?] CAT FROG]
END

TO PLAY
PR []
PR [PICK AN ANIMAL, ANY ANIMAL]
EXPLORE :BIGTREE
IF YESP [DO YOU WANT TO PLAY AGAIN?] [PLAY]
END
```

Remember that every time you play the animal game and it loses, :BIGTREE gets “bigger.” And the bigger the game’s “tree of knowledge,” the smarter the game appears to be.

Since :BIGTREE is a global variable, it remains in your workspace after you’ve finished a session with the animal game (that is, after you answer “no” to the game’s question, DO YOU WANT TO PLAY AGAIN?). If you save this workspace, :BIGTREE will be saved as well. At another session, you could make the game’s knowledge even bigger.

If you ever want to *erase* the game's knowledge, stop playing the game and call MAKETREE. MAKETREE causes the game to forget everything it has ever learned.

Other Procedures Used by the Game

All these procedures were mentioned earlier but we did not look at how they work.

The input to A.OR.AN should be an animal name. Its output is the animal name preceded by an appropriate article—either “a” or “an.”

```
TO A.OR.AN :ANIMAL
IF MEMBERP FIRST :ANIMAL [A E I O U] [OP SE "AN :ANIMAL"]
OP SE "A :ANIMAL
END
```

YESP and COMPLAIN get a yes-or-no answer to a question. The question is the input to YESP.

```
TO YESP :QUESTION
PR :QUESTION
MAKE "ANS RL
IF NOT OR (EQUALP :ANS [YES]) (EQUALP :ANS [NO])
  [COMPLAIN OP YESP :QUESTION]
OP EQUALP :ANS [YES]
END
```

```
TO COMPLAIN
PR [PLEASE ANSWER YES OR NO]
END
```

Here's an example.

```
?PR YESP [IS IT FURRY?]
IS IT FURRY?
SORT OF
PLEASE ANSWER YES OR NO
IS IT FURRY?
YES
TRUE
?
```

3LIST outputs a list of its three inputs.

```
TO 3LIST :ONE :TWO :THREE
OP FPUT :ONE FPUT :TWO FPUT :THREE []
END
```

SUGGESTIONS

You can play this game with exotic animal names such as armadillo, gnu, gazelle, iguana. You could even use fantastic animals like centaurs or pushme-pullyous. Some people say that it's most fun to play it with the names of your friends!

PROGRAM LISTING

```

TO QUESTION :TREE
OP FIRST :TREE
END

TO YES.BRANCH :TREE
OP FIRST BF :TREE
END

TO NO.BRANCH :TREE
OP FIRST BF BF :TREE
END

TO EXPLORE :TREE
IF WORDP :TREE [FINISH.UP :TREE STOP]
IF YESP QUESTION :TREE [EXPLORE ▶
    YES.BRANCH :TREE] [EXPLORE ▶
    NO.BRANCH :TREE]
END

TO FINISH.UP :BEAST
IF GUESS :BEAST [BRAG] [LEARN :BEAST ▶
    [] []]
END

TO GUESS :BEAST
OP YESP (SE [IS IT] A.OR.AN :BEAST ▶
    [?])
END

TO BRAG
PR [I WIN!]
PR [I WIN!]
END

TO GET.RIGHT.GUESS
PR [I GIVE UP. WHAT IS IT?]
OP LAST RL
END

TO GET.NEW.QUESTION
PR [PLEASE TYPE IN A NEW QUESTION]
(PR [WHOSE ANSWER IS 'YES' FOR] ▶
    :RIGHT.GUESS)
(PR [AND 'NO' FOR] :WRONG.GUESS)
OP RL
END

TO LEARN :WRONG.GUESS :RIGHT.GUESS ▶
    :NEW.QUESTION
MAKE "RIGHT.GUESS GET.RIGHT.GUESS
MAKE "NEW.QUESTION GET.NEW.QUESTION
MAKE "BIGTREE ALTER :BIGTREE ▶
    :NEW.QUESTION :RIGHT.GUESS ▶
    :WRONG.GUESS
END

TO ALTER :TREE :NEW.QUESTION ▶
    :RIGHT.GUESS :WRONG.GUESS
IF :TREE = :WRONG.GUESS [OP 3LIST ▶
    :NEW.QUESTION :RIGHT.GUESS ▶
    :WRONG.GUESS]
IF WORDP :TREE [OP :TREE]
OP 3LIST (QUESTION :TREE) (ALTER ▶
    YES.BRANCH :TREE :NEW.QUESTION ▶
    :RIGHT.GUESS :WRONG.GUESS) (ALTER ▶
    NO.BRANCH :TREE :NEW.QUESTION ▶
    :RIGHT.GUESS :WRONG.GUESS))
END

TO ANIMALGAME
IF NOT NAMEP "BIGTREE [MAKETREE]
PLAY
END

TO MAKETREE
MAKE "BIGTREE [[IS IT FURRY?]] CAT ▶
    FROG]
END

TO PLAY
PR []
PR [PICK AN ANIMAL, ANY ANIMAL]
EXPLORE :BIGTREE
IF YESP [DO YOU WANT TO PLAY AGAIN?] ▶
    [PLAY]
END

TO A.OR.AN :ANIMAL
IF MEMBERP FIRST :ANIMAL [A E I O U] ▶
    [OP SE "AN :ANIMAL]
OP SE "A :ANIMAL
END

```

```

TO YESP :QUESTION
PR :QUESTION
MAKE "ANS RL
IF NOT OR (EQUALP :ANS [YES]) (EQUALP ▶
    :ANS [NO]) [COMPLAIN OP YESP ▶
    :QUESTION]
OP EQUALP :ANS [YES]
END

TO COMPLAIN
PR [PLEASE ANSWER YES OR NO]
END

TO 3LIST :ONE :TWO :THREE
OP FPUT :ONE FPUT :TWO FPUT :THREE []
END

MAKE "BIGTREE [[IS IT FURRY?] CAT ▶
    FROG]

```

Dictionary

The idea for this project came about while I was hiking with some friends. During our climb up the mountain, we tried to stump each other by asking the meaning of unusual words. I began to think about developing a dictionary project using Logo.

I wanted to be able to do several things with my dictionary:

- Add a new word and its definition.
- Print the definition of a word.
- Remove a word and its definition.
- Print the entire dictionary.

The Dictionary

My first task was to decide how to store the words. I decided that the dictionary would be a list of entries. Each entry would be a list composed of a word and its definition. Here are two examples.

[ICE [FROZEN WATER]]

or

[HAT [COVERING FOR HEAD]]

I named the dictionary ENTRY.LIST. Here's how I created it.

```

MAKE "ENTRY.LIST [[EGREGIOUS [CONSPICUOUSLY BAD]]
    [PROSY [COMMONPLACE]]
    [AUTO-DA-FE [BURNING OF A HERETIC]]]

```

Using the Dictionary

When you type DICTIONARY, the following is printed on your screen:

WORDPLAY

WELCOME TO THE DICTIONARY.

HERE ARE THE COMMANDS:

```
TYPE A - TO ADD NEW ENTRY
TYPE D - TO PRINT DEFINITION OF WORD
TYPE P - TO PRINT DICTIONARY
TYPE Q - TO QUIT
TYPE R - TO REMOVE ENTRY
TYPE ? - TO PRINT COMMANDS
```

TYPE COMMAND.

>

DICTIONARY calls INIT, which checks to see if you already have a dictionary. If you do not, INIT creates one.

```
TO DICTIONARY
INIT
PR [WELCOME TO THE DICTIONARY.]
PR []
PR [HERE ARE THE COMMANDS:]
DO .CHOICE "?"
END

TO INIT
CT TS
IF NOT NAMEP "ENTRY.LIST" [MAKE "ENTRY.LIST"
  [[EGREGIOUS [CONSPICUOUSLY BAD]]
   [PROSY [COMMONPLACE]]
   [[AUTO-DA-FE] [BURNING OF A HERETIC]]]]
END
```

DO .CHOICE has the job of figuring out whether the character you type matches one of the expected commands. If there is no match or if you type ?, DO .CHOICE prints the list of possible choices.

```
TO DO .CHOICE :LTR
PR []
IF EQUALP "A :LTR" [ADD.ENTRY]
IF EQUALP "D :LTR" [PRINT.DEFINITION]
IF EQUALP "P :LTR" [PRINT.DICTIONARY]
IF EQUALP "Q :LTR" [STOP]
IF EQUALP "R :LTR" [REMOVE.ENTRY]
IF NOT MEMBERP :LTR [A D P Q R] [PRINT.CHOICES]
PR []
PR [TYPE COMMAND.]
TYPE ">"
MAKE "LTR RC
PR :LTR
DO .CHOICE :LTR
END
```

```

TO PRINT.CHOICES
PR [TYPE A - TO ADD NEW ENTRY]
PR [TYPE D - TO PRINT DEFINITION OF WORD]
PR [TYPE P - TO PRINT DICTIONARY]
PR [TYPE Q - TO QUIT]
PR [TYPE R - TO REMOVE ENTRY]
PR [TYPE ? - TO PRINT COMMANDS]
END

```

Adding a New Word and Definition

To add a word, you type A while running DICTIONARY. Here's an example of what happens.

```

TYPE NEW WORD.
FLUMP
TYPE DEFINITION OF NEW WORD.
DROP OR MOVE HEAVILY

TYPE COMMAND
>

```

If you try to add a word that is already in the dictionary, this happens:

```

TYPE NEW WORD
EGREGIOUS
EGREGIOUS IS ALREADY IN DICTIONARY.

```

ADD.ENTRY is the procedure that lets you add a new entry to the dictionary.

```

TO ADD.ENTRY
PR [TYPE NEW WORD .]
ADD.ENTRY1 FIRST RL
END

```

ADD.ENTRY1 calls GET.ENTRY to see if the word you want to add is already in the dictionary. If the word is not in the dictionary, then the word and its definition become a new entry.

```

TO ADD.ENTRY1 :WRD
IF NOT EMPTYP GET.ENTRY :WRD :ENTRY.LIST
  [PR SE :WRD [IS ALREADY IN DICTIONARY.] STOP]
PR [TYPE DEFINITION OF NEW WORD.]
MAKE.ENTRY LIST :WRD RL
END

```

GET.ENTRY has the task of finding an entry in the dictionary. It does this by attempting to match an input word with the first word in each entry.

```

TO GET.ENTRY :WRD :LST
IF EMPTYP :LST [OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP FIRST :LST]
OP GET.ENTRY :WRD BF :LST
END

```

WORDPLAY

MAKE.ENTRY adds a new entry to the dictionary.

```
TO MAKE.ENTRY :NEW.ENTRY
MAKE "ENTRY.LIST FPUT :NEW.ENTRY :ENTRY.LIST
END
```

Printing the Definition of a Word

This is what happens when you type D.

```
TYPE WORD WHOSE DEFINITION
YOU WANT PRINTED.
EGREGIOUS
[CONSPICUOUSLY BAD]
```

PRINT.DEFINITION calls PRINT.DEF1 to print out the definition of a word.

```
TO PRINT.DEFINITION
PR [TYPE WORD WHOSE DEFINITION]
PR [YOU WANT PRINTED.]
PRINT.DEF1 FIRST RL
END
```

```
TO PRINT.DEF1 :WRD
PRINT.DEF2 :WRD GET.ENTRY :WRD :ENTRY.LIST
END
```

PRINT.DEF1 then calls PRINT.DEF2 with the word to be defined and its entry in the dictionary. If the entry is in the dictionary, PRINT.DEF2 prints the definition.

```
TO PRINT.DEF2 :WRD :LST
IF EMPTYP :LST [PR SE :WRD [IS NOT IN DICTIONARY.] STOP]
PR BF :LST
END
```

Removing an Entry from the Dictionary

To remove an entry, you type R. Here is an example.

```
TYPE WORD
YOU WANT TO REMOVE.
FLUMP
```

REMOVE.ENTRY uses REMOVE to output a dictionary, minus the unwanted entry.

```
TO REMOVE.ENTRY
PR [TYPE WORD]
PR [YOU WANT TO REMOVE.]
MAKE "ENTRY.LIST REMOVE FIRST RL :ENTRY.LIST
END
```

```
TO REMOVE :WRD :LST
IF EMPTYP :LST [PR SE :WRD [IS NOT IN DICTIONARY.] OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP BF :LST]
OP PPUT FIRST :LST REMOVE :WRD BF :LST
END
```

Printing the Dictionary

Here's what happens when you type P. I've added some words that I thought were interesting to the dictionary.

```
IMPUSSANT
[WEAK; IMPOTENT]

ACCRETIVE
[ADDING IN GROWTH]

DENTILOQUY
[THE ACT OR HABIT OF SPEAKING WITH TEETH CLOSED]

CENOSITY
[FILTHINESS; SQUALOR]

DELIQUESCE
[TO MELT AWAY]

FETOR
[STRONG OFFENSIVE SMELL]

BRUMAL
[INDICATIVE OF OR OCCURRING IN WINTER]

**TYPE ANY CHARACTER
TO SEE MORE**
```

Note: At this point you press any key to see the next seven (or remaining) entries.

```
EGREGIOUS
[CONSPICUOUSLY BAD]
```

```
PROSY
[COMMONPLACE]
```

```
AUTO - DA - FE
[BURNING OF A HERETIC]
```

The procedures PRINT.DICTIONARY, FORMAT, and PRINT.ENTRY work together to print ENTRY.LIST in an easy-to-read format. There is room on the screen for seven entries. FORMAT counts the number of entries. When the screen is full, FORMAT pauses and waits until you type any character before printing the next seven or remaining entries.

WORDPLAY

```

TO PRINT.DICTIONARY
FORMAT @ :ENTRY.LIST
END

TO FORMAT :CTR :LST
IF EMPTYP :LST [STOP]
IF :CTR = 7
  [PR [**TYPE ANY CHARACTER]
   PR [TO SEE MORE**] PR RC]
PRINT.ENTRY FIRST :LST
FORMAT IF :CTR = 7 [1] [:CTR + 1] BF :LST
END

TO PRINT.ENTRY :ENTRY
PR FIRST :ENTRY
PR BF :ENTRY
PR []
END

```

PROGRAM LISTING

```

TO DICTIONARY
INIT
PR [WELCOME TO THE DICTIONARY.]
PR []
PR [HERE ARE THE COMMANDS.]
DO.CHOICE "?"
END

TO INIT
CT TS
IF NOT NAMEP "ENTRY.LIST [MAKE ►
  "ENTRY.LIST [[EGREGIOUS ►
  [CONSPICUOUSLY BAD]] [PROSY
  [COMMONPLACE]] [[AUTO-DA-FE] ►
  [BURNING OF A HERETIC]]]
END

TO DO.CHOICE :LTR
PR []
IF EQUALP "A :LTR [ADD.ENTRY]
IF EQUALP "D :LTR [PRINT.DEFINITION]
IF EQUALP "P :LTR [PRINT.DICTIONARY]
IF EQUALP "Q :LTR [STOP]
IF EQUALP "R :LTR [REMOVE.ENTRY]
IF NOT MEMBERP :LTR [A D P Q R] ►
  [PRINT.CHOICES]
PR []
PR [TYPE COMMAND.]
TYPE ">"
MAKE "LTR RC
PR :LTR
DO.CHOICE :LTR
END

TO PRINT.CHOICES
PR [TYPE A - TO ADD NEW ENTRY]
PR [TYPE D - TO PRINT DEFINITION OF ▶
  WORD]
PR [TYPE P - TO PRINT DICTIONARY]
PR [TYPE Q - TO QUIT]
PR [TYPE R - TO REMOVE ENTRY]
PR [TYPE ? - TO PRINT COMMANDS]
END

TO ADD.ENTRY
PR [TYPE NEW WORD.]
ADD.ENTRY1 FIRST RL
END

TO ADD.ENTRY1 :WRD
IF NOT EMPTYP GET.ENTRY :WRD ►
  :ENTRY.LIST [PR SE :WRD [IS ▶
  ALREADY IN DICTIONARY.] STOP]
PR [TYPE DEFINITION OF NEW WORD.]
MAKE.ENTRY LIST :WRD RL
END

TO GET.ENTRY :WRD :LST
IF EMPTYP :LST [OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP ►
  FIRST :LST]
OP GET.ENTRY :WRD BF :LST
END

```

```

TO MAKE.ENTRY :NEW.ENTRY
MAKE "ENTRY.LIST FPUT :NEW.ENTRY ▶
:ENTRY.LIST
END

TO PRINT.DEFINITION
PR [TYPE WORD WHOSE DEFINITION]
PR [YOU WANT PRINTED.]
PRINT.DEF1 FIRST RL
END

TO PRINT.DEF1 :WRD
PRINT.DEF2 :WRD GET.ENTRY :WRD ▶
:ENTRY.LIST
END

TO PRINT.DEF2 :WRD :LST
IF EMPTYP :LST [PR SE :WRD [IS NOT IN ▶
DICTIONARY.] STOP]
PR BF :LST
END

TO REMOVE.ENTRY
PR [TYPE WORD]
PR [YOU WANT TO REMOVE.]
MAKE "ENTRY.LIST REMOVE FIRST RL ▶
:ENTRY.LIST
END

TO REMOVE :WRD :LST
IF EMPTYP :LST [PR SE :WRD [IS NOT IN ▶
DICTIONARY.] OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP BF ▶
:LST]
OP FPUT FIRST :LST REMOVE :WRD BF :LST
END

TO PRINT.DICTIONARY
FORMAT @ :ENTRY.LIST
END

TO FORMAT :CTR :LST
IF EMPTYP :LST [STOP]
IF :CTR = 7 [PR [**TYPE ANY CHARACTER] ▶
PR [TO SEE MORE**] PR RC]
PRINT.ENTRY FIRST :LST
FORMAT IF :CTR = 7 [1] [:CTR + 1] BF ▶
:LST
END

TO PRINT.ENTRY :ENTRY
PR FIRST :ENTRY
PR BF :ENTRY
PR []
END

MAKE "ENTRY.LIST [[IMPUSSANT [WEAK; ▶
IMPOTENT]] [ACCRETIVE [ADDING IN ▶
GROWTH]] [DENTILOQUY [THE ACT OR ▶
HABIT OF SPEAKING WITH TEETH ▶
CLOSED]] [CENOSITY [FILTHINESS; ▶
SQUALOR]] [DELICHESCE [TO MELT ▶
AWAY]] [FETOR [STRONG OFFENSIVE ▶
SMELL]] [BRUMAL [INDICATIVE OF OR ▶
OCCURRING IN WINTER]] [EGREARIOUS ▶
[CONSPICUOUSLY BAD]] [PROSY ▶
[COMMONPLACE]] [[AUTO - DA - FE] ▶
[BURNING OF A HERETIC]]]
```

Hangman

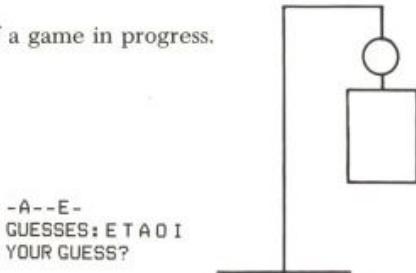
HANGMAN is based on the popular two-person pencil-and-paper game in which one player thinks up a secret word and the other player tries to discover what the word is by guessing what letters are in the word. A gallows is drawn, and for each incorrect guess, part of a stick figure is added to the drawing. The player who is guessing wins the game by guessing the entire word before the stick figure is completed.

By Brian Harvey.

WORDPLAY

In this version, the program chooses the secret word and you do the guessing. At each turn, you can guess either a single letter or the entire word.

Here is a picture of a game in progress.



The secret word is shown as -A--E-. This means that it has six letters, two of which have been guessed. You have made five guesses. A and E were correct. The others, T, O, and I, were wrong. Because of these three wrong guesses, the program has drawn the head, neck, and body of the person being hanged. If you make more wrong guesses, the program will draw the person's arms and legs.

I like this program because it combines text processing with graphics. The top-level procedure divides the program into two parts: setting up and playing the game.

```
TO HANGMAN
SETUP
PLAY
END
```

Setting Up

SETUP has two jobs; it gives initial values to certain variables, including the secret word, and it draws the gallows.

```
TO SETUP
MAKE "MYWORD PICKWORD
MAKE "GUESSES []
MAKE "WON "FALSE
MAKE "SPACES "
REPEAT 18 [MAKE "SPACES WORD :SPACES CHAR 32]
MAKE "GOTTEN []
MAKE "TRIES 7
CT
GALLOWS
END
```

SETUP uses two main subprocedures, one to pick the secret word and one to draw the gallows. PICKWORD outputs the secret word, which SETUP remembers in the global variable MYWORD. To choose the word from a list of possible words, PICKWORD uses the procedures PICK and ITEM, which appear as examples in the *Atari Logo Reference Manual*.

```
TO PICKWORD
OP PICK [POTSTICKER COMPUTER IRAQ GAZEBO THRUSH STYLE FOILED
SWARM ZEBRA AWFUL WILY YELLOW BARKED STOIC]
END
```

GALLWS positions a turtle for drawing the gallows, sets the pen down, and uses GALL1 to do the actual drawing. The reason to make GALL1 a subprocedure is that it will be used again, with the eraser down, to erase the gallows if you win by guessing the word.

```

TO GALLWS
  TELL [Ø 1 2 3]
  CS HT
  TELL Ø
  PU SETPOS [-4Ø -6Ø]
  RT 9Ø
  PD
  GALL1
  END

TO GALL1
  FD 8Ø
  BK 4Ø
  LT 9Ø
  FD 17Ø
  RT 9Ø
  FD 6Ø
  RT 9Ø
  FD 2Ø
  END

```



Variables Created by SETUP

The variable MYWORD is one of several that are used throughout the hangman program to keep track of the progress of the game. For example, the program must remember what letters have been guessed and how many wrong guesses are allowed before you lose. Several of these variables are given their initial values by SETUP.

MYWORD	The secret word.
GUESSES	A list of the letters you have guessed.
WON	TRUE if you win by guessing the word or the last missing letter in it.
SPACES	A word of eighteen spaces, which is typed to erase messages from the program in the text part of the screen.
GOTTEN	The number of letters in the secret word that you have guessed correctly. (If a letter occurs more than once in the secret word, the number of letters guessed correctly may be more than the number of correct guesses you have made, because one correct guess may reveal several letters in the word.)
TRIES	The number of incorrect guesses remaining before you lose.

Playing the Game

The central part of the hangman program is the procedure PLAY and its subprocedure GETGUESS, which is called each time you make a guess.

WORDPLAY

```

TO PLAY
IF :TRIES=0 [LOSE STOP]
GETGUESS
IF :WON [SETCURSOR [0 23] STOP]
PLAY
END

TO GETGUESS
DISPLAY
MAKE "GUESS FIRST RL
CLEARMESSAGE
IF (COUNT :GUESS) > 1 [TESTWORD STOP]
IF MEMBERP :GUESS :GUESSES [ALREADY GETGUESS STOP]
MAKE "GUESSES SE :GUESSES :GUESS
IF MEMP :GUESS :MYWORD [FIXGOT :GUESS :MYWORD] [BADTRY]
IF EQUALP :GOTTEN COUNT :MYWORD [WIN]
END

```

PLAY calls GETGUESS repeatedly, checking between times to see if you've won (the variable WON made TRUE) or lost (no more TRIES left). GETGUESS uses several subprocedures to display the current state of the game, read a guess from the keyboard, and test the guess. A guess can be either a single letter or the entire word. These cases are distinguished by checking the COUNT of the guess; if it's more than one letter, the procedure TESTWORD is used to compare the guess to the secret word. Otherwise, the program checks if you have already guessed the letter; if not, it checks to see if the guessed letter is actually in the word. If the letter is in the word, FIXGOT is called to update the number of letters correctly guessed. If not, BADTRY draws another piece of the body under the gallows.

Keeping Track of the Text Screen

The text part of the screen in the middle of a game might look like this:

-A-E--	YOU GUESSED THAT!
GUESSES: E T A	
YOUR GUESS? _	

In the top left corner is the display of the secret word, with some letters already guessed and the others indicated by hyphens. In the top right corner is the *message area*. You have just repeated a guess already made, and the program has complained about it. The next line shows the list of letters already guessed. The third line invites you to make another guess, and the cursor is positioned for reading that guess.

The message area is maintained by the procedure SAY. Two subprocedures of GETGUESS show simple examples of how SAY is used:

```

TO SAY :STUFF
SETCURSOR [19 20]
TYPE :STUFF
END

```

```

TO ALREADY
SAY [YOU GUessed THAT!]
END

TO CLEARMESSAGE
SAY :SPACES
END

```

(The underlining in this listing represents inverse-video characters on the screen.) The CLEARMESSAGE procedure types spaces into the message area, erasing any leftover messages. The procedure ALREADY is called by GETGUESS if you repeat a previous guess.

The rest of the text screen, apart from the message area, is maintained by the DISPLAY procedure:

```

TO DISPLAY
SETCURSOR [0 20]
DISWORD :MYWORD
SETCURSOR [0 21]
TYPE "GUESSES:"
SETCURSOR [9 21]
TYPE :GUESSES
SETCURSOR [0 22]
TYPE [YOUR GUESS?]
SETCURSOR [12 22]
END

```

For each letter of the secret word, DISWORD looks in the list of letters already guessed. If this letter has been guessed, DISWORD types it. If not, DISWORD types a hyphen.

```

TO DISWORD :WORD
IF EMPTYP :WORD [STOP]
IF MEMBERP FIRST :WORD :GUESSES [TYPE FIRST :WORD] [TYPE "-"]
DISWORD BF :WORD
END

```

When You Guess a Letter

When you guess a letter (that hasn't been guessed already), GETGUESS calls either FIXGOT or BADTRY depending on whether the guess is correct or incorrect. To test the correctness of the guess, GETGUESS uses MEMP, which is like the primitive MEMBERP except that it checks whether a letter is an element of a word, instead of whether a word is an element of a list.

```

TO MEMP :LETTER :WORD
IF EMPTYP :WORD [OP "FALSE"]
IF EQUALP :LETTER FIRST :WORD [OP "TRUE"]
OP MEMP :LETTER BF :WORD
END

```

(Actually, MEMP would work equally well testing for membership in a list, like MEMBERP, but we need it only to check for membership in a word.)

If the guess is correct, the task of FIXGOT is to calculate a new value

WORDPLAY

for the variable GOTEN, which counts the number of correctly guessed letters in the secret word. We can't just add 1 to GOTEN, because the letter you guessed may appear more than once in the secret word. For example, if the secret word is "thrush" and you guess H, FIXGOT must add 2 to GOTEN. So FIXGOT must examine each letter of the secret word.

```
TO FIXGOT :GUESS :WORD
IF EMPTYP :WORD [STOP]
IF EQUALP :GUESS FIRST :WORD [MAKE "GOTTEN" :GOTTEN+1]
FIXGOT :GUESS BF :WORD
END
```

Note that FIXGOT does not actually display the newly guessed letters on the screen. This will be done by DISPLAY the next time through GETGUESS.

What Happens on a Wrong Guess

If the guess is incorrect, BADTRY is called to count down the number of turns until you lose and to draw part of the body under the gallows:

```
TO BADTRY
RUN SE WORD "DRAW" :TRIES []
MAKE "TRIES" :TRIES-1
END
```

The RUN command is used to select a subprocedure to draw the appropriate part of the body, based on the number of tries remaining. For example, the variable TRIES is initially 7, and the procedure DRAW7 draws a head. DRAW6 draws the neck, DRAW5 the torso, DRAW4 and DRAW3 the arms, and DRAW2 and DRAW1 the legs:

```
TO DRAW7
PU
SETPOS [60 90]
SETH 105
PD
REPEAT 12 [FD 6 RT 30]
RT 75
END
```



```
TO DRAW6
PU
SETPOS [60 66]
SETH 180
PD
FD 10
END
```



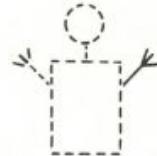
```
TO DRAW5
PU
SETPOS [40 56]
SETH 90
PD
REPEAT 2 [FD 40 RT 90 FD 60 RT 90]
END
```



```
TO DRAW4
PU
SETPOS [40 40]
SETH -45
PD
ARM
END
```

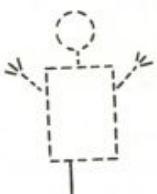


```
TO DRAW3
PU
SETPOS [80 40]
SETH 45
PD
ARM
END
```



```
TO ARM
FD 30
BK 14
LT 25
FD 14
BK 14
RT 50
FD 14
END
```

```
TO DRAW2
PU
SETPOS [52 -4]
SETH 180
PD
FD 30
RT 90
FD 8
END
```



```
TO DRAW1
PU
SETPOS [68 -4]
SETH 180
PD
FD 30
LT 90
FD 8
END
```



None of the procedures DRAW1, DRAW2, and so forth, assume that the turtle is at any particular position. This is because if you win, the program will erase the gallows and then finish drawing in the body, so any of these

WORDPLAY

procedures might be called with the turtle at the end of the gallows, rather than at the end of the previous body part.

When You Guess a Word

We have looked at the procedures that deal with a guess of a single letter. You may also guess the entire word; if so, the GETGUESS procedure calls TESTWORD.

```
TO TESTWORD
IF EQUALP :GUESS :MYWORD [WIN] [SAY [NOPE!!] BADTRY]
END
```

An incorrect guess of the entire word is handled by BADTRY, just like an incorrect guess of a letter. But if you guess the entire word correctly, there is no need to call FIXGOT. We can simply call WIN, because you have won the game.

When You Lose the Game

We have now looked at all the procedures involved in playing the game, up to the point of winning or losing. The case of losing is easier to understand. You lose by running out of tries. This means that the entire body has already been drawn.

```
TO LOSE
SAY [YOU LOSE!! SORRY.]
SETCURSOR [0 20]
TYPE :MYWORD
SETCURSOR [0 23]
EYES
FROWN
END

TO EYES
PU
SETPOS [52 82]
SETH 90
PD
FD 4
PU
FD 6
PD
FD 4
END

TO FROWN
PU
SETPOS [66 72]
SETH -9
MOUTH
END
```



```

TO MOUTH
PD
LT 18
REPEAT 8 [FD 2 LT 18]
END

```

The program tells you what the secret word was, moves the cursor down to the last screen line, and fills in the already-drawn head with a frowning face. When the program stops, Logo will print its prompt on the last line without obscuring what is written in the text area. (LOSE is called only by PLAY, which then stops, returning to HANGMAN, which stops. So when LOSE stops, the entire program is done.)

When You Win the Game

What if you win? In this case, the body is not yet entirely drawn. We want to erase the gallows, finish drawing the body, notify the winner, and stop the program.

```

TO WIN
SETCURSOR [0 20]
DISWORD :MYWORD
SAY [YOU WIN!!!]
MAKE "WON "TRUE
UNGALL
FINISH :TRIES
EYES
SMILE
END

TO UNGALL
PU
SETPOS [-40 -60]
SETH 90
PE
GALL1
END

TO FINISH :NUM
IF :NUM=0 [STOP]
RUN SE WORD "DRAW :NUM []
FINISH :NUM-1
END

TO SMILE
PU
SETPOS [54 76]
SETH 171
MOUTH
END

```

UNGALL is like GALLWS except that it draws the gallows in PE (penerase). FINISH calls each of the yet-undone drawing procedures (DRAW1, etc.) to



WORDPLAY

finish drawing the body. And SMILE draws the same mouth as FROWN, but right-side up.

Unlike LOSE, the WIN procedure can be called from two places in the program: TESTWORD and GETGUESS. Because these places are deeper in the chain of subprocedures, we must set the variable WON so that the PLAY procedure can test it, to know when to stop the game program.

Utilities

To complete the listing of procedures used in this project, here are the utility procedures PICK and ITEM:

```
TO PICK :LIST
OP ITEM (1 + RANDOM COUNT :LIST) :LIST
END

TO ITEM :N :LIST
IF :N=1 [OP FIRST :LIST]
OP ITEM :N-1 BF :LIST
END
```

PROGRAM LISTING

TO HANGMAN	TO GALL1
SETUP	FD 80
PLAY	BK 40
END	LT 90
TO SETUP	FD 170
MAKE "MYWORD PICKWORD	RT 90
MAKE "GUESSES []	FD 60
MAKE "WON "FALSE	RT 90
MAKE "SPACES "	FD 20
REPEAT 18 [MAKE "SPACES WORD :SPACES ▶	END
CHAR 32]	TO PLAY
MAKE "GOTTEN 0	IF :TRIES=0 [LOSE STOP]
MAKE "TRIES 7	GETGUESS
CT	IF :WON [SETCURSOR [0 23] STOP]
GALLOWS	PLAY
END	END
TO GALLOWS	TO GETGUESS
TELL [0 1 2 3]	DISPLAY
CS HT	MAKE "GUESS FIRST RL
TELL 0	CLEARMESSAGE
PU SETPOS [-40 -60]	IF (COUNT :GUESS) > 1 [TESTWORD STOP]
RT 90	IF MEMBERP :GUESS :GUESSES [ALREADY ▶
PD	GETGUESS STOP]
GALL1	MAKE "GUESSES SE :GUESSES :GUESS
END	IF MEMP :GUESS :MYWORD [FIXGOT :GUESS ▶
	:MYWORD] [BADTRY]
	IF EQUALP :GOTTEN COUNT :MYWORD [WIN]
	END

```

TO PICKWORD
OP PICK [POTSTICKER COMPUTER IRAQ ▶
    GAZEBO THRUSH STYLE FOILED SWARM ▶
    ZEBRA AWFUL WILY YELLOW BARKED ▶
    STOIC]
END

TO SAY :STUFF
SETCURSOR [19 20]
TYPE :STUFF
END

TO ALREADY
SAY [YOU_GUESSED THAT!]
END

TO CLEARMESSAGE
SAY :SPACES
END

TO DISPLAY
SETCURSOR [0 20]
DISWORD :MYWORD
SETCURSOR [0 21]
TYPE "GUESSES:"
SETCURSOR [9 21]
TYPE :GUESSES
SETCURSOR [0 22]
TYPE [YOUR GUESS?]
SETCURSOR [12 22]
END

TO DISWORD :WORD
IF EMPTYYP :WORD [STOP]
IF MEMBERTP FIRST :WORD :GUESSES [TYPE ▶
    FIRST :WORD] [TYPE "-"]
DISWORD BF :WORD
END

TO MEMP :LETTER :WORD
IF EMPTYYP :WORD [OP "FALSE"]
IF EQUALYP :LETTER FIRST :WORD [OP ▶
    "TRUE"]
OP MEMP :LETTER BF :WORD
END

TO FIXGOT :GUESS :WORD
IF EMPTYYP :WORD [STOP]
IF EQUALYP :GUESS FIRST :WORD [MAKE ▶
    "GOTTEN :GOTTEN+1"]
FIXGOT :GUESS BF :WORD
END

TO BADTRY
RUN SE WORD "DRAW :TRIES []"
END

MAKE "TRIES :TRIES-1"
END

TO DRAW7
PU
SETPOS [60 90]
SETH 105
PD
REPEAT 12 [FD 6 RT 30]
RT 75
END

TO DRAW6
PU
SETPOS [60 66]
SETH 180
PD
FD 10
END

TO DRAW5
PU
SETPOS [40 56]
SETH 90
PD
REPEAT 2 [FD 40 RT 90 FD 60 RT 90]
END

TO DRAW4
PU
SETPOS [40 40]
SETH -45
PD
ARM
END

TO DRAW3
PU
SETPOS [80 40]
SETH 45
PD
ARM
END

TO ARM
FD 30
BK 14
LT 25
FD 14
BK 14
RT 50
FD 14
END

```

```

TO DRAW2
PU
SETPOS [52 -4]
SETH 180
PD
FD 30
RT 90
FD 8
END

TO DRAW1
PU
SETPOS [68 -4]
SETH 180
PD
FD 30
LT 90
FD 8
END

TO TESTWORD
IF EQUALP :GUESS :MYWORD [WIN] [SAY ▶
  [NOPE!] BADTRY]
END

TO LOSE
SAY [YOU LOSE!! SORRY.]
SETCURSOR [0 20]
TYPE :MYWORD
SETCURSOR [0 23]
EYES
FROWN
END

TO EYES
PU
SETPOS [52 82]
SETH 90
PD
FD 4
PU
FD 6
PD
FD 4
END

TO FROWN
PU
SETPOS [66 72]
SETH -9
MOUTH
END

TO MOUTH
PD
LT 18
REPEAT 8 [FD 2 LT 18]
END

TO WIN
SETCURSOR [0 20]
DISWORD :MYWORD
SAY [YOU WIN!!!]
MAKE "WON "TRUE
UNGALL
FINISH :TRIES
EYES
SMILE
END

TO UNGALL
PU
SETPOS [-40 -60]
SETH 90
PE
GALL1
END

TO FINISH :NUM
IF :NUM=0 [STOP]
RUN SE WORD "DRAW :NUM []
FINISH :NUM-1
END

TO SMILE
PU
SETPOS [54 76]
SETH 171
MOUTH
END

TO PICK :LIST
OP ITEM (1 + RANDOM COUNT :LIST) :LIST
END

TO ITEM :N :LIST
IF :N=1 [OP FIRST :LIST]
OP ITEM :N-1 BF :LIST
END

```

Math: A Sentence Generator

When we think of computers making up sentences, we most often think of them making up English or French sentences. We rarely think of them making up math sentences. This project is about developing a math sentence generator. It is set in the context of developing an interactive program. A sentence is made up in the form $3 + X = 5$ and the user is asked **WHAT IS X?**.

The first example involves only addition sentences. Then the program is modified to include multiplication, subtraction, and division. Later the program is changed once more to vary the form of the math sentences and keep track of the number of times the user responds to the same question.

I boldface what the user types.

```
?MATH
6 + X = 7
WHAT IS X? 1
RIGHT

7 + X = 16
WHAT IS X? 5
NOPE, X IS 9

7 + X = 10
WHAT IS X? 3
RIGHT
```

As the example shows, MATH makes addition sentences of the form $2 + X = 3$ and not of the form $X + 2 = 3$. Later we will change MATH so that it uses both forms.

MATH randomly chooses two of the integers to be used in the math sentence. ADD then presents the addition problem and checks on your answer. The numbers MATH chooses are less than ten, but you can easily adjust the procedure and make the numbers larger.

```
TO MATH
ADD RANDOM 10 RANDOM 10
PR []
MATH
END

TO ADD :NUM1 :NUM2
PR (SE :NUM1 [+ X =] :NUM1 + :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END
```

In the addition sentences, the value of X is :NUM2, which is the second input to ADD. The sum of the two inputs is computed by ADD.

There are different ways to expand this program. You could design the

WORDPLAY

program so that it gives you three chances to get the answer right. You could expand the program so that it gives you problems in subtraction, division, and multiplication. You could make it keep track of the number of problems you do and the number you respond correctly to. You might decide to help the user. Some of these suggestions are explored in the next section.

Making MATH Subtract, Multiply, and Divide

One way to extend MATH is to make three more procedures, SUBTRACT, MULTIPLY, and DIVIDE.

```

TO SUBTRACT :NUM1 :NUM2
PR (SE :NUM1 [- X =] :NUM1 - :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END

TO MULTIPLY :NUM1 :NUM2
PR (SE :NUM1 [* X =] :NUM1 * :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END

TO DIVIDE :NUM1 :NUM2
PR (SE :NUM1 [/ X =] :NUM1 / :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END

```

Try these procedures to see if there are any bugs. Modifying MATH is a good way to try these new procedures.

```

TO MATH
ADD RANDOM 10 RANDOM 10
SUBTRACT RANDOM 10 RANDOM 10
MULTIPLY RANDOM 10 RANDOM 10
DIVIDE RANDOM 10 RANDOM 10
PR []
MATH
END

MATH
4 + X = 7          6 * X = 18
WHAT IS X? 3        WHAT IS X? 3
RIGHT              RIGHT
3 - X = 4          3 / X = 1
WHAT IS X? 1        WHAT IS X? 3
NOPE, X IS -1      RIGHT

```

What do you think? The program seems to work, but there are some possible problems. For example, in the subtraction sentences x might have a negative value. Perhaps you want to use this program without negative numbers for answers. We can adjust SUBTRACT so that the value of x is always positive.

Notice that the sentences are of the form $3 - x = 2$. The form $x - 3 = 4$ might be easier to solve, and so you might want to make sentences in that form.

There is a potential bug with multiplication and division. For example, division by 0 will cause Logo to stop the program and print out an error message. Attempts to divide by 0 must be prevented. One way to make sure of this is to add one to the random number used as DIVIDE's second input. Multiplication by 0 can cause a different sort of problem when you try to figure out what $0 * x$ is.

Although the preceding examples do not show x being a fractional number like .5, it is possible. You might want to guard against that happening. Since the sentences are generated by the program, we can make sure that the computation is performed so that x is always a whole number.

In the next section MATH is extended to include some of these ideas. The procedures are rewritten. A new procedure is introduced called ANSWER. It is used by ADD, MULTIPLY, SUBTRACT, and DIVIDE to print out the sentence and get the user's response to what x is.

Extending MATH

In this section, the first extensions to MATH guard against multiplication or division by 0 and give the user three chances to figure out what x is. All math sentences are still written in the form $3 + x = 5$ and expect integer answers. The program generates two random numbers and then computes a third. Here is an example of the program in action.

```
?MATH  
HERE ARE SOME MATH PROBLEMS.  
8 + X = 13  
WHAT IS X?
```

Now if you type 5, Logo responds:

```
RIGHT ON
```

If you type anything else, Logo responds:

```
TRY AGAIN
```

```
8 + X = 13  
WHAT IS X?
```

You are given three tries to get the answer. If you are still wrong, Logo responds:

```
NOPE, X IS 5
```

WORDPLAY

MATH has MATH1 present sentences in subtraction, multiplication, and division as well as addition.

```
TO MATH
  TS
  CT
  PR [HERE ARE SOME MATH PROBLEMS.]
  MATH1
  END

  TO MATH1
    ADD RANDOM 11 RANDOM 11
    SUBTRACT RANDOM 11 RANDOM 11
    MULTIPLY 1 + RANDOM 10 1 + RANDOM 10
    DIVIDE 1 + RANDOM 10 1 + RANDOM 10
    PR [] WAIT 60
    MATH1
    END
```

After ADD computes :RESULT, ANSWER takes over the job of printing out the sentence and checking the user's response.

```
TO ADD :NUM1 :NUM2
  MAKE "RESULT :NUM1 + :NUM2
  ANSWER [+ X =] 1
  END
```

ADD gives ANSWER the form [+ X =] as its first input. The second input represents the number of times the user responds to the question WHAT IS X?. :NUM1, :NUM2, and :RESULT are used by ANSWER's subprocedures ANSWER1 and GETINP. The variables are not given as inputs to ANSWER or its subprocedures. As far as these procedures are concerned, these are global variables. The value of X is still :NUM2.

ANSWER prints the mathematical sentence with the help of ANSWER1. After the sentence is printed, ANSWER asks for the value of X. It then turns the job over to GETINP along with the user's response.

```
TO ANSWER :PHRASE :TIMES
  PR []
  ANSWER1 :PHRASE
  TYPE SE [WHAT IS X?] "\ "
  GETINP RL
  END

  TO ANSWER1 :PHRASE
  PR (SE :NUM1 :PHRASE :RESULT)
  END
```

(\ is the way to quote special characters like space. ANSWER prints two spaces after the question mark.) GETINP plays an important role. It determines what to do next. If :INP is empty, GETINP assumes this is the user's signal to do something else and so calls MATH1. If :INP is not the same as :NUM2, then GETINP calls ANSWER adding 1 to :TIMES, unless this is the user's third try. On the third try GETINP gives the answer.

```

TO GETINP :INP
IF EMPTYP :INP [MATH1 STOP]
IF :NUM2 = FIRST :INP [PR [RIGHT ON] STOP]
IF :TIMES = 3 [PR SE [NOPE, X IS] :NUM2 STOP]
PR [TRY AGAIN]
ANSWER :PHRASE :TIMES + 1
END

```

The MULTIPLY procedure is similar to ADD in structure.

```

TO MULTIPLY :NUM1 :NUM2
MAKE "RESULT :NUM1 * :NUM2
ANSWER [* X =] 1
END

```

A couple of tricks are used here so that ANSWER will work for MULTIPLY, DIVIDE, and SUBTRACT. :NUM2 is always the value of X. :NUM1 is always on the left side of the equals sign and :RESULT is always on the right of the equals sign. What does change is which of these numbers are inputs to a procedure and which are computed in the procedure. For example, SUBTRACT computes the value of NUM1 while :RESULT and :NUM2 are inputs. But the value of X is still :NUM2.

```

TO SUBTRACT :RESULT :NUM2
MAKE "NUM1 :RESULT + :NUM2
ANSWER [- X =] 1
END

```

DIVIDE makes sure that the value of X is always an integer by shifting the role of its first input, which becomes :RESULT. DIVIDE is given :RESULT and computes :NUM1.

```

TO DIVIDE :RESULT :NUM2
MAKE "NUM1 :RESULT * :NUM2
ANSWER [/ X =] 1
END

```

Extensions

There are many modifications you might want to make to this kind of program. The modification I chose is to allow sentences to be in either of two forms.

```

3 + X = 5
X + 3 = 5

```

The changed procedures follow. Notice that the decision as to which form to use is based on whether RANDOM 2 outputs 0 or 1.

```

TO ADD :NUM1 :NUM2
MAKE "RESULT :NUM1 + :NUM2
IF 0 = RANDOM 2 [ANSWER [X +] 0 STOP]
ANSWER [+ X =] 1
END

```

WORDPLAY

```
TO MULTIPLY :NUM1 :NUM2
MAKE "RESULT :NUM1 * :NUM2
IF 0 = RANDOM 2 [ANSWER [X *] 0 STOP]
ANSWER [* X =] 1
END
```

The new forms for ADD and MULTIPLY are

$$\begin{aligned} X + 3 &= 5 \\ X * 3 &= 6 \end{aligned}$$

where the value of X is still :NUM2.

When SUBTRACT generates a sentence in the form $6 - X = 2$, its inputs, :RESULT and :NUM2, are added together to be :NUM1. In the example $6 - X = 2$, :NUM1 is 6 and :RESULT is 2.

When the sentence is in the form $X - 4 = 2$, then SUB2 computes :NUM2 by adding the inputs :RESULT and :NUM1. In this case :RESULT is 2 and :NUM1 is 4.

```
TO SUBTRACT :RESULT :NUM2
IF 0 = RANDOM 2 [SUB2 :RESULT :NUM2 STOP]
MAKE "NUM1 :RESULT + :NUM2
ANSWER [- X =] 1
END
```

```
TO SUB2 :RESULT :NUM1
MAKE "NUM2 :RESULT + :NUM1
ANSWER [X -] 1
END
```

DIVIDE computes :NUM1 as :RESULT * :NUM2 when the form is $6 / X = 2$. DIVIDE computes :NUM2 as :NUM1 * :RESULT when the form is $X / 3 = 2$.

```
TO DIVIDE :RESULT :NUM2
IF 0 = RANDOM 2 [DIV2 :RESULT :NUM2 STOP]
MAKE "NUM1 :RESULT * :NUM2
ANSWER [/ X =] 1
END
```

```
TO DIV2 :RESULT :NUM1
MAKE "NUM2 :RESULT * :NUM1
ANSWER [X /] 1
END
```

ANSWER needed to be changed as well.

```
TO ANSWER :PHRASE :TIMES
PR []
IF "X = FIRST :PHRASE [ANSWER2 :PHRASE] [ANSWER1 :PHRASE]
TYPE SE [WHAT IS X?] "\ \
GETINP RL
END

TO ANSWER1 :PHRASE
PR (SE :NUM1 :PHRASE :RESULT)
END

TO ANSWER2 :PHRASE
PR (SE :PHRASE :NUM1 "= :RESULT)
END
```

PROGRAM LISTING

```

TO MATH
  TS
  CT
  PR [HERE ARE SOME MATH PROBLEMS.]
  MATH1
  END

  TO MATH1
    ADD RANDOM 11 RANDOM 11
    SUBTRACT RANDOM 11 RANDOM 11
    MULTIPLY 1 + RANDOM 10 1 + RANDOM 10
    DIVIDE 1 + RANDOM 10 1 + RANDOM 10
    PR [] WAIT 60
    MATH1
    END

  TO ADD :NUM1 :NUM2
    MAKE "RESULT :NUM1 + :NUM2
    IF 0 = RANDOM 2 [ANSWER [X +] 0 STOP]
    ANSWER [+ X =] 1
    END

  TO MULTIPLY :NUM1 :NUM2
    MAKE "RESULT :NUM1 * :NUM2
    IF 0 = RANDOM 2 [ANSWER [X *] 0 STOP]
    ANSWER [* X =] 1
    END

  TO SUBTRACT :RESULT :NUM2
    IF 0 = RANDOM 2 [SUB2 :RESULT :NUM2 ►
      STOP]
    MAKE "NUM1 :RESULT + :NUM2
    ANSWER [- X =] 1
    END

  TO SUB2 :RESULT :NUM1
    MAKE "NUM2 :RESULT + :NUM1
    ANSWER [X -] 1
    END

    TO DIVIDE :RESULT :NUM2
      IF 0 = RANDOM 2 [DIV2 :RESULT :NUM2 ►
        STOP]
      MAKE "NUM1 :RESULT * :NUM2
      ANSWER [/ X =] 1
      END

      TO DIV2 :RESULT :NUM1
        MAKE "NUM2 :RESULT * :NUM1
        ANSWER [X /] 1
        END

        TO ANSWER :PHRASE :TIMES
          PR []
          IF "X = FIRST :PHRASE [ANSWER2 ►
            :PHRASE] [ANSWER1 :PHRASE]
          TYPE SE [WHAT IS X?] "\ "
          GETINP RL
          END

          TO ANSWER1 :PHRASE
            PR (SE :NUM1 :PHRASE :RESULT)
            END

          TO ANSWER2 :PHRASE
            PR (SE :PHRASE :NUM1 "= :RESULT)
            END

            TO GETINP*:INP
              IF EMPTYP :INP [MATH1 STOP]
              IF :NUM2 = FIRST :INP [PR [RIGHT ON] ►
                STOP]
              IF :TIMES = 3 [PR SE [NOPE, X IS] ►
                :NUM2 STOP]
              PR [TRY AGAIN]
              ANSWER :PHRASE :TIMES + 1
              END

```

Number Speller

```
?PRINT SPELL 1427
ONE THOUSAND FOUR HUNDRED TWENTY SEVEN
?
```

This program takes a whole number as input and outputs the number spelled out in words.

The general idea is to divide the number into groups of three digits. For example, the number 1234567890 is 1 billion, 234 million, 567 thousand, 890. For each such group we must spell out its three-digit number and also find the word (like “million”) that indicates the position of that group in the entire number.

Spelling a Group of Three

Let's start by writing a procedure, SPELL.GROUP, that spells out a number of up to three digits.

```
TO SPELL.GROUP :GROUP
IF :GROUP>99 [OUTPUT (SE DIGIT FIRST :GROUP "HUNDRED
    SPELL.GROUP BF :GROUP)]
IF 3=COUNT :GROUP [MAKE "GROUP BF :GROUP]
IF AND :GROUP>10 :GROUP<20 [OUTPUT TEEN :GROUP-10]
OUTPUT SE (IF :GROUP>9 [TENS FIRST :GROUP] [])
    (IF 0<LAST :GROUP [DIGIT LAST :GROUP] [])
END
```

```
?PRINT SPELL.GROUP 425
FOUR HUNDRED TWENTY FIVE
?
```

Subprocedures DIGIT, TEEN, and TENS select words corresponding to a particular digit in different positions. DIGIT selects words like “three”; TEEN words like “thirteen”; and TENS words like “thirty.”

The first instruction in SPELL.GROUP deals with a nonzero hundreds digit of the group, if any. Next, a possible leading zero is eliminated from the group. Then the procedure recognizes the special case of a number greater than ten and less than twenty. These numbers are special because they are represented all in one word, like “thirteen.” Other two-digit numbers are represented by one word for the tens digit and one for the ones digit, like “eighty seven.” If the number isn't a teen, the procedure then deals with its tens digit and its ones digit separately.

A trick used in SPELL.GROUP looks like this:

```
IF predicate [ expression ] []
```

Here is an example:

```
IF :GROUP>9 [TENS FIRST :GROUP] []
```

If the predicate tested by `IF` is `FALSE`, the value of this expression is the empty list (`[]`), so it contributes nothing to the final result when combined with other things using `SE`.

`SPELL.GROUP` outputs the empty list, not the word `ZERO`, if its input is `0`. This is okay because we want to say “zero” only if the entire number we’re spelling is `0`, not just one group. (Remember that the reason we wrote `SPELL.GROUP` for numbers up to three digits is that groups of three are the building blocks of larger numbers.) For example, the number `1000234` is spelled “one million two hundred thirty four,” not “one million zero thousand two hundred thirty four.” We’ll have to remember to notice, later on, if the entire number we’re spelling is `0`.

Here are the procedures that select the words for each digit.

```
TO TENS :DIG
OUTPUT ITEM :DIG [TEN TWENTY THIRTY FORTY FIFTY
SIXTY SEVENTY EIGHTY NINETY]
END

TO TEEN :DIG
OUTPUT ITEM :DIG [ELEVEN TWELVE THIRTEEN FOURTEEN FIFTEEN
SIXTEEN SEVENTEEN EIGHTEEN NINETEEN]
END

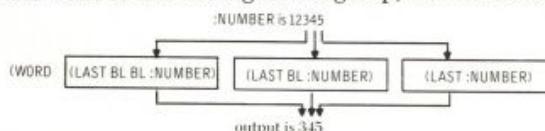
TO DIGIT :DIG
OUTPUT ITEM :DIG [ONE TWO THREE FOUR FIVE SIX SEVEN
EIGHT NINE]
END
```

These use the common subprocedure `ITEM`.

```
TO ITEM :NUM :STUFF
IF :NUM=1 [OP FIRST :STUFF]
OP ITEM :NUM-1 BF :STUFF
END
```

Spelling a Large Number

Now we have to divide a large number into groups of three, so that we can use `SPELL.GROUP` on each of the triads. One complication is that in dealing with very large numbers, we can’t rely on Logo’s arithmetic operations, because if we do, the numbers will be rounded off. Logo ordinarily handles numbers only up to ten digits without rounding. We’ll use Logo’s word-manipulation operations. For example, if we’re spelling out the number `12345` and want to find the rightmost group, we’ll do something like this:



In other words, we must treat a large number as a word that happens to be composed of digits instead of letters.

Note: In order to convince Logo not to round off numbers longer than ten digits, you have to type them in with a quotation mark like this:

```
PRINT SPELL "1234567890987654321
```

WORDPLAY

We can work up from SPELL.GROUP. One thing we need is a procedure to combine a spelled-out group with the name of its place in the complete number (thousand, million, etc.):

```
TO TRIAD :GROUP :PLACE
IF :GROUP>0 [OP SE SPELL.GROUP :GROUP :PLACE]
OP []
END
```

The test for :GROUP>0 is there to deal with cases like 1000234, where the entire thousands group should be omitted.

At this point, it's important to decide whether we are working on the number from left to right or from right to left. The most obvious thing is probably left to right, because that's the way we actually read numbers, starting with the leftmost group. That's the approach I took the first time I wrote this program. But it turns out to be much simpler to write the program if we start from the right. There are two reasons for this.

The first reason is this: suppose you see a long number like 123,456,-234,345,567,678,346,765,654,987. What is the name of the place associated with the leftmost group? To answer that question you have to count the groups, starting from the right. The 987 group is the ones group, the 654 group is the thousands group, the 765 group is the millions group, and so on. So in a sense we have to start from the right in order to know what to do with the 123 group on the left. The second reason is related to the first. Sometimes numbers are written with commas separating the groups. But in Logo we don't use commas inside numbers this way. Suppose you see a number like 1234567890987654321. What is the leftmost group? You might guess 123, but that would be true only if the number of digits in the entire number were a multiple of three. Actually, this number is 1 quintillion 234 quadrillion and so on. In order to know the number of digits in the leftmost group, we have to count off by threes from the right.

Working from right to left, the overall pattern of the program will be more or less like the following. I've written this in lower case to emphasize that it isn't a completed Logo procedure.

```
to spell.number :number
op se (spell.number butlast3 :number) (triad last3 :number)
end
```

Two things are missing from this partially written procedure. First, there is no *stop rule* to tell the procedure when it has reached the end (the leftmost end, that is) of the number. Second, we haven't provided for the place-name input to TRIAD. The solution to the first problem is that when the number of digits in the number we're spelling is three or fewer, we're down to the last group. The solution to the second problem involves providing a list of group place names as another input to this partly written procedure. Putting these things together results in two procedures.

```
TO SPELL :NUMBER
IF :NUMBER=0 [OP [ZERO]]
OP SPELL1 :NUMBER [[] THOUSAND MILLION BILLION TRILLION
QUADRILLION QUINTILLION]
END
```

```

TO SPELL1 :NUMBER :PLACES
IF (COUNT :NUMBER)<4 [OP TRIAD :NUMBER FIRST :PLACES]
OP SE (SPELL1 BUTLAST3 :NUMBER BF :PLACES)
    (TRIAD LAST3 :NUMBER FIRST :PLACES)
END

```

The top-level procedure, SPELL, recognizes the special case of the number 0. In its subprocessure SPELL1, two auxiliary procedures are used that we haven't written yet. LAST3 and BUTLAST3 are operations like LAST and BUTLAST, but they output (all but) the last three letters of a word instead of (all but) the last one. Here they are:

```

TO LAST3 :WORD
OP (WORD (LAST BL BL :WORD) (LAST BL :WORD) (LAST :WORD))
END

```

```

TO BUTLAST3 :WORD
OP BL BL BL :WORD
END

```

SUGGESTIONS

- What do you have to do to make this program spell out numbers in a language other than English? The main thing, of course, is to change the lists of words in SPELL, DIGIT, TENS, and TEEN. But what *structural* differences are there in different languages? For example, in French there are no special names for 70 and 90. Instead, numbers are added to the names for 60 and 80. That is, 70 is "soixante-dix," or "sixty-ten"; 73 is "soixante-treize" or "sixty-thirteen." (This is true of French as spoken in France; the dialect of French spoken in Belgium *does* have special words for 70 and 90!)
- Can you modify the program to spell out numbers including a decimal fraction, so SPELL 3.14 will output [THREE AND FOURTEEN ONE-HUNDREDTHS]? What about exponential notation, so that SPELL 4E3 will output [FOUR THOUSAND]?
- What about translating to or from Roman numerals? In what ways would a program to do that be similar to this one? How would it be different?
- What about translating backward? That is, write a program that will accept a list of words representing a number and output the number.

PROGRAM LISTING

```

TO SPELL.GROUP :GROUP
IF :GROUP>99 [OUTPUT (SE DIGIT FIRST ▶
    :GROUP "HUNDRED SPELL.GROUP BF ▶
    :GROUP)]
IF 3=COUNT :GROUP [MAKE "GROUP BF ▶
    :GROUP]
IF AND :GROUP>10 :GROUP<20 [OUTPUT ▶
    TEEN :GROUP-10]
OUTPUT SE (IF :GROUP>9 [TENS FIRST ▶
    :GROUP] [[]]) (IF 0<LAST :GROUP ▶
    [DIGIT LAST :GROUP] [[]])
END
TO TENS :DIG
OUTPUT ITEM :DIG [TEN TWENTY THIRTY ▶
    FORTY FIFTY SIXTY SEVENTY EIGHTY ▶
    NINETY]
END

```

```

TO TEEN :DIG
OUTPUT ITEM :DIG [ELEVEN TWELVE ▶
    THIRTEEN FOURTEEN FIFTEEN SIXTEEN ▶
    SEVENTEEN EIGHTEEN NINETEEN]
END

TO DIGIT :DIG
OUTPUT ITEM :DIG [ONE TWO THREE FOUR ▶
    FIVE SIX SEVEN
    EIGHT NINE]
END

TO ITEM :NUM :STUFF
IF :NUM=1 [OP FIRST :STUFF]
OP ITEM :NUM-1 BF :STUFF
END

TO TRIAD :GROUP :PLACE
IF :GROUP>0 [OP SE SPELL.GROUP :GROUP ▶
    :PLACE]
OP []
END

TO SPELL :NUMBER
IF :NUMBER=0 [OP [ZERO]]
OP SPELL1 :NUMBER [|] THOUSAND MILLION ▶
    BILLION TRILLION QUADRILLION ▶
    QUINTILLION]
END

TO SPELL1 :NUMBER :PLACES
IF (COUNT :NUMBER)<4 [OP TRIAD :NUMBER ▶
    FIRST :PLACES]
OP SE (SPELL1 BUTLAST3 :NUMBER BF ▶
    :PLACES) (TRIAD LAST3 :NUMBER ▶
    FIRST :PLACES)
END

TO LAST3 :WORD
OP (WORD (LAST BL BL :WORD) (LAST BL ▶
    :WORD) (LAST :WORD))
END

TO BUTLAST3 :WORD
OP BL BL BL :WORD
END

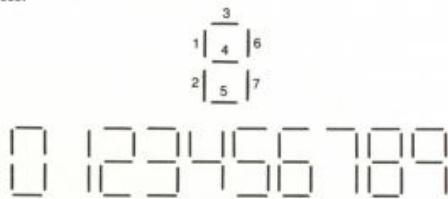
```

Drawing Letters

This project lets the turtle draw letters using a multiple-segment system like that of digital watches. It illustrates Logo's list processing capability and the use of RUN with program-generated Logo instructions. That is, instead of just carrying out procedures that were written ahead of time, this program actually assembles lists of Logo instructions and then carries out those instructions to draw the letters.

Drawing Letters in Segments

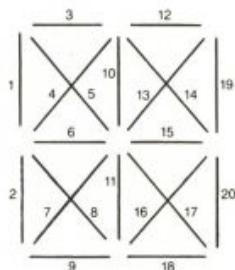
Digital watches, which only have to display digits, generally use a seven-segment system.



Seven-segment display for digits

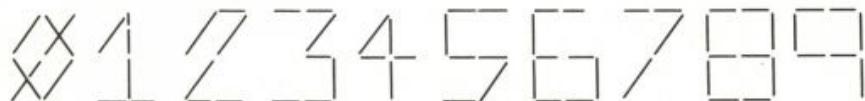
By Brian Harvey

To display all the letters of the alphabet, I chose to use a twenty-segment system, illustrated below.



Twenty-segment display for letters

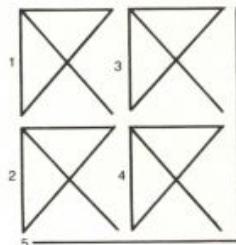
Of course, it would be possible to write a separate procedure for each letter, giving explicit turtle motion commands to shape the letter precisely. The advantage of the segment idea is that it makes it possible to write a single program, then design the individual letters very quickly. For example, after I had finished the letters of the alphabet, it was very easy for me to add the ten digits, even though I hadn't planned for them initially.



Twenty-segment digits

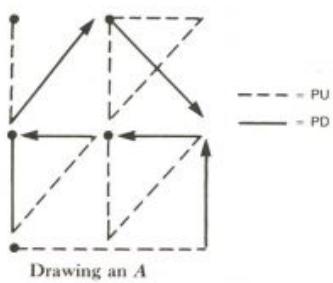
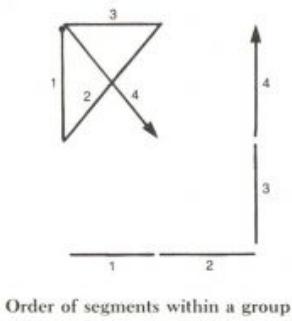
I could have written twenty procedures, one for each segment. Each would start from a "base" position, move the turtle to one end of the segment, draw the segment, and return to the base position. Then each letter could be described as a list of numbers, identifying the segments that are used to draw the letter. Instead, I chose to try to find some regularities in the way the segments are arranged. I divided the twenty segments into five groups of four each. In each group, the segments can be drawn in a single continuous path, without drawing any segment twice. (I would have liked to be able to draw the entire group of twenty segments continuously without duplication, but that's impossible.) Four of my five groups are identical in shape; the fifth is special.

The five groups are numbered in a specific order. Within a group, the segments are also numbered in a specific order; this is shown in the next figure. The program is written so that it draws segments in this order. That is, to draw a letter, the program first draws the four segments that make up the arrow-shaped group in the top left corner. Then the program goes on



Dividing twenty segments into five sets of four each

WORDPLAY



to the second group, the arrow-shaped one at the bottom left, and so on. Within each group, the program first draws segment 1, then 2, 3, and 4.

Not all segments are used in every letter, of course. Therefore, the turtle lifts its pen while tracing some of the segments. For example, consider this representation of the letter *A*.

```
MAKE "A [[PU PD] [PD PU PD] [PU PU PU PD]
          [PU PU PD] [PU PUPD]]
```

The variable *A* contains a list of five lists. Each of these smaller lists corresponds to one of the five groups of segments. The first sublist is `[PU PD]`; this means that the turtle's pen should be up during the first segment and down during the second segment. (There could be up to four words in each sublist. In this case, since there are only two words, the program will stop tracing the first group of segments after the second segment in the group.) This figure shows how the program draws the letter *A*; compare it to the list just given.

The Letter-Drawing Procedures

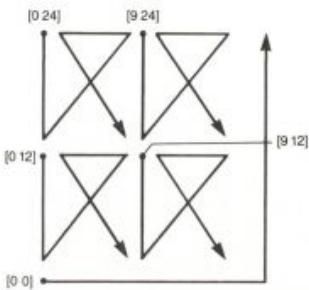
The procedure *LETTER* draws a letter. It takes two inputs. The first is a list like the one stored in the variable *A*; the second is a position, that is, a list of two numbers. The letter described by the list is drawn at the position. (Actually it is the lower left corner of the letter that is at the given position.) For example, if we have defined the variable *A* as just given, we could say

```
LETTER :A [0 0]
```

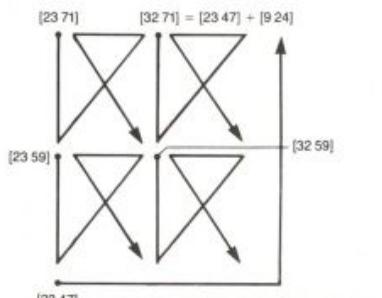
Here is the procedure:

```
TO LETTER :LET :POS
SEGMENTS :LET [[0 24] ARROW] [[0 12] ARROW]
           [[9 24] ARROW] [[9 12] ARROW] [[0 0] FINISH] :POS
END
```

The *LETTER* procedure uses a subprocedure *SEGMENTS*. The second input to *SEGMENTS* is a list that describes the overall layout of the groups of segments. Like the letter descriptions, it is a list containing five lists. But each of the five lists has only two elements: the starting position of the group of segments and the name of a procedure to draw the group of segments. This procedure is called *ARROW* for the first four groups and *FINISH* for the fifth group. The “position” of the beginning of the segment group is actually relative to the position of the letter as a whole, not an absolute screen position. For example, if the position of the letter is `[23 47]` and the relative position of the third segment group is `[9 24]`, then the actual screen position for that group is `[32 71]`.



Starting points of segments in relative coordinates



Starting points of segments for a letter drawn at *POS*=[23 47]

To know why the position numbers are what they are, you must know that I chose to base the segment lengths on a 3-4-5 right triangle. The horizontal segments are 9 turtle steps long, the vertical ones 12 steps long, and the diagonal ones 15 steps long. This conveniently makes all the FORWARD commands use whole-number inputs. It is also a reasonable shape for the overall letters.

The procedure SEGMENTS has three inputs. The third is the position of the letter. The first two are both five-element lists of lists. One is a letter description; the other is the overall layout description. The job of SEGMENTS is to match each element of the letter with the corresponding element of the description. It invokes the subprocedure SEGMENT with these sublists as inputs:

```
TO SEGMENTS :LET :TEMPLATE :POS
IF EMPTYP :LET [STOP]
IF NOT EMPTYP FIRST :LET
  [SEGMENT FIRST :LET FIRST :TEMPLATE :POS]
SEGMENTS BF :LET BF :TEMPLATE :POS
END
```

Let's see how this works with a particular example. Suppose we ask Logo to draw the letter *A* with this instruction:

```
LETTER :A [23 47]
```

This ends up invoking SEGMENTS this way:

```
SEGMENTS [[PU PD] [PD PU PD] [PU PU PU PD]
          [PU PU PD] [PU PU PD]]
  [[[0 24] ARROW] [[[0 12] ARROW]
    [[[9 24] ARROW] [[[9 12] ARROW] [[0 0] FINISH]]]
  [23 47]
```

Then SEGMENTS invokes SEGMENT five times.

```
SEGMENT [PU PD] [[[0 24] ARROW] [23 47]
SEGMENT [PD PU PD] [[[0 12] ARROW] [23 47]
SEGMENT [PU PU PU PD] [[[9 24] ARROW] [23 47]
SEGMENT [PU PU PD] [[[9 12] ARROW] [23 47]
SEGMENT [PU PU PD] [[[0 0] FINISH] [23 47]
```

Each element of the list that is specific to the letter *A* (for example, [PU PD]) is matched with an element of the list that describes the layout of letters in general (for example, [[[0 24] ARROW]]).

Drawing Each Segment Group

Remember that each sublist of the template (the overall layout description) has two pieces: the relative position of the group and the name of the procedure that draws the group. SEGMENT first has to position the turtle, then invoke the correct procedure. To position the turtle, SEGMENT uses a subprocedure called ADDPOS, which adds two position lists just as we did a few paragraphs ago. Then it uses the RUN command to invoke the procedure ARROW or FINISH, as the case may be. These procedures take the letter

WORDPLAY

description sublist as input, so the procedure name must be linked with that list to form the Logo instruction for RUN.

```
TO SEGMENT :LETPART :TEMPPART :POS
PU
SETPOS ADDPOS :POS FIRST :TEMPPART
RUN LIST LAST :TEMPPART :LETPART
END
```

For example, the first use of SEGMENT in drawing the letter *A* in our example is

```
SEGMENT [PU PD] [[Ø 24] ARROW] [23 47]
```

This is equivalent to the following Logo instructions.

```
PU
SETPOS ADDPOS [23 47] [Ø 24]
RUN LIST "ARROW [PU PD]
```

This is, in turn, equivalent to

```
PU
SETPOS [23 71]
ARROW [PU PD]
```

The tricky (but exciting!) thing to understand here is that the instruction ARROW [PU PD] doesn't actually appear in any Logo procedure in this program. Instead, this instruction is put together as the program is run. SEGMENT combines the word ARROW (which it found in the template list) with the list [PU PD] (which it found in the letter description list) into one big list. It then uses the RUN command to interpret that list as a Logo instruction. We'll use the same trick again later.

The procedures ARROW and FINISH have to follow a certain path, setting the turtle's pen up or down between steps as specified in the letter description. They use a common subprocessure DRAW, which knows how to do that. One of the inputs to DRAW is the letter description sublist with the PU and PD commands; the other input is a list of four Logo instruction lists, one for each segment of the group.

```
TO ARROW :PENS
DRAW :PENS [[SETH 180 FD 12] [LT 143.13 FD 15]
              [LT 126.87 FD 9] [LT 126.87 FD 15]]
END

TO FINISH :PENS
DRAW :PENS [[SETH 90 FD 9] [FD 9] [LT 90 FD 12] [FD 12]]
END

TO DRAW :PENS :CMDS
IF EMPTYP :PENS [STOP]
RUN FPUT FIRST :PENS FIRST :CMDS
DRAW BF :PENS BF :CMDS
END
```

Here is how this works out in our example with the letter A. The five invocations of SEGMENT listed earlier result in four invocations of ARROW and one of FINISH.

```
ARROW [PU PD]
ARROW [PD PU PD]
ARROW [PU PU PU PD]
ARROW [PU PU PD]
FINISH [PU PU PD]
```

We'll look at the first invocation of ARROW in more detail. ARROW invokes DRAW like this:

```
DRAW [PU PD] [[SETH 180 FD 12] [LT 143.13 FD 15]
[LT 126.87 FD 9] [LT 126.87 FD 15]]
```

Just as SEGMENTS paired elements of its list inputs, so does DRAW. It ends up executing these Logo instructions:

```
RUN FPUT "PU [SETH 180 FD 12]
RUN FPUT "PD [LT 143.13 FD 15]
```

There might have been up to four of these RUN instructions, because there are four segments in an ARROW group, but in this case there were only two pen commands in the input list :PENS. If we look at what the RUN instructions actually do in this example, we see that the final effect is just as if the procedure contained these instructions:

```
PU SETH 180 FD 12
PD LT 143.13 FD 15
```

This is a straightforward series of turtle graphics commands. Again, though, it's important to understand that that series of commands is not actually part of any procedure. Instead, the commands were *generated* by the DRAW procedure by putting together pieces of its inputs.

Final Details

Here is ADDPOS, the subprocedure of SEGMENT that turns the relative position of a segment group into an absolute position:

```
TO ADDPOS :POS1 :POS2
OUTPUT LIST (FIRST :POS1)+FIRST :POS2
(LAST :POS1)+LAST :POS2
END
```

Finally, the procedure SAY takes an entire word as input and draws the letters in that word one by one. It's used like this:

```
SAY "HELLO [Ø Ø]
```

and here it is.

```
TO SAY :WORD :POS
IF EMPTYP :WORD [STOP]
LETTER THING FIRST :WORD :POS
SAY BF :WORD ADDPOS :POS [24 Ø]
END
```

WORDPLAY

Here are the definitions for my letters:

```

MAKE "A [[PU PD] [PD PU PD] [PU PU PU PD]
      [PU PU PD] [PU PU PD]]
MAKE "B [[PD PU PD] [PD PU PD] [PU PD PD]
      [PU PU PU PD] [PD PD]]
MAKE "C [[PU PD] [PU PU PU PD] [PU PU PD] [] [PU PD]]
MAKE "D [[PD PU PD] [PD] [PU PU PU PD] [PU PD] [PD]]
MAKE "E [[PD PU PD] [PD PU PD] [PU PU PD] [] [PD PD]]
MAKE "F [[PD PU PD] [PD PU PD] [PU PU PD] []]
MAKE "G [[PD PU PD] [PD] [PU PU PD] [PU PU PD] [PD PD PD]]
MAKE "H [[PD] [PD PU PD] [] [PU PU PD] [PU PU PD PD]]
MAKE "I [[PU PU PD] [] [PD PU PD] [PD] [PD PD]]
MAKE "J [[PU PU PD] [] [PD PU PD] [PD] [PD]]
MAKE "K [[PD] [PD PU PD] [PU PD] [PU PU PU PD]]
MAKE "L [[PD] [PD] [] [] [PD PD]]
MAKE "M [[PD PU PU PD] [PD] [PU PD] [] [PU PU PD PD]]
MAKE "N [[PD PU PU PD] [PD] [] [PU PU PU PD] [PU PU PD PD]]
MAKE "O [[PD PU PD] [PD] [PU PU PD] [] [PD PD PD PD]]
MAKE "P [[PD PU PD] [PD PU PD] [PU PU PD]
      [PU PU PD] [PU PU PU PD]]
MAKE "Q [[PD PU PD] [PD]
      [PU PU PD] [PU PD PU PD] [PD PU PU PD]]
MAKE "R [[PD PU PD] [PD PU PD] [PU PU PD]
      [PU PU PD] [PU PU PU PD]]
MAKE "S [[PU PD] [PU PU PD] [PU PU PD] [PU PD PD] [PD]]
MAKE "T [[PU PU PD] [] [PD PU PD] [PD]]
MAKE "U [[PD] [PD] [] [] [PD PD PD PD]]
MAKE "V [[PD] [PU PU PU PD] [] [PU PD] [PU PU PU PD]]
MAKE "W [[PD] [PD PD] [] [PU PU PU PD] [PU PU PD PD]]
MAKE "X [[PU PU PU PD] [PU PD] [PU PD] [PU PU PU PD]]
MAKE "Y [[PU PU PU PD] [] [PU PD] [PD]]
MAKE "Z [[PU PU PD] [PU PD] [PU PD PD] [] [PD PD]]

```

SUGGESTIONS

- Make up descriptions for the twenty-segment digits shown near the beginning of this write-up.
- The letter *L* is described very efficiently by this scheme; the turtle takes no unnecessary steps to draw it. The letter *A*, on the other hand, is not very efficiently described. Each *PU* in its description represents a step that the turtle takes without drawing anything; to draw six strokes, the turtle travels over fifteen segments. Can you work out a way to group the segments that makes more letters more efficient? (I don't have any secret answer to this; I haven't tried it myself.)
- Modify the procedures so that the size of the letters can be varied. You could have an input called *SIZE* and use $3^* : \text{SIZE}$ for the horizontal segments, and so forth.
- Modify the procedures so that the aspect ratio of the letters (the ratio of the vertical segment length to the horizontal segment length) is variable. This is much harder; in general, it requires using trigonometry.
- Make up descriptions for lower-case letters. This may require changing the whole arrangement of segments, since some lower-case let-

ters have *descenders*. That is, they extend below the baseline of the capital letters. These letters are *g*, *j*, *p*, *q*, and *y*. Manufacturers of computer terminals don't always use descenders for lower-case letters. Some avoid it by printing those letters higher than they should be; others just use **SMALL CAPITALS** instead of lower case.

- Without changing the letter descriptions, change the shapes embodied in the procedures **ARROW** and **FINAL**. See if you can invent an interesting new alphabet this way.
- Modify the procedures so that you can write words at an angle, not just horizontally across the screen.

PROGRAM LISTING

```

TO LETTER :LET :POS
SEGMENTS :LET [[0 24] ARROW] [[0 12] ▶
    ARROW] [[9 24] ARROW] [[9 12] ▶
    ARROW] [[0 0] FINISH] 1 :POS
END

TO SEGMENTS :LET :TEMPLATE :POS
IF EMPTYYP :LET [STOP]
IF NOT EMPTYYP FIRST :LET [SEGMENT ▶
    FIRST :LET FIRST :TEMPLATE :POS]
SEGMENTS BF :LET BF :TEMPLATE :POS
END

TO SEGMENT :LETPART :TEMPPART :POS
PU
SETPOS ADDPOS :POS FIRST :TEMPPART
RUN LIST LAST :TEMPPART :LETPART
END

TO ARROW :PENS
DRAW :PENS [[SETH 180 FD 12] [LT ▶
    143.13 FD 15] [LT 126.87 FD 9] ▶
    [LT 126.87 FD 15]]
END

TO FINISH :PENS
DRAW :PENS [[SETH 90 FD 9] [FD 9] [LT ▶
    90 FD 12] [FD 12]]
END

TO DRAW :PENS :CMDS
IF EMPTYYP :PENS [STOP]
RUN FPUT FIRST :PENS FIRST :CMDS
DRAW BF :PENS BF :CMDS
END

TO ADDPOS :POS1 :POS2
OUTPUT LIST (FIRST :POS1)+FIRST :POS2 ▶
    (LAST :POS1)+LAST :POS2
END

TO SAY :WORD :POS
IF EMPTYYP :WORD [STOP]
LETTER THING FIRST :WORD :POS
SAY BF :WORD ADDPOS :POS [24 0]
END

MAKE "A [[PU PD] [PD PU PD] [PU PU PU ▶
    PD] [PU PU PD] [PU PU PD]]
MAKE "B [[PD PU PD] [PD PU PD] [PU PD ▶
    PD] [PU PU PU PD] [PD PD]]
MAKE "C [[PU PD] [PU PU PU PD] [PU PU ▶
    PD] [] [PU PD]]
MAKE "D [[PD PU PD] [PD] [PU PU PU PD] ▶
    [PU PD] [PD]]
MAKE "E [[PD PU PD] [PD PU PD] [PU PU ▶
    PD] [] [PD PD]]
MAKE "F [[PD PU PD] [PD PU PD] [PU PU ▶
    PD] []]
MAKE "G [[PD PU PD] [PD] [PU PU PD] ▶
    [PU PU PD] [PD PD PD]]
MAKE "H [[PD] [PD PU PD] [] [PU PU PD] ▶
    [PU PU PD PD]]
MAKE "I [[PU PU PD] [] [PD PU PD] [PD] ▶
    [PD PD]]
MAKE "J [[PU PU PD] [] [PD PU PD] [PD] ▶
    [PD]]
MAKE "K [[PD] [PD PU PD] [PU PD] [PU ▶
    PU PU PD]]
MAKE "L [[PD] [PD] [] [] [PD PD]]
MAKE "M [[PD PU PU PD] [PD] [] [PU PU ▶
    PU PD] [PU PU PD PD]]
MAKE "N [[PD PU PU PD] [PD] [] [PU PU ▶
    PU PD] [PU PU PD PD]]
MAKE "O [[PD PU PD] [PD] [PU PU PD] [] ▶
    [PD PD PD PD]]
MAKE "P [[PD PU PD] [PD PU PD] [PU PU ▶
    PD] [PU PU PD] [PU PU PU PD]]
MAKE "Q [[PD PU PD] [PD] [PU PU PD] ▶
    [PU PD PU PD] [PD PU PU PD]]
```

```

MAKE "R [[PD PU PD] [PD PU PD] [PU PU ►      [PU PU PU PD]]
      PD] [PU PU PD PD] [PU PU PU PD]] ►      MAKE "W [[PD] [PD PD] [] [PU PU PU PD] ►
MAKE "S [[PU PD] [PU PU PD] [PU PU PD] ►      [PU PU PD PD]]
      [PU PD PD] [PD]] ►      MAKE "X [[PU PU PU PD] [PU PD] [PU PD] ►
MAKE "T [[PU PU PD] [] [PD PU PD] ►      [PU PU PU PD]]
      [PD]] ►      MAKE "Y [[PU PU PU PD] [] [PU PD] ►
MAKE "U [[PD] [PD] [] [] [PD PD PD ►      [PD]]
      PD]] ►      MAKE "Z [[PU PU PD] [PU PD] [PU PD PD] ►
MAKE "V [[PD] [PU PU PU PD] [] [PU PD] ►      [] [PD PD]]
```

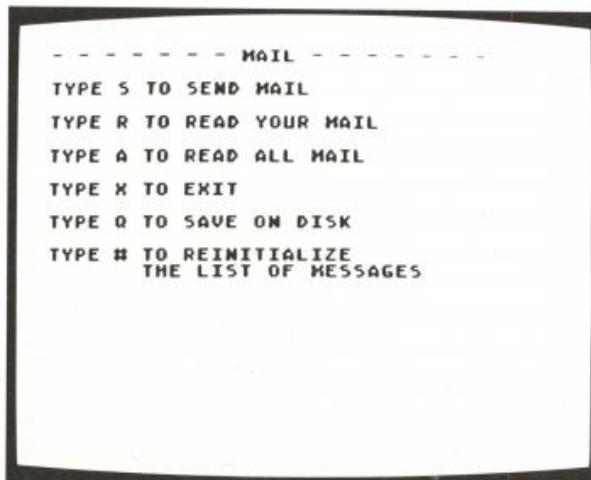
Mail

When I was a kid in school, my friends and I liked passing notes to each other. It was reflection on this experience that inspired me to write a mail program. In those olden days, the suspense was great as we waited to see if we could send messages from one side of the room to another without getting caught. With this modern method of letting Logo be the mail carrier, students today find different pleasures.

Using the Program

Since Logo has no mail system of its own, I decided to build one. The essential actions are sending and receiving mail. This project is just one example of an electronic mail system. *em*.

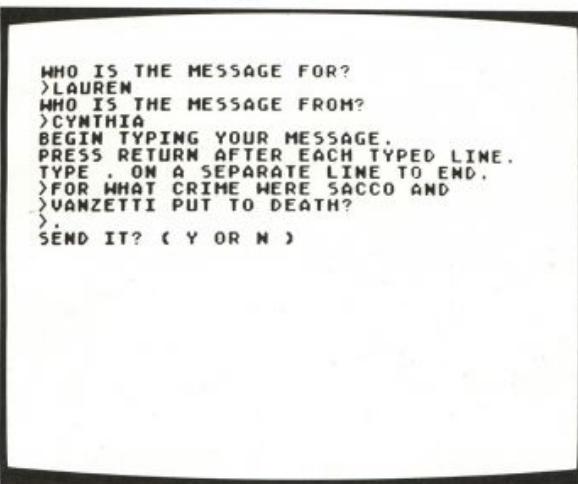
The program assumes that you have a disk on which daily mail can be saved. For convenience, you should reserve one diskette specifically to hold the messages and the mail program.* To start the program, type MAIL. You will get a screen that looks like this:



*You may also change the mail program so that you can use a cassette recorder. Then you would save to the cassette instead of to the disk.

Sending Mail

If you want to send mail, type S.

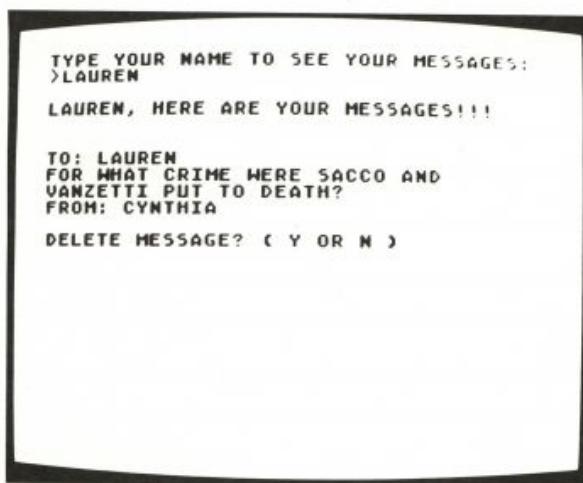


First you are asked who the message is for. A prompt (>) appears, and you type in the name of the person to whom you want to send mail. You are then asked who the message is from, and you type in your name. Next you receive instructions. After you type in your message, you are asked if you really want to send it. If you do, you are informed that the message is in the mailbox.

Reading Your Mail

To read your mail, type R.

You are asked to type in your name. Once you do, your messages appear on the screen.

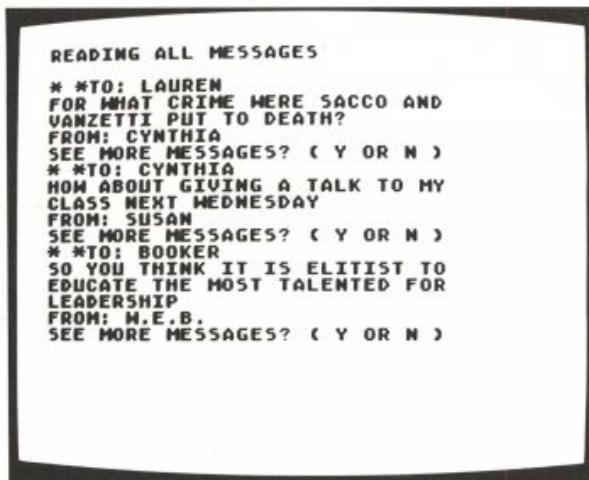


WORDPLAY

After reading each message, you are asked if you want to delete it. If you type Y, that message is deleted.

Reading All the Mail

If for some reason you want to read all the messages that have been written, type A.



After each message, you are asked if you want to see more messages. If you type Y, you see another message, otherwise you exit from reading all messages.

Other Commands

- Q Automatically saves all messages on the diskette. You are asked if the mail disk is in the drive. If you type Y, the program and all messages are saved on diskette; otherwise the program stops.
- X Stops the program.
- # Deletes all messages.

Structure of the Mail Program**The Data Base**

All the messages are organized into one list named ALL.MESSAGES. For example, :ALL.MESSAGES might look like this:

```

[[[TO: JRD] [WHO GOT THE VOTE FIRST: BLACKS OR] [WOMEN]
[FROM: DAVE]]
[[TO: LAUREN] [FOR WHAT CRIME WERE SACCO AND]
[VANZETTI PUT TO DEATH] [FROM: CYNTHIA]]
```

```
[[TO: LISA][DID THEY EVER DECIDE WHETHER]
[BLACK ENGLISH IS A LANGUAGE OR A DIALECT?]
[FROM: MARGARET]]
[[TO: JAN] [I HEARD THAT THE BLUEBERRY CROP IN]
[MAINE WILL BE GREAT THIS YEAR.]
[BECAUSE OF THE ACID RAIN.]
[DO YOU BELIEVE IT ?] [FROM: TOM]]
[[TO: BOOKER] [DO YOU AGREE WITH ME THAT THE]
[TALENTED TENTH SHOULD RECEIVE]
[EDUCATION FOR LEADERSHIP] [FROM: W.E.B.]]]
```

Each message is itself a list of lists.

The first message in this example is

```
[[TO: JRD]
[WHO GOT THE VOTE FIRST: BLACKS OR]
[WOMEN]
[FROM: DAVE]]
```

This message contains four sublists:

The first is [TO: JRD].
The second is [WHO GOT THE VOTE FIRST: BLACKS OR].
The third is [WOMEN].
The last is [FROM: DAVE].

The word TO and the receiver's name make up the first list in each message, while the word FROM and the sender's name make up the last list in the message.

The Main Procedure

MAIL is the main procedure of the program. It displays the help text, gets a character command from the user, and checks to see if the command is valid. If it is, it calls the appropriate procedures to carry out the actions. These procedures are SEND.MAIL, MY.MESSAGES, READ.ALL.MAIL, REMOVE.ALL.MESSAGES, and DISK.DUMP.

```
TO MAIL
HELP
MAKE "CHAR RC
IF NOT MEMBERP :CHAR [R S A X Q #] [PR [] PR
[!!!NOT A COMMAND.]
IF :CHAR = "R [MY.MESSAGES]
IF :CHAR = "S [SEND.MAIL]
IF :CHAR = "A [READ.ALL.MAIL]
IF :CHAR = "X [STOP]
IF :CHAR = "Q [DISK.DUMP STOP]
IF :CHAR = "#" [REMOVE.ALL.MESSAGES]
PR []
MAIL
END
```

HELP puts the menu of possible actions on the text screen.

WORDPLAY

```

TO HELP
WAIT 50
CT
PR [----- MAIL -----]
PR []
PR [TYPE S TO SEND A MESSAGE]
PR []
PR [TYPE R TO READ YOUR MAIL]
PR []
PR [TYPE A TO READ ALL MAIL]
PR []
PR [TYPE X TO EXIT]
PR []
PR [TYPE Q TO SAVE ON DISK]
PR []
PR [TYPE # TO REINITIALIZE]
SETCURSOR [7 13]
PR [THE LIST OF MESSAGES]
END

```

Sending Mail

SEND.MAIL is the main procedure for sending mail. First it asks for the name of the person who is to receive the message. It then asks for your name (the sender). You are then given instructions for typing the message. Finally you are given a chance to change your mind about sending it. If you decide that you want to send it, the message is included in the list of all messages.

```

TO SEND.MAIL
CT
PR []
PR [WHO IS THE MESSAGE FOR?]
MAKE "ANS RECEIVER'S.NAME
PR [WHO IS THE MESSAGE FROM?]
MAKE "FROM SENDER'S.NAME
PR [BEGIN TYPING YOUR MESSAGE.]
PR [PRESS RETURN AFTER EACH TYPED LINE.]
PR [TYPE . ON A SEPARATE LINE TO END.]
MAKE "PRESENT.MESSAGE :PUT :ANS GET.MESSAGE
[] LPUT :FROM []
PR [SEND IT? ( Y OR N )]
IF RC = "N [PR [!!!!MESSAGE DELETED!!!!] WAIT 50 STOP]
IF EMPTY :PRESENT.MESSAGE [STOP]
ADD.THE.MESSAGE :PRESENT.MESSAGE
PR []
PR [* * * * IT'S IN THE MAILBOX * * * *]
PR []
END

```

SEND.MAIL uses four subprocedures: RECEIVER'S.NAME, SENDER'S.NAME, GET.MESSAGE, and ADD.THE.MESSAGE.

RECEIVER'S.NAME outputs a sentence of the word TO: and the name of the person who is to receive the message. SENDER'S.NAME works similarly.

```
TO RECEIVER'S.NAME
TYPE ">
OP SE "TO: RL
END
```

```
TO SENDER'S.NAME
TYPE ">
OP SE "FROM: RL
END
```

GET.MESSAGE lets you type in a message, line by line. A ".," typed on a separate line signals the completion of the message.

```
TO GET.MESSAGE :MSG
TYPE ">
MAKE "EACH.LINE RL
IF :EACH.LINE = [.] [OP :MSG]
OP GET.MESSAGE LPUT :EACH.LINE :MSG
END
```

In ADD.THE.MESSAGE, the typed message is added to :ALL.MESSAGES.

```
TO ADD.THE.MESSAGE :PRESENT.MESSAGE
MAKE "ALL.MESSAGES FPUT :PRESENT.MESSAGE :ALL.MESSAGES
END
```

Reading Your Mail

MY.MESSAGES is the main procedure for reading your own messages. It gets your name and checks to see if you have any mail by calling CHECK.MY.MESSAGES.

```
TO MY.MESSAGES
CT
IF EMPTYP :ALL.MESSAGES [STOP]
PR [TYPE YOUR NAME TO SEE YOUR MESSAGES]
TYPE ">
MAKE "ANS RL
IF EMPTYP :ANS [MY.MESSAGES STOP]
PR []
PR SE WORD FIRST :ANS ", [HERE ARE YOUR MESSAGES!!!!]
PR []
CHECK.MY.MESSAGES :ANS :ALL.MESSAGES @
END
```

CHECK.MY.MESSAGES checks each message in :LIST to see if it is for you. CHECK.MY.MESSAGES takes three inputs. The first is :WHO, the name of the person (you) whose mail it is looking for. The second, :LIST, is the list of messages. The third, :COUNTER, is a message counter that is needed if you should decide to delete a message.

WORDPLAY

```

TO CHECK.MY.MESSAGES :WHO :LIST :COUNTER
PR []
IF EMPTYP :LIST [PR [* * * * * THAT'S IT * * * * *]
    STOP]
IF EQUALP FIRST :WHO FIRST BF FIRST FIRST :LIST
    [PRINT.AND.DELETE FIRST :LIST :COUNTER]
PR []
PR []
CHECK.MY.MESSAGES :WHO BF :LIST ( 1 + :COUNTER )
END

```

CHECK.MY.MESSAGES calls PRINT.AND.DELETE, which prints a message and asks if you want to delete it.

```

TO PRINT.AND.DELETE :MESSAGE :COUNTER
PRINT.MESSAGE :MESSAGE
PR []
TYPE [DELETE MESSAGE? ( Y OR N )]
IF RC = "Y" [MAKE "ALL.MESSAGES DELETE :COUNTER :ALL.MESSAGES
    PR [] PR [!!!!MESSAGE DELETED!!!!]]
END

```

PRINT.AND.DELETE uses PRINT.MESSAGE and DELETE.
PRINT.MESSAGE prints a single message, consisting of the receiver's name, then the message, and finally the sender's name.

```

TO PRINT.MESSAGE :MSG
IF EMPTYP :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END

TO DELETE :N :LIST
IF EMPTYP :LIST [OP []]
IF :N = 0 [OP BF :LIST]
OP FPUT FIRST :LIST DELETE :N - 1 BF :LIST
END

```

Reading All the Mail

The main procedure for reading all the mail is READ.ALL.MAIL; it calls the message-printing procedure, PRINT.ALL.MESSAGES.

```

TO READ.ALL.MAIL
CT
PR [READING ALL MESSAGES]
PRINT.ALL.MESSAGES :ALL.MESSAGES
END

```

PRINT.ALL.MESSAGES prints each message and asks you if you want to see more messages.

```

TO PRINT.ALL.MESSAGES :ALL
PR []
IF EMPTYP :ALL [PR [* * * * NO MORE MESSAGES * * * *]
STOP]
TYPE [* *]
PRINT.MESSAGE FIRST :ALL
TYPE [READ MORE MESSAGES? (Y OR N)]
IF RC = "N [PR [] PR [* * * YOU EXITED MAIL READER * * *]
STOP]
PRINT.ALL.MESSAGES BF :ALL
END

TO PRINT.MESSAGE :MSG
IF EMPTYP :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END

```

Saving on the Diskette

DISK.DUMP saves on the disk. First it reminds you to put the disk in the drive.*

```

TO DISK.DUMP
PR []
PR []
PR []
PR []
PR [IS THE DISK IN THE DRIVE?????? (Y OR N)]
IF RC = "Y [SAVE "D:MAIL"]
END

```

Reinitializing the List of Messages

REMOVE.ALL.MESSAGES clears all messages from the list of messages.

```

TO REMOVE.ALL.MESSAGES
MAKE "ALL.MESSAGES []"
END

```

PROGRAM LISTING

TO MAIL	IF :CHAR = "Q [DISK.DUMP STOP]
HELP	IF :CHAR = "# [REMOVE.ALL.MESSAGES]
MAKE "CHAR RC	PR []
IF NOT MEMBERP :CHAR [R S A X Q #] [PR ▶	MAIL
[] PR [!!!!NOT A COMMAND.]]	END
IF :CHAR = "R [MY.MESSAGES]	TO REMOVE.ALL.MESSAGES
IF :CHAR = "S [SEND.MAIL]	MAKE "ALL.MESSAGES []
IF :CHAR = "A [READ.ALL.MAIL]	END
IF :CHAR = "X [STOP]	

*If you are using a cassette instead of a diskette, you must change the last instruction in DISK.DUMP so that it saves to a cassette: IF RC = "Y [SAVE "C:]

```

TO HELP
WAIT 50
CT
PR [----- MAIL -----]
PR []
PR [TYPE S TO SEND A MESSAGE]
PR []
PR [TYPE R TO READ YOUR MAIL]
PR []
PR [TYPE A TO READ ALL MAIL]
PR []
PR [TYPE X TO EXIT]
PR []
PR [TYPE Q TO SAVE ON DISK]
PR []
PR [TYPE # TO REINITIALIZE]
SETCURSOR [7 13]
PR [THE LIST OF MESSAGES]
END

TO SEND.MAIL
CT
PR []
PR [WHO IS THE MESSAGE FOR?]
MAKE "ANS RECEIVER'S.NAME"
PR [WHO IS THE MESSAGE FROM?]
MAKE "FROM SENDER'S.NAME"
PR [BEGIN TYPING YOUR MESSAGE.]
PR [PRESS RETURN AFTER EACH TYPED ►
LINE.]
PR [TYPE . ON A SEPARATE LINE TO END.]
MAKE "PRESENT.MESSAGE SE FPUT :ANS ▶
GET.MESSAGE [] LPUT :FROM []"
PR [SEND IT? ( Y OR N )]
IF RC = "N [PR [!!!!MESSAGE ►
DELETED!!!!] WAIT 50 STOP]
IF EMPTYP :PRESENT.MESSAGE [STOP]
ADD.THE.MESSAGE :PRESENT.MESSAGE
PR []
PR [* * * * IT'S IN THE MAILBOX * * * ▶
*]
PR []
END

TO RECEIVER'S.NAME
TYPE ">"
OP SE "TO: RL
END

TO SENDER'S.NAME
TYPE ">"
OP SE "FROM: RL
END

TO GET.MESSAGE :MSG
TYPE ">"
MAKE "EACH.LINE RL
IF :EACH.LINE = [.] [OP :MSG]
OP GET.MESSAGE LPUT :EACH.LINE :MSG
END

TO ADD.THE.MESSAGE :PRESENT.MESSAGE
MAKE "ALL.MESSAGES FPUT ▶
:PRESENT.MESSAGE :ALL.MESSAGES
END

TO MY.MESSAGES
CT
IF EMPTYP :ALL.MESSAGES [STOP]
PR [TYPE YOUR NAME TO SEE YOUR ▶
MESSAGES]
TYPE ">"
MAKE "ANS RL
IF EMPTYP :ANS [MY.MESSAGES STOP]
PR []
PR SE WORD FIRST :ANS ", [THERE ARE ▶
YOUR MESSAGES!!!]
PR []
CHECK.MY.MESSAGES :ANS :ALL.MESSAGES 0
END

TO CHECK.MY.MESSAGES :WHO :LIST ▶
:COUNTER
PR []
IF EMPTYP :LIST [PR [* * * * * ▶
THAT'S IT * * * * *] STOP]
IF EQUALP FIRST :WHO FIRST BF FIRST ▶
FIRST :LIST [PRINT.AND.DELETE ▶
FIRST :LIST :COUNTER]
PR []
PR []
CHECK.MY.MESSAGES :WHO BF :LIST ( 1 + ▶
:COUNTER )
END

TO PRINT.AND.DELETE :MESSAGE :COUNTER
PRINT.MESSAGE :MESSAGE
PR []
TYPE [DELETE MESSAGE? ( Y OR N )]
IF RC = "Y [MAKE "ALL.MESSAGES DELETE ▶
:COUNTER :ALL.MESSAGES PR [] PR ▶
[!!!!MESSAGE DELETED!!!!]]
END

```

```

TO DELETE :N :LIST
IF EMPTYP :LIST [OP []]
IF :N = 0 [OP BF :LIST]
OP FPUT FIRST :LIST DELETE :N - 1 BF ▶
    :LIST
END

TO READ.ALL.MAIL
CT
PR [READING ALL MESSAGES]
PRINT.ALL.MESSAGES :ALL.MESSAGES
END

TO PRINT.ALL.MESSAGES :ALL
PR []
IF EMPTYP :ALL [PR [* * * * NO MORE ▶
    MESSAGES * * * *] STOP]
TYPE [* *]
PRINT.MESSAGE FIRST :ALL
TYPE [READ MORE MESSAGES? (Y OR N)]
IF RC = "N [PR [] PR [* * * YOU ▶
    EXITED MAIL READER * * *] STOP]
PRINT.ALL.MESSAGES BF :ALL
END

TO PRINT.MESSAGE :MSG
IF EMPTYP :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END

TO DISK.DUMP
PR []
PR []
PR []
PR [IS THE DISK IN THE DRIVE?????? (Y ▶
    OR N)]
IF RC = "Y [SAVE "D:MAIL"]
END

MAKE "ALL.MESSAGES [[TO: LAUREN] [FOR ▶
    WHAT CRIME WERE SACCO AND] ▶
    [EVANZETTI PUT TO DEATH?] [FROM: ▶
    CYNTHIA]] [[TO: CYNTHIA] [HOW ▶
    ABOUT GIVING A TALK TO MY] [CLASS ▶
    NEXT WEDNESDAY] [FROM: SUSAN]] ▶
    [[TO: BOOKER] [SO YOU THINK IT IS ▶
    ELITIST TO] [EDUCATE THE MOST ▶
    TALENTED FOR] [LEADERSHIP] [FROM: ▶
    W.E.B.]] [[TO: LAUREN] [I HEARD ▶
    THAT THE BLUEBERRY] [CROP IN ▶
    MAINE WILL BE GREAT] [NEXT YEAR ▶
    BECAUSE OF THE ACID RAIN.] [FROM: ▶
    TOM]] [[TO: TO LISA] [IS BLACK ▶
    ENGLISH CONSIDERED] [A LANGUAGE ▶
    OR A DIALECT?] [FROM: MARGARET]] ▶
    [[TO: JRD] [WHO GOT THE VOTE ▶
    FIRST] [BLACKS OR WOMEN?] [FROM: ▶
    DAVE]]]

```

Wordscram

A Word Guessing Game

WORDSCRAM picks a word, scrambles the letters, and shows you the scrambled version of the word. Your job is to guess the word. (In this sample game, the word is chosen from a list of thirty or forty technical Logo terms.) WORDSCRAM helps you by showing which letters in your guess are in the correct spot. You can also type HINT if you need a hint, or HELP if you want to give up. Here is a sample of WORDSCRAM in action.

WORDPLAY**?WORDSCRAM**

WELCOME TO WORDSCRAM !
 DO YOU WANT INSTRUCTIONS ? N
 THINKING....

OK. HERE IS YOUR SCRAMBLED WORD:
 RSNRUOCIEI

WHAT'S YOUR GUESS ?

RE

Two letters correct.

WHAT'S YOUR GUESS ?

RE

**

WHAT'S YOUR GUESS ?

HINT

WELL...OK...TRY REC

Get a hint.

Computer responds.

WHAT'S YOUR GUESS ?

RECUSR

S and R not correct.

*****??

WHAT'S YOUR GUESS ?

RECURSION

DOING GREAT !

Got it!

DO YOU WANT ANOTHER WORD ? Y

THINKING....

OK. HERE IS YOUR SCRAMBLED WORD:
 PUUTOT

WHAT'S YOUR GUESS ?

HELP

I give up.

THE WORD WAS OUTPUT

DO YOU WANT ANOTHER WORD ? N

Program ends.

Scrambling a Word

The heart of WORDSCRAM is SCRAMBLE. It takes a word as input and outputs a scrambled version of it. The strategy goes something like this. Let's say the word to scramble is "draw."

1. Pick a letter from the word at random.
2. To make sure that the letter does not get picked again, remove it from the word.
3. Join the letter just picked to the result of scrambling the remaining letters of the word. Continue until there are no more letters left.

Using the word "draw" as an example, we might get this result:

```

SCRAMBLE "DRAW
W + SCRAMBLE "DRA
R + SCRAMBLE "DA
A + SCRAMBLE "D
D + SCRAMBLE "

```

The assembled word is "wrad."

SCRAMBLE picks a letter from the word, then uses that letter in two ways: it removes the letter from the word (to get the input for the recursive invocation of SCRAMBLE), and it sticks the same letter back onto the beginning of the scrambled word. To make this work, after SCRAMBLE picks a letter, it invokes a subprocedure, SCRAMBLE1, whose second input is the letter to remove from the word.

```

TO SCRAMBLE :WORD
IF EMPTYP :WORD [OP ]
OP SCRAMBLE1 :WORD RANPICK :WORD
END

TO SCRAMBLE1 :WORD :LETTER
OP WORD :LETTER (SCRAMBLE REMOVE :LETTER :WORD)
END

```

Here is how SCRAMBLE and SCRAMBLE1 interact, in the same example we looked at before.

```

SCRAMBLE "DRAW
SCRAMBLE1 "DRAW "W
SCRAMBLE "DRA
SCRAMBLE1 "DRA "R
SCRAMBLE "DA
SCRAMBLE1 "DA "A
SCRAMBLE "D
SCRAMBLE1 "D "D
SCRAMBLE "
SCRAMBLE outputs "
SCRAMBLE1 outputs "D which is WORD "D "
SCRAMBLE outputs "D
SCRAMBLE1 outputs "AD which is WORD "A "D
SCRAMBLE outputs "AD
SCRAMBLE1 outputs "RAD which is WORD "R "AD
SCRAMBLE outputs "RAD
SCRAMBLE1 outputs "WRAD which is WORD "W "RAD
SCRAMBLE outputs "WRAD

```

Removing a Letter from a Word

REMOVE takes two inputs, a letter and a word. It compares the input letter with each letter of the input word. When it finds a matching letter, it outputs the word with that letter removed.

WORDPLAY

```
?PRINT REMOVE "U "RECURSION
RECRSION
?
```

REMOVE works by comparing the input letter with the first letter of the input word. If they match, then the BUTFIRST of the word is the output we want. Otherwise, the output is formed by joining the first letter of the input word with the result of REMOVEing the input letter from the rest of the word.

```
TO REMOVE :LETTER :WORD
IF :LETTER=FIRST :WORD [OP BF :WORD]           Send back the rest.
OP WORD FIRST :WORD REMOVE :LETTER BF :WORD
END
```

Here is how the preceding example (using RECURSION as the word) happens.

```
REMOVE "U "RECURSION
REMOVE "U "ECURSION
REMOVE "U "CURSION
REMOVE "U "URSION
REMOVE outputs "RSION
REMOVE outputs "CRSION which is WORD "C "RSION
REMOVE outputs "ECSRION which is WORD "E "CRSION
REMOVE outputs "RECRSION which is WORD "R "ECSRION
```

The remaining procedures in this program are straightforward and won't be explained in detail. You can look at the program listing to see what they are.

SUGGESTIONS

Here are a few ideas for changing WORDSCRAM.

- Change the list of words it knows.
- Tell the player how many guesses it took to get the word.
- After the player guesses the word, ask if she or he would like to see the definition of the word. Since WORDSCRAM's words are technical Logo terms, this would be an interesting way to learn about Logo.
- Add some new messages.
- Do some psychology experiments. Some words look very strange when scrambled. Does this "strangeness" vary from person to person? Some people are better at unscrambling words than others. Why? What sort of strategy do you apply to unscrambling a word? Does it resemble other problem-solving strategies you use?

PROGRAM LISTING

```
TO WORDSCRAM
TS
CT
PR [WELCOME TO WORDSCRAM !]
PR []
PR [DO YOU WANT THE INSTRUCTIONS ?]
IF GETANSWER RC [INSTRUCTIONS] [PR []]
PLAYGAME WIN.MESSAGES GETWORDS
END
```

SEE IF THE USER WANTS INSTRUCTIONS

```
TO GETANSWER :ANS
IF :ANS = "Y" [TYPE "YES. OP "TRUE]
IF :ANS = "N" [TYPE "NO. OP "FALSE]
PR [PLEASE ANSWER WITH Y OR N]
OP GETANSWER RC
END

TO INSTRUCTIONS
TS CT
PR [SE [FROM A LIST OF] COUNT GETWORDS [LOGO WORDS, THE])
PR [COMPUTER WILL PICK ONE AT RANDOM AND]
PR [SCRAMBLE IT FOR YOU. YOUR JOB IS]
PR [TO UNSCRAMBLE IT.]
PR []
PR [IT IS NOT NECESSARY TO GUESS THE WORD]
PR [ON THE FIRST TRY. THE COMPUTER WILL]
PR [TELL YOU WHICH LETTERS YOU HAVE IN]
PR [THE RIGHT POSITION BY PRINTING A]
PR [STAR UNDER EACH CORRECT LETTER. A]
PR [LETTER IN THE WRONG POSITION WILL]
PR [HAVE A ? UNDER IT.]
PR []
PR [IF YOU ARE REALLY STUCK, TYPE HINT]
PR [FOR A HINT OR HELP TO SEE THE WORD.]
PR []
PR [GOOD LUCK.]
PR []
TYPE [PRESS ANY KEY TO START...]
MAKE "DUMMY RC
END
```

STARTING THE GAME PLAY

```
TO PLAYGAME :WIN.MESSAGES :WORDS
CT
PR [THINKING ...]
PLAYGAME1 RANPICK :WORDS
IF ANOTHER? [PLAYGAME :WIN.MESSAGES :WORDS] [PR []]
END
```

```

TO PLAYGAME1 :WORD
MAKE "SCRAMBLED SCRAMBLE :WORD
PR []
PR [OK. HERE IS YOUR SCRAMBLED WORD:]
PR :SCRAMBLED
MAKE "TOO.MANY.HINTS "FALSE
MAKE "GUESSED.WORDS SE FIRST :WORD []
GET
END

```

SCRAMBLING THE WORD

```

TO SCRAMBLE :WORD
IF :WORD = " [OP "
OP SCRAMBLE1 :WORD RANPICK :WORD
END

TO SCRAMBLE1 :WORD :LETTER
OP WORD :LETTER ( SCRAMBLE REMOVE :LETTER :WORD )
END

TO REMOVE :LETTER :WORD
IF :LETTER = FIRST :WORD [OUTPUT BF :WORD]
OUTPUT WORD FIRST :WORD REMOVE :LETTER BF :WORD
END

```

GETTING THE USER'S GUESS

```

TO GET
PR []
PR [WHAT'S YOUR GUESS ?]
IF ( NOT ( ROW < 23 ) ) [REFRESH.SCREEN]
GETGUESS FIRST RL
END

TO ROW
OP .EXAMINE 171
END

TO REFRESH.SCREEN
SAVE.CURSOR
REDISPLAY
RESTORE.CURSOR
END

TO GETGUESS :GUESS
IF EMPTYP :GUESS [OP GETGUESS FIRST RL]
IF :GUESS = "HELP [SHOW.WORD STOP]
IF :GUESS = "HINT [HINT LAST :GUESSED.WORDS GET STOP]
ADDGUESS :GUESS COMPARE :GUESS :WORD
IF :GUESS = :WORD [PR [] PR RANPICK :WIN.MESSAGES ►
STOP] [GET]
END

```

CHECK THE GUESS FOR CORRECT AND INCORRECT LETTERS

```

TO ADDGUESS :GUESS
MAKE "GUessed.WORDS LPUT :GUESS :GUessed.WORDS
END

TO COMPARE :GUESS :CORRECT
IF ( OR ( :GUESS = " " ) ( :CORRECT = " " ) ) [PR [] STOP]
IF ( ( FIRST :GUESS ) = ( FIRST :CORRECT ) ) ►
    [TYPE [*]] [TYPE ?]
COMPARE BF :GUESS BF :CORRECT
END

```

HINT AND HELP

```

TO HINT :G
IF ( OR ( :G = BL :WORD ) ( :G = ( BL BL :WORD ) ) ) ►
    [MAKE "TOO.MANY.HINTS "TRUE]
IF :TOO.MANY.HINTS [PR [YOU DON'T NEED A HINT!] ►
    PR [I THINK SOME MORE.] STOP]
TYPE [WELL]
DODOTS ( 1 + RANDOM 7 )
TYPE [OK]
DODOTS ( 1 + RANDOM 5 )
PR SE [TRY] HINTWORD1 :G :WORD
END

TO HINTWORD1 :W1 :W2
IF EMPTYP :W2 [OP []]
IF EMPTYP :W1 [OP FIRST :W2]
IF NOT EQUALP FIRST :W1 FIRST :W2 [OP FIRST :W2]
OP WORD FIRST :W2 HINTWORD1 BF :W1 BF :W2
END

TO DODOTS :N
IF :N = 0 [STOP]
WAIT 5
TYPE [.]
DODOTS :N - 1
END

TO SHOW.WORD
PR []
PR SE [THE WORD WAS] :WORD
END

```

MISCELLANEOUS PROCEDURES

```

TO ITEM :N :OBJECT
IF :N = 1 [OUTPUT FIRST :OBJECT]
OUTPUT ITEM :N - 1 BF :OBJECT
END

```

```

TO RESTORE.CURSOR
SETCURSOR :CURSOR
END

TO REDISPLAY
SETCURSOR [0 0]
PR [THINKING.....]
PR []
PR [OK. HERE IS YOUR SCRAMBLED WORD:]
PR :SCRAMBLED
PR []
END

TO SAVE.CURSOR
PR []
MAKE "CURSOR LIST ( .EXAMINE 172 ) - 1 ( .EXAMINE 171 ) - 1
END

TO ANOTHER?
PR []
PR [DO YOU WANT ANOTHER WORD ?]
OP GETANSWER RC
END

TO RANPICK :L
OP ITEM ( 1 + RANDOM COUNT :L ) :L
END

TO GETWORDS
OP [CATALOG TURTLE FORWARD BACK LEFT RIGHT PROCEDURE      ▶
    INPUT RECURSION SETBG CIRCLE SQUARE LOGO GRAPHICS      ▶
    EDIT REPEAT POTS DEFINE COUNT HEADING MEMBERP NODES      ▶
    PADDLE DYNATURTLE INSTANT BUTFIRST PENDOWN PENUP      ▶
    PRODUCT RANDOM SETCURSOR SETPC WINDOW TOUCHING MAKE      ▶
    BUTLAST OUTPUT HIDETURTLE SQRT]
END

TO WIN.MESSAGES
OP [[HEY, YOU'RE PRETTY SMART] [WHAT A FLUKE!]      ▶
    [WE ALL GET LUCKY ONCE IN A WHILE!]      ▶
    [A GOLD STAR FOR YOU] [1 POINT FOR YOU!]      ▶
    [DOING GREAT!!] [KEEP UP THE GOOD WORK...]]
END

```

Madlibs™

This project plays the game of Madlibs.* The program asks for words or phrases with which to fill in the blanks in an already-prepared story. Then it prints the resulting story.

*"Madlibs" is a trademark of Price/Stern/Sloan.

By Brian Harvey; story template by Susan Cotten.

Here is an example of a story to be used with the program.

AT _____, _____ WAS _____ING DOWN THE
time of day person way to move

STREET. _____ SPOTTED A _____ DIGGING IN A
person animal

GARBAGE CAN ACROSS THE STREET. _____ BEGAN TO
person

_____ IN THE OPPOSITE DIRECTION BUT IT WAS TOO LATE.
way to move

THE _____ SAW _____. IT BEGAN TO CHASE
animal person

_____. _____ TRIPPED AND FELL. THE
person person

_____ CAME UP BESIDE _____. AND BEGAN TO WAG ITS
animal person

_____. _____ REALIZED THERE WAS NOTHING TO
body part person

FEAR. _____ REACHED OUT AND PATTED THE _____.
person animal

Here is what happens when you use the program with this story:

?MADLIB :STORY1
TELL ME A TIME OF DAY
DUSK
TELL ME A PERSON'S NAME
URSULA
TELL ME A WAY TO MOVE
JUMP
TELL ME AN ANIMAL YOU FEAR
RAT
TELL ME A BODY PART
TOE
AT DUSK, URSULA WAS JUMPING DOWN THE STREET.
URSULA SPOTTED A RAT DIGGING IN A GARBAGE CAN ACROSS
THE STREET. URSULA BEGAN TO JUMP IN THE OPPOSITE
DIRECTION BUT IT WAS TOO LATE. THE RAT SAW URSULA. IT
BEGAN TO CHASE URSULA. URSULA TRIPPED AND FELL. THE
RAT CAME UP BESIDE URSULA AND BEGAN TO WAG ITS TOE.
URSULA REALIZED THAT THERE WAS NOTHING TO FEAR. URSULA
REACHED OUT AND PATTED THE RAT.
?

How a Story Is Represented

A story is represented as a list that contains words and lists (which we'll refer to as *sublists*). The sublists are the blanks of the story. Here is the list that represents the preceding example.

```
MAKE "STORY1 [AT * [HOUR TIME OF DAY]
, [PERSON PERSON'S NAME]
WAS * [MOTION WAY TO MOVE] ING DOWN THE STREET. [PERSON]
SPOTTED A [ANIMAL ANIMAL YOU FEAR] DIGGING IN A GARBAGE CAN
ACROSS THE STREET. [PERSON] BEGAN TO [MOTION] IN THE
OPPOSITE DIRECTION BUT IT WAS TOO LATE. THE [ANIMAL] SAW
* [PERSON] . IT BEGAN TO CHASE * [PERSON] . [PERSON] TRIPPED
AND FELL. THE [ANIMAL] CAME UP BESIDE [PERSON] AND BEGAN
TO WAG ITS * [ANATOMY BODY PART] . [PERSON] REALIZED THERE
WAS NOTHING TO FEAR. [PERSON] REACHED OUT AND PATTED
THE * [ANIMAL] .]
```

Each word or phrase that the user types to replace a blank is given a *name*, so that the program is able to remember it. The named phrase can be used to fill more than one blank. The sublist

[MOTION WAY TO MOVE]

signals the program to type

TELL ME A WAY TO MOVE

and to give what the user types the name MOTION. Later the sublist [MOTION] appears in :STORY1 without the prompting phrase WAY TO MOVE. This signals the program to fill the blank with the word or phrase named MOTION, without asking for a new motion.

The Procedures

The top-level procedure is MADLIB.

```
TO MADLIB :STORY
PRINT FILL.IN :STORY
END
```

MADLIB invokes FILL.IN and prints its output, which is a story list with the blanks filled in.

The job of FILL.IN is to go through the story list, one element at a time. If an element is a word, that word itself should be part of the output. If the element is a list, it has to fill a blank. Here is the procedure.

```
TO FILL.IN :STORY
IF EMPTYP :STORY [OP []]
IF WORDP FIRST :STORY
  [OP FPUT FIRST :STORY FILL.IN BF :STORY]
IF NOT EMPTYP BF FIRST :STORY [FILL.BLANK FIRST :STORY]
  OP SE THING FIRST FIRST :STORY FILL.IN BF :STORY
END
```

This procedure has the overall structure of a recursive operation that does something to every element of a list.

The first instruction is the end test for the input list being empty.

The next line checks for the case in which the first element of the list is a word. In that case, we want to put the word itself in the output.

If the first element isn't a word, it's a blank to be filled. There are two cases. If the list contains more than one word, like [MOTION WAY TO MOVE], that means that the user must be asked for a WAY TO MOVE to fill the blank. The name for what the user types is the first word of the list, MOTION. FILL.BLANK handles this interaction.

```
?SHOW FILL.IN [ [MOTION WAY TO MOVE] QUICKLY! ]
TELL ME A WAY TO MOVE
PERAMBULATE
[PERAMBULATE QUICKLY!]
?
```

If the first element is a list that has only one word, like [MOTION], then we use the word or phrase that was remembered under that name.

```
?SHOW FILL.IN [HELLO, [PERSON NAME];HOW IS [PERSON] TODAY?]
TELL ME A NAME
JONATHAN
[HELLO, JONATHAN ; HOW IS JONATHAN TODAY?]
?
```

The last line of FILL.IN provides the output for both kinds of sublists.

Filling Blanks by Asking Questions

FILL.BLANK has two tasks: it asks the user for a word or phrase, and it gives what the user types a name.

```
TO FILL.BLANK :BLANK
PR SE [TELL ME] ARTICLE BF :BLANK
MAKE FIRST :BLANK RL
END
```

By the way, this is a good example of the use of MAKE with a first input that is not a quoted word. The name of the variable we want to set is part of the story list and does not appear in the text of the procedure.

An elegant detail of FILL.BLANK is that it figures out whether to use A or AN in prompting for a word or phrase. Here is the subprocedure that does the figuring.

```
TO ARTICLE :PROMPT
IF VOWELP FIRST FIRST :PROMPT [OP SE "AN :PROMPT"]
OP SE "A :PROMPT
END

TO VOWELP :LETTER
OP MEMBERP :LETTER [A E I O U]
END
```

Handling Punctuation

If a blank to be filled is the last thing in a sentence in the story, there is the problem of putting a punctuation mark at the end, without making it a separate word. For example, in our story we have a sentence that ends

SAW [PERSON] .

If the variable PERSON contains the word URSULA, we'd like the finished story to end

SAW URSULA.

But if we don't treat this as a special case, the period will be a word by itself:

SAW URSULA .

The solution I chose is to use an asterisk in the story to mean "take the next two elements in the list and combine them as one word." That's a slight simplification, though, because the next element may be an entire phrase, and only the last word of the phrase can be combined with the punctuation character that follows. The procedure that does the combining is this.

```
TO PUNCTUATE :STUFF :PUNCT
IF WORDP :STUFF [OP WORD :STUFF :PUNCT]
OP SE BL :STUFF WORD LAST :STUFF :PUNCT
END
```

Here is a revised version of FILL.IN that uses PUNCTUATE.

```
TO FILL.IN :STORY
IF EMPTYP :STORY [OP []]
IF EQUALP FIRST :STORY ** [OP SE (PUNCTUATE FILL.IN
    (FPUT FIRST BF :STORY []) FIRST BF BF :STORY)
    FILL.IN BF BF BF :STORY]
IF WORDP FIRST :STORY
    [OP FPUT FIRST :STORY FILL.IN BF :STORY]
IF NOT EMPTYP BF FIRST :STORY [FILL.BLANK FIRST :STORY]
OP SE THING FIRST FIRST :STORY FILL.IN BF :STORY
END
```

PROGRAM LISTING

TO MADLIB :STORY	[FILL.BLANK FIRST :STORY]
PRINT FILL.IN :STORY	OP SE THING FIRST FIRST :STORY FILL.IN ▶
END	BF :STORY
END	
TO FILL.IN :STORY	TO FILL.BLANK :BLANK
IF EMPTYP :STORY [OP []]	PR SE [TELL ME] ARTICLE BF :BLANK
IF EQUALP FIRST :STORY ** [OP SE ▶	MAKE FIRST :BLANK RL
(PUNCTUATE FILL.IN (FPUT FIRST BF ▶	END
:STORY []) FIRST BF BF :STORY) ▶	
FILL.IN BF BF BF :STORY]	
IF WORDP FIRST :STORY [OP FPUT FIRST ▶	TO VOWELP :LETTER
:STORY FILL.IN BF :STORY]	OP MEMBERP :LETTER [A E I O U]
IF NOT EMPTYP BF FIRST :STORY ▶	END

```
TO ARTICLE :PROMPT
IF VOWELP FIRST FIRST :PROMPT [OP SE ▶
    "AN :PROMPT]
OP SE "A :PROMPT
END

TO PUNCTUATE :STUFF :PUNCT
IF WORDP :STUFF [OP WORD :STUFF ▶
    :PUNCT]
OP SE BL :STUFF WORD LAST :STUFF ▶
    :PUNCT
END

MAKE "STORY1 [AT * [HOUR TIME OF DAY] ▶
    , [PERSON PERSON'S NAME] WAS * ▶
```

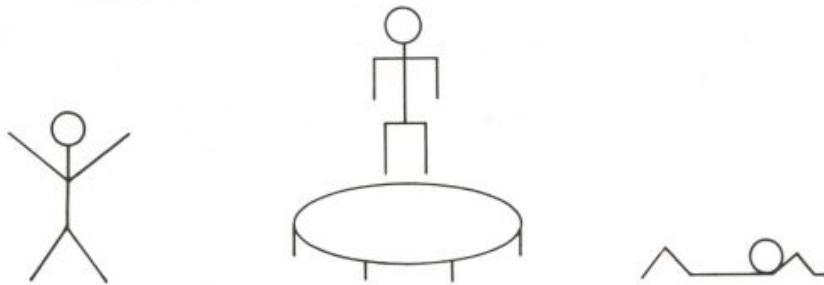
```
[MOTION WAY TO MOVE] ING DOWN THE ▶
STREET. [PERSON] SPOTTED A ▶
[ANIMAL ANIMAL YOU FEAR] DIGGING ▶
IN A GARBAGE CAN ACROSS THE ▶
STREET. [PERSON] BEGAN TO ▶
[MOTION] IN THE OPPOSITE ▶
DIRECTION BUT IT WAS TOO LATE. ▶
THE [ANIMAL] SAW * [PERSON] . IT ▶
BEGAN TO CHASE * [PERSON] . ▶
[PERSON] TRIPPED AND FELL. THE ▶
[ANIMAL] CAME UP BESIDE [PERSON] ▶
AND BEGAN TO WAG ITS * [ANATOMY ▶
BODY PART] . [PERSON] REALIZED ▶
THERE WAS NOTHING TO FEAR. ▶
[PERSON] REACHED OUT AND PATTED ▶
THE * [ANIMAL] .]
```

2

Stories

Exercise

In this project you watch a stick figure go through exercises consisting of jumping jacks and jumping on a trampoline. In the end, the figure collapses from exhaustion.



The idea for this animated story came about when I was playing with the shape editor. I wanted a stick figure to perform a jumping jack. I divided the jumping jack exercise into two parts. In the first part, the figure has its arms extended upward and legs apart; in the second part, the figure has its arms at its side and legs together. I designed the following two shapes in the shape editor to represent these stances.



Next I wrote a short procedure to see what these shapes would look like when put together.

```
TO TEST
SETSH 1
WAIT 5
SETSH 2
WAIT 5
TEST
END
```

By Susan Cotten.

The movement was jerky. I made a third shape, which had its arms extended downward and its legs apart. I added this shape to TEST. The three shapes, interspersed with WAIT, made for a fairly smooth jumping jack.

Here is the sequence of shapes that I use for the jumping jack.



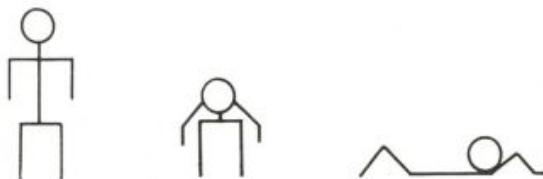
I began to think of other things I could do with my figure. What about jumping on a trampoline? I sketched a trampoline. I saw that the trampoline bed could be drawn using two arcs. After I'd figured out how to draw the trampoline, I put the first stick figure in the sequence on it and wrote a procedure to make the figure jump up and down on the trampoline. I also added a TOOT that sounded a *boing* whenever the figure bounced on the trampoline.

I then made a shape for the turtle to wear while getting up on the trampoline.

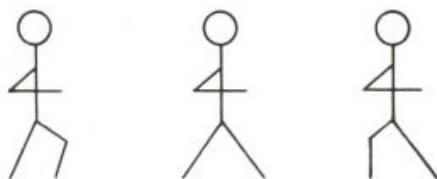


What next? I recalled how I felt after a good workout. It might be interesting to make the figure collapse on the floor. The figure would begin this sequence standing upright, then would bend over and collapse.

Here are the shapes.



I hadn't yet written the procedures to connect the exercises, but my general plan was to have the figure walk from sequence to sequence. These are the three shapes that I used for the walking motion.



STORIES

I used nine shapes in my story. I named these shapes FIGURE1, FIGURE2, and so forth. The shapes as seen in the editor and the list of numbers describing them are printed at the end of this write-up.

Putting It All Together

Here are the procedures that make up EXERCISE. They are listed in order of use.

The top-level procedure in this program is EXERCISE.

```
TO EXERCISE
SETUP
JUMPING.JACKS
MOVE.TO.TRAMPOLINE
JUMPING.ON.TRAMPOLINE
GET.OFF.TRAMPOLINE
COLLAPSE
END
```

SETUP readies the screen and turtle for the first exercise sequence.

```
TO SETUP
TELL 0
CS HT FS
SETBG 72
READY.SHAPES
DRAW.TRAMPOLINE
READY.JUMPING.JACKS
END
```

READY.SHAPES takes care of putting the shapes into the shape table.

```
TO READY.SHAPES
PUTSH 1 :FIGURE1
PUTSH 2 :FIGURE2
PUTSH 3 :FIGURE3
PUTSH 4 :FIGURE4
PUTSH 5 :FIGURE5
PUTSH 6 :FIGURE6
PUTSH 7 :FIGURE7
PUTSH 8 :FIGURE8
PUTSH 9 :FIGURE9
END
```

DRAW.TRAMPOLINE draws the trampoline in the middle of the screen.

```
TO DRAW.TRAMPOLINE
SETPN 0
SETPC 0 37
TRAMP.BED
TRAMP.LEGS
END
```

```
TO TRAMP.BED
```

```
PD
```

```
RT 45
```

```
ARCR 8 5
```

```
RT 90
```

```
ARCR 8 5
```

```
SETH 0
```

```
END
```

```
TO ARCR :STEPS :TIMES
```

```
REPEAT :TIMES [RT 9 FD :STEPS RT 9]
```

```
END
```

```
TO TRAMP.LEGS
```

```
FRONT.LEG
```

```
SETPOS [8.67 -5]
```

```
BACK.LEG
```

```
SETPOS [28.67 -5]
```

```
BACK.LEG
```

```
SETPOS [36.67 0]
```

```
FRONT.LEG
```

```
END
```

```
TO FRONT.LEG
```

```
PD
```

```
BK 8
```

```
FD 8
```

```
PU
```

```
END
```

```
TO BACK.LEG
```

```
PD
```

```
BK 3
```

```
FD 3
```

```
PU
```

```
END
```

READY.JUMPING.JACKS moves the turtle over to the left side of the screen and sets its shape for the first exercise.

```
TO READY.JUMPING.JACKS
```

```
PU SETPOS [-60 0] PD
```

```
SETH 3 SETC 7 ST
```

```
END
```



The figure performs six jumping jacks. Whenever the figure touches the ground, Logo makes a sound, a TOOT. At the end of this exercise, the turtle sets its shape to the standing figure.

```
TO JUMPING.JACKS
```

```
REPEAT 6 [JUMPING.JACKS1]
```

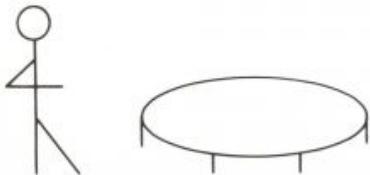
```
SETH 3
```

```
END
```

STORIES

```
TO JUMPING.JACK$1
SETSH 3 WAIT 20
SETSH 2 TOOT 0 80 10 5 WAIT 20
SETSH 1 WAIT 20
END
```

MOVE.TO.TRAMPOLINE takes care of moving the turtle over to and then up on the trampoline.



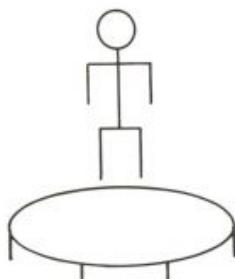
```
TO MOVE.TO.TRAMPOLINE
WALK 10
SETSH 7
SETH 45 FD 12
SETH 90 FD 20
SETH 0 BK 2
SETSH 3
END
```

The input to WALK tells it how far to move, that is, how many times to repeat USE.SHAPES.WALK.

```
TO WALK :DISTANCE
PU SETH 90
REPEAT :DISTANCE [USE.SHAPES.WALK]
END
```

```
TO USE.SHAPES.WALK
SETSH 4 WAIT 5 FD 5
SETSH 6 WAIT 5
SETSH 5 WAIT 5
SETSH 6 TOOT 0 200 15 2 WAIT 5
END
```

The figure is now standing on the trampoline.
JUMPING.ON.TRAMPOLINE makes the figure jump up and down by running JUMP fifteen times.



```
TO JUMPING.ON.TRAMPOLINE
REPEAT 15 [JUMP]
END

TO JUMP
FD 15 WAIT 2
BK 15 WAIT 10
TOOT 0 80 10 5 WAIT 12
END
```

At this point the figure is through exercising. GET.OFF.TRAMPOLINE moves the figure down off the trampoline and then walks it a short distance.



```
TO GET.OFF.TRAMPOLINE
WALK 3
SETH 0
BK 10
WALK 5
END
```

The figure stops and lets out a musical sigh as it collapses on the floor. COLLAPSE relies on SIGH to do the work. SIGH is given a pitch to play and a list of shapes.

```

TO COLLAPSE
WAIT 35
SIGH 150 [3 3 3 8 8 8 9 9 9 9]
END

TO SIGH :NOTE :LIST
IF EMPTYP :LIST [STOP]
SETSH FIRST :LIST
TOOT 0 :NOTE 15 5
WAIT 5
SIGH :NOTE - 8 BF :LIST
END

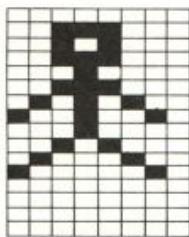
```



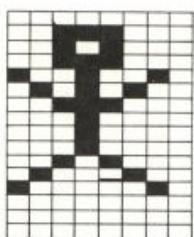
PROGRAM LISTING

<pre> TO EXERCISE SETUP JUMPING.JACKS MOVE.TO.TRAMPOLINE JUMPING.ON.TRAMPOLINE GET.OFF.TRAMPOLINE COLLAPSE END TO SETUP TELL 0 CS HT FS SETBG 72 READY.SHAPES DRAW.TRAMPOLINE READY.JUMPING.JACKS END TO READY.SHAPES PUTSH 1 :FIGURE1 PUTSH 2 :FIGURE2 PUTSH 3 :FIGURE3 PUTSH 4 :FIGURE4 PUTSH 5 :FIGURE5 PUTSH 6 :FIGURE6 PUTSH 7 :FIGURE7 PUTSH 8 :FIGURE8 PUTSH 9 :FIGURE9 END </pre>	<pre> TO DRAW.TRAMPOLINE SETPN 0 SETPC 0 37 TRAMP.BED TRAMP.LEGS END TO TRAMP.BED PD RT 45 ARCR 8 5 RT 90 ARCR 8 5 SETH 0 END TO ARCR :STEPS :TIMES REPEAT :TIMES [RT 9 FD :STEPS RT 9] END TO TRAMP.LEGS FRONT.LEG SETPOS [8.67 -5] BACK.LEG SETPOS [28.67 -5] BACK.LEG SETPOS [36.67 0] FRONT.LEG END </pre>
---	---

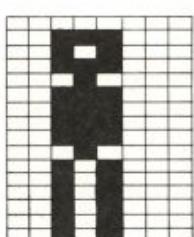
SHAPES



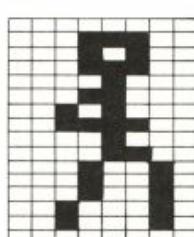
:FIGURE1



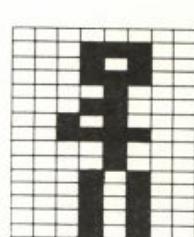
:FIGURE2



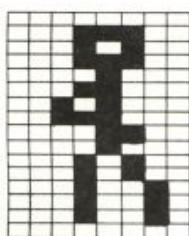
:FIGURE3



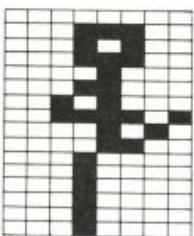
:FIGURE4



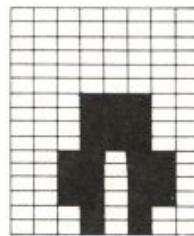
:FIGURE5



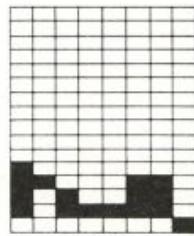
:FIGURE6



:FIGURE7



:FIGURE8



:FIGURE9

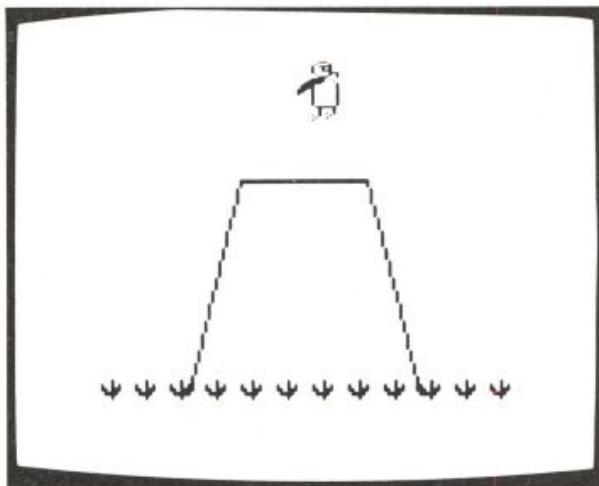
Cartoon

In CARTOON a bird flies around chirping, flapping its wings, and fertilizing the world. Eventually it settles on a mountaintop and lays an egg, whereupon the bird flies off forever. The egg hatches into a little bird, which also flies away.

This project is one that I developed over a long period of time. I started out wanting to explore graphics and dynamics with turtles. As I began to sketch out ideas in the shape editor, I stumbled onto a design that looked like a bird's head and upper body. I used a cartooning technique of drawing in outline a side view of the bird. As soon as the bird emerged as the central character, the rest of the project began to suggest itself. The overall goal became clear: to make a bird and then make it fly.

In the following discussion I try to convey the process I went through in developing this project. If you want to look at the completed program, turn to the listing at the end.

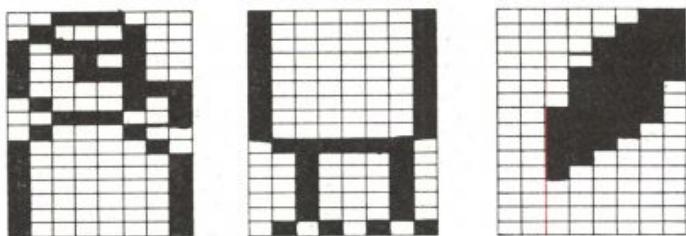
By Erric Solomon.



Making the Bird

Making the bird was a long process of sketching the parts in the shape editor and then testing them out in relation to the other parts. I used one shape for the bird's head and the top of its body and another shape for the bottom of the body and the legs. I also used a separate shape for the wing. Since I decided on a side view, only one wing was needed. I put the first wing shape in shape 3 and used shape 1 for the top of the bird and shape 2 for the bottom of the bird. I chose a yellowish color for the outline of the bird and picked a purple shade for the wing, which I made as a solid figure.

Here is the way these shapes look in the shape editor.



I named the shapes TOPBIRD, BOTTOMBIRD, and WING1.

```
MAKE "TOPBIRD GETSH 1
MAKE "BOTTOMBIRD GETSH 2
MAKE "WING1 GETSH 3
```

I set up the bird giving each of the three turtles a shape and a position. These instructions make up the procedure BIRD.



```

TO BIRD
PUTSH 1 :TOPBIRD
PUTSH 2 :BOTTOMBIRD
PUTSH 3 :WING1
TELL 0 SETPOS [-9 -10] SETSH 3 SETC 53
TELL 1 SETPOS [0 0] SETSH 1 SETC 12
TELL 2 SETY -15 SETSH 2 SETC 12
TELL [0 1 2] ST
END

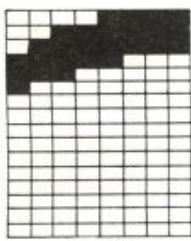
```

Turtle 0 is :WING1 (shape 3), turtle 1 is :TOPBIRD (shape 1), and turtle 2 is :BOTTOMBIRD (shape 2).

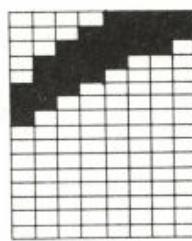
I used a trick here; turtle 0 assumes the role of the wing so that it is visible over the bird's body. (Lower-numbered turtles cover higher-numbered ones.)

Animating the Bird

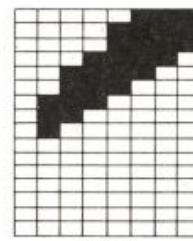
I thought I would make several wing shapes to represent different flapping positions. Making the wings took a lot of experimentation; I made several until I had three that were satisfactory. Here are those three shapes.



:WING2



:WING3



:WING4

I wrote MOVEWING to change the wing by putting new shapes in slot 3 as they were needed for the animation. The input to MOVEWING indicates how long Logo waits before the shape is changed.

```

TO MOVEWING :DELAY
PUTSH 3 :WING1 WAIT :DELAY
PUTSH 3 :WING4 WAIT :DELAY
PUTSH 3 :WING3 WAIT :DELAY
PUTSH 3 :WING2 WAIT :DELAY
PUTSH 3 :WING3 WAIT :DELAY
PUTSH 3 :WING4 WAIT :DELAY
END

```

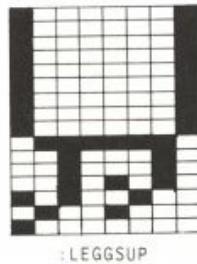
STORIES

At first I was going to put all the wings in the shape table at the same time, but I had made so many shapes I exhausted the shape table slots. Instead I used only slot 3.

I coordinated the speed the bird travels with its wing movement and finally settled on the following:

```
BIRD
SETH 90 SETSP 5
REPEAT 30 [MOVEWING 5]
```

Although I was pleased with the way the wings looked, I didn't like the bird's legs. I wanted them to curl up for flying. So I drew a new bird bottom with the legs up, which I referred to as LEGGSUP.

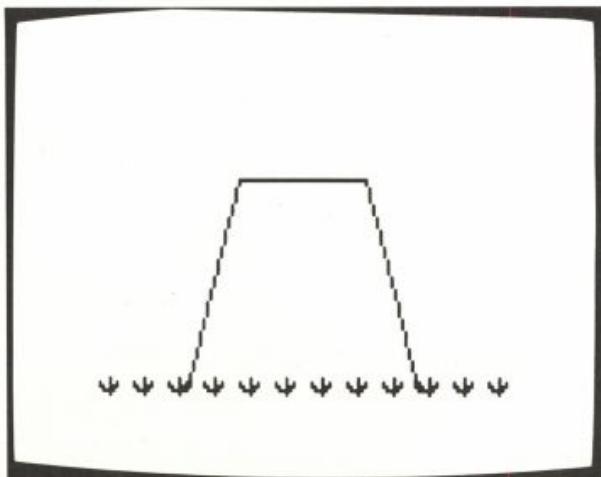


The Scenery

The bird needed a place to fly from, so I made a mountain. In the process I decided to add clumps of grass at the bottom of the mountain.

```
TO MOUNTAIN
PU SETPOS [-65 -90]
PD SETPOS [-35 37]
RT 90 FD 70
SETPOS [65 -90]
END
```

The grass was made in clumps.



```

TO GRASS
SETH -60 FD 5 BK 5
RT 30 FD 8 BK 8
RT 30 FD 10 BK 10
RT 30 FD 8 BK 8
RT 30 FD 5 BK 5
END

```

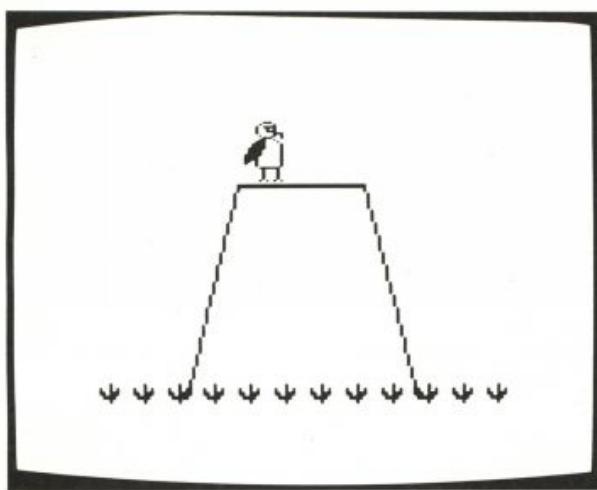
I used all four turtles to draw the grass so that this drawing takes place faster.

```

TO DRAWGRASS
TELL [0 1 2 3] SETPN 0 PU
ASK 0 [SETPOS [-110 -90]]
ASK 1 [SETPOS [-90 -90]]
ASK 2 [SETPOS [-70 -90]]
ASK 3 [SETPOS [-50 -90]]
REPEAT 3 [PD GRASS SETH 90 PU FD 80]
END

```

I then changed BIRD so that the bird perched on the mountaintop.



```

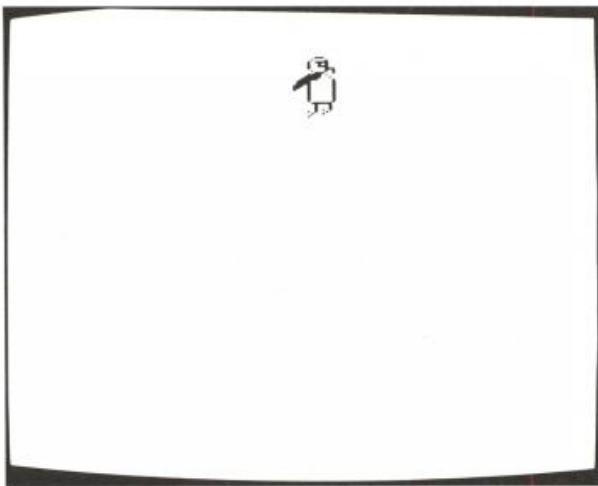
TO BIRD
PUTSH 1 :TOPBIRD
PUTSH 2 :BOTTOMBIRD
PUTSH 3 :WING1
TELL 0 SETPOS [-9 -10] SETSH 3 SETC 53
TELL 1 SETPOS [0 0] SETSH 1 SETC 12
TELL 2 SETY -15 SETSH 2 SETC 12
TELL [0 1 2]
EACH [SETPOS SE XCOR - 20 YCOR + 65]
ST
END

```

Making the Bird Fly

After getting the bird to sit on the mountain, I wanted it to take off, fly around, and then land. To produce the animated portion of the story, I anticipated needing several procedures like TAKEOFF, FLAP, and LANDING.

TAKEOFF adjusts the bird's heading and speed for its ascent and leveling off. In the process the bird's legs are raised and its wing flaps.



```
TO TAKEOFF
PUTSH 2 :LEGGSUP
SETH 45
SETSP 5 FLAPFLAP 5 5
SETSP 10 SETH 67
FLAPFLAP 10 5
SETH 90
END
```

FLAPFLAP calls MOVEWING.

```
TO FLAPFLAP :TIMES :DELAY
REPEAT :TIMES [MOVEWING :DELAY]
END
```

(Note that the bigger :DELAY is, the slower the wing flaps.)

Assembling CARTOON

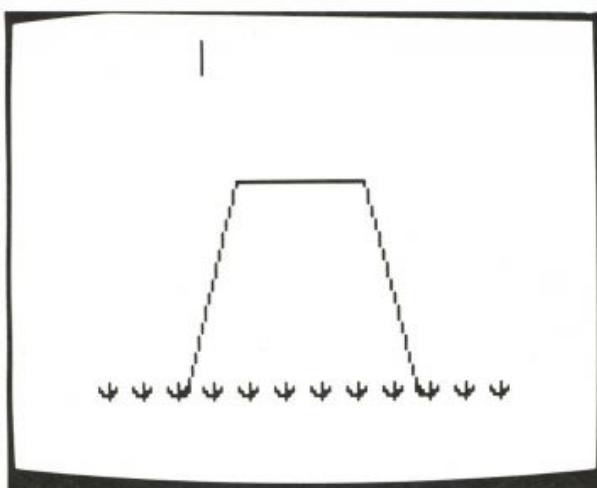
I put all these instructions into a procedure.

```
TO CARTOON
  TELL [0 1 2 3] HT CS FS
  SETBG 0 SETPN 0 SETPC 0 6
  MOUNTAIN
  DRAWGRASS
  BIRD
  TAKEOFF
END
```

Landing the Bird

Once I got the bird to take off, I wanted to make it land. I had to figure out a way to invoke the landing process and then coordinate the bird's heading so that it would land on the mountain. I also had to find a way to stop the bird once it reached the mountaintop.

An obvious way to stop the bird once it landed on the mountain was to set up a demon to watch for turtle 2 colliding with the mountaintop drawn by pen 0. A much harder problem was figuring out how to start the descent. I wanted to use a demon, but I wasn't sure how. I finally happened upon a technique that I found extremely useful in other parts of this project. I made a wall in the sky by drawing a vertical line above the mountain in the background color. Turtle 2 would inevitably collide with this line. I set up a demon to watch for this event and then invoke the landing.



SET.MARKS draws the line. I put SET.MARKS in CARTOON directly after MOUNTAIN draws the mountain using turtle 0 with pen 0. This mark is drawn by turtle 0 with pen 1.

STORIES

```
TO SET.MARKS
SETPN 1 SETPC 1 BG
PU SETPOS [-55 100]
PD SETY 120 PU
END
```

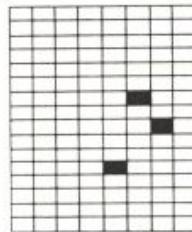
I put the WHEN instruction in CARTOON immediately following DRAWGRASS.
WHEN OVER 2 1 [LANDING]

LANDING changes the bird's flight direction and sets up a demon to look for a collision between the bird and the mountain.

```
TO LANDING
TELL [0 1 2 3] SETH 130
PUTSH 3 :WING1
WHEN OVER 2 0 [ PUTSH 2 :BOTTOOMBIRD SETSP 0]
END
```

Fertilizing the World

While birds fly around they also drop sprinklings of digested matter. This bird is no different. I designed TURD and put it in shape 4.



I wanted TURD to suddenly appear and fall to the ground as the bird continues its flight. I used turtle 3 for this role. Turtle 3 would travel with turtle 2, but invisibly. To camouflage turtle 3, I set its color to the background color until the proper moment. I changed BIRD:

```
TO BIRD
PUTSH 1 :TOPBIRD
PUTSH 2 :BOTTOOMBIRD
PUTSH 3 :WING1
PUTSH 4 :TURD
TELL 0 SETPOS [-9 -10] SETSH 3 SETC 53
TELL 1 SETPOS [0 0] SETSH 1 SETC 12
TELL 2 SETY -15 SETSH 2 SETC 12
TELL 3 SETPOS [-9 -15] SETSH 4 SETC BG
TELL [0 1 2 3]
EACH [SETPOS SE XCOR - 20 YCOR + 65]
ST
END
```

Here is the path I planned for the fertilizer.

```
TELL 3 SETH 180 SETC 7
SETSP 25 ST
```

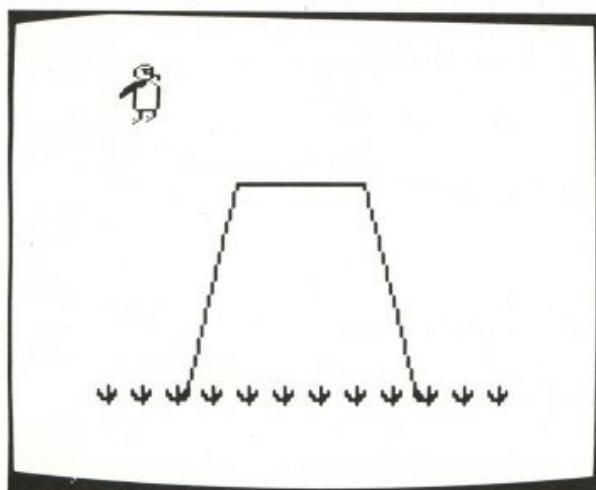
I made one addition; I added sound effects to attract the viewer's attention. I then redirected the turtle commands to the bird turtles.

```
TOOT 1 30 7 20
TELL [0 1 2]
```

I put these instructions in a new procedure called `DROP.TURD`.

I had to face a couple of problems: when would the bird release the fertilizer and how would turtle 3 know when it hit the ground and so stop?

I concentrated on how to activate `DROP.TURD`. I wanted the bird to take off and fly for a while before its musical gift descended to earth. The bird would perform this feat after crossing the screen once.



I used the same technique I used for `LANDING`. To do this, I drew an invisible wall on the screen running vertically at the place where I wanted turtle 3 to begin its visible descent. I used pen 2 to draw the line. The pencolor was the same as the background color; it was invisible to the viewer, but not to a turtle or a demon. I changed `SET.MARKS`.

```
TO SET.MARKS
TELL 0
SETPN 1 SETPC 1 BG
PU SETPOS [-55 100]
PD SETY 120 PU
SETPN 2 SETPC 2 BG
SETPOS [-50 100]
PD SETY 120 PU
END
```

I set up a demon to watch for turtle 2 (`:LEGGSUP`) colliding with this line.

```
WHEN OVER 2 2 [DROP.TURD]
```

STORIES

I had another problem. When does this demon get activated? I wanted this event to happen before the bird starts its landing.

I changed the WHEN instruction so that the demon invokes DROPANDLAND instead of DROP.TURD.

```
WHEN OVER 2 2 [DROPANDLAND]
```

DROPANDLAND dismisses the currently active demon, runs DROP.TURD, and sets up a new demon to invoke LANDING.

```
TO DROPANDLAND
WHEN OVER 2 2 []
DROP.TURD
WHEN OVER 2 1 [LANDING]
END
```

I used the same trick for making turtle 3 as TURD disappear. This time I drew an invisible line at the bottom of the mountain in the background color. Since the visible scenery was drawn with pen 0, I decided to use pen 2 for the line drawn in the background color.

```
SETPN 2 SETPC 2 BG
PU SETPOS [110 -90]
PD SETX -110 PU
```

I added these instructions to SET.MARKS.

I set up a demon to watch for the collision between turtle 3 and pen line 2. The demon invokes a procedure to make the turtle vanish.

Now DROP.TURD looks like this:

```
TO DROP.TURD
WHEN OVER 3 2 [DISAPPEAR]
TELL 3 SETH 180 SETC 7
SETSP 25 ST
TOOT 1 30 7 20
TELL [0 1 2]
END

TO DISAPPEAR
ASK 3 [SETSP 0 HT]
WHEN OVER 3 2 []
END
```

CARTOON *Updated*

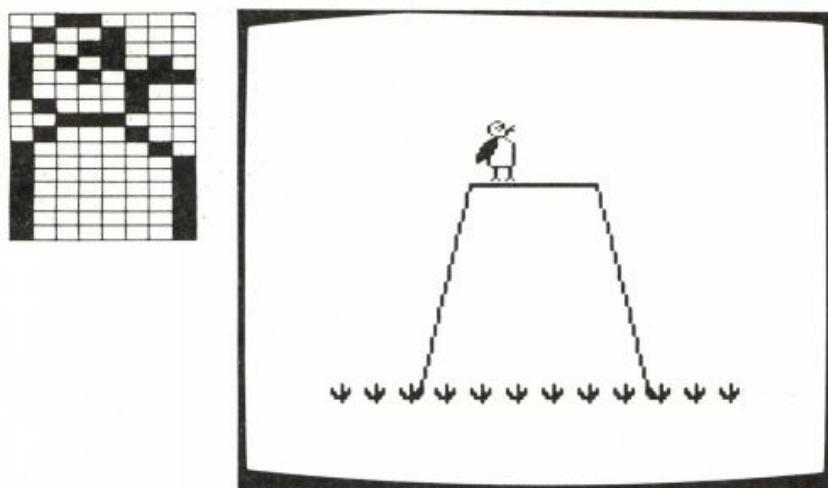
```
TO CARTOON
TELL [0 1 2 3] HT CS FS
SETBG 0 SETPN 0 SETPC 0 6
MOUNTAIN
SET.MARKS
DRAWGRASS
WHEN OVER 2 2 [DROPANDLAND]
RECYCLE
BIRD
TAKEOFF
END
```

Additional Sound Effects and Graphics

I liked the sound effects in DROP.TURD and decided that the bird should sing before taking off in flight. To do this the bird needed a tune.

```
TO SONG
TOOT @ 1000 7 7
TOOT @ 1100 7 7
TOOT @ 1150 7 7
TOOT @ 1050 7 7
END
```

When birds sing, they open and close their mouths. This action required another shape for the bird.



I refer to this shape as TOPBIRD2. SING alternates TOPBIRD with TOPBIRD2 as it calls SONG.

```
TO SING :TIMES
REPEAT :TIMES [PUTSH 1 :TOPBIRD2 SONG
  PUTSH 1 :TOPBIRD WAIT 20 + RANDOM 30]
END
```

FLY incorporated SING with the bird in flight.

```
TO FLY
TELL [@ 1 2 3]
WAIT 60 SING 5 WAIT 60
TAKEOFF SETSP 15
FLAP 7
WAIT 30 SING 5 WAIT 30
END
```

STORIES

FLY uses TAKEOFF and also FLAP. FLAP stops when the bird lands on the mountain.

```
TO FLAP :DELAY
MOVEWING :DELAY
IF COND OVER 2 0 [STOP]
FLAP :DELAY
END
```

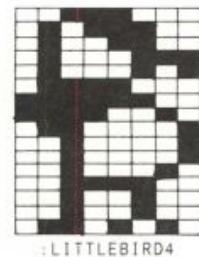
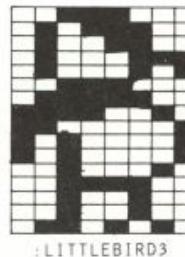
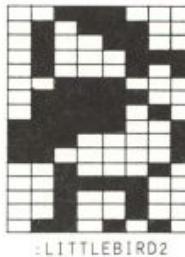
I put FLY in CARTOON in place of TAKEOFF.

More Frills

The bird lays an egg and flies away. The egg hatches and a little bird flies off. First I made a little bird. This bird fits in one turtle shape.



Since I had already made one bird fly, this one was not a problem. I made a total of four shapes for this little bird to show it in various stages of flight.



I used the same technique for this animation as I did for the wing flapping for the big bird.

```
TO FLIP
PUTSH 4 :LITTLEBIRD1 WAIT 10
PUTSH 4 :LITTLEBIRD2 WAIT 10
PUTSH 4 :LITTLEBIRD3 WAIT 10
PUTSH 4 :LITTLEBIRD4 WAIT 8.5
PUTSH 4 :LITTLEBIRD3 WAIT 10
PUTSH 4 :LITTLEBIRD2 WAIT 10
END
```

The bird shapes are put in shape 4 in the shape table. Turtle 3 would now be the little bird.*

FLYOFF makes the little bird fly away.

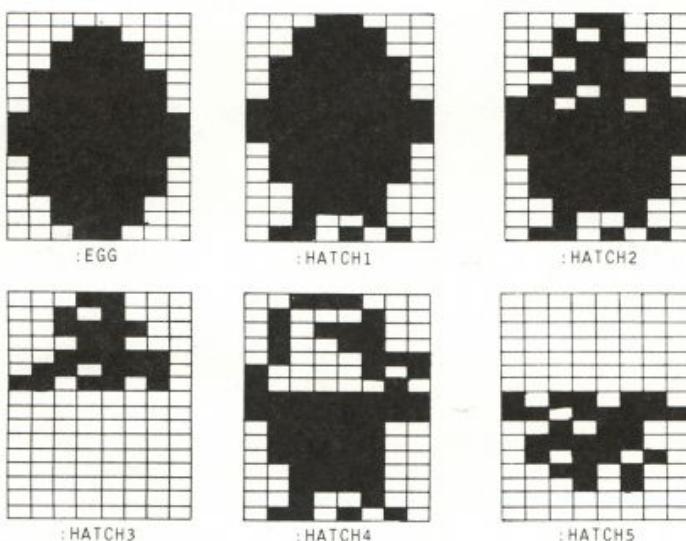
```
TO FLYOFF
  TELL 3 SETH 45
  SETSP 5 WING 3
  SETSP 10 SETH 67 WING 5
  SETH 90
  WING 5 HT
END

TO WING :TIMES
  REPEAT :TIMES [FLIP]
END
```

Hatching the Egg

Turtle 3 eventually is changed from an egg to a bird. The egg hatching process consisted of designing shapes and playing around with how they interact with one another.

I drew the egg shape and five more shapes to depict the hatching process.



:HATCH3 and :HATCH5 are fragments of the top and the bottom of the egg. Having these two shapes superimposed on the more complete shapes of :HATCH4 and :LITTLEBIRD1 enhances the hatching animation. It simplifies the pictures I had to draw. This superimposition happens by having turtle

*Notice that the delay for :LITTLEBIRD4 is shorter than the other delays. I did this because the wing in that shape is clipped. It extends beyond the shape's dimensions. With the shorter delay you don't notice the missing wing part, but you do get an uninterrupted sense of a flapping motion.

0 assume these fragments. The shape assumed by turtle 3 shows as well as the added details offered by turtle 0. In fact, this is a frill. Even without turtle 0's participation, the hatching scenario is quite acceptable.

```
TO HATCH
  TELL 0 SET.EGG
  SETSH 1
  PUTSH 4 :EGG WAIT 30
  PUTSH 4 :HATCH1 WAIT 30
  PUTSH 4 :HATCH2 WAIT 30
  PUTSH 1 :HATCH3 WAIT 30
  ST
  PUTSH 4 :HATCH4 WAIT 50
  HT
  PUTSH 1 :HATCH5 WAIT 50
  ST
  PUTSH 4 :LITTLEBIRD1 WAIT 50
  HT
END
```

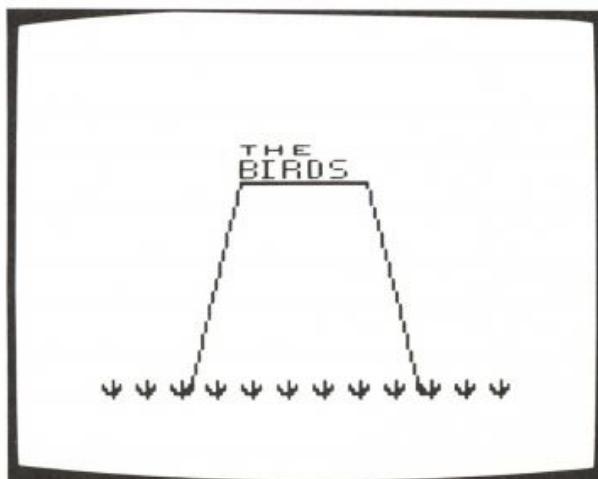
The following instructions prepare for HATCH.

```
CARTOON
  TELL 3 SETSP 0
  SETPOS [-7 48]
  SETC 7
  PUTSH 4 :EGG ST
  TELL [0 1 2] TAKEOFF
  FLAPFLAP 1 15 HT
```

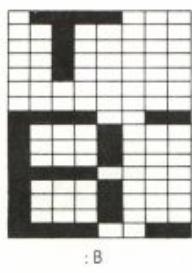
Then

HATCH

As a final touch I added a title at the beginning and one at the end. These procedures and shapes are included in the following listing.



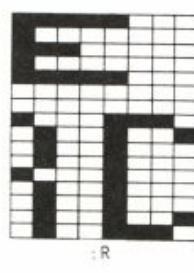
```
TO BIRDS.SIGN
  TELL [0 1 2 3] HT
  PU SETH 90
  SETPOS [-30 50]
  ASK 0 [SETSH 1]
  PUTSH 1 :B
  ASK 1 [FD 15 SETSH 2]
  PUTSH 2 :I
  ASK 2 [FD 30 SETSH 3]
  PUTSH 3 :R
  ASK 3 [FD 45 SETSH 4]
  PUTSH 4 :D
  TELL :ALL SETC BG ST
  SETC 56 ST
  WAIT 20
  REPEAT 4 [SETC COLOR + 1 WAIT 20]
END
```



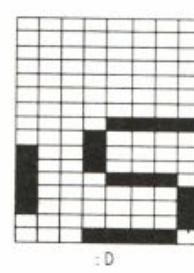
:B



:I



:R

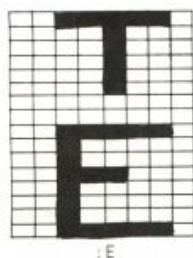
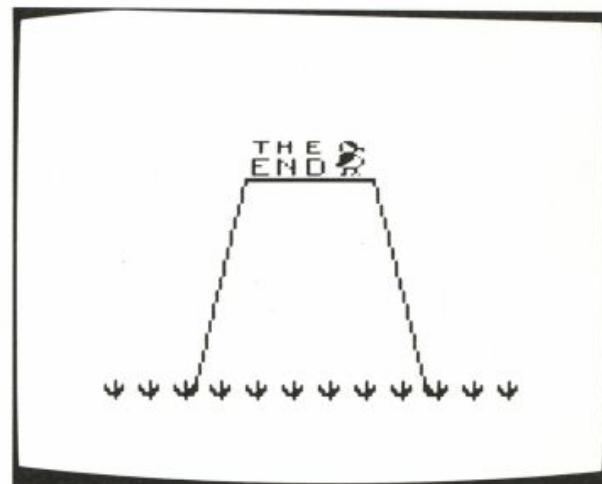


:S

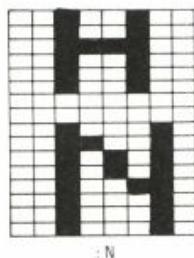
```

TO THEEND.SIGN
TELL [0 1 2 3] HT
PU SETH 90
SETPOS [-30 50]
ASK 0 [SETSH 1]
PUTSH 1 :E
ASK 1 [FD 15 SETSH 2]
PUTSH 2 :N
ASK 2 [FD 30 SETSH 3]
PUTSH 3 :T
ASK 3 [FD 50 SETSH 4]
PUTSH 4 :LITTLEBIRD1
SETC BG ST
SETC 88 WAIT 20
REPEAT 4 [SETC COLOR + 1 WAIT 20]
SETBG 74 SETPC 1 74 SETPC 2 74
WAIT 60
END

```



:E



:N



:T

I put all these new instructions into CARTOON and made a new procedure
SET.EGG,

```

TO SET.EGG
SETSP 0 SETPOS [-7 48]
SETC 7
END

```

Some other nice project ideas for these birds are to make them walk, make one of them find a worm and then fly off, make different background scenery, and so on.

PROGRAM LISTING

```

TO CARTOON
RECYCLE
TELL [0 1 2 3] HT CS FS
SETBG 0 SETPN 0 SETPC 0 6
MOUNTAIN
SET.MARKS
DRAWGRASS
WHEN OVER 2 2 [DROPANDLAND]
BIRDS.SIGN RECYCLE
WAIT 60
BIRD
FLY
ASK 3 [SET.EGG PUTSH 4 :EGG ST]
TELL [0 1 2] TAKEOFF
FLAPFLAP 1 15 HT
HATCH WAIT 30
ASK 3 [FLYOFF]
TELL [0 1 2 3] SETSP 0
THEEND.SIGN
END

TO MOUNTAIN
PU SETPOS [-65 -90]
PD SETPOS [-35 37]
RT 90 FD 70
SETPOS [65 -90]
END

TO SET.MARKS
TELL 0
SETPN 1 SETPC 1 BG
PU SETPOS [-55 100]
PD SETY 120 PU
SETPN 2 SETPC 2 BG
SETPOS [-50 100]
PD SETY 120 PU
SETPOS [110 -90]
PD SETX -110 PU
END

TO DRAWGRASS
TELL [0 1 2 3] SETPN 0 PU
ASK 0 [SETPOS [-110 -90]]
ASK 1 [SETPOS [-90 -90]]
ASK 2 [SETPOS [-70 -90]]
ASK 3 [SETPOS [-50 -90]]
REPEAT 3 [PD GRASS SETH 90 PU FD 80]
END

TO GRASS
SETH -60 FD 5 BK 5
RT 30 FD 8 BK 8
RT 30 FD 10 BK 10
RT 30 FD 8 BK 8
RT 30 FD 5 BK 5
END

TO BIRDS.SIGN
TELL [0 1 2 3] HT
PU SETH 90
SETPOS [-30 50]
ASK 0 [SETH 1]
PUTSH 1 :B
ASK 1 [FD 15 SETSH 2]
PUTSH 2 :I
ASK 2 [FD 30 SETSH 3]
PUTSH 3 :R
ASK 3 [FD 45 SETSH 4]
PUTSH 4 :D
SETC 56 ST
WAIT 20
REPEAT 4 [SETC COLOR + 1 WAIT 20]
END

TO BIRD
PUTSH 1 :TOPBIRD
PUTSH 2 :BOTTOMBIRD
PUTSH 3 :WING1
PUTSH 4 :TURD
TELL 0 SETPOS [-9 -10] SETSH 3 SETC 53
TELL 1 SETPOS [0 0] SETSH 1 SETC 12
TELL 2 SETY -15 SETSH 2 SETC 12
TELL 3 SETPOS [-9 -15] SETSH 4 SETC ▶
    BG
TELL [0 1 2 3]
EACH [SETPOS SE XCOR - 20 YCOR + 65]
ST
END

TO DROPANDLAND
WHEN OVER 2 2 []
DROP.TURD
WHEN OVER 2 1 [LANDING]
END

TO DROP.TURD
WHEN OVER 3 2 [DISAPPEAR]
TELL 3 SETH 180 SETC 7
SETSP 25 ST
TOOT 1 30 7 20
TELL [0 1 2]
END

```

```

TO DISAPPEAR
ASK 3 [SETSP 0 HT]
WHEN OVER 3 2 []
END

TO LANDING
TELL [0 1 2 3] SETH 130
PUTSH 3 :WING1
WHEN OVER 2 0 [PUTSH 2 :BOTTOOMBIRD ▶
    SETSP 0]
END

TO TAKEOFF
PUTSH 2 :LEGGSUP
SETH 45
SETSP 5 FLAPFLAP 5 5
SETSP 10 SETH 67
FLAPFLAP 10 5
SETH 90
SING 1
END

TO FLY
TELL [0 1 2 3]
WAIT 60 SING 5 WAIT 60
TAKEOFF SETSP 15
FLAP 7
WAIT 30 SING 5 WAIT 30
END

TO FLAPFLAP :TIMES :SPEED
REPEAT :TIMES [MOVEWING :SPEED]
END

TO MOVEWING :SPEED
PUTSH 3 :WING1 WAIT :SPEED
PUTSH 3 :WING4 WAIT :SPEED
PUTSH 3 :WING3 WAIT :SPEED
PUTSH 3 :WING2 WAIT :SPEED
PUTSH 3 :WING3 WAIT :SPEED
PUTSH 3 :WING4 WAIT :SPEED
END

TO FLYOFF
SETH 45
SETSP 5 WING 3
SETSP 10 SETH 67 WING 5
SETH 90
WING 5 SONG HT
END

TO WING :TIMES
REPEAT :TIMES [FLIP]
END

TO FLAP :SPEED
MOVEWING :SPEED
IF COND OVER 2 0 [STOP]
FLAP :SPEED
END

TO FLIP
PUTSH 4 :LITTLEBIRD1 WAIT 10
PUTSH 4 :LITTLEBIRD2 WAIT 10
PUTSH 4 :LITTLEBIRD3 WAIT 10
PUTSH 4 :LITTLEBIRD4 WAIT 8.5
PUTSH 4 :LITTLEBIRD3 WAIT 10
PUTSH 4 :LITTLEBIRD2 WAIT 10
END

TO SET.EGG
SETSP 0 SETPOS [-7 48]
SETC 7
END

TO HATCH
TELL [0 1 2 3] SET.EGG
PUTSH 4 :EGG SETSH 1 WAIT 30
PUTSH 4 :HATCH1 WAIT 30
PUTSH 4 :HATCH2 WAIT 30
PUTSH 1 :HATCH3 WAIT 30
TELL 0 ST
PUTSH 4 :HATCH4 WAIT 50
HT
PUTSH 1 :HATCH5 WAIT 50
ST
PUTSH 4 :LITTLEBIRD1 WAIT 50
HT
END

TO THEEND.SIGN
TELL [0 1 2 3] HT
PU SETH 90
SETPOS [-30 50]
ASK 0 [SETSH 1]
PUTSH 1 :E
ASK 1 [FD 15 SETSH 2]
PUTSH 2 :N
ASK 2 [FD 30 SETSH 3]
PUTSH 3 :T
ASK 3 [FD 50 SETSH 4]
PUTSH 4 :LITTLEBIRD1
SETC BG ST
SETC 88 WAIT 20
REPEAT 4 [SETC COLOR + 1 WAIT 20]
SETBG 74 SETPC 1 74 SETPC 2 74
WAIT 60
END

```

```

TO SING :TIMES
REPEAT :TIMES [PUTSH 1 :TOPBIRD2 SONG ▶
    PUTSH 1 :TOPBIRD WAIT 20 + RANDOM ▶
    30]
END

TO SONG
TOOT 0 1000 7 7
TOOT 0 1100 7 7
TOOT 0 1150 7 7
TOOT 0 1050 7 7
END

MAKE "TOPBIRD2 [48 72 153 170 151 132 ▶
    68 56 68 130 129 129 129 129 ▶
    129]
MAKE "LITTLEBIRD4 [56 92 76 71 69 120 ▶
    252 242 97 33 33 34 60 36 36 106]
MAKE "LITTLEBIRD3 [56 92 76 71 69 56 ▶
    124 242 193 161 161 34 60 36 36 ▶
    106]
MAKE "LITTLEBIRD2 [56 92 76 71 69 56 ▶
    124 122 249 225 193 34 60 36 36 ▶
    106]
MAKE "HATCH5 [0 0 0 0 0 0 0 182 223 44 ▶
    124 90 52 20 0 0]
MAKE "HATCH4 [56 68 92 76 71 133 130 ▶
    255 255 126 126 126 60 60 36 106]
MAKE "HATCH3 [24 40 60 88 62 110 218 0 ▶
    0 0 0 0 0 0 0]
MAKE "HATCH2 [24 40 60 88 62 110 218 ▶
    255 255 126 126 126 60 36 ▶
    106]
MAKE "HATCH1 [24 60 60 126 126 126 255 ▶
    255 255 126 126 126 60 60 36 ▶
    106]
MAKE "LEGGSUP [129 129 129 129 129 129 ▶
    129 129 129 126 34 34 42 164 72 ▶
    128]
MAKE "TURD [0 0 0 0 0 0 4 0 2 0 0 8 0 ▶
    0 0 0]
MAKE "TOPBIRD [56 68 188 148 156 135 ▶
    69 57 68 130 129 129 129 129 129 ▶
    129]
MAKE "BOTTOOMBIRD [129 129 129 129 129 ▶
    129 129 129 126 34 34 34 34 ▶
    34 85]
MAKE "EGG [24 60 60 126 126 126 255 ▶
    255 255 126 126 126 60 60 24 0]
MAKE "WING2 [15 63 127 248 224 128 0 0 ▶
    0 0 0 0 0 0 0]
MAKE "WING3 [15 31 62 120 112 224 192 ▶
    128 0 0 0 0 0 0]
MAKE "WING4 [7 15 15 30 60 56 112 96 ▶
    64 0 0 0 0 0 0]
MAKE "WING1 [3 7 15 15 31 30 30 62 60 ▶
    56 48 32 0 0 0]
MAKE "D [0 0 0 0 0 0 0 15 16 144 144 ▶
    142 129 129 1 30]
MAKE "R [248 128 224 128 248 0 0 142 ▶
    73 72 72 136 72 72 73 78]
MAKE "I [136 136 248 136 136 0 0 231 ▶
    132 132 132 135 132 132 132 228]
MAKE "B [248 32 32 32 32 0 0 243 136 ▶
    136 136 240 136 136 136 243]
MAKE "LITTLEBIRD1 [56 92 76 71 69 56 ▶
    124 122 121 113 225 194 60 36 36 ▶
    106]
MAKE "T [60 32 56 32 32 60 0 0 60 34 ▶
    34 34 34 34 34 60]
MAKE "N [36 36 60 36 36 36 36 0 0 34 50 ▶
    42 42 38 34 34 34]
MAKE "E [62 8 8 8 8 8 0 0 62 32 32 56 ▶
    32 32 32 62]

```

Jack and Jill

This animated story plays freely with two nursery rhymes I knew by heart when I was very young. The rhymes are “The House That Jack Built” and “Jack and Jill.” I did not try to retell either of them in any literal sense. Instead, I wanted the story to have in it some of my feelings for these memories of childhood. I wanted the animation to be lively, humorous, and whimsical. I let the story turn on graphics and animation features, and concentrated on how the animation felt.

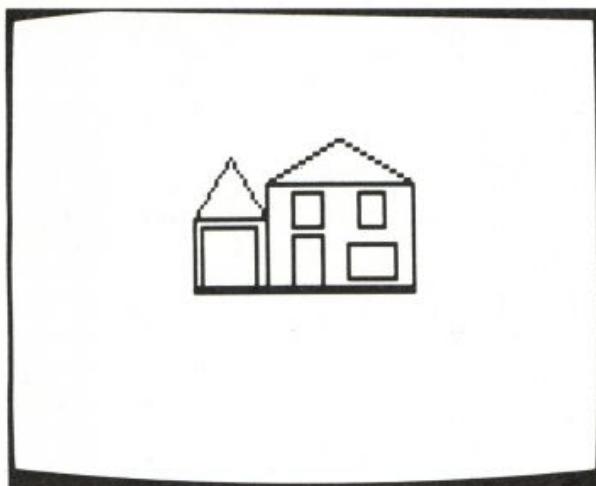
I built this story from a number of smaller projects I had written. I never planned these small projects as parts of a single animated feature, but

By Michael Grandfield.

I realized at some point that they could be used in this way. This write-up describes how the parts were made; it does not offer a description of the final animated story. However, a computer listing of the program is provided at the end of this write-up.

Building a House

Here is a house. Drawing it was the beginning of this whole project.



The funny thing about a picture of a house is that it doesn't let you know anything about how it was drawn. A nice thing about animation is that you can make the way the house is drawn an important, interesting process; and that is what the project was about.

One way to draw this house would be to write individual procedures for each of the shapes that are in the design. The house needs three shapes: a square for the house; a rectangle for the garage, the doors, and the windows; and a triangle for the roofs. Once these procedures are written, you can put together a procedure that uses them all to draw a house.

```
TO DRAW.HOUSE
HOUSE
GARAGE
WINDOWS
ROOFS
END
```

This is a fine way for one turtle to make a drawing. But I wanted to try something different. I chose to play with having all four turtles cooperate in drawing the house. They became the JackBuilt Construction Company.

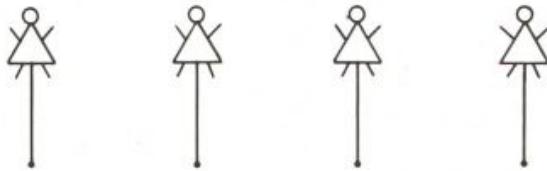
The tricky part of this project was making all the turtles draw different parts of the house at the same time. Here are a series of examples that illustrate the problem and how I decided to solve it.

In this illustration, all of the turtles are visible and placed apart from one other.



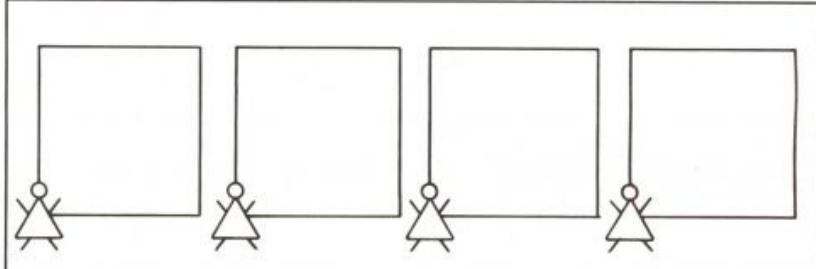
First, let's give the turtles something to do all at once.

TELL [0 1 2 3] FD 50



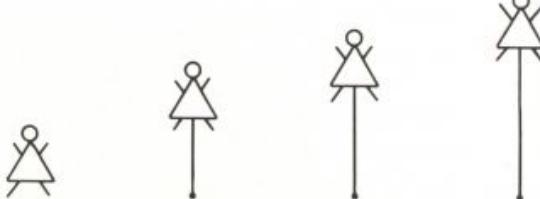
Next, the turtles can simultaneously draw the same figure.

SQUARE



Now let's have the turtles each do something different.

TELL [0 1 2 3] EACH [FD WHO * 20]



This command is very different. The turtles are moving one after another in very rapid succession. Yet as long as the instruction given to each turtle is not too long, they will appear to be moving all at once.

This is the key to the project.

The turtles can appear to be drawing a house simultaneously if each instruction to the turtles is pretty short. The procedure needs to explicitly tell each of them to draw each single line. The way I thought about this was to give each turtle at the job site its own work to do. One turtle laid the foundation, another put up walls, another built roofs, and the last made doors and windows. Each turtle had a list of things to do. When each had finished the first item on its list, they all went on to the next item.

```
MAKE "CARPENTER0 [[SETPOS [30 20] SETH 90 PD]
                  [FD 120 SETY 21]
                  [BK 120 SETY 22] [FD 120 SETY 23]
                  [BK 120 SETY 24] [FD 120 PU]
                  [SETPOS [70 85] SETH 90 PD FD 80 PU HOME]]
MAKE "CARPENTER1 [[SETPOS [150 25] PD SETH 0] [FD 60]
                  [LT 60 FD 46 PU]
                  [SETPOS [135 60] SETH 270 PD] [WIN PU]
                  [SETPOS [140 30] PD DOOR] [PU HOME]]
MAKE "CARPENTER2 [[SETPOS [30 25] SETH 0 PD] [FD 40]
                  [RT 30 FD 40] [RT 120 FD 40]
                  [RT 120 FD 40 PU]
                  [SETPOS [35 25] SETH 0 PD]
                  [REPEAT 2 [FD 35 RT 90 FD 30 RT 90]
                  PU HOME]]
MAKE "CARPENTER3 [[SETPOS [70 25] SETH 0 PD] [FD 60]
                  [RT 60 FD 46 PU]
                  [SETPOS [85 80] SETH 90 PD] [WIN PU]
                  [SETPOS [85 55] PD BAY] [PU HOME]]
```

It is important that each of these lists have the same number of items. Also, these lists call three short procedures to draw the door and windows. Here they are.

```
TO DOOR
REPEAT 2 [FD 30 RT 90 FD 15 RT 90]
END

TO WIN
REPEAT 4 [FD 20 RT 90]
END

TO BAY
REPEAT 2 [FD 20 RT 90 FD 30 RT 90]
END
```

Here's a procedure that can use the CARPENTER variables to draw a house.

```
TO BUILD.HOUSE
TELL [0 1 2 3] PU
BUILD.HOUSE1 :CARPENTER0 :CARPENTER1 :CARPENTER2 :CARPENTER3
END
```

BUILD.HOUSE uses BUILD.HOUSE1.

```
TO BUILD.HOUSE1 :LIST0 :LIST1 :LIST2 :LIST3
IF EMPTYP :LIST0 [STOP]
EACH IRUN FIRST THING WORD "LIST WHO TOOT 0 240 15 2"
BUILD.HOUSE1 BF :LIST0 BF :LIST1 BF :LIST2 BF :LIST3
END
```

Getting to the Job Site

Here is the second project. It's about carpenters and their truck.

It seemed important that the turtles look like carpenters and that they arrive at the job site in some believable way. I decided that driving up in a truck was the right thing to do. Here are the shapes for the carpenters and the truck. The listings for these shapes are provided at the end of this section.



As you can see, I decided to use only two shapes for the truck.

It didn't take long for me to discover that I had a bit of a problem. The turtles that carried the truck shapes had to become carpenters as soon as the truck arrived at the job site! This meant that the truck would disappear. I decided to solve this problem by having the carpenters drive up *as* the truck, rather than *in* the truck. When the truck arrived, it would be transformed into the carpenters. When the house was finished, the carpenters would transform back into the truck.

Here are the shapes I made for the transformation.



And here's what that transformation looked like on the screen.



The interesting part of this project was making the shapes and making the animation work effectively.

Here are the procedures.

```

TO SETUP
PUTSH 1 :SHAPE1
PUTSH 2 :SHAPE2
PUTSH 3 :SHAPE3
PUTSH 4 :SHAPE4
PUTSH 5 :SHAPE5
PUTSH 6 :SHAPE6
PUTSH 7 :SHAPE7
MAKE "CARPENTER 1
MAKE "REAR 2
MAKE "FRONT 3
END

TO ENTER.TRUCK
TELL [0 1]
PU SETPOS [-155 20] SETH 90
ASK 0 [SETSH :REAR]
ASK 1 [SETSH :FRONT FD 15]
ST
REPEAT 80 [FD 2 TOOT 0 20 15 2]
END
  
```

Remember, TELL [0 1] is still in effect.

```

TO TRANSFORM
ASK 2 [SETPOS ASK 0 [POS]]
ASK 3 [SETPOS ASK 1 [POS]]
EACH [SETSH SUM WHO 5]
ASK [2 3] [SETSH 3 ST WAIT 30 SETSH 4]
ASK 0 [BK 15]
ASK 1 [FD 15]
RT 90 FD 8
SETSH 3 WAIT 30 SETSH 4
END
  
```

Now the carpenters are ready to BUILD.HOUSE. When they have finished the job, they turn back into their truck and go home.

```

TO TRANSFORM.BACK
SETH 90
EACH [SETPOS LIST SUM 12 WHO * 30 15]
ASK 2 [SETPOS LIST SUM 8 ASK 0 [XCOR] 15]
ASK 3 [SETPOS LIST SUM -8 ASK 1 [XCOR] 15]
EACH [SETSH SUM WHO 5]
ASK 0 [FD 8]
ASK 1 [BK 8]
ASK [2 3] [WAIT 30 SETSH 4 WAIT 30 SETSH 3 WAIT 30 HT]
EACH [SETSH SUM 1 WHO]
END

TO EXIT
REPEAT 62 [FD 2 TOOT 0 20 15 2]
ASK 1 [HT]
REPEAT 8 [FD 2 TOOT 0 20 15 2]
HT
END

```

Here are the listings for the shapes. They are listed in the same order as they appeared in this section.

```

MAKE "SHAPE1 [24 24 24 24 0 28 26 26 26
              24 36 36 68 68 132 132]
MAKE "SHAPE2 [0 0 0 0 0 0 255 255
              255 255 255 255 255 255 96 96]
MAKE "SHAPE3 [0 0 0 0 0 0 224 224
              239 239 255 255 255 255 102 102]
MAKE "SHAPE4 [0 0 0 0 0 0 0 0 0 0 0 0 255 255 255 96 96]
MAKE "SHAPE5 [0 0 0 0 0 0 0 0 0 0 0 0 239 255 255 102 102]
MAKE "SHAPE6 [0 0 0 0 32 16 139 135 255 255 0 0 0 0 0 0]
MAKE "SHAPE7 [35 19 11 7 6 142 254 12 136 240 0 0 0 0 0 0]

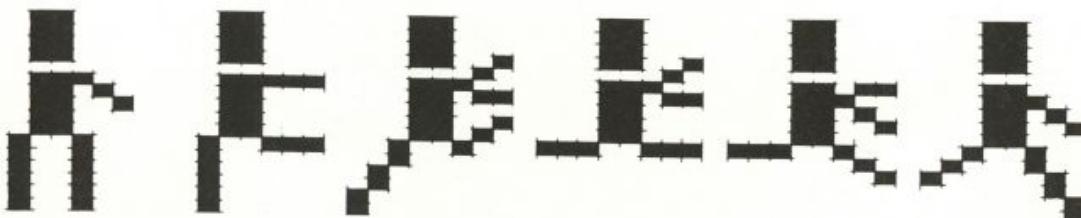
```

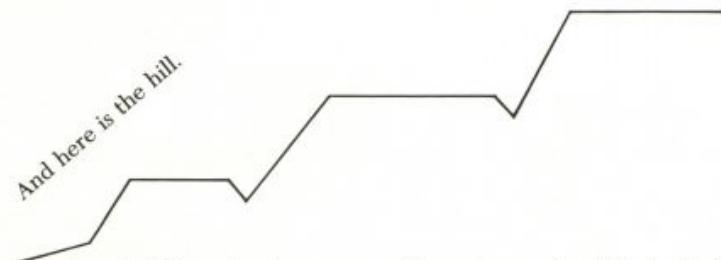
Jack and Jill

This is the third project. It's about animating figures.

I wanted to animate Jack leaping up a hill, meeting Jill, and then Jack and Jill leaping down the hill. I decided to forget about "*Jack fell down and broke his crown/And Jill came tumbling after.*" Getting them to leap without falling down was enough to do. I decided, however, that Jack should be a bit clumsy. He would always bump into things before leaping over them.

Here are the leaping shapes. The listings appear in order at the end of this section.





Jack, agile fellow that he is, sets off leaping up the hill. At the first opportunity, he bumps into the rise in the hill. He is saved from a fall by his guardian demon. Every time Jack stubs his toe, this demon stops him and gets him to back up a bit. Here's the demon.

```
WHEN OVER 0 0 [SETSP 0 BK 5]
```

Now let's look at the procedures that bring this animation to life. *JACK.AND.JILL* is the main procedure.

```
TO JACK.AND.JILL
TELL 0 HT
ASK [1 2 3] [HT]
DRAW.HILL
SETUP
UP.THE.HILL
DOWN.THE.HILL
END

TO DRAW.HILL
PU SETPOS [-155 -50] SETH 90 PD
REPEAT 3 [FD 40 LT 60 FD 30 RT 120 FD 5 SETH 90]
FD 40 PU
END
```

SETUP activates the demon and places Jack and Jill.

```
TO SETUP
PUTSH 1 :SHAPE1
PUTSH 2 :SHAPE2
PUTSH 3 :SHAPE3
PUTSH 4 :SHAPE4
PUTSH 5 :SHAPE5
PUTSH 6 :SHAPE6
PUTSH 7 :SHAPE7
MAKE "JACK 0
MAKE "JILL 1
MAKE "LEAP.UP [1 2 3 4 5 6]
MAKE "LEAP.DN [6 5 4 3 2 1]
TELL :JACK PU SETPOS [-150 -38] SETH 90 SETSH 1 SETC 7 ST
ASK :JILL [PU SETPOS [52 28] SETH 270 SETSH 1 SETC 7 ST]
WHEN OVER 0 0 [SETSP 0 BK 5 MAKE "FREQ 200]
END
```

UP.THE.HILL animates Jack's ascent.

STORIES

```
TO UP.THE.HILL
SETSP 10
ANIMATE :LEAP.UP
SETSP 0
END
```

ANIMATE stops only when Jack has reached the top of the hill. Until then, it repeatedly invokes ANIMATE1.

```
TO ANIMATE :SHAPES
IF OR XCOR > 50 XCOR < -155 [STOP]
ANIMATE1 :SHAPES
ANIMATE :SHAPES
END
```

ANIMATE1 does all the work.

```
TO ANIMATE1 :SHAPES
IF EMPTYP :SHAPES [STOP]
IF SPEED = 0 [PICK.JUMP]
SETH FIRST :SHAPES
TOOT 0 440 10 1
ANIMATE1 BF :SHAPES
END
```

Here's the procedure that is invoked when the collision demon has stopped Jack.

```
TO PICK.JUMP
IF WHO = 0 [JUMP.UP] [JUMP.DOWN]
END
```

JUMP.UP is invoked to get Jack up the hill.

```
TO JUMP.UP
SETH 3 SETH 30
REPEAT 16 [TOOT 1 :FREQ 15 2 FD 2 MAKE "FREQ :FREQ + 200]
SETH 4 SETH 90
REPEAT 8 [FD 2]
SETH 1 SETH 150
REPEAT 6 [TOOT 0 :FREQ 15 2 FD 1 MAKE "FREQ :FREQ - 400]
SETH 90 SETSP 10
END
```

JUMP.DOWN is invoked to get Jack and Jill down the hill.

```
TO JUMP.DOWN
WAIT 5
SETH 330 SETH 3
REPEAT 7 [TOOT 0 :FREQ 15 2 FD 1 MAKE "FREQ :FREQ + 600]
SETH 270 SETH 4
REPEAT 15 [FD 2]
SETH 210 SETH 1
REPEAT 14 [TOOT 1 :FREQ 15 2 FD 2 MAKE "FREQ :FREQ - 200]
SETH 270 SETSP 10
END
```

Now Jack and Jill can leap down the hill.

```
TO DOWN.THE.HILL
  TELL LIST :JACK :JILL
  ASK :JACK [SETPOS [43 28]]
  SETH 270 SETSP 10
  ANIMATE :LEAP.DN
  SETSP 0
END
```

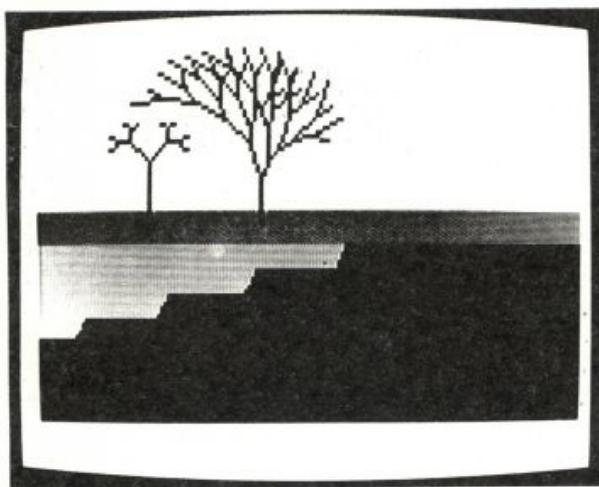
Here are the listings for the shapes.

```
MAKE "SHAPE1 [24 24 24 24 0 28
              26 25 24 24 36 36 36 36 36]
MAKE "SHAPE2 [24 24 24 24 0 31
              24 24 24 24 39 32 32 32 32]
MAKE "SHAPE3 [24 24 24 25 2 28
              27 24 25 26 36 32 64 64 128 128]
MAKE "SHAPE4 [24 24 24 25 2 28 27 24 24 24 24 231 0 0 0 0 0]
MAKE "SHAPE5 [24 24 24 24 0 27 28 26 25 24 228 2 1 0 0 0]
MAKE "SHAPE6 [24 24 24 24 0 24 28 26 25 24 36 68 130 2 1 1]
```

Setting the Scene

This is the fourth project. It's about drawing scenery and choosing colors that complement one another.

I wanted to learn about how to write programs for scenery. So I drew lots of different scenes and I learned some very useful techniques that make drawing faster and easier. Here is the scenery I use in Jack and Jill.



SET.SCENE is the main procedure.

```
TO SET.SCENE
  SETUP
  GRASS
  ROAD
  HILL
  TREES
END
```

SETUP is the setup procedure.

```
TO SETUP
SETBG 80
SETPC 0 7
SETPC 1 4
SETPC 2 61
FS
TELL 0 HT
END
```

In this scene I wanted to fill in a large area of the screen with a single color. This can be a slow process. Here is a technique for overcoming this difficulty.

```
TO GRASS
SETPN 0 SETH 90
PU SETPOS [-160 -105] PD
FD 320
SETH 89.6
REPEAT 2 [FD 7700]
SETH 90
FD 320
PU
END
```

GRASS assumes that the screen is in WRAP. It begins at the bottom of the screen and draws a single line across the screen. Then the interesting part happens. The turtle is turned 0.4 degrees and is instructed to go forward many steps. Try this out.

Now let's look at the rest of SETSCENE1.

```
TO SET.SCENE1
SETUP
GRASS
ROAD
HILL
TREES
END
```

ROAD works in exactly the same way as GRASS.

```
TO ROAD
SETPN 1 SETH 270
PU SETPOS [160 3] PD
FD 320
SETH 270.4
FD 2750
SETH 270
FD 320
PU
END
```

HILL invokes HILL1 to draw the several levels of the hill.

```
TO HILL
SETPN 2 SETH 0
PU SETPOS [-158 -54] PD
HILL1 0 -135
HILL1 0 -85
HILL1 0 -35
HILL1 0 15
PU
END
```

This particular hill was made with fearless Jack in mind.

```
TO HILL1 :A :X
IF :A = 7 [STOP]
SETX :X + :A
SETY SUM YCOR 1
SETX -158
SETY SUM YCOR 1
HILL1 :A + 1 :X
END
```

I decided to add trees to this scene, and I remembered the cover illustration from Harold Abelson and Andrea DiSessa's *Turtle Geometry* (Cambridge, Mass: MIT Press, 1981).

```
TO TREES
SETPN 0 SETH 0
PU SETPOS [-90 22] PD
BRANCH 30 5
PU SETPOS [-25 22] PD
LBRANCH 10 20 5
PU
END
```

Here are the four procedures from *Turtle Geometry*.

```
TO RBRANCH :LENGTH :ANGLE :NUM
FD :LENGTH
NODE :LENGTH :ANGLE :NUM
BK :LENGTH
END

TO LBRANCH :LENGTH :ANGLE :NUM
FD 2 * :LENGTH
NODE :LENGTH :ANGLE :NUM
BK 2 * :LENGTH
END

TO NODE :LEN :ANG :TIMES
IF :TIMES = 0 [STOP]
LT :ANGLE
LBRANCH :LEN :ANG :TIMES - 1
RT 2 * :ANGLE
RBRANCH :LEN :ANG :TIMES - 1
LT :ANGLE
END
```

```

TO BRANCH :LENGTH :TIMES
IF :TIMES = 0 [STOP]
FD :LENGTH
LT 45
BRANCH :LENGTH / 2 :TIMES - 1
RT 90
BRANCH :LENGTH / 2 :TIMES - 1
LT 45
BK :LENGTH
END

```

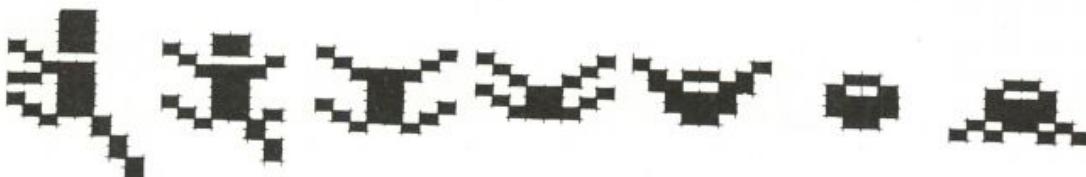
The Whole Ball of Wax

I realized, while I was making up a diskette with all of my projects on it, that I had all of the pieces I needed to make a long animated cartoon. Jack and Jill could become an opus.

Here's the storyline. The JackBuilt Construction Company drives down a tree-lined road that runs along the crest of a terraced hillside. Their truck stops at a house site and the carpenters build a home for Jack and Jill. When the house is built and the carpenters are gone, Jill appears in the doorway and waits for Jack to leap up the hill. Jack sees Jill waiting but rings the doorbell anyway, and Jill comes out. Together they leap off down the road, magically transforming themselves into birds as they leap. The two beautiful birds fly off through the trees, and the story ends.

Animating the birds was the finishing touch to this story. I love magic, and I feel that all children's stories should have some magic in them. I decided to transform Jack and Jill into birds in midleap and have them fly away. This transformation delights your eye and surprises your expectations for the story. It's magic.

Here are the shapes I designed for this transformation sequence. Jack and Jill are traveling from right to left across the screen, so this sequence is also displayed here from right to left. The listings appear in order at the end of this section.



The transformation begins with both figures carrying a leaping shape. BECOME.BIRDS accomplishes the actual transformation.

```

TO BECOME.BIRDS
SETSH 1
SETC 46
WAIT 3
PUTSH 1 :BECOME.B1

```

```

SETH 300
WAIT 3
PUTSH 1 :BECOME.B2
SETSP 20
WAIT 3
PUTSH 1 :BECOME.B3
WAIT 3
PUTSH 1 :BECOME.B4
END

```

Notice that throughout the transformation the figures are carrying shape 1. I decided to use PUTSH to change shape 1 several times.

FLY is the final procedure in the story. It animates the birds and synchronizes the sounds they make as they fly away.

```

TO FLY
EACH [IF XCOR < -150 [STOP]]
PUTSH 1 :FLY1
EACH [TOOT WHO SUM WHO 1 * 1760 10 3]
PUTSH 1 :FLY2
EACH [TOOT WHO SUM WHO 1 * 1760 10 3]
PUTSH 1 :FLY3
EACH [TOOT WHO SUM WHO 1 * 1760 10 3]
FLY
END

```

Here are the shapes.

```

MAKE "BECOME.B1 [24 24 24 152 64 56 216 24
                  152 88 36 4 2 2 1 1]
MAKE "BECOME.B2 [0 0 24 152 66 60 24 24 152 90 36 4 2 2 0 0]
MAKE "BECOME.B3 [0 0 0 129 66 60 24 24 153 90 36 0 0 0 0 0]
MAKE "BECOME.B4 [0 0 0 129 66 36 153 90 60 0 0 0 0 0 0 0]
MAKE "FLY1 [0 0 0 129 90 102 60 60 24 0 0 0 0 0 0 0]
MAKE "FLY2 [0 0 0 0 24 36 60 60 24 0 0 0 0 0 0 0]
MAKE "FLY3 [0 0 0 0 24 36 60 60 90 165 0 0 0 0 0 0]

```

PROGRAM LISTING

This program is divided into five files. You only need to load the first file D:JACK; the others are loaded by the program at the appropriate times.

D:JACK

The main procedure is JACK.

TO JACK	TO SETUP
SETUP	LOAD.SHAPES 1
SET.SCENE	SETBG 80
ERALL	SETPC 0 7
LOAD "D:JACK1	SETPC 1 4
CT SS	SETPC 2 61
PR [TYPE "C" TO CONTINUE]	SET.DRIVE
END	FS
	ASK [0 1 2 3] [PU HT SETC 39]
	END

```

TO LOAD.SHAPES :NUM
IF :NUM > 15 [STOP]
PUTSH :NUM THING WORD "SHAPE :NUM"
ERN WORD "SHAPE :NUM"
LOAD.SHAPES :NUM + 1
END

TO SET.DRIVE
TS CT
SETCURSOR [4 10]
PR [INSERT DEMO DISK IN DRIVE #1,]
SETCURSOR [10 11]
PR [THEN PRESS RETURN]
IF NOT RC = CHAR 155 [SET.DRIVE]
END

TO SET.SCENE
TELL @
GRASS
ROAD
HILL
TREES
END

TO GRASS
SETPN @ SETH 90
PU SETPOS [-160 -105] PD
FD 320
SETH 89.6
REPEAT 2 [FD 7700]
SETH 90
FD 320
PU
END

TO ROAD
SETPN 1 SETH 270
PU SETPOS [160 3] PD
FD 320

SETH 270.4
FD 2750
SETH 270
FD 320
PU
END

TO HILL
SETPN 2 SETH 0
PU SETPOS [-158 -54] PD
HILL1 0 -135
HILL1 0 -85
HILL1 0 -35
HILL1 0 15
PU
END

TO HILL1 :A :X
IF :A = 7 [STOP]
SETX :X + :A
SETY SUM YCOR 1
SETX -158
SETY SUM YCOR 1
HILL1 :A + 1 :X
END

TO TREES
SETPN @ SETH 0
PU SETPOS [-90 22] PD
BRANCH 30 5
PU SETPOS [-25 22] PD
LBRANCH 10 20 5
PU
END

TO RBRANCH :LENGTH :ANGLE :NUM
FD :LENGTH
NODE :LENGTH :ANGLE :NUM
BK :LENGTH
END

TO LBRANCH :LENGTH :ANGLE :NUM
FD 2 * :LENGTH
NODE :LENGTH :ANGLE :NUM
BK 2 * :LENGTH
END

TO NODE :LEN :ANG :TIMES
IF :TIMES = 0 [STOP]
LT :ANGLE
LBRANCH :LEN :ANG :TIMES - 1
RT 2 * :ANGLE
RBRANCH :LEN :ANG :TIMES - 1
LT :ANGLE
END

TO BRANCH :LENGTH :TIMES
IF :TIMES = 0 [STOP]
FD :LENGTH
LT 45
BRANCH :LENGTH / 2 :TIMES - 1
RT 90
BRANCH :LENGTH / 2 :TIMES - 1
LT 45
BK :LENGTH
END

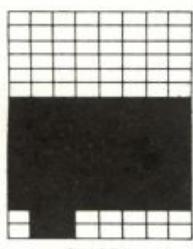
MAKE "SHAPE1 [0 0 0 0 0 255 255 255 ▶
255 255 255 255 255 96 96]"
MAKE "SHAPE2 [0 0 0 0 0 224 224 239 ▶
239 255 255 255 255 102 102]"

```

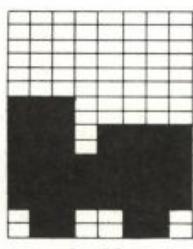
```

MAKE "SHAPE3 [0 0 0 0 32 16 139 135 ▶
  255 255 0 0 0 0 0 0]
MAKE "SHAPE4 [35 19 11 7 6 142 254 12 ▶
  136 240 0 0 0 0 0 0]
MAKE "SHAPE5 [0 0 0 0 0 0 0 0 0 0 0 0 ▶
  255 255 255 96 96]
MAKE "SHAPE6 [0 0 0 0 0 0 0 0 0 0 0 0 ▶
  239 255 255 102 102]
MAKE "SHAPE7 [24 24 24 24 0 28 26 26 ▶
  26 24 36 36 68 68 132 132]
MAKE "SHAPE8 [24 24 24 24 0 56 88 152 ▶
  24 24 36 36 36 36 36 36]
MAKE "SHAPE9 [24 24 24 24 0 248 24 24 ▶
  24 24 228 4 4 4 4 4]
MAKE "SHAPE10 [24 24 24 152 64 56 216 ▶
  24 152 88 36 4 2 2 1 1]
MAKE "SHAPE11 [24 24 24 152 64 56 216 ▶
  24 24 24 231 0 0 0 0 0]
MAKE "SHAPE12 [24 24 24 24 0 216 56 88 ▶
  152 24 39 64 128 0 0 0]
MAKE "SHAPE13 [24 24 24 0 36 24 24 24 ▶
  24 36 36 66 66 129 129 129]
MAKE "SHAPE14 [24 24 24 24 0 24 56 88 ▶
  152 24 36 34 65 64 128 128]
MAKE "SHAPE15 [24 24 153 90 36 24 24 ▶
  24 24 24 36 36 66 129 129]

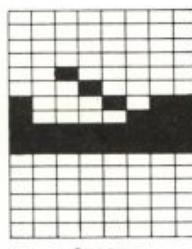
```



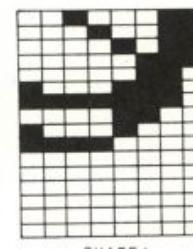
:SHAPE1



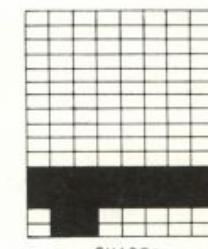
:SHAPE2



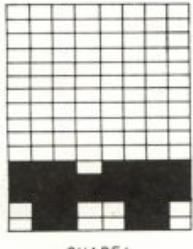
:SHAPE3



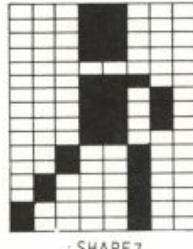
:SHAPE4



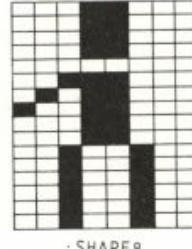
:SHAPE5



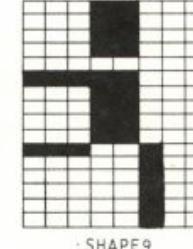
:SHAPE6



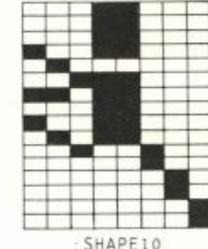
:SHAPE7



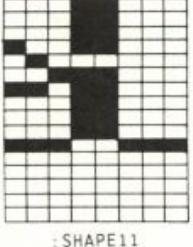
:SHAPE8



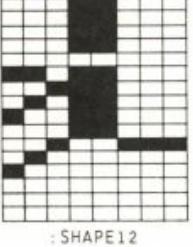
:SHAPE9



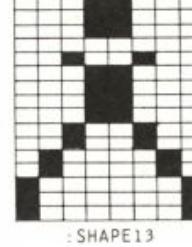
:SHAPE10



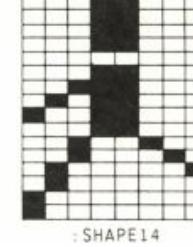
:SHAPE11



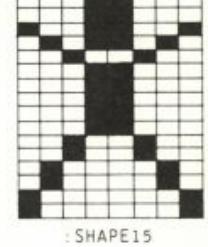
:SHAPE12



:SHAPE13



:SHAPE14



:SHAPE15

```

D:JACK1

TO C
ER "C
FS
LOAD "D:JACK2
ASK [Ø 1 2 3] [PU HT]
TELL [Ø 1]
ENTER.TRUCK
RECYCLE
TRANSFORM
FIRST.CLEANUP
TELL [Ø 1 2 3]
BUILD.HOUSE
TELL [Ø 1]
REFORM.AND.EXIT.TRUCK
SECOND.CLEANUP
JACK.AND.JILL
AFTERMATH
END

TO TRANSFORM
ASK 2 [SETPOS ASK Ø [POS]]
ASK 3 [SETPOS ASK 1 [POS]]
EACH [SETSH SUM WHO 5]
ASK [2 3] [SETSH 3 ST WAIT 3Ø SETSH 4]
ASK Ø [BK 15]
ASK 1 [FD 15]
RT 9Ø FD 8
ASK [Ø 1] [SETSH 3 WAIT 3Ø SETSH 4]
END

TO ENTER.TRUCK
SETH 9Ø
SETPOS [-155 2Ø]
ASK 1 [FD 15]
EACH [SETSH WHO + 1]
ST
REPEAT 8Ø [FD 2 TOOT Ø 2Ø 15 2]
END

TO FIRST.CLEANUP
ER "FIRST.CLEANUP
ER [ENTER.TRUCK TRANSFORM]
RECYCLE
END

TO REFORM.AND.EXIT.TRUCK
SETH 9Ø
EACH [SETPOS LIST SUM 12 WHO * 3Ø 15]
ASK 2 [SETPOS LIST SUM 8 ASK Ø [XCOR] ▶
    15]
ASK 3 [SETPOS LIST SUM -8 ASK 1 [XCOR] ▶
    15]

EACH [SETSH SUM WHO 5]
ASK Ø [FD 8]
ASK 1 [BK 8]
ASK [2 3] [WAIT 3Ø SETSH 4 WAIT 3Ø ▶
    SETSH 3 WAIT 3Ø HT]
EACH [SETSH SUM 1 WHO]
RECYCLE
REPEAT 62 [FD 2 TOOT Ø 2Ø 15 2]
ASK 1 [HT]
REPEAT 8 [FD 2 TOOT Ø 2Ø 15 2]
HT
END

TO BUILD.HOUSE
SETSH 14
SETPN 2
BUILD :CARPØ :CARP1 :CARP2 :CARP3
BUILD :CARP1Ø :CARP11 :CARP12 :CARP13
PU
WAIT 3Ø
END

TO SECOND.CLEANUP
ER "SECOND.CLEANUP
ACTIVATE [1 2 3 4 5 6] 65
ERN [CARPØ CARP1 CARP2 CARP3 CARP1Ø ▶
    CARP11 CARP12 CARP13]
ER [BUILD.HOUSE BUILD DOOR BAY WIN ▶
    REFORM.AND.EXIT.TRUCK ACTIVATE]
RECYCLE
END

TO JACK.AND.JILL
ER "JACK.AND.JILL
SETUP.JJ
UP.THE.HILL
DOOR.BELL
TELL [Ø 1]
DOWN.THE.HILL?
BECOME.BIRDS
FLY
ASK [Ø 1] [HT SETSP Ø]
END

TO AFTERMATH
TELL Ø ASK [Ø 1 2 3] [SETPOS [Ø Ø]]
ERALL
LOAD "D:JACK3
LOAD "D:JACK4
CT SS
PR [TYPE "RERUN" TO SEE THIS DEMO ▶
    AGAIN]
PR [TYPE "RESET" TO STOP THIS DEMO]
END

```

```

TO BUILD :N0 :N1 :N2 :N3          WHEN Ø []
IF EMPTYYP :NØ [STOP]           RECYCLE
EACH [RUN FIRST ( THING WORD "N WHO ) ▶ END
      TOOT Ø 240 15 2]
BUILD BF :NØ BF :N1 BF :N2 BF :N3
END

TO ACTIVATE :LIST :N1           TO DOWN.THE.HILL?
IF EMPTYYP :LIST [STOP]          PUTSH 1 GETSH 9
PUTSH FIRST :LIST THING CHAR :N1 SETY 1Ø SETH 280
ERN WORD " CHAR :N1            SETSP 1Ø
ACTIVATE BF :LIST SUM 1 :N1     ANIMATE 8 14
END                             END

TO DOOR                         TO BECOME.BIRDS
REPEAT 2 [FD 25 RT 90 FD 20 RT 90] SETC 46
END                               SETSH 1
                                 WAIT 5
                                 PUTSH 1 :AA
                                 SETH 300
                                 WAIT 3
                                 PUTSH 1 :BB
                                 SETSP 20
                                 WAIT 3
                                 PUTSH 1 :CC
                                 WAIT 5
                                 PUTSH 1 :DD
END

TO BAY                           END

TO WIN                           TO FLY
REPEAT 2 [FD 15 RT 90 FD 30 RT 90] EACH [IF XCOR < -150 [STOP]]
END                               PUTSH 1 :FF
                                 EACH [TOOT WHO SUM WHO 1 * 1760 1Ø 3]
                                 PUTSH 1 :EE
                                 EACH [TOOT WHO SUM WHO 1 * 1760 1Ø 3]
                                 PUTSH 1 :GG
                                 EACH [TOOT WHO SUM WHO 1 * 1760 1Ø 3]
                                 FLY
                                 END

TO SETUP.JJ                      TO JUMP.OVER.RT
ER "SETUP.JJ                     SETSP Ø
WHEN Ø [JUMP.OVER.RT]           BK 5
MAKE "NUM 1                       SETSH 3 SETH 20
TELL Ø                           RAMP 200 3000 200
SETPOS [-155 -38]                SETSH 4 SETH 90
SETH 90 SETSH 1 SETC 7           REPEAT 7 [FD 2]
ASK 1 [SETPOS [92 35] SETH 270 SETSH 8 ▶ SETSH 15 SETH 150
      SETC 7 ST]                  RAMP 3000 400 400
ASK 2 [SETPOS [127 65]]          SETH 90
ASK 3 [SETPOS [127 35]]          SETSP 1Ø
ST                                END
RECYCLE
END

TO UP.THE.HILL                   TO ANIMATE :FIRST :LAST
SETSP 1Ø                          IF XCOR > 90 [SETSP Ø SETH 270 SETX 80 ▶
ANIMATE 1 7                         STOP]
SETY 22                           IF YCOR > 25 [STOP]
END

TO DOOR.BELL                     ER [UP.THE.HILL JUMP.OVER.RT]
ER "DOOR.BELL
TOOT Ø 440 5 15 TOOT 1 660 5 15
TOOT Ø 660 5 15 TOOT 1 880 5 15
TOOT Ø 440 5 15 TOOT 1 660 5 15
TOOT Ø 880 5 60 TOOT 1 1320 5 60
ER [UP.THE.HILL JUMP.OVER.RT]

```

```

IF :FIRST > :LAST [MAKE "FIRST :LAST - ▶
  6]
SETSH :FIRST
TOOT 0 440 10 1
ANIMATE :FIRST + 1 :LAST
END

TO RAMP :S :F :R
IF :F < :S [REPEAT ( :S - :F ) / :R ▶
  [TOOT 1 :S 15 2 FD 2 MAKE "S :S - ▶
  :R]]
IF :S < :F [REPEAT ( :F - :S ) / :R ▶
  [TOOT 1 :S 15 2 FD 2 MAKE "S :S + ▶
  :R]]
END

TO REFORM
SETH 90
EACH [SETPOS LIST SUM 12 WHO * 30 15]
ASK 2 [SETPOS LIST SUM 8 ASK 0 [XCOR] ▶
  15]
ASK 3 [SETPOS LIST SUM -8 ASK 1 [XCOR] ▶
  15]
EACH [SETSH SUM WHO 5]

  ASK 0 [FD 8]
  ASK 1 [BK 8]
  ASK [2 3] [WAIT 30 SETSH 4 WAIT 30 ▶
    SETSH 3 WAIT 30 HT]
  EACH [SETSH SUM 1 WHO]
  REPEAT 80 [FD 2 TOOT 0 20 15 2]
  END

  MAKE "NUM 1
  MAKE "DD [0 0 0 129 66 36 153 90 60 0 ▶
    0 0 0 0 0 0]
  MAKE "CC [0 0 0 129 66 60 24 24 153 90 ▶
    36 0 0 0 0]
  MAKE "BB [0 0 24 152 66 60 24 24 152 ▶
    90 36 4 2 2 0 0]
  MAKE "AA [24 24 24 152 64 56 216 24 ▶
    152 88 36 4 2 2 1 1]
  MAKE "GG [0 0 0 0 24 36 60 60 90 165 0 ▶
    0 0 0 0 0]
  MAKE "EE [0 0 0 129 90 102 60 60 24 0 ▶
    0 0 0 0 0]
  MAKE "FF [0 0 0 0 24 36 60 60 24 0 0 0 ▶
    0 0 0 0]

```

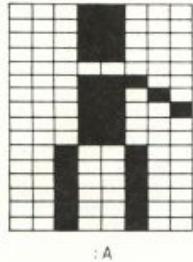
D:JACK2

```

MAKE "A [24 24 24 24 0 28 26 25 24 24 ▶
  36 36 36 36 36 36]
MAKE "B [24 24 24 24 0 31 24 24 24 24 ▶
  39 32 32 32 32 32]
MAKE "C [24 24 24 25 2 28 27 24 25 26 ▶
  36 32 64 64 128 128]
MAKE "D [24 24 24 25 2 28 27 24 24 24 ▶
  231 0 0 0 0]
MAKE "E [24 24 24 24 0 27 28 26 25 24 ▶
  228 2 1 0 0]
MAKE "F [24 24 24 24 0 24 28 26 25 24 ▶
  36 68 130 2 1 1]

MAKE "AA [24 24 24 152 64 56 216 24 ▶
  152 88 36 4 2 2 1 1]
MAKE "BB [0 0 24 152 66 60 24 24 152 ▶
  90 36 4 2 2 0 0]
MAKE "CC [0 0 0 129 66 60 24 24 153 90 ▶
  36 0 0 0 0 0]
MAKE "DD [0 0 0 129 66 36 153 90 60 0 ▶
  0 0 0 0 0 0]
MAKE "EE [0 0 0 129 90 102 60 60 24 0 ▶
  0 0 0 0 0 0]
MAKE "FF [0 0 0 0 24 36 60 60 24 0 0 0 ▶
  0 0 0 0]
  MAKE "GG [0 0 0 0 24 36 60 60 90 165 0 ▶
    0 0 0 0 0]
  MAKE "CARP0 [[SETPOS [30 25] PD SETH ▶
    90] [FD 120] [SETY 24 RT 180 FD ▶
    120] [SETY 23 RT 180 FD 120]]
  MAKE "CARP1 [[SETPOS [150 25] PD SETH ▶
    0] [FD 60] [LT 60 FD 46 PU] ▶
    [SETPOS [135 60] SETH 270 PD]]
  MAKE "CARP2 [[SETPOS [30 25] PD SETH ▶
    0] [FD 40] [RT 30 FD 40] [RT 120 ▶
    FD 40]]
  MAKE "CARP3 [[SETPOS [70 25] PD SETH ▶
    0] [FD 60] [RT 60 FD 46 PU] ▶
    [SETPOS [85 80] SETH 90 PD]]
  MAKE "CARP10 [[SETY 22 RT 180 FD 120 ▶
    PU] [SETPOS [70 85] SETH 90 PD FD ▶
    80]]
  MAKE "CARP11 [[WIN PU] [SETPOS [140 ▶
    30] PD DOOR]]
  MAKE "CARP12 [[RT 120 FD 40 PU] ▶
    [SETPOS [35 25] SETH 0 PD REPEAT ▶
    2 [FD 35 RT 90 FD 30 RT 90]]]
  MAKE "CARP13 [[WIN PU] [SETPOS [85 55] ▶
    PD BAY]]

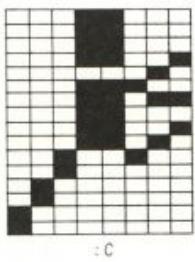
```



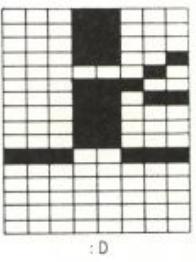
:A



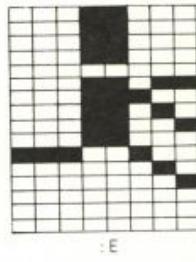
:B



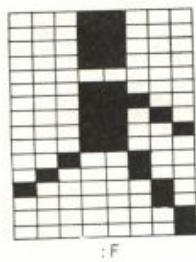
:C



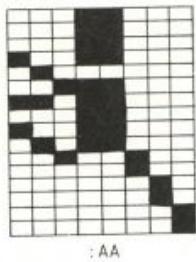
:D



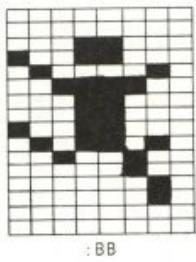
:E



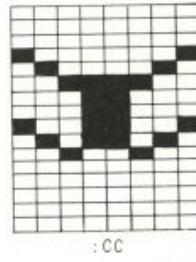
:F



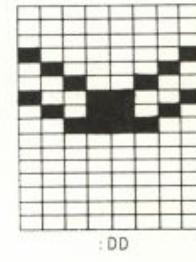
:AA



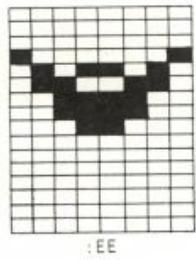
:BB



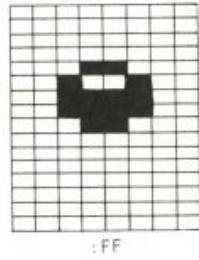
:CC



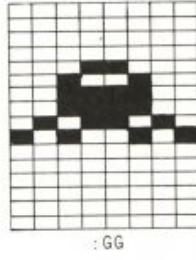
:DD



:EE



:FF



:GG

D:JACK3

```

TO RESET
ERALL
WRAP
SETBG 74
ASK [0 1 2 3] [SETC 7 SETSH 0 CS PD]
TELL 0
END

TO RERUN
ER "RERUN
FS

```

```

LOAD "D:JACK1
ACTIVATE [1 2 3 4 5 6] 65
TELL [0 1 2 3]
BUILD.HOUSE
ERNS
TELL 0
ASK [0 1 2 3] [SETSP 0 HT SETC 22 PU ▶
    HOME]
RECYCLE
C
END

```

```

MAKE "CARP0 [[SETPOS [30 25] PE SETH ►
  90] [FD 120] [SETY 24 RT 180 FD ►
  120] [SETY 23 RT 180 FD 120]]
MAKE "CARP1 [[SETPOS [150 25] PE SETH ►
  0] [FD 60] [LT 60 FD 46 PU] ►
  [SETPOS [135 60] SETH 270 PE]]
MAKE "CARP2 [[SETPOS [30 25] PE SETH ►
  0] [FD 40] [RT 30 FD 40] [RT 120 ►
  FD 40]]
MAKE "CARP3 [[SETPOS [70 25] PE SETH ►
  0] [FD 60] [RT 60 FD 46 PU] ►
  [SETPOS [85 80] SETH 90 PE]]

```

D:JACK4

```

MAKE "CARP10 [[SETY 22 RT 180 FD 120 ►
  PU] [SETPOS [70 85] SETH 90 PE FD ►
  80]]
MAKE "CARP11 [[WIN PU] [SETPOS [140 ►
  30] PE DOOR]]
MAKE "CARP12 [[RT 120 FD 40 PU] ►
  [SETPOS [35 25] SETH 0 PE REPEAT ►
  2 [FD 35 RT 90 FD 30 RT 90]]]
MAKE "CARP13 [[WIN PU] [SETPOS [85 55] ►
  PE BAY]]
MAKE "A [0 0 0 0 0 255 255 255 ►
  255 255 255 255 96 96]
MAKE "B [0 0 0 0 0 224 224 239 239 ►
  255 255 255 255 102 102]
MAKE "C [0 0 0 0 32 16 139 135 255 255 ►
  0 0 0 0 0 0]
MAKE "D [35 19 11 7 6 142 254 12 136 ►
  240 0 0 0 0 0 0]
MAKE "E [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 255 ►
  255 96 96]
MAKE "F [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 239 255 ►
  255 102 102]

```

The six shapes :A through :F in this file (D:JACK4) are the same as the shapes :SHAPE1 through :SHAPE6 in the file D:JACK.

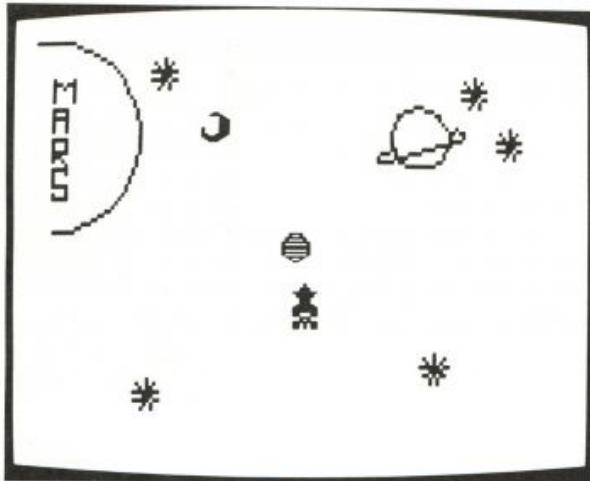
Rocket

ROCKET creates an outer-space scenario. Imagine that you're standing on Demos, a mythical planet orbiting somewhere in the galaxy. Views of Mars, Saturn, a few stars, and some mountains surround you. A rocketship appears and blasts off with musical fanfare. After traveling halfway up the screen, the ship's crew realizes that it has forgotten the pilot and returns to the original take-off site. The pilot races across the screen and boards the ship. The rocketship takes off again, amid blast-off sounds and color flashes, and heads due north. As the ship travels through space, the view of the mountains disappears and is replaced by a couple of stars. Soon one of the moons orbiting Mars is in view, as well as a mysterious green planet moving west. Eventually the rocketship and green planet collide, and a message appears on the screen that tells you the crew has reached a planet they wish to explore and your adventure is over, and bids you farewell.

Overview

This discussion presents an overview of how the program works and describes the process I went through in designing it. Few of the procedures

By Lauren Young.



are listed within the discussion. They are all listed at the end of this write-up.

The Rocket

This whole project began with a lone rocketship I had designed, traveling up the screen. Before it took off on a flight through the galaxy, I added fire to it. That is, I designed a shape that I put beneath the rocketship; it looks like fire emerging from the ship. The shapes are in slots 2 and 3.

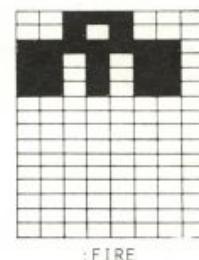
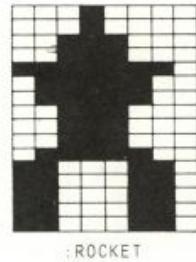
Turtle 1 carries the shape of the rocketship; turtle 2 carries that of the fire. SET.SHIP sets turtles 1 and 2 near the bottom of the screen, one above the other.

```
TO SET.SHIP
  TELL [1 2]
  PU
  ASK 1 [SETSH 2 SETPOS [0 -90] SETC 7]
  ASK 2 [SETSH 3 SETPOS [0 -110] SETC 32]
  ST FS
END
```

The Background Scenery

Saturn, Mars, Stars, and Mountains

I wanted something interesting and colorful, but what? I thought about what one might see on a voyage through the galaxy—a planet! That's where I began. When I finished, I had written procedures for two planets, Mars and Saturn, three stars, and some mountains: PLANET.MARS, PLANET.SATURN, STARS, and MOUNTAINS.



I picked Mars because its name is short. This was important because I wanted to draw the planet's name within the outline of the planet. I designed the letters M, A, R, and S and the procedure MARS that puts the letters inside a semicircle on the screen. PLANET.MARS is the procedure that connects these procedures.

I wanted to include a familiar and smaller—seemingly more distant—planet in the upper-right quadrant of the screen. Saturn was an easy choice from among the planets because of its ring.

What is a galaxy without stars? The procedure STARS tells turtles 0, 1, and 2 to each draw a star.

I was satisfied with the upper half of the screen, filled with two planets and three stars. The bottom half was still bare and needed a landscape. I thought about mountains and decided to adapt two procedures, MOUNTAINS and SUBMOUNTAIN, from the *Atari Logo Reference Manual*.*

Putting It Together

BACKGROUND.SCENE sets up the entire scene.

```
TO BACKGROUND.SCENE
TELL 0
SETBG 72
PLANET.MARS
STARS
PLANET.SATURN
MOUNTAINS
END
```

Ready, Set, Action!

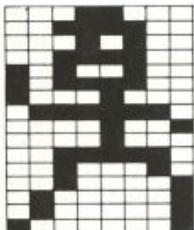
Now, for some more action! I wanted to try animating a person walking across the bottom of the screen. Susan Cotten's Exercise project gave me some ideas, and I started experimenting. I designed three shapes (PERSON.1, PERSON.2, and PERSON.3), each with the same head and torso, but with arms and legs in different stances. As in Exercise, I also wanted footprint sounds with each step the person took. The pilot appears to run by having turtle 0 rapidly change its shape to PERSON.2, PERSON.3, and PERSON.1.

Launching the Rocketship

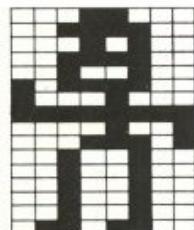
BLAST.OFF sets the rocketship in motion. I put in a few sound effects to jazz up the takeoff. About midscreen, the rocketship stops and a message from the ship's crew is printed:

WAIT!!! WE FORGOT THE PILOT.

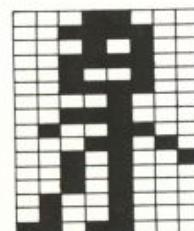
The rocketship reverses its direction and returns to its blast-off site and stops. The pilot races across the bottom of the screen and boards the rocketship. When turtle 0 (the pilot) and turtle 2 (the fire on the rocketship) touch,



:PERSON.1



:PERSON.2



:PERSON.3

*The original version of these procedures is on p. 124 of the manual.

the pilot disappears (HT).

BLAST.OFF2 makes the rocketship take off again. There are take-off sounds and changing background colors; the mountains disappear. The rocketship is on its way. As the rocket continues to travel in space, two more stars are drawn. A message, "GOODBYE ALL!!," is printed on the screen.

Traveling in Space

As the ship travels past Saturn and Mars, a moon that belongs to Mars begins to orbit across the screen. I set up a green planet to orbit in the opposite direction.

When the green planet (turtle 0) and the rocketship (turtle 1) touch, HIT.GREEN.PLANET is called. This procedure makes a siren sound (HIT.SOUND) and prints out a few messages:

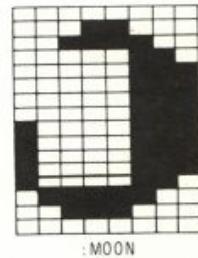
```
WOW!!! THIS LOOKS LIKE A GOOD
PLACE TO EXPLORE. LET'S STOP.
```

```
THIS IS THE END OF OUR ADVENTURE.
IT WAS GOOD HAVING YOU ABOARD.
COME BACK AGAIN SOMETIME.
```

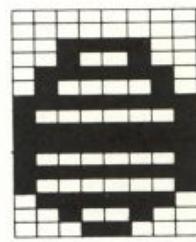
```
SO LONG.
```

TAKE.OFF.ACTION is the procedure that runs most of the action in the story.

```
TO TAKE.OFF.ACTION
SET.SHIP
RECYCLE
BLAST.OFF
GET.PILOT
BLAST.OFF2
MOON.PLANET
FLIGHT.SOUND
END
```



:MOON



:PLANET

Conversation with the User

To involve you, the user, in a personal way, I added the procedure CONVERSATION. While this is one of the first procedures run in ROCKET, it was one of the last added to the project. At the start of ROCKET, Logo prints out:

```
HI THERE.
WHAT'S YOUR NAME?
```

Logo waits for you to type in something. If you type ROXANNE, for example, it will respond:

```
WELCOME TO DEMOS, ROXANNE
PLEASE WAIT A MINUTE WHILE
I DRAW THE PLANETS AND THE STARS.
```

The name you type in at the beginning of the program is remembered throughout. I edited HIT.GREEN.PLANET, one of the procedures used near the end of the story, so that its last statements refers to you by name.

Rocket

ROCKET is the top-level procedure.

```
TO ROCKET
SET.UP
CONVERSATION
BACKGROUND.SCENE
TAKE.OFF.ACTION
CLEAN.UP
END
```

Setting Up and Cleaning Up

SET.UP puts the shapes into the slots in the shape memory and changes the turtles appropriately. SET.UP also tells Logo to change the background color to black (SETBG 0) and to clear the screen. As I also wanted to leave the turtles in their start-up state, I wrote CLEAN.UP.

PROGRAM LISTING

TO PLANET.MARS	A
HT	PU
SETPC 0 21	BK 18
DRAW.MARS	PD
MARS	R
HOME.AGAIN	PU
END	BK 22
	PD
TO DRAW.MARS	S
PU	HOME.AGAIN
SETPOS [-155 120]	END
SETH 90	TO HOME.AGAIN
PD	PU
REPEAT 18 [FD 10 RT 10]	HOME
END	PD
TO MARS	END
PU	TO M
SETPOS [-145 85]	FD 12
SETH 0	RT 135
PD	FD 8
M	LT 90
PU	FD 8
BK 18	RT 135
PD	FD 12

```

PU                               FD 8
BK 12                            RT 90
LT 135                           FD 2
BK 8                             END
RT 90
BK 8
LT 135
BK 12
END

TO A
FD 12
RT 90
FD 8
RT 90
FD 12
BK 6
RT 90
FD 8
PU
LT 90
FD 6
RT 180
END

TO R
FD 12
RT 90
FD 8
RT 90
FD 6
RT 90
FD 8
LT 135
FD 12
PU
BK 12
RT 45
FD 6
RT 180
END

TO S
FD 2
BK 2
RT 90
FD 8
LT 90
FD 8
LT 90
FD 8
RT 90
FD 8
RT 90

FD 8
RT 90
SETPN 0
ASK 0 [SETPLACE [100 90]]
ASK 1 [SETPLACE [120 60]]
ASK 2 [SETPLACE [-80 100]]
STAR
HOME.AGAIN
END

TO STAR
REPEAT 9 [FD 8 BK 8 RT 40]
END

TO SETPLACE :STAR
PU
SETPOS :STAR
PD
END

TO PLANET.SATURN
TELL 0
SETPLACE [50 60]
SATURN
SETRING
RING
HOME.AGAIN
END

TO SATURN
SETPN 2
SETPC 2 52
REPEAT 18 [FD 6 RT 20]
END

TO SETRING
RT 180
REPEAT 6 [FD 1 LT 18]
RT 180
FD 8
END

TO RING
REPEAT 10 [FD 1 LT 18]
FD 45
REPEAT 10 [FD 1 LT 18]
FD 5
END

```

```

TO MOUNTAINS
TELL 0
FS
PU
SETPOS [-158 -60]
PD
RT 45
SETPN 1
SETPC 1 32
SUBMOUNTAIN
END

TO SUBMOUNTAIN
FD 10 + RANDOM 15
IF OR YCOR > 50 YCOR < 0 [SETH 180 - ▶
    HEADING]
IF XCOR > 155 [HOME AGAIN STOP]
SUBMOUNTAIN
END

TO BACKGROUND.SCENE
TELL 0
SETBG 72
PLANET.MARS
STARS
PLANET.SATURN
MOUNTAINS
END

TO GET.PILOT
TELL 0
SETC 20
PU
SETPOS [155 -95]
ST
SETH 270
WHEN TOUCHING 0 2 [HT]
WHEN TOUCHING 0 2 []
REPEAT 10 [WALK]
HT
END

TO WALK
SETH 5
WALK.SOUND
SETH 6
WALK.SOUND
SETH 1
WALK.SOUND
END

TO WALK.SOUND
TOOT 1 200 15 2
WAIT 5
FD 5
END

TO BLAST.OFF
BLAST.OFF.UP
BLAST.OFF.DOWN
END

TO BLAST.OFF.UP
WAIT 60
SETSP 20
SOUND.UP 50
SETSP 0
WAIT 20
CT SS
PR [WAIT!!! WE FORGOT THE PILOT.]
WAIT 100
FS
END

TO SOUND.UP :FREQ
TOOT 1 :FREQ 10 10
IF :FREQ > 1700 [STOP]
SOUND.UP :FREQ + 50
END

TO BLAST.OFF.DOWN
SETSP -20
SOUND.DOWN 1700
SETSP 0
END

TO SOUND.DOWN :FREQ
TOOT 1 :FREQ 10 10
IF :FREQ = 50 [STOP]
SOUND.DOWN :FREQ - 50
END

TO BLAST.OFF2
RECYCLE
TAKE.OFF
BLAST.SOUND
FLASH.COLOR
MOUNTAINS.DISAPPEAR
GOODBYE
MORE.STARS
END

TO TAKE.OFF
TELL [1 2]
ST
WAIT 180
SETSP 21
END

```

```

TO BLAST.SOUND
REPEAT 10 [SOUND.OFF]
END

TO SOUND.OFF
TOOT 0 50 10 3
WAIT 5
TOOT 0 55 10 3
WAIT 5
END

TO FLASH.COLOR
WAIT 60
REPEAT 30 [SETBG RANDOM 128 WAIT 2]
SETBG 0
WAIT 180
END

TO MOUNTAINS.DISAPPEAR
SETPC 1 0
SETSP 39
END

TO GOODBYE
CT SS
PR [GOODBYE ALL!!!]
WAIT 60
FS
END

TO MORE.STARS
TELL 3
PU
SETPLACE [-90 -90]
PD
STAR
PU
SETPLACE [75 -75]
PD
STAR
PU
END

TO MOON.PLANET
PU
MOON.MARS
GREEN.PLANET
END

TO MOON.MARS
TELL 3
PU
SETPOS [-155 70]
ST

RT 90
SETSH 4
SETC 7
SETSP 5
END

TO GREEN.PLANET
TELL 0
SETSH 7
WHEN TOUCHING 0 1 [HIT.GREEN.PLANET ▶
STOP]
SETPOS [107 0]
SETH 270
SETC 99
SETSP 5
ST
END

TO HIT.GREEN.PLANET
HIT.SOUND
CT SS
TELL [0 1 2] SETSP 0
PR [WOW!!! THIS LOOKS LIKE A GOOD]
PR [PLACE TO EXPLORE. LET'S STOP.]
WAIT 480 CT
PR [THIS IS THE END OF OUR ADVENTURE.]
PR [IT WAS GOOD HAVING YOU ABOARD]
PR [SE :NAME .]
PR [COME BACK AGAIN SOMETIME.]
WAIT 540
FS
WAIT 180
CT SS
PR [SE [SO LONG,] :NAME]
END

TO HIT.SOUND
REPEAT 4 [NOISE]
END

TO NOISE
TOOT 0 1000 10 20
WAIT 10
TOOT 1 200 10 20
WAIT 10
END

TO TAKE.OFF.ACTION
SET.SHIP
RECYCLE
BLAST.OFF
GET.PILOT
BLAST.OFF2
MOON.PLANET
FLIGHT.SOUND
END

```

```

TO FLIGHT.SOUND
REPEAT 35 [BEAT]
END

TO BEAT
TOOT 0 100 10 3
WAIT 20
TOOT 1 110 10 3
WAIT 20
END

TO CONVERSATION
CT SS
PR [HI THERE.]
PR [WHAT'S YOUR NAME?]
MAKE "NAME RL
CT
PR (SE [WELCOME TO DEMOS,] :NAME)
PR [PLEASE WAIT A MINUTE WHILE]
PR [I DRAW THE PLANETS AND THE STARS.]
END

TO SET.UP
SETBG 0
PUTSHAPES
TELL 0
SETPN 0
SETPC 0 21
SETSHAPES
CS
END

TO PUTSHAPES
PUTSH 1 :PERSON.1
PUTSH 2 :ROCKET
PUTSH 3 :FIRE
PUTSH 4 :MOON
PUTSH 5 :PERSON.2
PUTSH 6 :PERSON.3
PUTSH 7 :PLANET
END

TO SETSHAPES
ASK [0 1 2 3] [EACH [SETSH WHO + 1 HT ▶
SETSP 0]]
END

TO SET.SHIP
TELL [1 2]
PU
ASK 1 [SETPOS [0 -90] SETC 7]
ASK 2 [SETPOS [0 -110] SETC 32]
ST FS
END

TO ROCKET
SET.UP
CONVERSATION
BACKGROUND.SCENE
TAKE.OFF.ACTION
CLEAN.UP
END

TO CLEAN.UP
TELL [0 1 2 3]
SETBG 0 SETC 7
CS HT
SETSH 0
TELL 0 ST
SETBG 74
CT SS
END

MAKE "ROCKET [16 16 56 56 254 124 124 ▶
56 56 56 124 254 254 254 198 198]
MAKE "FIRE [56 40 254 214 214 214 0 ▶
0 0 0 0 0 0 0 0]
MAKE "PERSON.1 [24 60 20 60 164 188 ▶
136 126 9 8 62 34 33 65 65 193]
MAKE "PERSON.2 [24 60 20 60 36 188 136 ▶
255 9 29 36 36 36 36 108]
MAKE "PERSON.3 [24 60 20 60 36 60 8 56 ▶
76 26 41 40 40 72 72 216]
MAKE "MOON [0 24 60 6 7 7 7 7 135 ▶
135 135 207 254 124 56 0]
MAKE "PLANET [0 0 60 36 126 66 255 129 ▶
255 255 129 255 129 126 36 24]

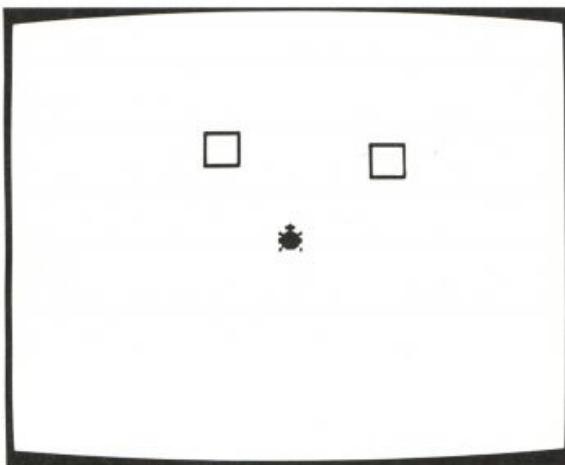
```

3

Games

Boxgame

When you type BOXGAME, two square boxes are put on the screen. They are the targets. A turtle appears in the center of the screen. The goal of the game is to put the turtle inside each box. After you put the turtle inside a box, the box vanishes. BOXGAME gives you experience moving and turning the turtle.



After BOXGAME sets up the boxes and the turtle, it activates demons to watch for the turtle crossing over the lines of the boxes. Then the procedure stops, and you take over and control the turtle directly using commands like FD, BK, LT, or RT. When one of the turtles collides with a line, a demon invokes instructions that make the box disappear.

BOXGAME can be modified so that you control the turtle in different ways. You might want to use special commands, like F for FD 10 and R for RT 15. You might prefer to use a joystick to control the turtle.

In the following discussion I begin by showing how BOXGAME was constructed so that you use Logo primitives like FD and RT. I also show how to introduce new commands like F and R to the game. Lisa Delpit then describes her version of BOXGAME, which she made for some young children

she was working with. I later describe how to change BOXGAME so that you control the turtle with a joystick in port 1. Then I add more frills to the joystick version.

The Procedures

BOXGAME sets up turtle 0 and then turns the rest of the job over to SETUP. The player is in direct control of the turtle.

```
TO BOXGAME
  TELL [0 1 2 3] HT
  TELL 0 SETSH 0 ST
  SETUP
  PR [NOW PUT THE TURTLE IN EACH BOX]
END
```

SETUP calls SETBOXES to put the two targets on the screen and then alerts the demons to watch for turtle 0 crossing over lines drawn by pens 0 or 1. When either event happens, the pen color is changed to the background color and thus the box becomes invisible.

```
TO SETUP
  CS PU
  SETBOXES
  WHEN OVER 0 0 [SETPC 0 BG]
  WHEN OVER 0 1 [SETPC 1 BG]
  SETPN 2 SETPC 2 24
END
```

The boxes are drawn in different colors. Their positions are chosen at random and are likely to be different each time the game is played.

```
TO SETBOXES
  SETPN 0 SETPC 0 7
  DRAWBOX RANDOM 90
  SETPN 1 SETPC 1 100
  DRAWBOX 0 - RANDOM 90
END
```

SETBOXES uses DRAWBOX to put a box on the screen.

```
TO DRAWBOX :ANGLE
  PU SETH :ANGLE
  FD RANDOM 100
  SETH 0 PD
  BOX 20
  PU HOME
END

TO BOX :SIDE
  REPEAT 4 [FD :SIDE RT 90]
END
```

This kind of game can be fun for a while. But it can also be hard work for very young children! Thus you might want to add procedures that will

let the user type in single-key commands for controlling the turtle. For example, when the user types F, the turtle moves forward twenty units.

```
TO F      TO B      TO L      TO R
FD 20     BK 20    LT 15    RT 15
END      END      END      END
```

Bye-Bye Boxes

(A Modification of Boxgame)

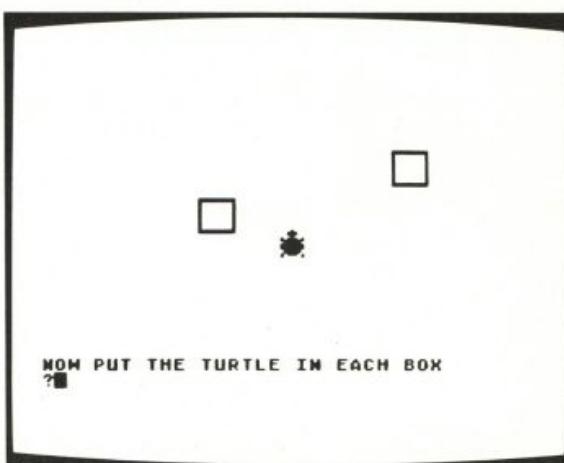
I used Cynthia's BOXGAME with a group of five-year-olds to help them in their left-right orientation, and they loved it. But while they improved their ability to direct the turtle when the turtle's direction was at HEADING 0 (that is, when the turtle's left and right were the same as their left and right), they were still thoroughly confused when the turtle was headed in any other direction. To help solve this problem I modified BOXGAME so that the squares appear in any of eight directions (0, 45, 90, 135, 180, 235, 270, or 315) on the screen at different distances from the center. I also set the turtle up in the center of the screen, but now facing in one of the eight directions. I added sound effects too, partially because I thought the kids would find it interesting but mostly because I enjoy playing with TOOT. The children came up with the catchy name.

The procedures are almost the same as those for BOXGAME, with the additions of GETTURN, which generates the number for the turtle's heading, and DEC.SOUND and INC.SOUND, which add the sound effects.

The Procedures

I will point out where I made changes to BOXGAME. BYEBYE is like BOXGAME except that it calls SETUP.BYE.

```
TO BYEBYE
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETUP.BYE
PR [NOW PUT THE TURTLE IN EACH BOX]
END
```



By Lisa Delpit.

In SETUP.BYE, I add sound to the instructions for the demons.

```
TO SETUP.BYE
CS PU
SETBOXES.BYE
WHEN OVER 0 0 [SETPC 0 BG DEC.SOUND 3500]
WHEN OVER 0 1 [SETPC 1 BG DEC.SOUND 3500]
SETPN 2 SETPC 2 24
END
```

I changed the colors of the squares in SETBOXES.BYE.

```
TO SETBOXES.BYE
SETPN 0 SETPC 0 7
DRAWBOX.BYE
SETPN 1 SETPC 1 100
DRAWBOX.BYE
END
```

DRAWBOX.BYE sets up the turtle for drawing each box at an angle that is a multiple of 45 and at a distance of 25 to 70 steps from the center. This distance is not far enough away to be hidden behind the text at the bottom of the screen. DRAWBOX.BYE, then, turns the turtle to a heading that is a multiple of 45, and the game begins.

```
TO DRAWBOX.BYE
PU
SETH GETTURN
FD 25 + RANDOM 45
SETH 0 PD
INC.SOUND 3000
BOX 20
PU HOME
END
```

I wrote GETTURN so that it outputs one of eight possible numbers, all multiples of 45.

```
TO GETTURN
OP 45 * RANDOM 8
END
```

A sound of increasing frequency accompanies the drawing of the box. A sound of decreasing frequency accompanies the disappearance of the box.

```
TO INC.SOUND :FRE
IF :FRE > 3500 [TOOT 1 4000 8 3 STOP]
TOOT 1 :FRE 6 3
INC.SOUND :FRE + 100
END
```

```
TO DEC.SOUND :FRE
IF :FRE < 3000 [TOOT 1 1000 8 3 STOP]
TOOT 1 :FRE 6 3
DEC.SOUND :FRE - 100
END
```

The following three procedures are unchanged.

```
TO BOX :X
REPEAT 4 [FD :X RT 90]
END
```

```
TO F
FD 20
END
```

```
TO B
BK 20
END
```

The next procedures are changed so that the turtle turns 45 degrees.

```
TO R
RT 45
END
```

```
TO L
LT 45
END
```

Back to Boxgame

Using a Joystick

Another variation of this game is to attach a joystick to the computer and use the joystick to control the turtle. In the next example, pressing the joystick button moves the turtle forward five steps. The joystick moved to the left turns the turtle left 15 degrees; the joystick moved to the right turns the turtle right 15 degrees. To do this BOXGAME has to be changed and a couple of new procedures have to be written for the joystick.

```
TO JOYGAME
CT SS
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETUP
PR [THE JOYSTICK TURNS THE TURTLE]
PR [THE BUTTON MOVES THE TURTLE]
PR [NOW PUT THE TURTLE IN EACH BOX]
JOYREAD
END

TO JOYREAD
IF JOYB 0 [FD 5 ]
IF EQUALP PC 0 PC 1 [STOP]
JOYRD JOY 0
JOYREAD
END
```

GAMES

```
TO JOYRD :STICK
IF :STICK = 6 [LT 15]
IF :STICK = 2 [RT 15]
END
```

Extending JOYGAME

JOYGAME will set the turtle's speed. I use a speed of 100, but you might want to change this. This time when the turtle goes over a pen line, the background changes color. Finally, the game starts up again.

JOYGAME needs to be changed. I also want to change the setting-up procedure. Let's rename the new versions of these procedures.

```
TO NEWGAME
CT SS
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETJOY
SETSP 100
PR [THE JOYSTICK TURNS THE TURTLE]
PR [THE BUTTON MOVES THE TURTLE]
PR [NOW PUT THE TURTLE IN EACH BOX]
JOYREAD
NEWGAME
END

TO SETJOY
CS PU
SETBOXES
WHEN OVER 0 0 [FLASH BG SETPC 0 BG]
WHEN OVER 0 1 [FLASH BG SETPC 1 BG]
SETPN 2 SETPC 2 24
END
```

Here is the new procedure.

```
TO FLASH :BG
REPEAT 6 [SETBG RANDOM 100 WAIT 5]
SETBG :BG
END
```

PROGRAM LISTING

TO BOXGAME	SETBOXES
TELL [0 1 2 3] HT	WHEN OVER 0 0 [SETPC 0 BG]
TELL 0 SETSH 0 ST	WHEN OVER 0 1 [SETPC 1 BG]
SETUP	SETPN 2 SETPC 2 24
PR [NOW PUT THE TURTLE IN EACH BOX]	END
END	
TO SETUP	TO BOX :SIDE
CS PU	REPEAT 4 [FD :SIDE RT 90]
	END

```

TO SETBOXES
SETPN 0 SETPC 0 7
DRAWBOX RANDOM 90
SETPN 1 SETPC 1 100
DRAWBOX 0 - RANDOM 90
END

TO DRAWBOX :ANGLE
PU SETH :ANGLE
FD RANDOM 100
SETH 0 PD
BOX 20
PU HOME
END

TO BYEBYE
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETUP.BYE
PR [NOW PUT THE TURTLE IN EACH BOX]
END

TO SETUP.BYE
CS PU
SETBOXES
WHEN OVER 0 0 [SETPC 0 BG DEC.SOUND ▶
 3500]
WHEN OVER 0 1 [SETPC 1 BG DEC.SOUND ▶
 3500]
SETPN 2 SETPC 2 24
END

TO SETBOXES.BYE
SETPN 0 SETPC 0 7
DRAWBOX.BYE
SETPN 1 SETPC 1 100
DRAWBOX.BYE
END)

TO DRAWBOX.BYE
PU
SETH GETTURN
FD 25 + RANDOM 45
SETH 0 PD
INC.SOUND 3000
BOX 20
PU HOME
END

TO GETTURN
OP 45 * RANDOM 8
END

TO INC.SOUND :FRE
IF :FRE > 3500 [TOOT 1 4000 8 3 STOP]
TOOT 1 :FRE 6 3
INC.SOUND :FRE + 100
END

TO DEC.SOUND :FRE
IF :FRE < 3000 [TOOT 1 1000 8 3 STOP]
TOOT 1 :FRE 6 3
DEC.SOUND :FRE - 100
END

TO F
FD 20
END

TO B
BK 20
END

TO R
RT 45
END

TO L
LT 45
END

TO JOYGAME
CT SS
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST
SETUP
PR [THE JOYSTICK TURNS THE TURTLE]
PR [THE BUTTON MOVES THE TURTLE]
PR [NOW PUT THE TURTLE IN EACH BOX]
JOYREAD
END

TO JOYREAD
IF JOYB 0 [FD 5 ]
IF EQUALP PC 0 PC 1 [STOP]
JOYRD JOY 0
JOYREAD
END

TO JOYRD :STICK
IF :STICK = 6 [LT 15]
IF :STICK = 2 [RT 15]
END

TO NEWGAME
CT SS
TELL [0 1 2 3] HT
TELL 0 SETSH 0 ST

```

```

SETJOY
SETSP 100
PR [THE JOYSTICK TURNS THE TURTLE]
PR [NOW PUT THE TURTLE IN EACH BOX]
JOYREAD.NEW
NEWGAME
END

TO JOYREAD.NEW
IF EQUALP PC 0 PC 1 [STOP]
JOYRD JOY 0
JOYREAD.NEW
END

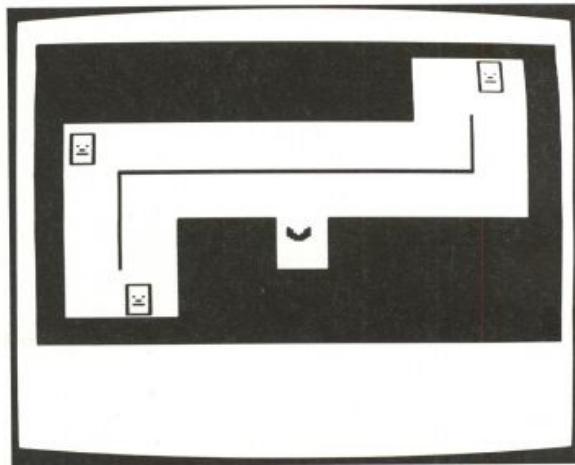
TO SETJOY
CS PU
SETBOXES
WHEN OVER 0 0 [FLASH BG SETPC 0 BG]
WHEN OVER 0 1 [FLASH BG SETPC 1 BG]
SETPN 2 SETPC 2 24
END

TO FLASH :BG
REPEAT 6 [SETBG RANDOM 100 WAIT 5]
SETBG :BG
END

```

Pacgame

PACGAME was inspired by PAC-MAN™ and designed as a learning tool for beginners. You play the game by using turtle commands to move the gobbling pacman around the game board. The game's special effects and features are activated entirely by demons. Thus a player types all commands directly to Logo and demons take care of the game's actions.



Here are the rules I decided on for PACGAME.

- The game is played on a game board. There is a pacman and three targets. Unlike PAC-MAN's ghosts, the targets in PACGAME are stationary.
- Once play begins, it continues until all three targets are gobbled.
- Each target is worth 10,000 points and explodes when it is gobbled.
- All turns need to be multiples of 90 degrees because the pacman can gobble in only four directions. If any other turn is made, the pacman will change back into a turtle and complain.

PAC-MAN is a trademark of Bally Midway Manufacturing Company

By Michael Grandfield.

In addition to the rules, the pacman bounces back onto the game board whenever it goes out of bounds.

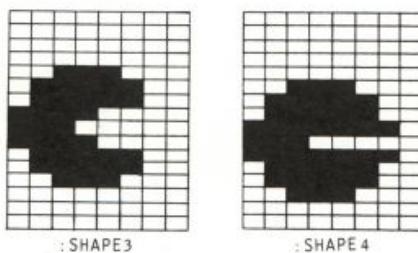
I'd like to present PACGAME by showing the playing pieces and game board I designed and talking about the demons that bring the game to life.

The Playing Pieces and the Game Board

Let's look at these one at a time.

The Pacman

I wanted to make a pacman that could gobble and also behave like a turtle. It took two shapes to animate the pacman, one for an open mouth and one for a closed mouth.

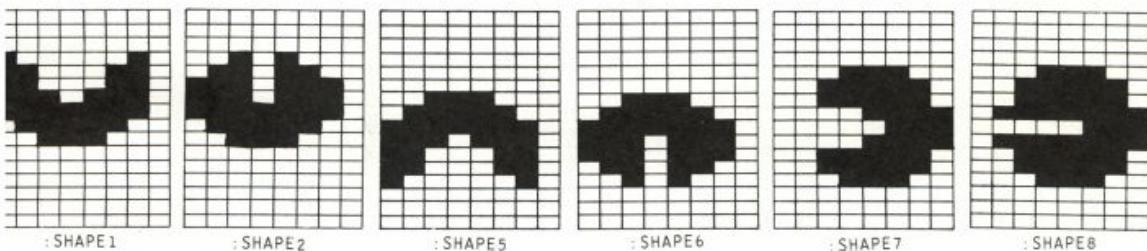


Animating the pacman without limiting its turtle capabilities or interfering with typing in a new command was a problem. However, I found an effective solution by using the once-per-second demon. Here it is.

```
WHEN 7 [SETSH 1 WAIT 30 SETSH 2]
```

This demon runs the instruction list [SETSH 1 WAIT 30 SETSH 2] once every second. This makes the animation continue steadily until the demon is halted.

Next I decided that the pacman should be able to gobble in four different directions (up, down, right, and left) and made six more pacman shapes.

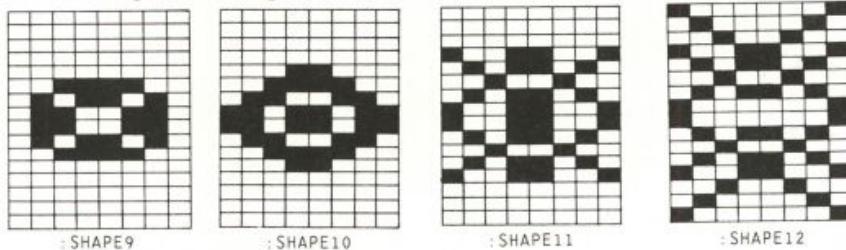


Later I explain how the pacman chooses the pair of shapes that corresponds to its heading.

The Targets

Next I designed a target. The design I settled on has a distinct outline to make it easy to see how far the pacman is from hitting it. It also has a sinister face.

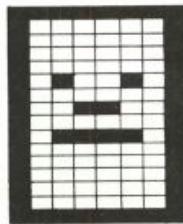
I wanted this target to explode as it was being gobbled, so I made this sequence of shapes.



In the game, three turtles are targets. Collision-detection demons tell when a target has been hit. Here's an example.

WHEN TOUCHING Ø 1 [EXPLODE 1]

:SHAPE13

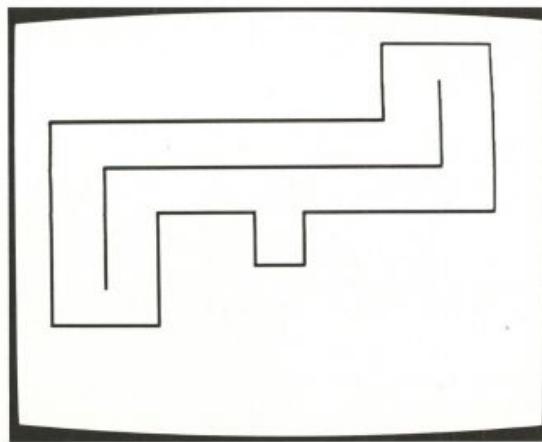


`EXPLODE` gives the turtle representing the target each of the explosion shapes in quick succession and also changes the turtle's color with each change of shape.

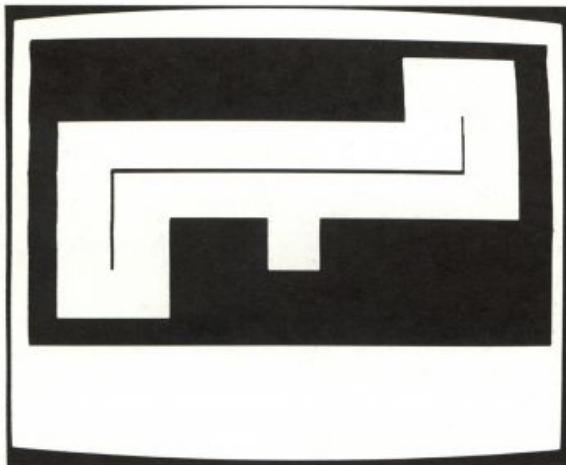
Later I added new instructions for these demons, in order to score the game and congratulate the player.

The Game Board

Originally the game board was a thin outline. It looked like this.



There was a problem with this approach. I wanted a demon to detect any collision between the pacman and a border of the game board. Also, I wanted to keep the pacman in bounds by having it bounce back from a collision. I discovered that these borders were too thin. The demon often



failed to detect a collision. My solution was to make the game board have very thick borders.

Here is the procedure that draws the game board. Notice that I used two pens of different colors to draw the board.

```
TO DRAW.BOARD
SETPC 0 22
SETPC 1 26
TELL 0 SETPN 0 PU HT
SETPOS [-160 120] PD SETH 90
REPEAT 10 [FD 320 SETY YCOR - 1]
REPEAT 40 [FD 230 SETPN 1 FD 70 SETPN 0 FD 20 SETY YCOR - 1]
REPEAT 60 [FD 20 SETPN 1 FD 280 SETPN 0 FD 20 SETY YCOR - 1]
FD 20 SETPN 1 FD 280 SETPN 0 FD 20
REPEAT 34 [FD 20 SETPN 1 FD 70 SETPN 0 FD 58 SETPN 1 FD 30
           SETPN 0 FD 142 SETY YCOR - 1]
REPEAT 30 [FD 20 SETPN 1 FD 70 SETPN 0 FD 230 SETY YCOR - 1]
REPEAT 15 [FD 320 SETY YCOR - 1]
PU
SETPOS [-105 -20] SETH 0 PD
FD 60 RT 90 FD 210 LT 90 FD 35
PU
END
```

Bringing the Game to Life

The main procedure is PACGAME. It calls SETUP and PLAY.

```
TO PACGAME
SETUP
PLAY
END
```

Setting Up

SETUP clears all graphics and text from the screen, and calls PUT.SHAPES and DRAW.BOARD.

GAMES

```
TO SETUP
CS SS CT
PUT.SHAPES 1
SETUP.VARS
DRAW.BOARD
END
```

PUT.SHAPES puts all the shapes, which are stored as lists of numbers, into the shape slots. It also erases the variables that contained the lists. (I guess I always like to free up as much of the workspace as I can.)

```
TO PUT.SHAPES :NUM
IF :NUM > 15 [STOP]
PUTSH :NUM THING WORD "SHAPE :NUM
ERN WORD "SHAPE :NUM
PUT.SHAPES :NUM + 1
END
```

SETUP.VARS sets up variables that the program uses.

```
TO SETUP.VARS
MAKE "270 7
MAKE "180 5
MAKE "90 3
MAKE "0 1
MAKE "POS3 [-95 -40]
MAKE "POS2 [-130 55]
MAKE "POS1 [116 100]
MAKE "PURPLE 70
MAKE "PACMAN 0
MAKE "BLACK 0
MAKE "TARGET 9
END
```

You have already seen DRAW.BOARD in the description of the game pieces.

The Play

The procedure PLAY calls two setup procedures, SET.PIECES and SET.DEMONS. Once these procedures have been run, the game begins.

```
TO PLAY
SET.PIECES
SET.DEMONS
END
```

SET.PIECES sets the score to 0 and places the pacman and the targets on the gameboard in the correct positions to begin the game. It also creates the variable PAC.POSITION.

During the game PAC.POSITION is given the value of the pacman's current position. This value changes at regular intervals, provided that the pacman remains in bounds. If the pacman goes out of bounds

PAC.POSITION is not given a new value, so the pacman can bounce back to the position that is the value of PAC.POSITION.

```
TO SET.PIECES
MAKE "SCORE 0
TELL 0 HT HOME SETC 44
ASK [1 2 3] [HT PU SETC 7 SETSH :TARGET]
ASK [1 2 3] [EACH [SETPOS THING WORD "POS WHO] ST]
CT ST
MAKE "PAC.POSITION POS
END
```

SET.DEMONS creates demons. These demons animate the pacman and detect collisions. The procedures that these demons call are the guts of the game.

```
TO SET.DEMONS
WHEN 7 [IF MEMBERP HEADING [0 90 180 270]
          [ANIMATE.PACMAN]
          [REVEAL.TURTLE]]
WHEN OVER 0 0 [STAY.IN.BOUNDS]
WHEN TOUCHING 0 1 [BULLSEYE 1]
WHEN TOUCHING 0 2 [BULLSEYE 2]
WHEN TOUCHING 0 3 [BULLSEYE 3]
END
```

Procedures Called by the Demons

Animating the Pacman

You can see that I have changed the instructions for the once-per-second demon (7) from the earlier example. The demon checks to see if the pacman has heading 0, 90, 180, or 270. If so, it animates the pacman; otherwise it reveals the original turtle shape. As you will see, this demon is able to do several different jobs neatly.

The procedure ANIMATE.PACMAN uses two interesting programming tricks.

```
TO ANIMATE.PACMAN
SETSH THING HEADING
WAIT 10
UPDATE
SETSH 1 + THING HEADING
END
```

The first trick is that I have given each shape a variable name that is the same as the heading associated with the shape. For example, the shape that gobbles upward is used when the turtle has a heading of 0, and is given the name "0". The variable for this shape is created by the instruction

```
MAKE "0 1
```

Now if I type

PR :0

or

PR THING 0

Logo will respond

1

If the pacman's heading is 0, I can also type

SETSH THING HEADING

and Logo will respond as if I had typed

SETSH 1

The second trick is that I decided to use the bit of time between shape changes to call the procedure UPDATE. This procedure checks to see if the pacman is still in bounds. If so, UPDATE gives a new value to the variable PAC.POSITION.

```
TO UPDATE
IF COND OVER 0 1 [MAKE "PAC.POSITION POS]
END
```

To sum up, ANIMATE.PACMAN changes the shape of the pacman to correspond to its heading, animates its gobbling, and calls UPDATE.

Here is the procedure REVEAL.TURTLE. It is called whenever a turn that is not a multiple of 90 degrees is made.

```
TO REVEAL.TURTLE
SETSH 0
SETCURSOR [0 19]
TYPE [\ FIX HEADING TO GET BACK IN THE GAME\ ]
END
```

REVEAL.TURTLE makes it clear when a turn is not within the rules by showing the turtle shape and protects the game from crashing by allowing the pacman to respond to any turning instruction.

Staying in Bounds

STAY.IN.BOUNDS is called whenever the pacman bumps into a boundary of the board. It makes a bumping sound, sends the pacman back to :PAC.POSITION, and complains OUCH!!!.

```
TO STAY.IN.BOUNDS
TOOT 0 100 7 15
TOOT 1 200 7 15
SETPOS :PAC.POSITION
PR [] PR "OUCH!!! PR []
END
```

Gobbling the Targets

BULLSEYE is called whenever the pacman hits a target. It temporarily stops the gobbling animation by halting the once-per-second demon and lets you know that the pacman has hit a target by changing the pacman's color. Next BULLSEYE updates the score and tells the gobbled target to explode. It also offers some congratulations and prints your score on the screen. Finally BULLSEYE checks the score to see whether to continue the game or declare a victory.

```
TO BULLSEYE :WHICH.TARGET
WHEN 7 [] SETC :BLACK
ADD.TO.SCORE
EXPLODE :WHICH.TARGET
( PR :SCORE [TO BE EXACT] )
WAIT 90
IF NOT :SCORE = 30000 [ON.WITH.THE.GAME] [VICTORY]
END

TO ADD.TO.SCORE
MAKE "SCORE 10000 + :SCORE
CT PR [BOINK!!! WOWIEE!!!]
PR [SCORE LOTS OF POINTS FOR YOU!]
PR []
END
```

The procedure EXPLODE animates the explosion by calling EXPLODE1.

```
TO EXPLODE :TARGET
ASK :TARGET [EXPLODE1 [9 10 11 12 13 12 13 12 13 12]
[22 31 55 79 55 79 55 79 55 7]
HT HOME]
END

TO EXPLODE1 :SHAPES :COLORS
IF EMPTYP :SHAPES [STOP]
TOOT 0 20 10 7 TOOT 1 25 10 6
SETSH FIRST :SHAPES
SETC FIRST :COLORS
EXPLODE1 BF :SHAPES BF :COLORS
END
```

The tricky part of EXPLODE1 was synchronizing sound and animation. I use both voices to emit a sound before each shape and color change. Thus I use two TOOT commands. The shape and color changes begin before the sound dies away, so all three events happen together.

On with the Game

ON.WITH.THE.GAME continues the game after a target has been gobbled. It resets the pacman's color and resets the once-per-second demon.

GAMES

```

TO ON.WITH.THE.GAME
TELL :PACMAN SETC :PURPLE
WHEN 7 [IF MEMBERP HEADING [0 90 180 270]
          [ANIMATE.PACMAN]
          [REVEAL.TURTLE]]
END

```

Winning the Game

VICTORY is called only when the score has reached 30,000. It celebrates winning the game by playing a rousing fanfare and flashing the background color in time to the music.

```

TO VICTORY
RECYCLE
TELL 0
CT
PR [THE WINNER, AND STILL CHAMP!]
FANFARE
WAIT 180
PR [DO YOU WANT TO PLAY AGAIN?]
END

```

FANFARE calls FANFARE1, which plays some rousing music to celebrate.

```

TO FANFARE
FANFARE1 55 110 39 40 7040
FANFARE1 35 70 31 48 7040
FANFARE1 30 60 23 56 7040
FANFARE1 25 50 15 64 7040
FANFARE1 30 50 78 71 7680
FANFARE1 120 200 63 56 7680
SETPC 0 22
SETPC 1 26
SETBG 64
SETENV 0 15 SETENV 1 15
TOOT 0 240 15 240 TOOT 1 400 15 240
END

```

FANFARE1 invokes TOOT for both voices, then changes the color of the game board and the background. Each time FANFARE1 repeats, the notes increase an octave in pitch, until the :HIGHFREQ limit is reached.

```

TO FANFARE1 :FREQ0 :FREQ1 :PENCOLOR
              :SCREENCOLOR :HIGHFREQ
IF :FREQ0 > 7040 [STOP]
TOOT 0 :FREQ0 15 10 TOOT 1 :FREQ1 15 7
SETPC 0 :PENCOLOR
SETPC 1 :PENCOLOR + 4
SETBG :SCREENCOLOR
FANFARE1 :FREQ0 * 2 :FREQ1 * 2
          :PENCOLOR - 1 :SCREENCOLOR + 1
          :HIGHFREQ
END

```

When FANFARE1 has been called for the final time, the game board is returned to its original colors.

Finally VICTORY asks DO YOU WANT TO PLAY AGAIN?. Here are the procedures, YES and NO, that are called by your response.

```
TO YES
PLAY
END
```

NO reinitializes Logo and clears the workspace.

```
TO NO
SETBG 74
SETPC 0 22
TELL [0 1 2 3] SETSH 0 CT
TELL 0 HOME ST SETPN 0
ERALL
RECYCLE
END
```

PROGRAM LISTING

```
TO PACGAME
SETUP
PLAY
END

TO SETUP
CS SS CT
PUT.SHAPES 1
SETUP.VARS
DRAW.BOARD
END

TO PUT.SHAPES :NUM
IF :NUM > 15 [STOP]
PUTSH :NUM THING WORD "SHAPE :NUM"
ERN WORD "SHAPE :NUM"
PUT.SHAPES :NUM + 1
END

TO SETUP.VARS
MAKE "270 7
MAKE "180 5
MAKE "90 3
MAKE "@ 1
MAKE "POS3 [-95 -40]
MAKE "POS2 [-130 55]
MAKE "POS1 [116 100]
MAKE "PURPLE 70
MAKE "PACMAN 0
MAKE "BLACK 0
MAKE "TARGET 9
END

TO DRAW.BOARD
SETPC 0 22
SETPC 1 26
TELL 0 SETPN 0 PU HT
SETPOS [-160 120] PD SETH 90
REPEAT 10 [FD 320 SETY YCOR - 1]
REPEAT 40 [FD 230 SETPN 1 FD 70 SETPN ▶
    0 FD 20 SETY YCOR - 1]
REPEAT 60 [FD 20 SETPN 1 FD 280 SETPN ▶
    0 FD 20 SETY YCOR - 1]
FD 20 SETPN 1 FD 280 SETPN 0 FD 20
REPEAT 34 [FD 20 SETPN 1 FD 70 SETPN 0 ▶
    FD 58 SETPN 1 FD 30 SETPN 0 FD 142 ▶
    SETY YCOR - 1]
REPEAT 30 [FD 20 SETPN 1 FD 70 SETPN 0 ▶
    FD 230 SETY YCOR - 1]
REPEAT 15 [FD 320 SETY YCOR - 1]
PU
SETPOS [-105 -20] SETH 0 PD
FD 60 RT 90 FD 210 LT 90 FD 35
PU
END

TO PLAY
SET.PIECES
SET.DEMONS
END

TO SET.PIECES
MAKE "SCORE 0
END
```

```

TELL Ø HT HOME SETC 44
ASK [1 2 3] [HT PU SETC 7 SETSH ▶
:TARGET]
ASK [1 2 3] [EACH [SETPOS THING WORD ▶
"POS WHO] ST]
CT ST
MAKE "PAC.POSITION POS
END

TO SET.DEMONS
WHEN 7 [IF MEMBERP HEADING [Ø 9Ø 18Ø ▶
27Ø] [ANIMATE.PACMAN] ▶
[REVEAL.TURTLE]]
WHEN OVER Ø Ø [STAY.IN.BOUNDS]
WHEN TOUCHING Ø 1 [BULLSEYE 1]
WHEN TOUCHING Ø 2 [BULLSEYE 2]
WHEN TOUCHING Ø 3 [BULLSEYE 3]
END

TO ANIMATE.PACMAN
SETSH THING HEADING
WAIT 1Ø
UPDATE
SETSH 1 + THING HEADING
END

TO UPDATE
IF COND OVER Ø 1 [MAKE "PAC.POSITION ▶
POS]
END

TO REVEAL.TURTLE
SETSH Ø
SETCURSOR [Ø 19]
TYPE [\ FIX HEADING TO GET BACK IN THE ▶
GAME\ ]
END

TO STAY.IN.BOUNDS
TOOT Ø 10Ø 7 15
TOOT 1 20Ø 7 15
SETPOS :PAC.POSITION
PR [] PR "OUCH!!! PR []
END

TO BULLSEYE :WHICH.TARGET
WHEN 7 [] SETC :BLACK
ADD.TO.SCORE
EXPLODE :WHICH.TARGET
( PR :SCORE [TO BE EXACT] )
WAIT 9Ø
IF NOT :SCORE = 30000 ▶
[ON.WITH.THE.GAME] [VICTORY]
END

TO ADD.TO.SCORE
MAKE "SCORE 10000 + :SCORE
CT PR [BOINK!!! WOWIEE!!!]
PR [SCORE LOTS OF POINTS FOR YOU!]
PR []
END

TO EXPLODE :TARGET
ASK :TARGET [EXPLODE1 [9 1Ø 11 12 13 ▶
12 13 12 13 12] [22 31 55 79 55 ▶
79 55 79 55 7] HT HOME]
END

TO EXPLODE1 :SHAPES :COLORS
IF EMPTYP :SHAPES [STOP]
TOOT Ø 2Ø 1Ø 7 TOOT 1 25 1Ø 6
SETSH FIRST :SHAPES
SETC FIRST :COLORS
EXPLODE1 BF :SHAPES BF :COLORS
END

TO ON.WITH.THE.GAME
TELL :PACMAN SETC :PURPLE
WHEN 7 [IF MEMBERP HEADING [Ø 9Ø 18Ø ▶
27Ø] [ANIMATE.PACMAN] ▶
[REVEAL.TURTLE]]
END

TO VICTORY
RECYCLE
TELL Ø
CT
PR [THE WINNER, AND STILL CHAMP!]
FAFAR
WAIT 18Ø
PR [DO YOU WANT TO PLAY AGAIN?]
END

TO FANFARE
FAFAR1 55 11Ø 39 4Ø 704Ø
FAFAR1 35 7Ø 31 48 704Ø
FAFAR1 3Ø 6Ø 23 56 704Ø
FAFAR1 25 5Ø 15 64 704Ø
FAFAR1 3Ø 5Ø 78 71 768Ø
FAFAR1 12Ø 20Ø 63 56 768Ø
SETPC Ø 22
SETPC 1 26
SETBG 64
SETENV Ø 15 SETENV 1 15
TOOT Ø 24Ø 15 24Ø TOOT 1 40Ø 15 24Ø
END

TO FANFARE1 :FREQØ :FREQ1 :PENCOLOR ▶
:SCREENCOLOR :HIGHFREQ

```

```

IF :FREQ0 > 7040 [STOP]
TOOT 0 :FREQ0 15 10 TOOT 1 :FREQ1 15 7
SETPC 0 :PENCOLOR
SETPC 1 :PENCOLOR + 4
SETBG :SCREENCOLOR
FANFARE1 :FREQ0 * 2 :FREQ1 * 2 ▶
    :PENCOLOR - 1 :SCREENCOLOR + 1 ▶
    :HIGHFREQ
END

TO YES
PLAY
END

TO NO
SETBG 74
SETPC 0 22
TELL [0 1 2 3] SETSH 0 CT
TELL 0 HOME ST SETPN 0
ERALL
RECYCLE
END

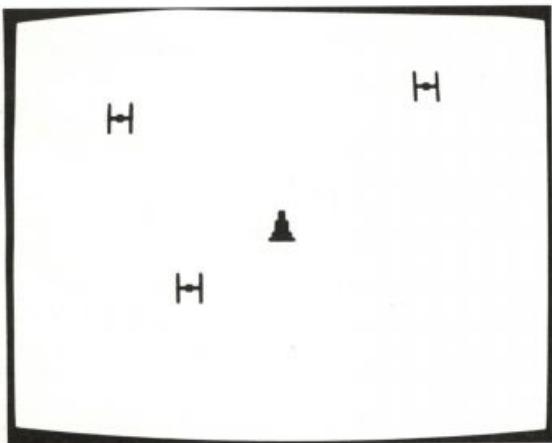
MAKE "SHAPE15 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ▶
    0 0 0 0]
MAKE "SHAPE14 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ▶
    0 0 0 0]

MAKE "SHAPE13 [129 66 36 153 90 36 90 ▶
    129 129 90 36 90 153 36 66 129]
MAKE "SHAPE12 [0 0 0 153 90 36 90 153 ▶
    153 90 36 90 153 0 0 0]
MAKE "SHAPE11 [0 0 0 0 24 60 102 219 ▶
    219 102 60 24 0 0 0 0]
MAKE "SHAPE10 [0 0 0 0 0 60 90 102 102 ▶
    90 60 0 0 0 0]
MAKE "SHAPE9 [255 129 129 129 129 129 ▶
    165 129 153 129 189 129 129 129 ▶
    129 255]
MAKE "SHAPE8 [0 0 0 0 28 62 62 127 7 ▶
    127 62 62 28 0 0 0]
MAKE "SHAPE7 [0 0 0 0 28 62 62 15 7 15 ▶
    62 62 28 0 0 0]
MAKE "SHAPE6 [0 0 0 0 0 0 56 124 254 ▶
    238 238 108 40 0 0 0]
MAKE "SHAPE5 [0 0 0 0 0 0 56 124 254 ▶
    238 198 198 130 0 0 0]
MAKE "SHAPE4 [0 0 0 0 56 124 124 254 ▶
    224 254 124 124 56 0 0 0]
MAKE "SHAPE3 [0 0 0 0 56 124 124 240 ▶
    224 240 124 124 56 0 0 0]
MAKE "SHAPE2 [0 0 0 0 40 108 238 238 ▶
    254 124 56 0 0 0 0]
MAKE "SHAPE1 [0 0 0 0 130 198 198 238 ▶
    254 124 56 0 0 0 0]

```

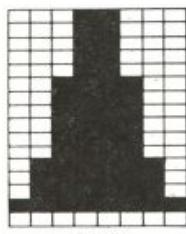
Blaster

That's your spaceship in the middle of the screen. You can steer with the joystick and fire lasers at the three enemy ships that surround you. You get points for hitting their ships. If an enemy ship collides with you, you lose a life. The game ends when you've lost five lives.

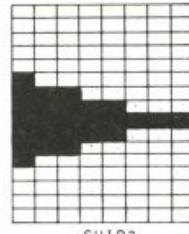


By Brian Harvey.

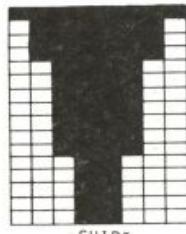
Turtle 0 represents your ship. It never moves from the center of the screen, but it can turn in different directions depending on the position of the joystick. It has eight possible shapes, each representing the ship facing one of the eight possible directions.



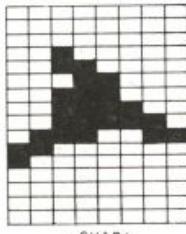
:SHIP1



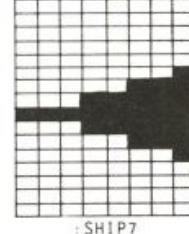
:SHIP3



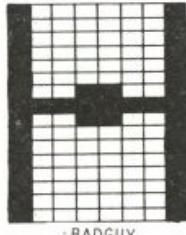
:SHIP5



:SHIP6



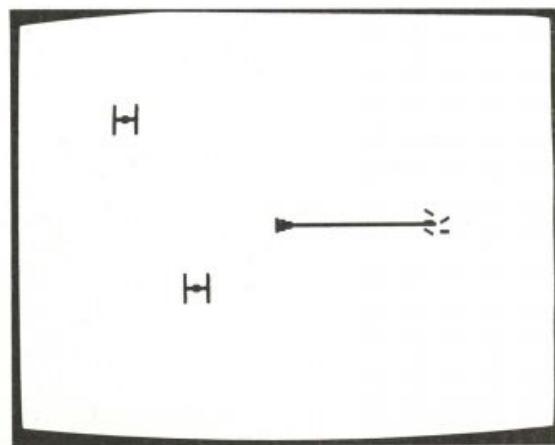
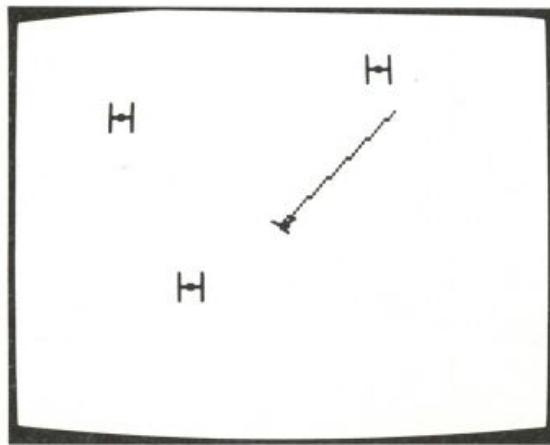
:SHIP7



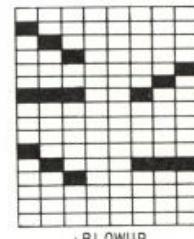
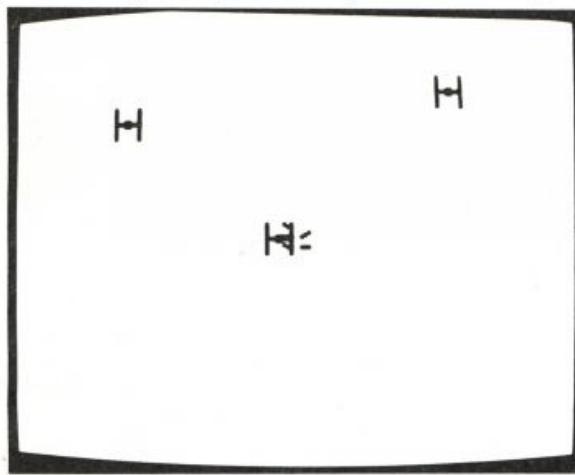
:BADGUY

Turtles 1, 2, and 3 are the bad guys. They move at random speeds. Their direction is always more or less toward you, but not necessarily directly toward you. They have only one shape.

You shoot by pressing the joystick button. This makes a red line appear for a moment, pointing in the direction the ship is facing. If this line hits one of the enemy ships, you score a point. Demons are used to detect the shot hitting an enemy.



Demons are also used to detect one of the enemy ships colliding with your ship. (The enemy ships don't fire at you; they have no weapons. All they can do is collide with you. Shame on you for firing at unarmed ships!) In the picture below, your ship has blown up because an enemy ship hit you.



LIVES: 3 POINTS: 14

To start the game, run the procedure BLASTER. It has two subprocedures, one to set up the screen and the other to play the game. The inputs to PLAY.BLASTER are the number of lives you're allowed and the number of points you start with.

```
TO BLASTER
SETUP.BLASTER
PLAY.BLASTER 5 0
END
```

Setting Up

The setup procedure sets colors and shapes, positions the turtles, and uses the PX command to set turtle 0 (your ship) in *penreverse* so that when you shoot, it can display and then erase the blast by retracing the line. Here is SETUP.BLASTER and its subprocedures.

```
TO SETUP.BLASTER
CT
SETBG 0
SETPC 0 40
PUTSHAPES 1 [SHIP1 SHIP2 SHIP3 SHIP4 SHIP5
SHIP6 SHIP7 SHIP8 BADGUY BLOWUP]
MAKE "ENEMIES [1 2 3]
SETUP.ENEMIES
SETUP.PLAYER
END
```

PUTSHAPES takes two inputs. The first input is the starting shape number. The second is a list of names containing shapes (in the list form output

GAMES

by GETSH). It uses PUTSH to copy those shapes into Logo's shape slots. In effect, this procedure replaces what would otherwise be ten individual PUTSH instructions.

```
TO PUTSHAPES :NUMBER :LIST
IF EMPTYP :LIST [STOP]
PUTSH :NUMBER THING FIRST :LIST
PUTSHAPES :NUMBER+1 BF :LIST
END
```

`SETUP.ENEMIES` sets the shape and color of the enemy ships.

```
TO SETUP.ENEMIES
TELL :ENEMIES
HT
SETSH 9
SETC 74
PU
END
```

`SETUP.PLAYER` sets the shape and color of your ship, and tells the turtle to penreverse, as explained earlier.

```
TO SETUP.PLAYER
TELL 0
ST
SETSH 1
SETC 7
CS
PX
END
```

Playing the Game

The main job of `PLAY.BLASTER` is to set up several demons. There is one for the joystick button, to fire a shot; three for the enemy ships colliding with pen 0, when you shoot them; three for the enemy ships colliding with turtle 0, when they hit you; and one for the joystick, to steer your ship. The procedure also puts the enemy ships in random positions, prints the initial score, and invokes `BLASTER LOOP` to play the game.

Two variables are used throughout this part of the program to keep track of scoring. These variables are the two inputs to `PLAY.BLASTER`, called `LIVES` and `POINTS`.

<code>LIVES</code>	The number of times an enemy ship can ram your ship before the game is over.
<code>POINTS</code>	The number of times you've hit an enemy ship. This is your score.

Here are `PLAY.BLASTER` and its subprocedures.

```
TO PLAY.BLASTER :LIVES :POINTS
SETUP.DEMONS
TELL :ENEMIES
```

```
EACH [SETPOS RANDOM.POS]
SHOW.SCORE
ST
BLASTER.LOOP
END
```

SETUP.DEMONS starts the demons for the joystick button (firing), turtle-pen collisions (you shooting an enemy), turtle-turtle collisions (an enemy ramming you), and the joystick (steering).

```
TO SETUP.DEMONS
WHEN 3 [ASK 0 [FIRE]]
WHEN OVER 1 0 [ASK 1 [EXPLODE]]
WHEN OVER 2 0 [ASK 2 [EXPLODE]]
WHEN OVER 3 0 [ASK 3 [EXPLODE]]
WHEN TOUCHING 0 1 [ASK 0 [DIE 1]]
WHEN TOUCHING 0 2 [ASK 0 [DIE 2]]
WHEN TOUCHING 0 3 [ASK 0 [DIE 3]]
WHEN 15 [ASK 0 [STEER JOY 0]]
END
```

When nothing special is happening, the program spends most of its time in BLASTER.LOOP. It checks to see if you've run out of lives, in which case the game ends. Otherwise, it steers the enemy ships, shows the score, and continues. The ships are steered within 30 degrees of the direction toward you, so they tend to get closer to you but don't always move straight to you. (Their heading is chosen using the TOWARDS procedure, which is in the Towards and Arctan project.)

```
TO BLASTER.LOOP
IF :LIVES<1 [CS TS STOP]
EACH [SETH (TOWARDS [0 0])+(RANDOM 60)-30]
      SETSP 30+RANDOM 150 WAIT RANDOM 30]
SHOW.SCORE
BLASTER.LOOP
END
```

When you move the joystick, a demon invokes the STEER procedure with turtle 0 active. The demon wakes up whenever the joystick is moved, including when it is returned to the center position. In that case, the input to STEER is -1 and nothing is done. Otherwise, we have to change the turtle's heading (so it can fire properly) and its shape.

```
TO STEER :WHERE
IF :WHERE<0 [STOP]
SETH 45*:WHERE
SETSH 1+:WHERE
END
```

When you push the joystick button, a demon invokes the FIRE procedure with turtle 0 active. This procedure draws and then erases a line representing your shot. It hides the turtle while drawing so that the ship doesn't appear to move.

```
TO FIRE
HT
FD 80
BK 80
ST
END
```

When your shot hits an enemy ship, a collision demon invokes the EXPLODE procedure, using ASK to make the turtle representing that ship become the current turtle. This procedure changes this turtle to an explosion shape, blinks it on and off, makes a noise, and then repositions the enemy ship somewhere else at random on the screen. It also adds one to your score.

```
TO EXPLODE
SETSH 10
TOOT 0 14 15 60
TOOT 1 16 15 60
REPEAT 5 [HT WAIT 2 ST WAIT 2]
HT
SETPOS RANDOM.POS
SETSH 9
ST
MAKE "POINTS :POINTS+1
END
```

When an enemy ship hits your ship, a collision demon invokes the DIE procedure with turtle 0 active. The turtle number of the ship that hit you is an input to the procedure. That ship is moved to a random position, your ship explodes, the game stops for a second, and the count of how many lives remain is reduced by one. When your ship reappears, its shape is chosen to match its heading.

```
TO DIE :KILLER
ASK :KILLER [SETPOS RANDOM.POS]
SETSH 10
TOOT 0 25 15 60
TOOT 1 27 15 60
REPEAT 5 [HT WAIT 2 ST WAIT 2]
HT
WAIT 60
SETSH 1+INT (HEADING/45)
ST
MAKE "LIVES :LIVES-1
END
```

Every so often, the BLASTER.LOOP procedure calls SHOW.SCORE to update the display of how many lives remain and how many points you've earned. This isn't done instantly when you get a point or lose a life, because it would slow down the play of the game. Because of this, you might sometimes get an extra (bonus) life if BLASTER.LOOP doesn't notice your death soon enough. That's why the procedure checks for a negative number of lives remaining and displays it as zero.

```

TO SHOW.SCORE
IF :LIVES<0 [MAKE "LIVES 0]
SETCURSOR [3 22]
TYPE SE "LIVES: :LIVES
SETCURSOR [20 22]
PRINT SE "POINTS: :POINTS
END

```

RANDOM.POS is the utility procedure that outputs a random position on the screen for use with SETPOS.

```

TO RANDOM.POS
OUTPUT LIST RANDOM 320 RANDOM 240
END

```

Shapes

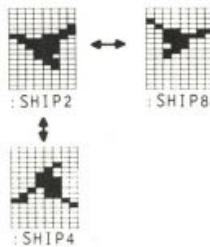
Here are the shapes.

```

MAKE "SHIP1 [24 24 24 24 24 60 60 60
           60 60 60 126 126 126 255 255]
MAKE "SHIP2 [0 0 0 0 1 3 206 124 60 28 24 12 4 0 0 0]
MAKE "SHIP3 [0 0 0 0 128 224 248 255
           255 248 224 128 0 0 0 0]
MAKE "SHIP4 [0 0 0 4 12 24 28 60 124 206 3 1 0 0 0 0]
MAKE "SHIP5 [255 255 126 126 126 60
           60 60 60 60 24 24 24 24]
MAKE "SHIP6 [0 0 0 32 48 24 56 60 62 115 192 128 0 0 0 0]
MAKE "SHIP7 [0 0 0 0 1 7 31 255 255 31 7 1 0 0 0 0]
MAKE "SHIP8 [0 0 0 0 128 192 115 62 60 56 24 48 32 0 0 0]
MAKE "BADGUY [129 129 129 129 129 129 153 255
           255 153 129 129 129 129 129 129]
MAKE "BLOWUP [128 192 96 32 1 3 230 228
           0 0 128 199 103 32 0 0]

```

Notice that SHIP4 to SHIP8 are simply mirror images of the first three ship shapes. For example, SHIP4 is the same as SHIP2 but reflected around a horizontal axis. SHIP8 is the same shape, but reflected around a vertical axis. These relationships can be seen in the lists of numbers representing the shape. For example, the list representing SHIP4 is the same as the list for SHIP2, but with the numbers in reverse order. (The relationship between a shape and its vertical-axis reflection is more complicated.) I actually only made the first three ship shapes in the shape editor; I created the others by calculating the appropriate numbers.



SUGGESTIONS

- Make the enemy ships fire back instead of just ramming you.
- The game hasn't been "playtuned." Should it be easier or harder? For example, the enemy ships move within 30 degrees of your direction. If that number were smaller, they'd hit you more often. The range of speeds could be changed too.
- You could start with a limited number of shots available. On the other hand, there could be a limit to the number of enemy ships that appear. (As it is, there is no way to "win" the game by destroying all the enemies.)
- You could add nice touches like stars in the background (remember to use a different pen for the stars and for the shots!) and sound effects between hits.
- It would be good to be able to move your own ship as well as steer it. This would require some way to indicate "thrust"; you could add a second joystick, or use the keyboard.

PROGRAM LISTING

The procedures from the Towards and Arctan project (p. 212) are also used in this program.

```

TO BLASTER
SETUP.BLASTER
PLAY.BLASTER 5 0
END

TO SETUP.BLASTER
CT
SETBG 0
SETPC 0 40
PUTSHAPES 1 [SHIP1 SHIP2 SHIP3 SHIP4 ▶
SHIP5 SHIP6 SHIP7 SHIP8 BADGUY ▶
BLOWUP]
MAKE "ENEMIES [1 2 3]
SETUP.ENEMIES
SETUP.PLAYER
END

TO PUTSHAPES :NUMBER :LIST
IF EMPTYP :LIST [STOP]
PUTSH :NUMBER THING FIRST :LIST
PUTSHAPES :NUMBER+1 BF :LIST
END

TO SETUP.ENEMIES
TELL :ENEMIES
HT
SETSH 9
SETC 74
PU
END

TO SETUP.PLAYER
TELL 0
ST
SETSH 1
SETC 7
CS
PX
END

TO PLAY.BLASTER :LIVES :POINTS
SETUP.DEMONS
TELL :ENEMIES
EACH [SETPOS RANDOM.POS]
SHOW.SCORE
ST
BLASTER LOOP
END

TO SETUP.DEMONS
WHEN 3 [ASK 0 [FIRE]]
WHEN OVER 1 0 [ASK 1 [EXPLODE]]
WHEN OVER 2 0 [ASK 2 [EXPLODE]]
WHEN OVER 3 0 [ASK 3 [EXPLODE]]
WHEN TOUCHING 0 1 [ASK 0 [DIE 1]]
WHEN TOUCHING 0 2 [ASK 0 [DIE 2]]
WHEN TOUCHING 0 3 [ASK 0 [DIE 3]]
WHEN 15 [ASK 0 [STEER JOY 0]]
END

```

```

TO BLASTER.LOOP
IF :LIVES<1 [CS TS STOP]
EACH [SETH (TOWARDS [0 0])+(RANDOM ▶
  60)-30 SETSP 30+RANDOM 150 WAIT ▶
  RANDOM 30]
SHOW.SCORE
BLASTER.LOOP
END

TO STEER :WHERE
IF :WHERE<0 [STOP]
SETH 45*:WHERE
SETSH 1+:WHERE
END

TO FIRE
HT
FD 80
BK 80
ST
END

TO EXPLODE
SETSH 10
TOOT 0 14 15 60
TOOT 1 16 15 60
REPEAT 5 [HT WAIT 2 ST WAIT 2]
HT
SETPOS RANDOM.POS
SETSH 9
ST
MAKE "POINTS :POINTS+1
END

TO DIE :KILLER
ASK :KILLER [SETPOS RANDOM.POS]
SETSH 10
TOOT 0 25 15 60
TOOT 1 27 15 60
REPEAT 5 [HT WAIT 2 ST WAIT 2]
HT
WAIT 60
SETSH 1+INT (HEADING/45)
ST
MAKE "LIVES :LIVES-1
END

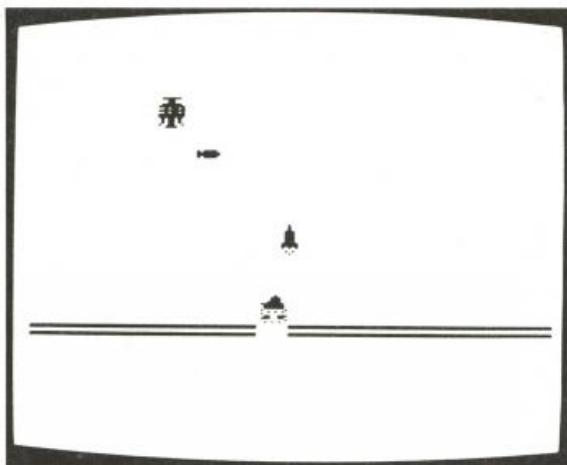
TO SHOW.SCORE
IF :LIVES<0 [MAKE "LIVES 0]
SETCURSOR [3 22]
TYPE SE "LIVES: :LIVES
SETCURSOR [20 22]
PRINT SE "POINTS: :POINTS
END

TO RANDOM.POS
OUTPUT LIST RANDOM 320 RANDOM 240
END

MAKE "SHIP1 [24 24 24 24 24 60 60 60 ▶
  60 60 60 126 126 126 255 255]
MAKE "SHIP2 [0 0 0 0 1 3 206 124 60 28 ▶
  24 12 4 0 0 0]
MAKE "SHIP3 [0 0 0 0 128 224 248 255 ▶
  255 248 224 128 0 0 0 0]
MAKE "SHIP4 [0 0 0 4 12 24 28 60 124 ▶
  206 3 1 0 0 0 0]
MAKE "SHIP5 [255 255 126 126 126 60 60 ▶
  60 60 60 24 24 24 24 24]
MAKE "SHIP6 [0 0 0 32 48 24 56 60 62 ▶
  115 192 128 0 0 0 0]
MAKE "SHIP7 [0 0 0 0 1 7 31 255 255 31 ▶
  7 1 0 0 0 0]
MAKE "SHIP8 [0 0 0 0 128 192 115 62 60 ▶
  56 24 48 32 0 0 0]
MAKE "BADGUY [129 129 129 129 129 129 ▶
  153 255 255 153 129 129 129 129 ▶
  129 129]
MAKE "BLOWUP [128 192 96 32 1 3 230 ▶
  228 0 0 128 199 103 32 0 0]


```

Alien



Here is a description of this program by its author, Jeanry Chandler.

Alien is basically a Space Invaders-type game. All you need to play is a deft hand, and possibly a severe case of xenophobia. You control a sturdy defender tank with a joystick; you launch your deadly cruise missiles by (you guessed it) pressing the joystick button.

The alien craft, intent upon landing, will slip ever downward while avoiding your missiles and dropping its own neutro-destroyer bombs. If the alien lands, you are in serious trouble indeed. Two little green creatures will emerge and try to plant a bomb on your tank. You can attempt to shoot the little pests, but your gun has jammed and you can only shoot in one direction, so you have to shoot one quickly and then use the magic of Logoland to wrap and face the other. This, of course, is nearly impossible.

In this write-up I talk about the overall structure of the program and the decisions made about how it keeps track of things; I do not cover all the procedures in detail.

This game is in two parts. If you manage to shoot the alien helicopter before it lands, you don't play the second part. I have organized the program so that the two parts have the same structure.

Before proceeding further, you may want to play Alien. To begin the game, run START. You are the defender, controlling your maneuverable tank with the joystick plugged into port 1. You can switch the direction you are moving with the joystick and fire missiles at the alien helicopter with the joystick button.

Structure of the First Part of the Game

In their roles as alien, missile, bomb, and defender, the turtles are given their positions, headings, states, shapes, and colors. Demons are created to watch for certain game conditions and then call procedures that keep score, create explosions, and set variables. Those variables are:

SCORE	The score.
WIN	TRUE if something has happened that means the player has won the game.
LOSE	TRUE if something has happened that means the player has lost the game and thus has been annihilated.
MISSILE	Used to tell if a missile can be launched. Its value is 1 if the defender's missile is in the air, 0 if not. The rule is that the defender cannot launch a missile while the previous one is still in the air. When the missile gets to the top of the screen, it disappears and :MISSILE is reset to 0.
BOMB	Used to tell when to drop a bomb. After the alien drops a bomb, :BOMB becomes 1. This is used to prevent another from being dropped. When the bomb hits a target or the ground, :BOMB is reset to 0 to reenable launching.

It is a good idea to know the roles played by the four turtles.

- 0 Defender (the player)
- 1 Alien
- 2 Bomb (alien's neuro-destroyer bomb)
- 3 Missile (defender's missile)

Naming Conventions

Most procedures that deal with the alien have **A** in them; those that deal with the defender have **D**. The procedures that deal with the missile generally have **MISSILE** in their names, and the ones that deal with the alien's bomb have **BOMB** in theirs. The variables that have to do with the weapons are **:MISSILE** and **:BOMB**.

Setting Up

At the start of the game, **SETUP** initializes the turtle shapes and the game variables. Then it calls **SETUP_DEMONS** to create all the demons that are used in the game. **SETUP** then calls **ASETUP** and **DSETUP** to initialize the alien's and defender's positions, headings, colors, and speeds.

GAMES

```

TO SETUP
CT CS
SETUPSHAPES
TELL [0 1 2 3] HOME HT PU
TELL [1 0] ST PU
MAKE "BOMB 0
MAKE "MISSILE 0
MAKE "SCORE 0
MAKE "WIN "FALSE
MAKE "LOSE "FALSE
SETUP.DEMONS
DSETUP
ASETUP
TELL 0
END

TO ASETUP
TELL 1
SETSH 1 SETSP 70 SETH 90
SETPOS [0 100]
SETSH 3 SETSP 65 SETH 270
ASK 2 [SETC 44]
END

TO DSETUP
TELL 0 SETPOS [0 -55]
SETPN 0
RT 90 PD FD 300 PU RT 90 FD 5 RT 90 PD FD 300 PU
SETPOS [0 -43]
SETSP 75
END

```

Setting Up Demons and Demon Instruction Conventions

One of the demons that SETUP.DEMONS creates carries out its instructions every time the joystick position changes. (It is created by the line WHEN 15 [DMOVE]). The demon instructions call the procedure DMOVE to let the joystick control the defender's motion.

```

TO SETUP.DEMONS
WHEN 15 [DMOVE]
MISSILE.DEMONS
BOMB.DEMONS
END

TO DMOVE
IF MEMBERP JOY 0 [1 2 3] [ASK 0 [SETH 90]]
IF MEMBERP JOY 0 [5 6 7] [ASK 0 [SETH 270]]
END

```

The MISSILE.DEMONS and BOMB.DEMONS procedures create demons for actions having to do with the missile and bomb. For example:

```
WHEN TOUCHING 2 3 [SCORE 20 EXPLODE.BOMB.MISSILE]
```

is a line in MISSILE.DEMONS that creates a demon. This demon waits for a collision between the defender's missile and the alien's bomb. This demon's instructions make the missile explode, thus neutralizing the bomb and protecting the defender from it. SCORE 20 gives the player points for having the good aim to hit the bomb. EXPLODE.BOMB.MISSILE makes the explosion graphics and sounds and sets :MISSILE and :BOMB to zero. This lets the game know that the missile and bomb have been destroyed.

MISSILE.DEMONS also creates a demon that waits for the joystick button to be pressed. When the button is pressed, a missile is fired.

```
TO MISSILE.DEMONS
WHEN 3 [IF :MISSILE < 1 [MISSILE]]
WHEN TOUCHING 2 3 [SCORE 20 EXPLODE.BOMB.MISSILE]
WHEN TOUCHING 3 1
    [EXPLODE.MISSILE.A SCORE 50 MAKE "WIN "TRUE]
WHEN OVER 3 0 [MAKE "MISSILE 0 ASK 3 [HT]]
END

TO EXPLODE.BOMB.MISSILE
ASK 3 [HT]
ASK 2 [SETSP 10]
ASK 2 [SETSH 6]
REPEAT 20 [TOOT 0 14 10 2]
ASK 2 [HT]
MAKE "BOMB 0
MAKE "MISSILE 0
END
```

The other explosion procedures are named for the things that cause each explosion. For example, the procedure that is invoked when the missile hits the alien is named EXPLODE.MISSILE.A. Remember that when this happens, the player wins the game. The same demon instructions that call EXPLODE.MISSILE.A also include MAKE "WIN "TRUE.

In this program, the demon instructions conventionally include a call to SCORE and a call to the appropriate explosion procedure. They also set :WIN or :LOSE if the game has been won or lost. For example, if the bomb hits the defender, the player has lost the game. The instructions run by the demon created for the collision between the bomb and the defender include the appropriate explosion procedure and also MAKE "LOSE "TRUE. This means you can systematically review all the conditions for winning and losing the game by reading through the demon setup procedures.

The Game Actions

The main loop is INVADE. It stops only if the game has been won or lost. Either this happens in the first part of the game, or the alien helicopter survives to land and INVADE calls LAND. If INVADE calls LAND, then INVADE stops when the second part of the game is finished.

GAMES

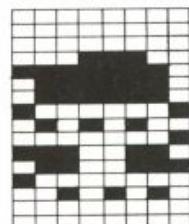
```

TO INVADE
IF :WIN [WIN STOP]
IF :LOSE [LOSE STOP]
TELL 0
SETSH :MISSILE + 8
BLADE
TELL 1
IF ( RANDOM 10 ) < 8 [SETY YCOR - 12]
IF YCOR < -45 [LAND STOP]
IF EQUALP RANDOM 3 1 [AMOVE]
IF EQUALP :BOMB 0 [BOMB]
BLADE
INVADE
END

```



Shape 8
`:MISSILE = 0`
`:MISSILE + 8 = 8`



Shape 9
`:MISSILE = 1`
`:MISSILE + 8 = 9`

INVADE calls most of the procedures that perform actions in the game. It does not call DMOVE or the explosion procedures; they are called by demons.

INVADE first checks :WIN and :LOSE to see if the end of the game has been signaled. If so, INVADE calls an appropriate procedure and stops. If not, INVADE updates the shape of the defender according to what is happening with the missile. It uses a trick with the shape numbers.

The trick is that INVADE uses :MISSILE to choose which shape to give the defender. :MISSILE gets changed by demon instructions. It is set to 1 when the missile is launched. This happens when the joystick button is pushed.* (See MISSILE_DEMONS.) When the missile hits something, or when it gets to the top of the screen, :MISSILE is set to 0 and the missile disappears. Demon instructions take care of this. While the missile is flying on the screen and :MISSILE is 1, INVADE gives the defender shape 9. When the missile is gone and :MISSILE is 0, INVADE gives the defender the ready-to-launch shape 8.

INVADE then calls BLADE, which makes the alien helicopter's rotor seem to turn by changing its shape. It also makes some sound effects to accompany the animation.

```

TO BLADE
TELL 1
SETSH 3
TOOT 0 80 12 1
WAIT 5
SETSH 10
TOOT 0 80 12 1
WAIT 5
SETSH 11
TOOT 0 80 12 1
WAIT 5
TELL 0
END

```

INVADE makes the alien drop closer to the ground 80 percent of the time. It uses `(RANDOM 10) < 8` to decide whether to drop the alien.

*The MISSILE procedure fires the missile up and in the same general direction as the defender is traveling. It uses the ADJUST procedure to decide on the heading of the missile.

INVADE checks whether the alien has reached ground level. If it has, INVADE calls LAND, the second part of the game. Otherwise, INVADE may call AMOVE to make the alien change direction toward the defender. It uses (RANDOM 3) = 1 to do this one third of the time.

```
TO AMOVE
IF XCOR > ( ASK 0 [XCOR] ) [SETH 90] [SETH 270]
END
```

Next INVADE checks if the bomb is on the screen. If it is not, INVADE calls BOMB to launch one. BOMB aims the bomb at the defender using TOWARDS.*

```
TO BOMB
MAKE "BOMB 1
TELL 2
SETPOS ASK 1 [POS]
PU SETSH 5 POINT.AT.D SETSP 80 ST
END
```

Last, INVADE calls BLADE again to create more animation of the alien, and then calls itself to continue the game process.

Winning and Losing

The WIN and LOSE procedures print the score and either a congratulatory or a gloomy message.

The Second Part of the Game

If the alien craft survives your attacks and reaches ground level, LAND is called. LAND controls all the action that happens at ground level. Since this part of the game has a similar structure to the first part, the programs look similar. Some of the same shapes are used. There is still a defender (with the same shape), two aliens (with animated walking shapes), and a bullet for the defender to shoot.

The turtles' new assignments are

- 0 Defender (the player)
- 1 Green alien walker
- 2 Green alien walker
- 3 Bullet (the player's)

Since there are two alien walkers, and you have to shoot *both* of them to win the game, there is a new game variable AWCOUNT. :AWCOUNT is the number of aliens still alive. This lets the game know when you have shot an alien, and whether it is the last one.

LAND corresponds to START. It calls SETUP.LAND, which cancels all the demons created in the first part of the game and sets up the new shapes and turtle states. Then it calls SETUP.LAND.DEMONS, which creates the demons used in this part of the game. For example, the line WHEN 15 [DMOVE] in SETUP.LAND.DEMONS creates a demon that lets the joystick control the defender's motion.

*TOWARDS is described in Brian Harvey's project, Towards and Arctan.

```

TO LAND
SETUP.LAND
SETUP.LAND.DEMONS
WALK
END

TO SETUP.LAND
CANCEL.DEMONS
TELL [1 2]
SETC 20
SETSH 13
SETY -43
SETSP 70
ST
TELL 1 SETH 270
TELL 2 SETH 90
MAKE "AWCOUNT 2
END

TO CANCEL.DEMONS
ASK 2 [HT CS]
END

TO SETUP.LAND.DEMONS
WHEN 15 [DMOVE]
WHEN TOUCHING 1 0 [EXPLODE.AW.D MAKE "LOSE "TRUE]
WHEN TOUCHING 2 0 [EXPLODE.AW.D MAKE "LOSE "TRUE]
WHEN TOUCHING 1 3 [EXPLODE.BULLET.AW 1]
WHEN TOUCHING 2 3 [EXPLODE.BULLET.AW 2]
END

```

There is a new set of explosion procedures for this part of the game. It is possible that some of the explosion procedures from the invaders part of the game could have been reused; they might have had appropriate actions for the turtles' new roles. I decided that it would be clearer to have a new set of procedures with names that go with the new roles: AW for green alien walker and BULLET for the bullet.

The WALK procedure corresponds to INVADE in the first part of the game. WALK controls most of the game actions. The sequence of SETSH 13, TOOT, and SETSH 12 creates the walking animation for turtles 1 and 2, the alien walkers. This sequence corresponds to BLADE in the first part of the game, which creates the helicopter blade animation. FIRE fires the bullet if the joystick button is pressed. It corresponds to MISSILE in the first part of the game. (I don't know why I put it in WALK instead of creating a demon to do it with WHEN 3 [FIRE].) Winning and losing the game happen exactly as in INVADE. WALK is recursive to keep the game process going.

```

TO WALK
IF :WIN [WIN STOP]
IF :LOSE [LOSE STOP]
IF JOYB 0 [FIRE]
TELL [1 2]
SETSH 13

```

```
TOOT 0 300 8 4
SETSH 12
WALK
END
```

Note that there is nothing corresponding to the MISSILE or BOMB variables in this part of the game. The BULLET keeps going once it is fired. One bullet has to hit both alien walkers for the game to be won.

SUGGESTIONS

Jeanry suggests that you change the shapes that the turtles carry to make a different game. You might make a flying saucer blinking its landing lights instead of a helicopter spinning its blades.

A very different kind of game could be created with this type of programming. One idea is to replace the defender shooting destructive missiles with a ground launching platform trying to send recharge fuel cylinders to a disabled spaceship. The fuel could enable the spaceship to turn on its brakes and land safely instead of crashing. You could even make the joystick control the refueled spaceship, so that the new challenge is to land the ship.

You can use some of Alien's techniques—sound effects, controlling turtles with the joystick, simple game play—in projects completely of your own imagination.

PROGRAM LISTING

THE FIRST PART OF THE GAME

TO START	TO INVADE
SETUP	IF :WIN [WIN STOP]
INVADE	IF :LOSE [LOSE STOP]
END	TELL 0
TO SETUP	SETSH :MISSILE + 8
CT CS	BLADE
SETUPSHAPES	TELL 1
TELL [0 1 2 3] HOME HT PU	IF (RANDOM 10) < 8 [SETY YCOR - 12]
TELL [1 0] ST PU	IF YCOR < -45 [LAND STOP]
MAKE "BOMB 0	IF EQUALP RANDOM 3 1 [AMOVE]
MAKE "MISSILE 0	IF EQUALP :BOMB 0 [BOMB]
MAKE "SCORE 0	BLADE
MAKE "WIN "FALSE	INVADE
MAKE "LOSE "FALSE	END
SETUP_DEMONS	
DSETUP	
ASETUP	
TELL 0	
END	

DEMONS

```

TO SETUP.DEMONS
WHEN 15 [DMOVE]
MISSILE.DEMONS
BOMB.DEMONS
END

TO BOMB.DEMONS
WHEN TOUCHING 2 0 [EXPLODE.BOMB.D MAKE ▶
  "LOSE "TRUE]
WHEN OVER 2 0 [EXPLODE.BOMB]
END

TO MISSILE.DEMONS
WHEN 3 [IF :MISSILE < 1 [MISSILE]]
WHEN TOUCHING 3 1 [EXPLODE.MISSILE.A ▶
  SCORE 50 MAKE "WIN "TRUE]
WHEN OVER 3 0 [MAKE "MISSILE 0 ASK 3 ▶
  [HT]]
END

```

SETUP FOR THE ALIEN AND DEFENDER

```

TO ASETUP
TELL 1
SETH 1 SETSP 70 SETH 90
SETPOS [0 100]
SETH 3 SETSP 65 SETH 270
ASK 2 [SETC 44]
END

TO DSETUP
TELL 0 SETPOS [0 -55]
SETPN 0
RT 90 PD FD 300 PU RT 90 FD 5 RT 90 PD ▶
  FD 300 PU
SETPOS [0 -43]
SETSP 75
END

```

GAME ACTIONS

```

TO BLADE
TELL 1
SETH 3
TOOT 0 80 12 1
WAIT 5
SETH 10
TOOT 0 80 12 1
WAIT 5
SETH 11

```

```

TOOT 0 80 12 1
WAIT 5
TELL 0
END

TO DMOVE
IF MEMBERP JOY 0 [1 2 3] [ASK 0 [SETH ▶
  90]]
IF MEMBERP JOY 0 [5 6 7] [ASK 0 [SETH ▶
  270]]
END

TO AMOVE
IF XCOR > ( ASK 0 [XCOR] ) [SETH 90] ▶
  [SETH 270]
END

TO BOMB
MAKE "BOMB 1
TELL 2
SETPOS ASK 1 [POS]
PU SETSH 5 POINT.AT.D SETSP 80 ST
END

TO POINT.AT.D
SETH TOWARDS ASK 0 [POS]
END

TO MISSILE
MAKE "MISSILE 1
ASK 0 [SETH 2]
ASK 3 [SETH 4 SETX ( ASK 0 [XCOR] ) ▶
  SETY -35 PU SETH ( ADJUST ( ASK 0 ▶
  [HEADING] ) ) SETSP 95 ST]
END

TO ADJUST :HEADING
IF :HEADING = 90 [OP 75] [OP 285]
END

TO LOSE
PR [YOU HAVE BEEN ANNIHILATED !!!]
TELL [0 1 2 3] CS
END

```

```

TO WIN
PR [YOU WIN!!!!]
TELL [0 1 2 3] CS
END

```

EXPLOSIONS, SCORING, AND WINNING/LOSING

```

TO EXPLODE.BOMB
ASK 2 [SETH 6]
REPEAT 2 [TOOT 0 20 15 5]
MAKE "BOMB 0
END

TO EXPLODE.BOMB.MISSILE
ASK 3 [HT]
ASK 2 [SETSP 10]
ASK 2 [SETH 6]
REPEAT 20 [TOOT 0 14 10 2]
ASK 2 [HT]
MAKE "BOMB 0
MAKE "MISSILE 0
END

TO EXPLODE.MISSILE.A
TELL 3 HT
TELL 1 SETSH 7 SETSP 5
REPEAT 20 [TOOT 0 420 15 10]
TELL 1 HT
END

TO EXPLODE.BOMB.D
TELL 2 HT
TELL 0 SETSH 7 SETSP 5
REPEAT 40 [TOOT 0 30 15 5]
TELL 0 HT
END

```

```

TO SETUP.LAND
CANCEL.DEMONS
TELL [1 2]
SETC 20
SETH 13
SETY -43
SETSP 70
ST
TELL 1 SETH 270
TELL 2 SETH 90
MAKE "AWCOUNT 2
END

```

DEMONS

```

TO CANCEL.DEMONS
ASK 2 [HT CS]
END

TO SETUP.LAND.DEMONS
WHEN 15 [DMOVE]
WHEN TOUCHING 1 0 [EXPLODE.AW.D MAKE ▶
    "LOSE "TRUE]
WHEN TOUCHING 2 0 [EXPLODE.AW.D MAKE ▶
    "LOSE "TRUE]
WHEN TOUCHING 1 3 [EXPLODE.BULLET.AW ▶
    1]
WHEN TOUCHING 2 3 [EXPLODE.BULLET.AW2]
END

```

SCORING PROCEDURES USED IN BOTH PARTS OF THE GAME

```

TO SCORE :S
MAKE "SCORE :SCORE + :S
PRINT SCORE
END

TO PRINT SCORE
PR SE [GOOD SHOT YOUR SCORE IS] :SCORE
END

```

GAME ACTIONS

```

TO WALK
IF :WIN [WIN STOP]
IF :LOSE [LOSE STOP]
IF JOYB 0 [FIRE]
TELL [1 2]
SETSH 13
TOOT 0 300 8 4
SETSH 12
WALK
END

```

THE SECOND PART OF THE GAME

```

TO LAND
SETUP.LAND
SETUP.LAND.DEMONS
WALK
END

```

```

TO FIRE
TELL 3
SETSH 14
SETH 270
ST SETSP 150 SETX ASK 0 [XCOR] SETY ▶
    -44
END

```

EXPLOSIONS, SCORING, AND WINNING/LOSING

```

TO EXPLODE.AW.D
TELL 0 SETSH 7 SETSP 5
REPEAT 40 [TOOT 0 30 15 5]
HT
END

TO EXPLODE.BULLET.AW :ALIEN
MAKE "AWCOUNT :AWCOUNT - 1
EXPLODE.B.AW :ALIEN
SCORE 30
IF :AWCOUNT = 0 [MAKE "WIN "TRUE]
END

TO EXPLODE.B.AW :A
ASK 3 [HT]
ASK :A [SETSH 7 SETSP 5]
REPEAT 10 [TOOT 0 420 15 10]
ASK :A [HT]
END

SETUPSHAPES, SAVE.SHAPES, AND SHAPES
TO SETUPSHAPES
PUTSH 1 :DEFENDER
PUTSH 2 :DEFENDER1
PUTSH 3 :ALIEN
PUTSH 4 :MISSILESHAPE
PUTSH 5 :BOMBSHAPE
PUTSH 6 :EXPLOSION1
PUTSH 7 :EXPLOSION2
PUTSH 8 :DEFENDER2
PUTSH 9 :DEFENDER3
PUTSH 10 :ALIEN2
PUTSH 11 :ALIEN3
PUTSH 12 :MAN1
PUTSH 13 :MAN2
PUTSH 14 :BULLET
END

TO SAVE.SHAPES
MAKE "DEFENDER GETSH 1
MAKE "DEFENDER1 GETSH 2

```

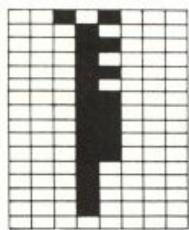
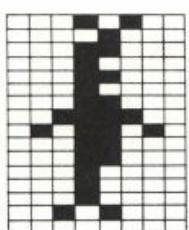
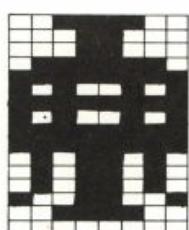
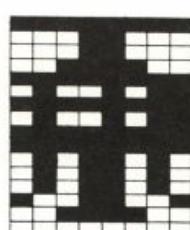
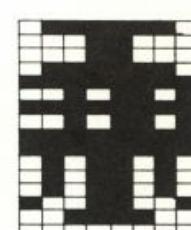
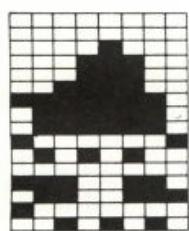
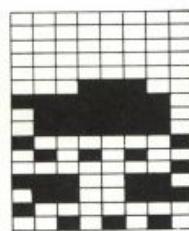
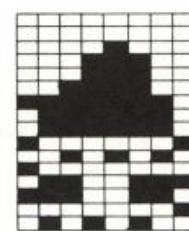
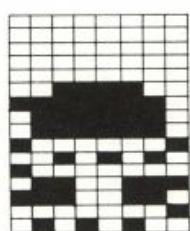
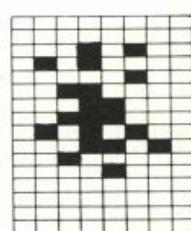
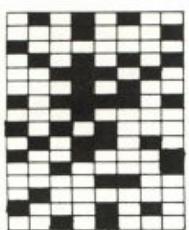
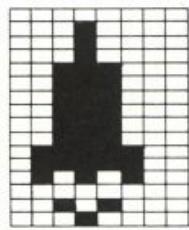
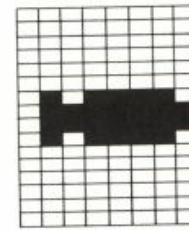
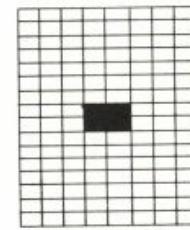
```

MAKE "ALIEN GETSH 3
MAKE "MISSILESHAPE GETSH 4
MAKE "BOMBSHAPE GETSH 5
MAKE "EXPLOSION1 GETSH 6
MAKE "EXPLOSION2 GETSH 7
MAKE "DEFENDER2 GETSH 8
MAKE "DEFENDER3 GETSH 9
MAKE "ALIEN2 GETSH 10
MAKE "ALIEN3 GETSH 11
MAKE "MAN1 GETSH 12
MAKE "MAN2 GETSH 13
MAKE "BULLET GETSH 14
END

MAKE "BULLET [0 0 0 0 0 0 0 24 24 0 0 ▶
0 0 0 0 0]
MAKE "MAN2 [20 8 24 16 24 16 24 60 90 ▶
24 24 16 16 16 40 0]
MAKE "MAN1 [40 16 24 16 24 16 24 24 24 ▶
24 24 16 16 16 0]
MAKE "ALIEN3 [126 24 24 126 255 45 255 ▶
45 255 255 90 90 90 153 60 0]
MAKE "ALIEN2 [255 24 24 60 255 75 255 ▶
75 255 255 90 90 90 153 60 0]
MAKE "DEFENDER3 [0 0 0 0 60 254 126 ▶
126 129 42 128 103 230 1 84]
MAKE "DEFENDER2 [0 0 8 28 28 60 254 ▶
126 126 129 42 128 103 230 1 84]
MAKE "EXPLOSION2 [84 17 128 85 58 20 ▶
189 16 168 90 145 33 12 64 138 ▶
40]
MAKE "EXPLOSION1 [0 0 20 80 4 48 26 56 ▶
84 26 32 8 0 0 0 0]
MAKE "BOMBSHAPE [0 0 0 0 0 0 94 127 ▶
127 94 0 0 0 0 0 0]
MAKE "MISSILESHAPE [0 16 16 16 56 56 ▶
56 56 56 56 124 124 84 0 40 16]
MAKE "ALIEN [60 24 24 126 255 165 255 ▶
165 255 255 90 90 90 153 60 0]
MAKE "DEFENDER1 [0 0 0 0 28 254 126 ▶
126 129 84 1 230 103 128 42]
MAKE "DEFENDER [0 0 8 28 28 60 254 126 ▶
126 129 84 1 230 103 128 42]

```

SHAPES

:MAN1
slot 12:MAN2
slot 13:ALIEN
slot 3:ALIEN2
slot 10:ALIEN3
slot 11:DEFENDER
slot 1:DEFENDER1
slot 2:DEFENDER2
slot 8:DEFENDER3
slot 9:EXPLOSION1
slot 6:EXPLOSION2
slot 7:MISSILESHAPE
slot 4:BOMBSHAPE
slot 5:BULLET
slot 14

Adventure

Adventure is one of a class of hundreds, if not thousands, of games inspired by Crowther and Woods's classic FORTRAN program. You play an adventure game by exploring a simulated world, and usually you win points for finding objects, solving riddles, or killing monsters. This version, however, awards no points.

There are three aspects to an adventure program:

- Language understanding. The program must recognize commands that you give in a simple language.
- Simulation. The program executes your commands.
- Language production. The program tells you the results.

As you read on you'll find out how my program does all three of these things.

This adventure is smaller than most because of Logo's space limitations. Other microcomputer adventures are usually written in assembly language and also use a disk to store more information. On the other hand, this version was easy to write and is easy to modify and extend.

Adventure Programs Understand a Simple Language

When you play Adventure you give the computer commands in the Adventure language, just as when you use Logo you use the Logo language. Adventure is a program written in Logo that understands the Adventure language.*

Sentences in this language are in one of three forms:

- | | |
|------------------|---|
| verb | Just a verb. The verb is one of these: LOOK, INVENTORY, or ?. |
| verb noun | A verb followed by a noun, for example, TAKE KEY. Verbs are TAKE, DROP, EXAMINE, UNLOCK, or DRINK. Nouns are objects you find while playing the game. |
| direction | The implied verb is MOVE, and the direction must be one of NORTH, EAST, SOUTH, WEST, UP or DOWN. |

Most of the verbs have the same meaning as their English counterpart. (You can find out more by using them.)

The syntax rules of the language are simple and strict. Each verb is in one of two classes: either it must always be followed by a noun (TAKE KEY), or it must never be used with a noun. The program won't understand what you mean if you supply an object to a verb that doesn't expect one (for example, LOOK NORTH) or if you omit a necessary noun.

*Logo, the program that interprets the Logo language, is itself a program, written in the machine language of the microcomputer. For more on this subject, see the preface and Logo Interpreter project in this book.

Using the Program

To use the program, invoke the top-level procedure ADVENTURE.

The program describes the area you're in, then prompts you (with a <) for a sentence.

```
YOU ARE IN A FOREST  
YOU SEE ASH AND BIRCH TREES  
YOU CAN GO NORTH  
YOU CAN GO EAST  
YOU CAN GO SOUTH  
YOU CAN GO WEST  
>
```

You type in a sentence telling the program what to do. If it understands you it does what you typed; otherwise it complains. If your action takes you to a new place, the program describes the new place. After telling you the consequences of your action, the program is ready for another command.

```
>NORTH  
YOU ARE AT THE BASE OF A CLIFF  
YOU SEE ASH AND BIRCH TREES  
YOU CAN GO SOUTH  
YOU CAN GO WEST  
A CLOSED STOUT IRON DOOR LEADS NORTH  
>
```

If you move north, you'll find yourself at the foot of a cliff, with a door leading in. But to open the door you'll have to explore the forest further. Be careful not to get lost; in Adventure it isn't always true that if you travel east to get from one place to another that traveling west will get you back where you were.

You can get a list of every word the program knows by typing ?.

You should probably play the game before reading further, because it is easier to understand the program if you've used it and because the game is much more fun if you don't know what to expect.

An Adventure Program Simulates a World

The program uses Logo *words* to represent the places and objects in the simulated world. The word CROWBAR represents the crowbar you'll find in the guard room. The word GUARD represents that guard room. There is one word for every place or object in the simulated world.

Objects and rooms in the simulation have different *attributes* (for example, weight). Each word has a list of all attributes of the thing it represents. A list of attributes is called a *property list*.*

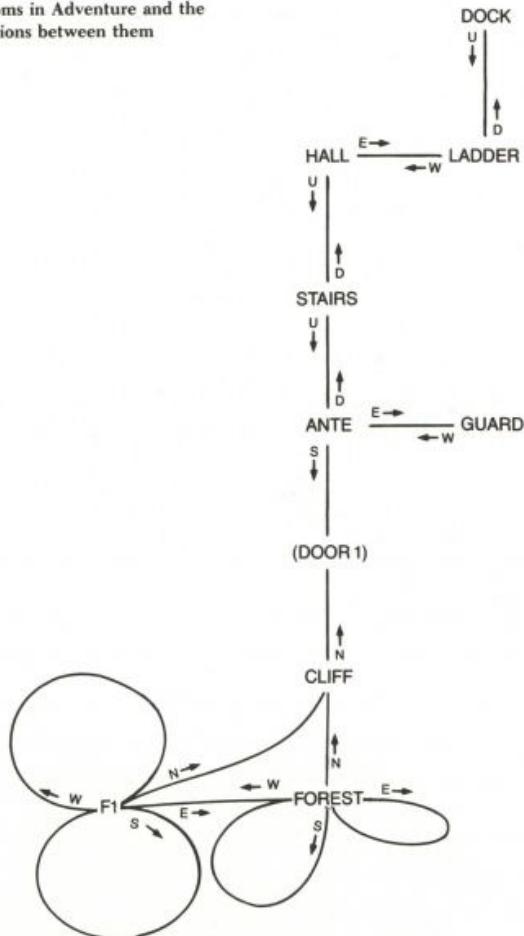
*This terminology comes from the programming language LISP. Some procedures in this project are tools for working with property lists: PPROP, GPROP, HASPROP, and PROPTURE?.

The Locales of the World Are Linked Rooms

Exploring in Adventure means moving from place to place and discovering objects in those places. The program thinks of all locales in the game (indoor rooms, the forest, stairways) as *rooms*. Each room is connected to at least one other room. You can move from one room to any connecting room—provided there's nothing preventing you from leaving, such as a shut door.

In Adventure there are six possible directions you can move: north, east, south, west, up, and down. Each room can have as many as six neighbors, one in each direction. This map shows how rooms are connected.

The rooms in Adventure and the connections between them



Each room is represented by a Logo word. Each word has an `EXITS` property that holds a list of six items, one for each possible exit direction from the room. Each item is the empty list if there is no exit in that direction; otherwise it is the word for the connecting room. Items in the

EXITS list appear in the order in which the direction options are presented: north, east, south, west, up, and down.

For example:

```
PPROP "STAIRS" "EXITS [ [] [] [] [] ] ANTE HALL"]
```

makes the room STAIRS have just two exits. (This makes sense, because we usually go up or down stairs, and there are exits at the top and bottom.) The up exit leads to ANTE, the down exit to HALL. (Actually, the exits lead to the rooms these words represent.)

I defined six variables to hold the positions of the exits in the exit list.

```
MAKE "NORTH 1  
MAKE "EAST 2  
MAKE "SOUTH 3  
MAKE "WEST 4  
MAKE "UP 5  
MAKE "DOWN 6
```

Then I defined ITEM, an operation that outputs the *n*th element of a list. This made it possible for me to extract the exit from a room for any direction. For example, I can get the up exit from STAIRS with the following instruction:

```
PR ITEM :UP GPROP "STAIRS" "EXITS  
ANTE
```

```
PR FIRST BF BF BF BF GPROP "STAIRS" "EXITS
```

The room exits are set up by INITROOMS. For a description, look at the listing at the end of this write-up.

Doors Were Hard to Add

In the outside world, nothing stops you from moving across open ground. But if you're in a building, there may be doors that stop you from getting into a room.* If a door is shut you must open it to get through, and if it is locked you must unlock it with a key before you can open it.

Doors are important in the real world, so I wanted to have doors in my program too. Doors were the most difficult part of the program to write. I tried a few different schemes before settling on one.

A door is a kind of exit from a room, but it isn't a destination in its own right. You may leave a room *through* a door, but you don't stay *in* the door. You go to the room on the other side.

Since doors are a type of exit, I decided that doors could go in the EXITS list just as rooms could. So I needed a predicate (DOORP) to distinguish doors from rooms, since both could be in the list. As you read on you'll discover other consequences of this decision.

*The word "indoors" is a reminder that one important thing about buildings is that they have doors, at least in most Western cultures.

You Leave a Room Through an Exit

When you move in a given direction, the program invokes the procedure MOVE. Its input is a number telling which way you want to move. (North is 1, east is 2, and so forth.)

```
TO MOVE :DIR
MOVE1 :DIR ( ITEM :DIR GPROP :#ROOM "EXITS" )
END

TO MOVE1 :DIR :THERE
IF EMPTYP :THERE [PR [YOU CAN'T GO THAT WAY!] STOP]
IF DOORP :THERE [TRYDOOR :THERE :DIR] [GOROOM :THERE]
END
```

The global variable #ROOM holds the word representing the current room. I used a global variable because I knew I'd refer to it in many places in the program, and it would have been a bother to pass it as an input to all the procedures that need it.*

MOVE outputs an empty list if there is no exit; otherwise it outputs the word for the room or door in that direction.

If the exit is a door, the program uses TRYDOOR; if the exit is a room, the program uses GOROOM to put you in the new room.

```
TO GOROOM :NEW
MAKE "#ROOM :NEW
LOOK
END
```

GOROOM sets the global variable #ROOM to the new room and describes the locale.

Doors Are Tricky

The program knows which rooms are on both sides of any door because doors have EXITS properties just as rooms do. In a way, the program uses doors as if they were small rooms that you move through automatically. If leaving a room to the east takes you to an (open) door, you will go through the door to whatever is to the east of the door.

A word that stands for a door has a DOOR property on its property list. The value of its DOOR property is TRUE. The DOORP predicate checks this property:

```
TO DOORP :OBJ
OP PROPTURE? :OBJ "DOOR"
END
```

A door also has a SHUT? property that is TRUE only if the door is shut;

*There are a few other global variables in the program. Almost all of them have names beginning with a sharp sign ("#") to distinguish them from procedure inputs. All global variables are set up by INITVARS at the start of the program.

a **LOCKED?** property that is **TRUE** only if the door is locked; and a **KEY** property that is the word representing the key that can unlock the door.

The procedure **TRYDOOR** tries to move you through a door in a certain direction. If the door is open, it finds the connected room by looking in the **EXITS** list for the door.

```
TO TRYDOOR :DOOR :DIR
IF GPROP :DOOR "SHUT? [PR [THE DOOR IS SHUT] STOP]
GOROOM ITEM :DIR GPROP :DOOR "EXITS
END
```

The adventure program in this project has only one door (**DOOR1**), but I designed it so that I could add more doors.

Adventure Programs Produce Language

The program prints descriptions of rooms and objects, tells you the results of things you do, and sometimes complains if you try something impossible. All the messages it prints are in fairly normal English.

The program describes what you'd see if you were really in the simulated world. When you enter a room or give the **LOOK** command, the program describes the room. When you give the **INVENTORY** command, the program describes whatever you're carrying. When you give the **EXAMINE** command, the program describes an object in more detail (sometimes).

Every object or room has a **DESCRIPT** property. For an object, this property is a noun phrase describing the object (for example, **A DULL SWORD**).^{*} For a room, this property is a prepositional phrase describing your relation to the room (for example, **IN A TWISTY MAZE OF LITTLE PASSAGES**). The descriptions are in different forms because the descriptions are used in different ways. A room description is for telling you where you are (*in* a room, *on* a ladder, *at* a computer terminal) and the object descriptions are for saying what a thing is.

LOOK describes the room you're in.

```
TO LOOK
( PR [YOU ARE] GPROP :#ROOM "DESCRIPT )
IF NOT EMPTYP GPROP :#ROOM "CONTAINS
  [TYPE [YOU SEE] DESCRIBE GPROP :#ROOM "CONTAINS]
PREXITS
PRDOORS
END
```

The value of the **CONTAINS** property of a room is a list of the words for the objects in that room. If you look at **INITROOMS** (in the full listing), you'll see how I set up the initial contents of each room.

DESCRIBE prints the descriptions of a list of objects:

^{*}The descriptions include the correct article (A or AN) for the word. I could have written a program to choose (checking whether the first letter is a vowel), but it would have taken up extra space and cost some extra time to execute. If there were seven hundred items instead of seven, I would have written the procedure, because it requires less space and less work to write it than to include an article in each of seven hundred descriptions.

```

TO DESCRIBE :LIST
IF EMPTYP :LIST [PR [] STOP]
TYPE "\"
TYPE GPROP FIRST :LIST "SCRIPT"
IF NOT EMPTYP BF :LIST [TYPE ",]
DESCRIBE BF :LIST
END

```

DESCRIBE gets each object's description from its SCRIPT property. It prints out the objects' descriptions, one after the other, all on the same line.

There are separate procedures for listing exits and doors because I thought it looked better to list them separately. PREXITS lists the directions in which you can leave a room, and PRDOORS lists the doors of the room. These procedures are similar.

```

TO PREXITS
PREXITS1 :#DIRNAMES GPROP :#ROOM "EXITS
END

TO PREXITS1 :DIRS :EXITS
IF EMPTYP :DIRS [STOP]
PREXIT FIRST :DIRS FIRST :EXITS
PREXITS1 BF :DIRS BF :EXITS
END

```

The variable #DIRNAMES holds a list of the names of all directions in the same order as they appear in the exit list. PREXITS looks at the first direction name in #DIRNAMES and the first exit in the EXITS list. Since the names and the exits are in the same order, it can tell what name to use for the direction of the exit. It maintains this one-to-one correspondence as it checks each item of the lists. An item is an exit if it is not empty and not a door.

```

TO PREXIT :DIRECTION :OUT
IF EMPTYP :OUT [STOP]
IF DOORP :OUT [STOP]
PR [YOU CAN GO] :DIRECTION
END

```

PRDOORS differs from PREXITS because it checks for doors instead of exits and tells you about the doors.

```

TO PRDOORS
PRDOORS1 :#DIRNAMES GPROP :#ROOM "EXITS
END

TO PRDOORS1 :DIRS :DOORS
IF EMPTYP :DIRS [STOP]
PRDOOR FIRST :DIRS FIRST :DOORS
PRDOORS1 BF :DIRS BF :DOORS
END

```

```

TO PRDOOR :DIRECTION :DOOR
IF EMPTYP :DOOR [STOP]
IF NOT DOORP :DOOR [STOP]
( PR IF GPROP :DOOR "SHUT? [[A CLOSED]] [[AN OPEN]]
GPROP :DOOR "DESCRIPT [LEADS] :DIRECTION)
END

```

The Syntax and Semantics of the Adventure Language

The first thing the program has to do to understand your sentence is to decide what type of word (noun, verb) each word in the sentence is. My program uses a very simple scheme: If the sentence is one word long, the first word must be a verb or a direction. If it is two words long, the first word must be a verb and the second a noun.

The procedure INTERPRET takes one input, a sentence.

```

TO INTERPRET :COM
IF ( COUNT :COM ) = 1 [INTER1 FIRST :COM STOP]
IF ( COUNT :COM ) = 2 [INTER2 FIRST :COM LAST :COM STOP]
PR [I KNOW ONLY ONE WORD AND TWO WORD SENTENCES.]
END

```

One-word and two-word sentences are handled by separate procedures. Anything else is an error.

```

TO INTER1 :VERB
IF MEMBERP :VERB :#DIRNAMES [MOVE THING :VERB STOP]
IF MEMBERP :VERB :#VERBS1 [RUN ( SE :VERB [] ) STOP]
PR [I DONT UNDERSTAND]
END

```

If you type NORTH, INTER1 finds the word NORTH in #DIRNAMES and knows you want to move in that direction. Each direction word has a value that is a number from 1 to 6 (an index into the EXITS list for the current room). The procedure uses THING to get the value of the direction word, and gives that as the input to the MOVE procedure.

Next the program looks at what the words *mean*. The “meaning” of a verb is given by a Logo procedure that carries out an action. For every verb in the language there is a Logo procedure. Conveniently, the Logo procedure has the same name as the verb.

The global variable #VERBS1 is a list of all single-word verbs.

```
MAKE "#VERBS1 [? LOOK INVENTORY]
```

If your single-word sentence is in #VERBS1, I use RUN to run the verb procedure. RUN wants a list as input, not a word, so I use SE to make a list containing only the verb.

INTER2 interprets two word sentences. Like INTER1, it checks whether the first word is a member of a list of verbs:

```
MAKE "#VERBS2 [TAKE DROP UNLOCK EXAMINE DRINK]
```

and runs a Logo procedure for the verb.

GAMES

```

TO INTER2 :VERB :NOUN
IF MEMBERP :VERB :#VERBS2
  [RUN LIST :VERB WORD "" :NOUN STOP]
PR [I DONT UNDERSTAND]
END

```

Each verb procedure takes one input, a noun. The noun is the second word in the sentence.

If you type the two-word sentence DRINK WINE, the program interprets the sentence by invoking the procedure DRINK (the verb) with the word WINE (the noun) as its input. In other words, the program carries out the Logo instruction

DRINK "WINE

That's why, in making the input list for RUN, I add the quote character ("") in front of the noun. Otherwise Logo would try to run the instruction

DRINK WINE

which is wrong.

Verbs in One-word Sentences

The simplest verb is ?, which just prints the names of all verbs that the interpreter knows. It exists so you won't have to remember the names.

```

TO ?
PR :#VERBS1
PR :#VERBS2
PR :#DIRNAMES
END

```

The verb INVENTORY takes an inventory of objects you've picked up. The global variable #INVENTORY holds the list of objects. The procedure DESCRIBE (explained earlier) prints the actual description.

```

TO INVENTORY
TYPE [YOU'RE CARRYING]
IF EMPTYP :#INVENTORY [PR "\ NOTHING] [DESCRIBE :#INVENTORY]
END

```

The verb LOOK is called to describe a room when you enter it.

Verbs with Objects

Nouns refer to objects in the simulated world. The program has to determine what a noun means. A sentence like "take rope" means that the user wants to pick up the rope. Somehow the program has to translate the word "rope" to the word the program uses to represent the rope.

My program has an extremely simple solution. The word the program uses is the *same* as the word in the Adventure language. That is why I had to be careful choosing names for the words I used in the program. They had to be the same as the words I thought users would use in their sentences.

So if you say "take rope" the word "rope" can only mean the piece of rope that the word ROPE represents.*

Objects in Your Inventory or Locale

The predicate CARRYINGP tests whether you've got an object with you. You do if it's in your inventory, or if it is contained within an object in your inventory. (For example, if you're carrying a satchel and there is a blowtorch in the satchel, then CARRYINGP "BLOWTORCH" outputs TRUE.)

```
TO CARRYINGP :OBJ
OP WITHIN :#INVENTORY :OBJ
END
```

The predicate INROOMP tests whether an object is somewhere in the room.

```
TO INROOMP :OBJ
OP WITHIN GPROP :#ROOM "CONTAINS" :OBJ
END
```

WITHIN checks for an object either in a list or contained in an object in that list.

```
TO WITHIN :LIST :THING
IF EMPTYP :LIST [OP "FALSE"]
IF EQUALP :THING FIRST :LIST [OP "TRUE"]
IF WITHIN :THING (GPROP FIRST :LIST "CONTAINS") [OP "TRUE"]
OP WITHIN BF :LIST :THING
END
```

The EXAMINE Verb

Like all verbs with objects, EXAMINE first checks whether or not the object you mention is present by using ABSENT.†

```
TO ABSENT :OBJ
IF PRESENTP :OBJ [OP "FALSE"]
PR [I DONT SEE IT HERE]
OP "TRUE"
END
```

An object is present if it's either in your inventory or lying loose in the room.

*This solution has drawbacks. First, the user must spell the word exactly as I do and must not use synonyms. There are other drawbacks as well, but I'll save them for later.

†I put this feature in, even though there are no carryable containers in the game, because it seemed elegant, and I might want to add containers later.

ABSENT combines two actions in one procedure. It is a predicate, the opposite of PRESENTP, and it also prints a message if the object is absent.

This extra action restricts the usefulness of ABSENT. It should only be called by a verb procedure, because otherwise it is not appropriate to print the message. Usually it's a bad idea to combine functions like this, but I did it after I discovered that only verb procedures used ABSENT and that each of them printed the same message if the object was absent. I combined the test and the message into one procedure to save space.

GAMES

```
TO PRESENTP :OBJ
OP OR CARRYINGP :OBJ INROOMP :OBJ
END
```

If an object is absent, the program just says so. The program deliberately doesn't distinguish between objects that exist somewhere but aren't nearby and objects that don't exist. If the object contains something, EXAMINE tells you about it. That's the only detail that EXAMINE ever gives.

```
TO EXAMINE :OBJ
IF ABSENT :OBJ [STOP]
IF EMPTYP GPROP :OBJ "CONTAINS [STOP]
TYPE [IT HOLDS] DESCRIBE GPROP :OBJ "CONTAINS
END
```

The TAKE and DROP Verbs Change Your Inventory

The TAKE program has to be more careful than EXAMINE. This is because there are many reasons that might prevent you from taking an object.

- It might not be there.
- You might already have it.
- It might be too heavy.
- It might be impossible to carry.
- Your arms could be full.

The program checks for each of these, making an appropriate complaint. If nothing prevents it, the object is added to your inventory and removed from the contents of the room.

```
TO TAKE :OBJ
IF ABSENT :OBJ [STOP]
IF CARRYINGP :OBJ [PR [YOU ALREADY HAVE IT!] STOP]
IF PROPTURE? :OBJ "IMMOBILE? [PR [IT'S TOO HEAVY] STOP]
IF PROPTURE? :OBJ "LIQUID? [PR [NO CONTAINER] STOP]
IF (COUNT :#INVENTORY) > 2 [PR [YOUR BAG IS FULL!] STOP]
MAKE "#INVENTORY FPUT :OBJ :#INVENTORY
PPROP :#ROOM "CONTAINS (REMOVE :OBJ GPROP :#ROOM "CONTAINS)
END
```

When I wrote this procedure I had to decide how to represent the mobility of objects. I could have given everything a WEIGHT property and compared that with a STRENGTH variable, but I rejected that as too much work. All I wanted was to prevent clearly impossible requests, such as picking up trees. For my purposes, objects are either heavy or not, so this suggested a property that was TRUE only if the object was liftable.

I chose to give heavy objects an IMMOBILE? property of TRUE instead of giving light objects a MOBILE? property because I knew I could save some space that way. I knew there would be only a few heavy objects and many light ones, and if I wrote my programs to assume that an object was light unless explicitly marked heavy I could avoid storing all the IMMOBILE? properties that were FALSE.

To make this assumption easier to program, I wrote the predicate PROPTURE?, which outputs TRUE only if the value of the property is TRUE. If an object doesn't have any value for a certain property, then GPROP outputs the empty list. The empty list isn't TRUE or FALSE, so I can't use GPROP directly in an IF.

```
TO PROPTURE? :OBJ :PROP
OP ( GPROP :OBJ :PROP ) = "TRUE"
END
```

I used the same kind of reasoning in defining the LIQUID? property. Most objects are dry.

It took a lot of thought to save a little space. Fortunately, the program is only a little harder to understand as a result. It would be foolish to make the program very complex just to save a little space.

DROP is the opposite of TAKE, but has to check only whether you are carrying the object.

```
TO DROP :OBJ
IF NOT CARRYINGP :OBJ [PR [YOU'RE NOT CARRYING IT!] STOP]
MAKE "#INVENTORY REMOVE :OBJ :#INVENTORY
PPROP :#ROOM "CONTAINS ( FPUT :OBJ GPROP :#ROOM "CONTAINS )
END
```

REMOVE takes as inputs a list and an item in the list and outputs the list with the item removed.

```
TO REMOVE :THING :LIST
IF EMPTYP :LIST [OP []]
IF :THING = FIRST :LIST [OP BF :LIST]
OP FPUT FIRST :LIST REMOVE :THING BF :LIST
END
```

The Game Program

Before showing you the rest of the verbs, I'd like to show you the top-level game loop. You start the program with ADVENTURE.

```
TO ADVENTURE
INIT
LOOK
GAMELOOP
END
```

INIT initializes everything.

```
TO INIT
INITVARS
INITOBJS
INITROOMS
END
```

INITVARS initializes all global variables, INITOBJS initializes all objects, and INITROOMS initializes all rooms. I won't include their definitions

here because they are just long lists of MAKEs and PPROPs. They are in the listing at the end of this write-up.

LOOK describes the room you're in. It was explained earlier.

The game loop is GAMELOOP:

```
TO GAMELOOP
INTERPRET GETINPUT
IF NOT EMPTYP :#RESULT [PR :#RESULT STOP]
GAMELOOP
END
```

GETINPUT prompts for a sentence and returns it.

```
TO GETINPUT
TYPE ">
OP RL
END
```

The game loop runs until the variable #RESULT becomes nonempty. Any verb can end the game by putting a message about the result of the game into that variable.

Now we'll look at the last two verbs.

UNLOCK *Unlocks a door*

Although it may not appear so at first, it's a little difficult for the program to understand UNLOCK DOOR because it's hard to tell what object the word DOOR refers to. Remember, I wanted it to be possible for there to be many doors.

The UNLOCK verb first ensures that you asked to unlock a DOOR. The program then uses GETDOOR to try to find a door. If there is one, UNLOCK looks at its KEY property to be sure that you have the key that unlocks it:

```
TO UNLOCK :OBJ
IF NOT EQUALP :OBJ "DOOR [PR [I CAN'T UNLOCK THAT] STOP]
MAKE "OBJ GETDOOR ( GPROP :#ROOM "EXITS ")
IF EMPTYP :OBJ [PR [NOTHING UNLOCKABLE HERE] STOP]
IF NOT CARRYINGP GPROP :OBJ "KEY [PR [YOU CAN'T] STOP]
PPROP :OBJ "LOCKED? "FALSE
PPROP :OBJ "SHUT? "FALSE
END
```

Unlocking a door also opens it automatically; you don't need to OPEN a door after you UNLOCK it.*

The procedure GETDOOR looks at every item in the EXITS list of the current room until it finds one that is a door, and outputs it. This door is assumed to be the object referred to by "DOOR" in the sentence "UNLOCK DOOR".

```
TO GETDOOR :EXITS
IF EMPTYP :EXITS [OP []]
```

*I didn't have enough space for a verb OPEN, and requiring you to OPEN the door slows the game down to no purpose, anyway.

```
IF DOORP FIRST :EXITS [OP FIRST :EXITS]
OP GETDOOR BF :EXITS
END .
```

DRINK*ing* Ends the Game

If you've played the game, you know the unfortunate effects of drinking wine.

```
TO DRINK :OBJ
IF ABSENT :OBJ [STOP]
IF NOT PROPTURE? :OBJ "LIQUID? [PR [NOT A LIQUID!] STOP]
PR [YOU DRINK THE WINE...]
WAIT 10
PR [IT'S DELICIOUS...] WAIT 10
PR [AND VERY POTENT. YOU GET DRUNK, AND FALL IN THE RIVER.]
MAKE "#RESULT [YOU DROWNED]
END
```

The DRINK verb is a little strange. It checks that you are drinking a liquid that is present, but then it makes two assumptions: that you are drinking wine and that you are drinking by the river. The program tells you the deadly result by setting the global variable #RESULT to a sentence. GAMELOOP notices, and ends the game.

The two assumptions are used in at least two ways. The first is that the message uses the words "wine" and "river" explicitly, and also says that the result is drunkenness. The second use of the assumption is that the result only is possible if you are by the river.

The assumption that you drank wine must be true because the only liquid in the game is wine. The assumption of locale is safe because the only wine in the game is in the barrel, and you can't move the barrel.

It is not a good thing to make assumptions like these in writing programs because it makes it hard to extend the program. (If I had added other liquids, I would have had to add an INTOXICATING? property to WINE to distinguish it from safer liquids.) I did it to save space, but I'm not proud of it.

You Can Change This Adventure in Many Ways

The easiest thing to do is to add new rooms. All you need to do is change the property assignments in INITROOMS. Make sure that you provide some path from every room to every other room.

You can add new doors in the same way as you add rooms. But it's hard to add a new key, for reasons I'll explain. So when making new doors, either they should not be locked or they should use the same key.

If you want to make a one-way exit from one room to another, do not include the first room in the EXITS of the second. (You can also make one-way doors.)

It's also easy to put new objects into the game. All the objects are created by INITOBJS.

You add new verbs by writing the procedure and modifying the list #VERBS1 or #VERBS2. But new verbs or objects may need some new properties. For example, if you added the verb EAT you'd want to give edible objects a FOOD? property of TRUE, and have EAT check it.

It might be fun to make a type of door that only opens after you take some action such as saying a password, or pressing a button.

Your verbs can end the game at any time by setting #RESULT.

It's easy to debug changes to Adventure. You can stop the program with the BREAK key, look at things, fix them, then resume with GAMELOOP. (This is also a good way to cheat.)

Some Problems with My Program

The scheme I use for semantics is not very good. The nouns you type must be the same words as those used by the program. That is why it would be hard to add another key (for example, a brass one). What Logo word would you use to represent it? You can't use KEY, because that word already stands for a different key, the iron one in the forest. When you type TAKE KEY the program looks for the word spelled "k-e-y". Suppose you use KEY1. The description of the new key would be [A BRASS KEY], and the user would try to refer to it with the word used in the description, namely KEY. The user would have no way to know that the "right" word is really KEY1, and even if the program printed out that word, it wouldn't be much like English. Can you imagine "You see a brass key1"?

For the same reason, there cannot be a second sword, or crowbar, or any such thing. The problem is most acute with keys, though, because that means that one key must be able to unlock all doors.

Note that this is not a problem for rooms, because the user never refers to a room in any way. It is also not much of a problem for verbs, because it would be easy to give each verb a property holding the name of a Logo procedure to run. Then verbs would not have to have the same name as the procedure that defines them.

One possible fix to this would be to give each item a property for what "kind" of thing it is.

```
PPROP "KEY" "KIND" "KEY"
PPROP "KEY1" "KIND" "KEY"
```

Then a reference to a "key" could be interpreted as meaning *any* object that was a key. This is similar to the way doors are identified by GETDOOR.

Another possible solution would be to give the noun KEY a property list of all the program words that are a "kind of" key.

There would still be problems with ambiguity. There might be two keys in the area. There is no way at present for the program to ask the user to say which key was meant. (This is a problem with GETDOOR as well. It takes the first door.) Perhaps the program could ask:

```
>TAKE KEY
DO YOU MEAN THE BRASS KEY, OR THE IRON KEY? BRASS
OK
```

Another problem is that room descriptions sometimes refer to things that the user might mention. For example, the description of the dock mentions an underground stream. People often try to drink the water, but the program doesn't even know there is a "stream" nearby, much less that a stream holds "water." The word "stream" is contained inside the description, and the program has no way to use it other than by printing it.

If scenes were described by properties, descriptions could perhaps be built from them, and the program would have access to the properties of the room. But generating good English from a set of properties is a difficult problem.

Some Adventuresome Improvements

There are many possible improvements to this game, some easy, others more difficult.

Writing programs that understand and produce natural language is a challenge for hundreds of researchers throughout the world. In a small way, Adventure is a part of this research.

First, you could fix the problems I just mentioned. But there is even more to do.

Consider a dialogue like

```
YOU ARE IN A FOOD STORE  
YOU SEE A GREEN CHEESE  
>TAKE IT
```

The program could use context to figure out what the word "it" means.

Or suppose you're carrying a baseball bat and a rock and are attacked by a vampire bat. The word "bat" in HIT BAT means the vampire bat, not the baseball bat.

It would also be very nice to have a richer syntax than the simple verb and noun scheme used here.

Many adventure games have autonomous characters. Usually they are your foes. It would be a fine challenge to add them to this game. Characters should move from room to room on their own, and sometimes the player should encounter them. The results need not always be woeful.

More complexly structured worlds are possible. The objects in my world are mostly decorative—there is nothing to pry with the crowbar, nowhere to climb with the rope.

If the Adventure language was extended such that you could use it to program, then you could teach a turtle how to explore, send it in to dangerous areas, and have it carry back things for you.

Writing good adventure programs is an art and a game of its own. Now that you've explored the simulated world of Adventure, perhaps it's time for you to begin exploring Adventure itself.

PROGRAM LISTING

```

TO ADVENTURE
INIT
LOOK
GAMELOOP
END

TO INIT
INITVARS
INITOBJS
INITROOMS
END

TO GAMELOOP
INTERPRET GETINPUT
IF NOT EMPTYP :#RESULT [PR :#RESULT ▶
    STOP]
GAMELOOP
END

TO GETINPUT
TYPE ">"
OP RL
END

TO INTERPRET :COM
IF ( COUNT :COM ) = 1 [INTER1 FIRST ▶
    :COM STOP]
IF ( COUNT :COM ) = 2 [INTER2 FIRST ▶
    :COM LAST :COM STOP]
PR [I KNOW ONLY ONE WORD AND TWO WORD ▶
    SENTENCES.]
END

TO INTER1 :VERB
IF MEMBERP :VERB :#DIRNAMES [MOVE ▶
    THING :VERB STOP]
IF MEMBERP :VERB :#VERBS1 [RUN SE ▶
    :VERB [] STOP]
PR [I DONT UNDERSTAND]
END

TO INTER2 :VERB :NOUN
IF MEMBERP :VERB :#VERBS2 [RUN LIST ▶
    :VERB WORD " " :NOUN STOP]
PR [I DONT UNDERSTAND]
END

TO MOVE :DIR
MOVE1 :DIR ( ITEM :DIR GPROP :#ROOM ▶
    "EXITS")
END

TO MOVE1 :DIR :THERE
IF EMPTYP :THERE [PR [YOU CAN'T GO ▶
    THAT WAY!] STOP]
IF DOORP :THERE [TRYDOOR :THERE :DIR] ▶
    [GOROOM :THERE]
END

TO DOORP :OBJ
IF EMPTYP :OBJ [OP "FALSE"]
OP PROPTURE? :OBJ "DOOR?"
END

TO GOROOM :NEW
MAKE "#ROOM :NEW
LOOK
END

TO LOOK
( PR [YOU ARE] GPROP :#ROOM "DESCRIPT ▶
    )
IF NOT EMPTYP GPROP :#ROOM "CONTAINS ▶
    [TYPE [YOU SEE] DESCRIBE GPROP ▶
        :#ROOM "CONTAINS]
PREXITS
PRDOORS
END

TO DESCRIBE :LIST
IF EMPTYP :LIST [PR [] STOP]
TYPE "\"
TYPE GPROP FIRST :LIST "DESCRIPT
IF NOT EMPTYP BF :LIST [TYPE ",]
DESCRIBE BF :LIST
END

TO PREXITS
PREXITS1 :#DIRNAMES GPROP :#ROOM ▶
    "EXITS
END

TO PREXITS1 :DIRS :EXITS
IF EMPTYP :DIRS [STOP]
PREXIT FIRST :DIRS FIRST :EXITS
PREXITS1 BF :DIRS BF :EXITS
END

TO PREXIT :DIRECTION :OUT
IF EMPTYP :OUT [STOP]
IF DOORP :OUT [STOP]
( PR [YOU CAN GO] :DIRECTION )
END

```

```

TO PRDOORS
PRDOORS1 :#DIRNAMES GPROP :#ROOM ▶
    "EXITS
END

TO PRDOORS1 :DIRS :DOORS
IF EMPTYP :DIRS [STOP]
PRDOOR FIRST :DIRS FIRST :DOORS
PRDOORS1 BF :DIRS BF :DOORS
END

TO PRDOOR :DIRECTION :DOOR
IF NOT DOORP :DOOR [STOP]
( PR IF PROPTURE? :DOOR "SHUT? [[A ▶
    CLOSED]] [[AN OPEN]] GPROP :DOOR ▶
    "DESCRIPT [LEADS] :DIRECTION )
END

TO TRYDOOR :DOOR :DIR
IF GPROP :DOOR "SHUT? [PR [THE DOOR IS ▶
    SHUT] STOP]
GOROOM ITEM :DIR GPROP :DOOR "EXITS
END

TO ?
PR :#VERBS1
PR :#VERBS2
PR :#DIRNAMES
END

TO INVENTORY
TYPE [YOU'RE CARRYING]
IF EMPTYP :#INVENTORY [PR " NOTHING] ▶
    [DESCRIBE :#INVENTORY]
END

TO TAKE :OBJ
IF ABSENT :OBJ [STOP]
IF CARRYINGP :OBJ [PR [YOU ALREADY ▶
    HAVE IT!] STOP]
IF PROPTURE? :OBJ "IMMOBILE? [PR [ITS ▶
    TOO HEAVY] STOP]
IF PROPTURE? :OBJ "LIQUID? [PR [NO ▶
    CONTAINER] STOP]
IF ( COUNT :#INVENTORY ) > 2 [PR [YOUR ▶
    BAG IS FULL!] STOP]
MAKE "#INVENTORY FPUT :OBJ :#INVENTORY
PPROP :#ROOM "CONTAINS ( REMOVE :OBJ ▶
    GPROP :#ROOM "CONTAINS )
END

TO CARRYINGP :OBJ
OP WITHIN :#INVENTORY :OBJ
END

TO ABSENT :OBJ
IF PRESENTP :OBJ [OP "FALSE]
PR [I DONT SEE IT HERE]
OP "TRUE
END

TO PRESENTP :OBJ
OP OR CARRYINGP :OBJ INROOMP :OBJ
END

TO WITHIN :LIST :THING
IF EMPTYP :LIST [OP "FALSE]
IF EQUALP :THING FIRST :LIST [OP ▶
    "TRUE]
IF WITHIN ( GPROP FIRST :LIST ▶
    "CONTAINS ) :THING [OP "TRUE]
OP WITHIN BF :LIST :THING
END

TO INROOMP :OBJ
OP WITHIN GPROP :#ROOM "CONTAINS :OBJ
END

TO DROP :OBJ
IF NOT CARRYINGP :OBJ [PR [YOUR NOT ▶
    CARRYING IT!] STOP]
MAKE "#INVENTORY REMOVE :OBJ ▶
    :#INVENTORY
PPROP :#ROOM "CONTAINS ( FPUT :OBJ ▶
    GPROP :#ROOM "CONTAINS )
END

TO EXAMINE :OBJ
IF ABSENT :OBJ [STOP]
IF EMPTYP GPROP :OBJ "CONTAINS [STOP]
TYPE [IT HOLDS] DESCRIBE GPROP :OBJ ▶
    "CONTAINS
END

TO UNLOCK :OBJ
IF NOT EQUALP :OBJ "DOOR [PR [I CANT ▶
    UNLOCK THAT] STOP]
MAKE "OBJ GETDOOR ( GPROP :#ROOM ▶
    "EXITS )
IF EMPTYP :OBJ [PR [NOTHING UNLOCKABLE ▶
    HERE] STOP]
IF NOT CARRYINGP GPROP :OBJ "KEY [PR ▶
    [YOU CAN'T] STOP]
PPROP :OBJ "LOCKED? "FALSE
PPROP :OBJ "SHUT? "FALSE
END

```

```

TO GETDOOR :EXITS
IF EMPTYP :EXITS [OP []]
IF DOORP FIRST :EXITS [OP FIRST ▶
    :EXITS]
OP GETDOOR BF :EXITS
END

TO DRINK :OBJ
IF ABSENT :OBJ [STOP]
IF NOT PROPTURE? :OBJ "LIQUID? [PR ▶
    [NOT A LIQUID!] STOP]
PR [YOU DRINK THE WINE...]
WAIT 10
PR [IT'S DELICIOUS...] WAIT 10
PR [AND VERY POTENT. YOU GET DRUNK, ▶
    AND FALL IN THE RIVER.]
MAKE "#RESULT [YOU DROWNED]
END

TO INITVARS
MAKE "NORTH 1
MAKE "EAST 2
MAKE "SOUTH 3
MAKE "WEST 4
MAKE "UP 5
MAKE "DOWN 6
MAKE "#DIRNAMES [NORTH EAST SOUTH WEST ▶
    UP DOWN]
MAKE "#VERBS1 [? LOOK INVENTORY]
MAKE "#VERBS2 [TAKE DROP UNLOCK ▶
    EXAMINE DRINK]
MAKE "#INVENTORY []
MAKE "#ROOM "FOREST
MAKE "#RESULT []
END

TO INITOBJS
PPROP "SWORD "DESCRIPT [A DULL SWORD]
PPROP "CROWBAR "DESCRIPT [A SEARS ▶
    CROWBAR]
PPROP "ROPE "DESCRIPT [A COIL OF HEMP ▶
    ROPE]
PPROP "KEY "DESCRIPT [AN IRON KEY]
PPROP "TREES "DESCRIPT [ASH AND BIRCH ▶
    TREES]
PPROP "TREES "IMMOBILE? "TRUE
PPROP "WINE "DESCRIPT [TASTY WINE]
PPROP "WINE "LIQUID? "TRUE
PPROP "BARREL "DESCRIPT [AN OAKEN ▶
    BARREL]
PPROP "BARREL "CONTAINS [WINE]
PPROP "BARREL "IMMOBILE? "TRUE
END

TO INITROOMS
PPROP "ANTE "EXITS [[] GUARD DOOR1 [] ▶
    [] STAIRS]
PPROP "ANTE "DESCRIPT [IN THE ANTEROOM ▶
    OF THE FORT]
PPROP "STAIRS "EXITS [[] [] [] [] ANTE ▶
    HALL]
PPROP "STAIRS "DESCRIPT [ON SOME ▶
    STAIRS]
PPROP "LADD "EXITS [[] [] [] HALL [] ▶
    DOCK]
PPROP "LADD "DESCRIPT [AT THE TOP OF A ▶
    LADDER]
PPROP "DOOR1 "EXITS [ANTE [] FOREST [] ▶
    [] []]
PPROP "DOOR1 "DESCRIPT [STOUT IRON ▶
    DOOR]
PPROP "DOOR1 "KEY "KEY
PPROP "DOOR1 "DOOR? "TRUE
PPROP "DOOR1 "LOCKED? "TRUE
PPROP "DOOR1 "SHUT? "TRUE
PPROP "DOCK "EXITS [[] [] [] LADD ▶
    []]
PPROP "DOCK "DESCRIPT [ON A STONE ▶
    DOCK, BESIDE AN UNDERGROUND ▶
    STREAM]
PPROP "DOCK "CONTAINS [BARREL]
PPROP "HALL "EXITS [[] LADD [] [] ▶
    STAIRS []]
PPROP "HALL "DESCRIPT [IN A HIGH, ▶
    NARROW HALL]
PPROP "HALL "CONTAINS [ROPE]
PPROP "GUARD "EXITS [[] [] [] ANTE [] ▶
    []]
PPROP "GUARD "DESCRIPT [IN AN OLD ▶
    GUARD ROOM]
PPROP "GUARD "CONTAINS [CROWBAR SWORD]
PPROP "CLIFF "EXITS [DOOR1 [] FOREST ▶
    F1 [] []]
PPROP "CLIFF "DESCRIPT [AT THE BASE OF ▶
    A CLIFF]
PPROP "CLIFF "CONTAINS [TREES]
PPROP "FOREST "EXITS [CLIFF FOREST ▶
    FOREST F1 [] []]
PPROP "FOREST "DESCRIPT [IN A FOREST]
PPROP "FOREST "CONTAINS [TREES]
PPROP "F1 "EXITS [CLIFF FOREST F1 F1 ▶
    [] []]
PPROP "F1 "DESCRIPT [IN A FOREST]
PPROP "F1 "CONTAINS [TREES KEY]
END

```

```

TO REMOVE :THING :LIST
IF EMPTYP :LIST [OP []]
IF :THING = FIRST :LIST [OP BF :LIST]
OP FPUT FIRST :LIST REMOVE :THING BF ▶
:LIST
END

TO ITEM :N :LIST
IF :N = 1 [OP FIRST :LIST]
OP ITEM :N - 1 BF :LIST
END

TO PPROP :NAME :PROP :VALUE
IF NOT NAMEP :NAME [MAKE :NAME []]
MAKE :NAME PPROP1 THING :NAME :PROP ▶
:VALUE
END

TO PPROP1 :PLIST :PROP :VALUE
IF EMPTYP :PLIST [OP LIST :PROP ▶
:VALUE]

IF :PROP = FIRST :PLIST [OP FPUT :PROP ▶
FPUT :VALUE BF BF :PLIST]
OP FPUT FIRST :PLIST FPUT FIRST BF ▶
:PLIST PPROP1 BF BF :PLIST :PROP ▶
:VALUE
END

TO GPROP :NAME :PROP
IF NOT NAMEP :NAME [OP []]
OP GPROP1 :PROP THING :NAME
END

TO GPROP1 :PROP :PLIST
IF EMPTYP :PLIST [OP []]
IF :PROP = FIRST :PLIST [OP FIRST BF ▶
:PLIST]
OP GPROP1 :PROP BF BF :PLIST
END

TO PROPTRUE? :OBJ :PROP
OP ( GPROP :OBJ :PROP ) = "TRUE"
END

```

Dungeon

Here's what the original author of this game, Jeanry Chandler, has to say about it.

Dungeon has all of the virtues of an adventure game and more. Not only can you play the part of the dauntless adventurer, exploring the dungeons in search of wonder and magic and fabulous treasures. You can also play the part of the all powerful dungeonmaster: laying the traps, preparing the monsters, and, of course, placing the treasure.

When you start DUNGEON, you are given the first level of the dungeon, four rooms of treasures, and traps, monsters, and money. The adventurer who enters this perilous palace can expect to face such horrific monsters as the nimble kobold, the mighty troll, and, worst of all, the fearsome man-eating Logo turtle!

But monsters are not the only inhabitants of the dreaded dungeon. As reward for defeating or avoiding those grumpy gargantuans, that same adventurer can fill his pockets with gold, jewels, magical potions, wands, enchanted swords, armor, and much more.

The successful adventurer who gains enough experience points could find himself going up a level of skill, thereby gaining fighting ability and strength.

After a period of blissful adventure, however, the aspiring adven-

turer will grow weary of the limits of my own possibly limited imagination. And this is where the dungeonmaster and the truly original facet of the game come into play.

The adventurer, or a friend, can become the dungeonmaster and design a new level filled with treasures and trolls. To start out, the dungeonmaster might simply make new rooms. Later he could design new shapes for monsters, new treasures, traps, and tricks for the adventurer to face.

More explanation would be detrimental to your understanding of the program. You must experience it to learn more!

As Jeanry says, there are two ways to enjoy Dungeon. One is to be the adventurer in his dungeon, and the other is to add to Jeanry's dungeon or to create your own with his tools. He's right, too, that you must experience it to understand it. So try it!

To start the game, type:

START

Playing the Dungeon Game

START begins the game. You control an adventuring player with a joystick in port 1.* You can move the adventurer with the joystick to avoid or confront monsters, to go through doors, and to get the contents of chests.



As in other adventure games, there are some commands you can type. I stands for inventory, D for drink, and W for wave wand. You must press RETURN after your commands.

*Remember that in Atari Logo, the joystick is referenced by JOY 0 when it is in port 1 of the Atari.

An Overview

The following sections present an overview of how this program works as the game is played and of how to modify the dungeon rooms and create your own dungeon. Then there are some suggestions for modifying and improving this program in more radical ways.

All of the procedures are listed at the end of this write-up. You may want to look at some of them as you read about them.

Rooms

Each room is represented by a single procedure (for example, ROOM1, ROOM2) that prints messages about what is in the room and makes turtles into monsters and treasure chests. Turtle 0 is the player, turtle 1 is usually a monster, and turtle 2 is usually a treasure chest.

MAKEROOM is called as a subprocedure from all rooms. It does stuff that needs to be done for each room: it draws walls and doors, sets variables with the dimensions of the room, and creates some demons. Two of the demons that it creates are those that wait for a collision between the player and the monster and between the player and a treasure chest. It also creates a demon that lets you control the player with the joystick and demons that keep the player and monster from drifting through the walls of the room.

The rest of the instructions in each room procedure customize the room. For example, look at ROOM2 in which turtle 2 is a chest and turtle 1 is a kind of monster called a kobold.

```

TO ROOM2
PR [THERE IS A BIG CHEST HERE]
PR [THERE IS A CUTE KOBOLD HERE]
RT 180
MAKEROOM 200 90 [[N ROOM3] [E ROOM1]]
PU
HOME FD 50
SETSP 20
SET.MONSTER 3 [-20 25] 270           Give monster kobold shape.
MAKE "MHITP 2
ASK 1 [SETSP 22]
ASK 2 [PU FD 30 LT 90 FD 25 ST]
END

```

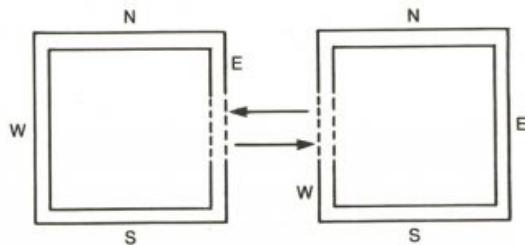
Walls and Doors

The walls and doors are set up for each room by the MAKEROOM procedure. The walls are drawn with pen 0 and the doors with pen 1 and pen 2. A room can have up to two doors.

The SETUP.DEMONS subprocedure of MAKEROOM creates a demon that waits for the player (turtle 0) to bump into a wall; the demon calls a procedure that makes the player bounce back. The condition for this demon is

OVER 0 0. Demons are also created to make the monster bounce off the walls and doors. The monster is not allowed to go through doors.

The DOOR.SIDE subprocess of MAKEROOM creates demons that wait for the player to bump into a door. These demons use conditions OVER 0 1 or OVER 0 2 to detect this. When the player bumps into a door, the player is moved into the adjoining room. The way this works is that the demon for that door calls the procedure that represents the adjoining room.



ROOM1 with connection W to ROOM2

The demon for this situation is created by DOOR.SIDE using

WHEN OVER 0 1 [ROOM2]

The Most Common Actions, FIGHT and CHEST

The SETUP.DEMONS procedure, called by MAKEROOM, creates the demons that wait for the player-monster and player-treasure chest collisions.

The demon that awaits collisions between the player and the monster is created using the instruction WHEN TOUCHING 0 1 [FIGHT]. Turtle 0 is the player and turtle 1 is a monster. When they collide, the demon calls the procedure FIGHT, which causes the two turtles to "fight." They swing at one another. The program considers the strengths and magical aids of the two combatants and determines if either one is hit. As the combatants continue to receive blows, they accumulate "hit points." If either sustains too much damage, it dies.

The demon that awaits collisions between the player and the treasure chest is created using the instruction WHEN TOUCHING 0 2 [CHEST]. Turtle 2 is a treasure chest. If the player collides with turtle 2, the demon calls CHEST. The procedure CHEST determines what treasures the chest contains and rewards the player with those treasures. Usually a treasure is given to the player by changing the value of a global variable such as :GOLD or :ARMOR.

Procedural and Demon-Based Representation

In this program, the only way to figure out all the details is to look at all the procedures. You might say that the program itself "figures it out as it goes along." You might contrast this with Jim Davis's Adventure game. Jim's program has global structures that contain information about his dun-

geon. For example, it has a list of all rooms. In Dungeon, almost all information is in the room procedures.

Programmer as Dungeonmaster: How to Create New and Better Dungeons

In this section I will discuss three kinds of changes to the Dungeon game. You can create new monsters, treasures, and whole rooms to put them in. You can create a new dungeon to replace Jeanry's. You can make changes to the workings of the game program itself to improve and change the game.

Creating New Rooms, Monsters, and Treasures

Looking at the room procedures (ROOM1, ROOM2, and so forth) will help you figure out how to make new rooms. You could make your very own completely new dungeon, or you could add rooms to the existing one. Remember that when you add a room, you might want to change some of the old rooms so that they connect to your new room. You might want to change the W procedure so that the wand can magically teleport the player into your new room.

You may have noticed that Jeanry's doors are *two-way*. That is, if a door leads *west* from ROOM1 to ROOM2, then there is a door that leads *east* from ROOM2 to ROOM1. Jeanry has made all of his doors match up. You might want to make all your doors match up too, or you could make some doors be one-way only. You could make some interesting and confusing dungeons this way.

The MAKEROOM procedure sets :ROOMLENGTH and :ROOMHEIGHT to be the length and height of the room. You can use these to position turtles or drawings in the rooms.

Here's an example of a new room I created.

```
TO ROOM5
PR [THIS IS MARGARET'S ROOM]
PR [BEWARE THE TROLL HERE]
PR [THERE IS A MYSTERIOUS ANCIENT CHEST HERE]
MAKEROOM 100 120 [[W ROOM4] [S ROOM2]]
PU
HOME
SETPOS SE (:ROOMLENGTH / 2) - 50 :ROOMHEIGHT / 2           Player in middle of room.
SETH 45
SETSP 20
SET.MONSTER 2 SE :ROOMLENGTH - 80 :ROOMHEIGHT - 25 300   Troll in corner.
MAKE "MNSTR 1
ASK 1 [SETSP 22]
ASK 2 [PU SETPOS SE :ROOMLENGTH - 90 25 ST]
END
```

This new room has a door leading west to ROOM4 and a door leading south to ROOM2. To make it possible for the player to get to this room, I put a door in ROOM4 leading east to ROOM5. To do this I changed a line in ROOM4 from

GAMES

```
MAKEROOM 100 100 [[S ROOM3]]
```

to

```
MAKEROOM 100 100 [[S ROOM3] [E ROOM5]]
```

I also changed a line in W from

```
IF EQUALP :WAND 2 [PR [YOU ARE TELEPORTED TO A NEW LOCATION]
RUN FPUT WORD "ROOM 1 + RANDOM 4 []]
```

to

```
IF EQUALP :WAND 2 [PR [YOU ARE TELEPORTED TO A NEW LOCATION]
RUN FPUT WORD "ROOM 1 + RANDOM 5 []]
```

To add a new kind of monster, you might want to make a new shape for it. You could add it to an old or new room, using the SET.MONSTER procedure. (You will probably want to add instructions about your monster shape in the START and SAVESH procedures.)

You can add new kinds of treasure. You must put your new kind of treasure in the CHEST procedure so it can be "in" the treasure chest. Then you must create a procedure to be run when your new treasure is found. If you want to represent your kind of treasure as a variable with a point value (like GOLD or WAND), you should initialize it in the START procedure.

Making a New Dungeon

Jeanry has left an opening in the program for you to add a complete new dungeon.

When you are playing the game and get to the stairs (in ROOM4), the program asks if you want to go down. If you answer Y, then the program types a message saying that you cannot go down to the lower dungeon unless you create it.

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
KEY "Y
PRINT [YOU CAN ONLY GO TO THE LOWER DUNGEON]
PRINT [IF YOU CREATE IT!]
END
```

Let's say you create several new rooms that connect to each other but do not connect to Jeanry's original four rooms. For example, let's say you make ROOM6, ROOM7, ROOM8, and ROOM9.

Then you could change STAIRS to

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
KEY "Y
ROOM7
END
```

Then STAIRS would plunk the player right into your dungeon. If you need more Logo workspace for your new dungeon, you could put your new room procedures in a separate file, for example D:LOWERDUNGEON. Then you could have STAIRS erase Jeanry's dungeon and load in yours. For example:

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
KEY "Y"
ER [ROOM1 ROOM2 ROOM3 ROOM4]
LOAD "D:LOWERDUNGEON
ROOM7
END
```

Improving and Changing the Dungeon Program

Right now there is only one kind of chest. It can contain any of the kinds of treasure in the game. You could change the game so that there are several kinds of chests with different kinds of treasures in them.

You could make a player who had certain treasures or lots of experience points become more powerful. For example, the player could bribe monsters to go away if he had enough gold. Or the player could be unable to see certain treasure chests unless he found some magic glasses. The Dungeon game could allow the player to go to the lower level only if he had accumulated enough experience points. Here's a way to implement that last suggestion.

```
TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
KEY "Y"
IF ((5 * :LEVEL) + :EXPERIENCE) > 18 [GO.DOWN]
[PRINT [YOU MUST BE WISER TO ENTER THE LOWER DUNGEON]]
END

TO GO.DOWN
ER [ROOM1 ROOM2 ROOM3 ROOM4]
LOAD "D:LOWERDUNGEON
ROOM7
END
```

You could create more typed commands similar to I, W, and D.

You could make the game smart about what direction you are going when you go through doors.

You could introduce global data structures to keep track of objects. This way the game could know when a monster in a particular room is dead and not display it again when you return to that room.

PROGRAM LISTING

SETTING UP

```

TO START
PUTSH 1 :PLAYER
PUTSH 2 :TROLL
PUTSH 3 :KOBOLD
PUTSH 4 :CHEST
PUTSH 5 :THRUST
PUTSH 6 :STAIRS
MAKE "PHITP 0           :PHITP is hit points against player.
MAKE "MHITP 0           :MHITP is hit points against monster.
MAKE "GOLD 1
MAKE "EXPERIENCE 1
MAKE "SWORD 0
MAKE "MNSTR 0
MAKE "POTION 0
MAKE "WAND 0
MAKE "LEVEL 0
MAKE "ARMOR 0
ASK [0 1 2 3] [PU HOME HT]
TELL 0
ROOM1
END

```

MAKEROOM, THE GENERAL ROOM MAKER

```

TO MAKEROOM :LEN :HGT :DOORS          MAKEROOM assumes WHO is 0.
CS HT
ASK [1 2 3] [HT]
HOME LT 90 FD 50 RT 90
SIDE :HGT :DOORS "W 1
SIDE :LEN :DOORS "N 1
SIDE :HGT :DOORS "E 1
SIDE :LEN :DOORS "S 1
DIMENSIONS :LEN :HGT
SETUP.DEMONS
PU ST
END

TO DIMENSIONS :LEN :HGT
MAKE "ROOMLENGTH :LEN
MAKE "ROOMHEIGHT :HGT
END

TO SETUP.DEMONS
WHEN OVER 0 0 [BK 10]           For when the player hits solid part of a wall.
WHEN TOUCHING 0 1 [FIGHT]        Turtle 1 is usually a monster.
WHEN TOUCHING 0 2 [CHEST]        Turtle 2 is usually a treasure chest.
WHEN 15 [MOVEPLAYER JOY 0]
WHEN OVER 1 0 [ASK 1 [BK 5 MOVE]]
WHEN OVER 1 1 [ASK 1 [BK 5 MOVE]]
WHEN OVER 1 2 [ASK 1 [BK 5 MOVE]]
END

```

STUFF USED BY MAKEROOM TO DRAW WALLS AND DOORS

```

TO SIDE :LEN :DOORS :WALL :PEN
IF EMPTYP :DOORS [SOLID.SIDE :LEN STOP]
IF EQUALP :WALL FIRST FIRST :DOORS ▶
    [DOOR.SIDE :LEN FIRST :DOORS :PEN STOP]
SIDE :LEN BF :DOORS :WALL :PEN + 1
END

TO SOLID.SIDE :LEN
PD
FD :LEN BK :LEN
RT 90
PU
FD 5
LT 90
PD
FD :LEN
RT 90
END

TO DOOR.SIDE :LEN :DOOR :PEN
PD
DOOR.LINE :LEN :PEN
PU
BK :LEN
RT 90
FD 5
LT 90
PD
DOOR.LINE :LEN :PEN
RT 90
WHEN OVER 0 :PEN BF :DOOR
END

TO DOOR.LINE :LEN :PEN
FD ( :LEN - 20 ) / 2
SETPN :PEN
FD 20
SETPN 0
FD ( :LEN - 20 ) / 2
END

```

INDIVIDUAL ROOMS

```

TO ROOM1
PR [THERE IS A GREAT BIG TROLL HERE!]
TELL 0
SETSH 1
PD
MAKEROOM 150 80 [[W ROOM2]]
HT
PU HOME FD 30 RT 90 FD 20 ST
SETSP 20

```

```

SET.MONSTER 2 [-35 40] 90
MAKE "MNSTR 2
END

TO ROOM2
PR [THERE IS A BIG CHEST HERE]
PR [THERE IS A CUTE KOBOLD HERE]
RT 180
MAKEROOM 200 90 [[N ROOM3] [E ROOM1]]
SETH 1
HOME FD 50
SETSP 20
SET.MONSTER 3 [-20 25] 270
ASK 1 [SETSP 22]
ASK 2 [PU SETPOS [-25 30] ST]
END

TO ROOM3
MAKEROOM 20 100 [[N ROOM4] [S ROOM2]]
SETH 1
SETPOS [-40 30]
SETSP 20
END

TO ROOM4
PRINT [THERE ARE STAIRS DOWN HERE]
PRINT [THERE IS A FEROCIOUS MAN EATING ►
GIANT LOGO TURTLE HERE]
PRINT [THERE IS A GREAT BIG BRONZE BOUND CHEST HERE]
MAKEROOM 100 100 [[S ROOM3]]
SET.MONSTER 0 [-20 50] 180
ASK 2 [ST PU SETH 4 SETPOS [-30 70]]
ASK 3 [ST PU SETH 6 SETPOS [20 20]]
WHEN TOUCHING 0 3 [STAIRS]
WHEN OVER 1 0 [ASK 1 [BK 9]]
SETH 1
SETPOS [-10 30]
SETSP 20
END

```

Give monster troll shape.
This is the chest.

Give monster kobold shape.
This is the chest.

Give monster turtle shape.
This is the chest.
This is the stairs shape.

ACTIONS THAT HAPPEN IN THE DUNGEON

```

TO MOVEPLAYER :JOY
IF :JOY < 0 [STOP]
SETH 45 * :JOY
ASK 1 [MOVE]
END

TO FIGHT
SETH 5
PR [YOU SWING...]
IF EQUALP RANDOM 3 1 [MHIT]
SETH 1
PR [IT ATTACKS...]

```

```
IF EQUALP RANDOM 3 1 [PHIT :MNSTR]
ASK 1 [BK 10]
BK 15
ASK 1 [MOVE]
END

TO CHEST
PR [CREAK,]
PR [{ THE CHEST OPENS }]
IF EQUALP RANDOM 5 1 [TRAP]
IF EQUALP RANDOM 3 1 [GOLD]
IF EQUALP RANDOM 2 1 [POTION]
IF EQUALP RANDOM 3 1 [JEWEL]
IF EQUALP RANDOM 3 1 [SWORD]
IF EQUALP RANDOM 3 1 [WAND]
IF EQUALP RANDOM 2 1 [ARMOR]
ASK 2 [HT]
END

TO STAIRS
PR [DO YOU WISH TO GO DOWN?]
IF NOT KEY "Y [BK 5 STOP]
PRINT [YOU CAN ONLY GO TO THE LOWER DUNGEON]
PRINT [IF YOU CREATE IT!]
ASK 3 [HT]
END

TO KEY :K
OP EQUALP RC :K
END
```

ACTIONS THAT HAPPEN BECAUSE OF GETTING TREASURES

```
TO TRAP
PR [YOU HIT A TRAP!]
PHIT 1
END

TO GOLD
PRINT "GOLD!
ADD "GOLD 5
ADD "EXPERIENCE 1
IF :EXPERIENCE > 20 [LEVEL]
END

TO POTION
PR [A POTION!]
MAKE "POTION 1 + RANDOM 4
END
```

```

TO JEWEL
PRINT "JEWEL!!
ADD "GOLD 50
ADD "EXPERIENCE 5
IF :EXPERIENCE > 20 [LEVEL]
END

TO SWORD
PR [A FINE ELFBLADE,WORTHY OF YOUR SKILL]
ADD "SWORD RANDOM 4
END

TO WAND
MAKE "WAND 1 + RANDOM 3
END

TO ARMOR
MAKE "ARMOR 1 + RANDOM 4
END

TO LEVEL
ADD "LEVEL 1
PR (SE [YOU NOW HAVE] :LEVEL [LEVELS OF EXPERIENCE])
MAKE "EXPERIENCE 0
END

```

FIGHTING

```

TO MHIT
PRINT "CRUNCH!
ADD "MHITP 1 + RANDOM 3
ADD "MHITP :SWORD
IF :MHITP > 5 [MDEAD]
ASK 1 [SETSP SUM SPEED 1]           Monster is mad after hit, gets faster.
END

TO MDEAD
PR [YOU KILLED THE MONSTER]
ADD "EXPERIENCE 5 + :MNSTR
IF :EXPERIENCE > 20 [LEVEL]
MAKE "MHITP 0
ASK 1 [SETSH 6 TOOT 0 255 5 20 HT ]
END

TO PHIT :STRENGTH
PRINT "OUCH
ADD "PHITP 1 + RANDOM 2
ADD "PHITP :STRENGTH
IF :PHITP > 10 [PDEAD]
END

TO PDEAD
PR [THOU ART SLAIN!]

```

```
TOOT Ø 255 1 1Ø
ASK [Ø 1 2 3] [CS HT]
END
```

I FOR INVENTORY

```
TO I
TYPE "GOLD PR :GOLD
TYPE "LEVEL PR :LEVEL
TYPE "EXPERIENCE PR :EXPERIENCE
TYPE [HIT POINTS] PR :PHITP
IF :SWORD > Ø [PR "ELFBLADE"]
IF :WAND > Ø [PR "WAND"]
IF :POTION > Ø [PR "POTION"]
IF :ARMOR > Ø [PR [MAGICAL CHAINMAIL]]
PR [LEATHER JERKIN]
PR [SACK]
PR [MACE]
END
```

D FOR DRINK

```
TO D
IF EQUALP :POTION Ø [PR [YOU HAVE NO POTION TO DRINK] STOP]
PR [YOU DRINK THE LIQUID]
IF EQUALP :POTION 1 [HPOTION MAKE "POTION Ø"]
IF EQUALP :POTION 2 [SPOTION MAKE "POTION Ø"]
IF EQUALP :POTION 3 [HPOTION MAKE "POTION Ø"]
IF EQUALP :POTION 4 [PPOTION]
END
```

POTIONS AND WHAT THEY DO

```
TO PPOTION
PR [YUCK! YOUR STOMACH WILL NEVER FORGIVE YOU ►
     FOR PUTTING THIS FOUL EXCREMENT IN IT]
PHIT Ø
END
```

```
TO SPOTION
ADD "STR 1
PR [YOU QUAFF THE FLUID AND YOUR MUSCLES BULGE!]
END
```

```
TO HPOTION
ADD "H -3
PRINT [YOU FEEL HEALED, TIS A FINE ELIXIR YOU HAVE QUAFFED!]
END
```

W FOR WAVE WAND

```

TO W
IF EQUALP :WAND 0 [STOP]
IF EQUALP :WAND 1 [PR [A BOLT OF LIGHTNING    ▶
  STREAKS FROM YOUR HAND...] MHIT]
IF EQUALP :WAND 2 [PR [YOU ARE TELEPORTED TO A NEW    ▶
  LOCATION] RUN FPUT WORD "ROOM 1 + RANDOM 4 []"]
IF EQUALP :WAND 3 [PR [THE WAND EXPLODES IN YOUR HAND!]    ▶
  PHIT 2]
END

```

SETTING UP PARTICULAR MONSTERS

```

TO SET.MONSTER :SHAPE :POS :HEADING
ASK 1 [SETSP 10 PU SETSH :SHAPE    ▶
  SETPOS :POS SETH :HEADING ST]
END

```

AIMING THE MONSTER TOWARD THE PLAYER

This procedure points the monster toward the quadrant the player is in.

```

TO MOVE
MAKE "XPLAYER ASK 0 [XCOR]
MAKE "YPLAYER ASK 0 [YCOR]
TELL 1
SETH 45
IF :YPLAYER < YCOR [SETH 135]
IF :XPLAYER < XCOR [IF :YPLAYER < YCOR    ▶
  [SETH 225] [SETH 315]]
TELL 0
END

```

UTILITIES

```

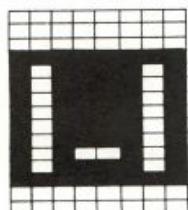
TO ADD :VAR :NUM
MAKE :VAR :NUM + THING :VAR
END

TO SAVESH
MAKE "PLAYER GETSH 1
MAKE "TROLL GETSH 2
MAKE "KOBOLD GETSH 3
MAKE "CHEST GETSH 4
MAKE "THRUST GETSH 5
MAKE "STAIRS GETSH 6
END

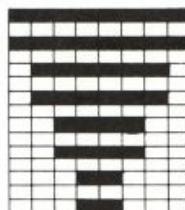
```

```
MAKE "THRUST [0 0 12 13 13 37 255 44  
12 12 12 12 60 36 39 101]  
MAKE "TROLL [12 11 28 63 60 120 126 122  
120 120 56 48 16 16 16 24]  
MAKE "PLAYER [0 64 88 88 88 72 254 90  
26 26 24 56 40 44 36 100]  
MAKE "KOBOLD [128 128 152 184 184 136 152 248  
152 24 24 8 8 8 8 24]  
MAKE "STAIRS [255 0 255 0 126 0 126 0 60 0 60 0 24 0 24 0]  
MAKE "CHEST [0 0 0 255 189 189 189 189  
189 189 165 189 255 0 0 0]
```

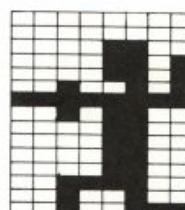
SHAPES



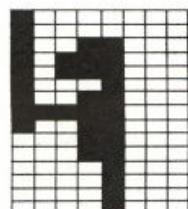
: CHEST
slot 4



: STAIRS
slot 6



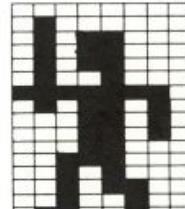
: THRUST
slot 5



: KOBOLD
slot 3



: TROLL
slot 2



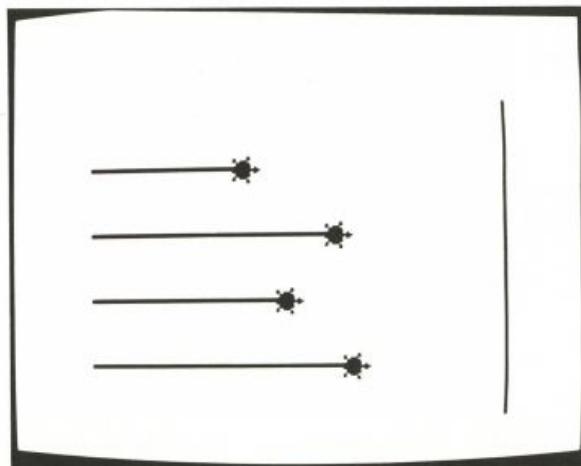
: PLAYER
slot 1

4

Turtle Geometry

Turtle Race

RACE shows four turtles racing from the left side of the screen to the right side. The winning turtle changes color. The background also changes color, to emphasize that someone has won the race. Here is a race in progress.



RACE.FROM does the real work by running first SETUP.RACE and then RUN.RACE. In other words, the program is divided into a setup part and an action part. The job of SETUP.RACE is to draw the racecourse and to assign colors and positions to the turtles. The job of RUN.RACE is to run the race.

```
TO RACE
RACE.FROM -120 120
END
```

The motion of the turtles is controlled by repeated use of the FORWARD command, not by using the dynamic (speed) ability of the turtles.

RACE is the top-level procedure. It knows where the left and right edges of the screen are and gives that information to RACE.FROM.

By Brian Harvey, with modifications by Jim Davis.

```
TO RACE.FROM :START :FINISH
SETUP.RACE
RUN.RACE
END
```

`START` and `FINISH` contain the *x* coordinates of the starting and ending positions. This information is used to set the turtles up at the start of the race and to find the winner.

Setting Up

`SETUP.RACE` has two tasks to do: set up the racecourse and prepare the four turtles as racers. It has one subprocedure to do each task. It hides the turtles when it starts, to avoid clutter during the setup.

```
TO SETUP.RACE
TELL [0 1 2 3]
HT
DRAW.RACETRACK
SETUP.RACERS
END
```

`DRAW.RACETRACK` is the procedure in charge of setting up the racecourse. This involves cleaning up the screen and setting its color, and then drawing the finish line.

```
TO DRAW.RACETRACK
SETBG 86
FS
CS
ASK 0 [DRAW.FINISHLINE]
END
```

`DRAW.FINISHLINE` draws a vertical line near the right edge of the screen.

```
TO DRAW.FINISHLINE
PU
SETPOS LIST :FINISH 80
SETPN 1
SETPC 1 105
PD
BK 190
PU
END
```

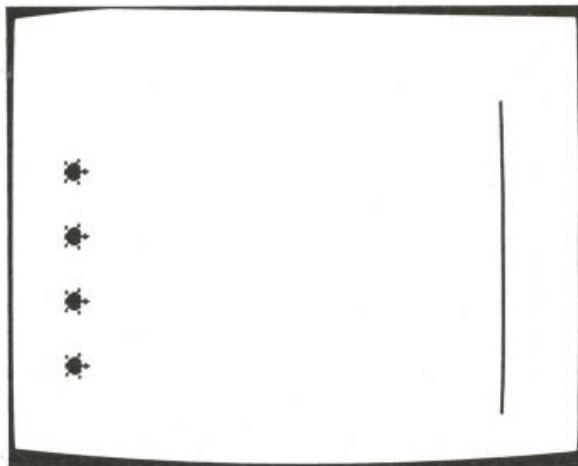
`SETUP.RACERS` positions the four turtles at the starting point of the race, near the left edge of the screen. Some things are the same for all the turtles, like the `RT 90` to point them toward the finish line. But two things are different for each turtle: the vertical position and the color. `SETUP.RACERS` uses the primitive command `EACH` to tell Logo to set these two properties for each turtle separately. In the instructions given as inputs to `EACH`, the particular value used for each turtle depends on the turtle number, represented by the primitive operation `WHO`. For example, the

TURTLE GEOMETRY

SETC instruction will give turtle 0 color 11 ($11+16*0$), turtle 1 color 27 ($11+16*1$), and so on.

```
TO SETUP.RACERS
PU
EACH [SETPOS LIST :START WHO*40-80]    Vertical position different
RT 90                                     for each turtle.
EACH [SETC 11+16*WHO]                      Each turtle has different hue
ST                                         but same intensity.
SETPN 0
SETPC 0 90
PD
END
```

The result of running SETUP.RACE is shown here.



Running the Race

The race itself is handled by RUN.RACE, which moves the turtles one at a time until there is a winner. The command EACH is used to accomplish this one-at-a-time motion.

After each turtle moves, the operation WONP checks whether or not the turtle that moved has reached :FINISH and thus has won the race. If so, the procedure SHOW.WINNER is called to congratulate the winning turtle by changing its color. If there is no winner, the race continues.

```
TO RUN.RACE
EACH [MOVE1 IF WONP [SHOW.WINNER STOP]]
RUN.RACE
END
```

MOVE1 is invoked for each turtle in turn. It moves the turtle a small random amount. The distances are small, so repeating this procedure over

and over will give a fairly smooth effect. The distances are random so that the race is different each time the program is run.

```
TO MOVE1
FD 6+RANDOM 20
END
```

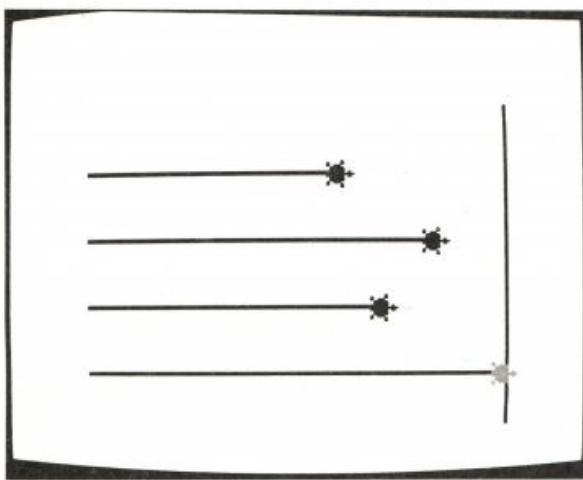
WONP checks the current turtle's position to see if it's past the finish line. If so, it outputs TRUE; otherwise, FALSE.

```
TO WONP
OP XCOR > :FINISH
END
```

SHOW.WINNER just changes the color of the winning turtle and the background color, to indicate that the race is over.

```
TO SHOW.WINNER
SETC 7
SETBG 84
END
```

This is the end of a race.



SUGGESTIONS

This race is unfair; lower-numbered turtles have a greater chance of winning, because they move first. Here's one way to fix it: judge the winner only when each turtle has had a chance to move. Then the operation WINNER would output a list of all winners. This is more egalitarian. Each could be bestowed an award.

How about a musical fanfare at the end?

On the other hand, if you like unfair races, perhaps the winning turtle should eat the other turtles, plunder their homelands, and so forth.

PROGRAM LISTING

```

TO RACE
RACE.FROM -120 120
END

TO RACE.FROM :START :FINISH
SETUP.RACE
RUN.RACE
END

TO SETUP.RACE
TELL [0 1 2 3]
HT
DRAW.RACETRACK
SETUP.RACERS
END

TO DRAW.RACETRACK
SETBG 86
FS
CS
ASK 0 [DRAW.FINISHLINE]
END

TO DRAW.FINISHLINE
PU
SETPOS LIST :FINISH 80
SETPN 1
SETPC 1 105
PD
BK 190
PU
END

TO SETUP.RACERS
PU
EACH [SETPOS LIST :START WHO*40-80]
RT 90
EACH [SETC 11+16*WHO]
ST
SETPN 0
SETPC 0 90
PD
END

TO RUN.RACE
EACH [MOVE1 IF WONP [SHOW.WINNER ▶
STOP]]
RUN.RACE
END

TO MOVE1
FD 6+RANDOM 20
END

TO WONP
OP XCOR > :FINISH
END

TO SHOW.WINNER
SETC 7
SETBG 84

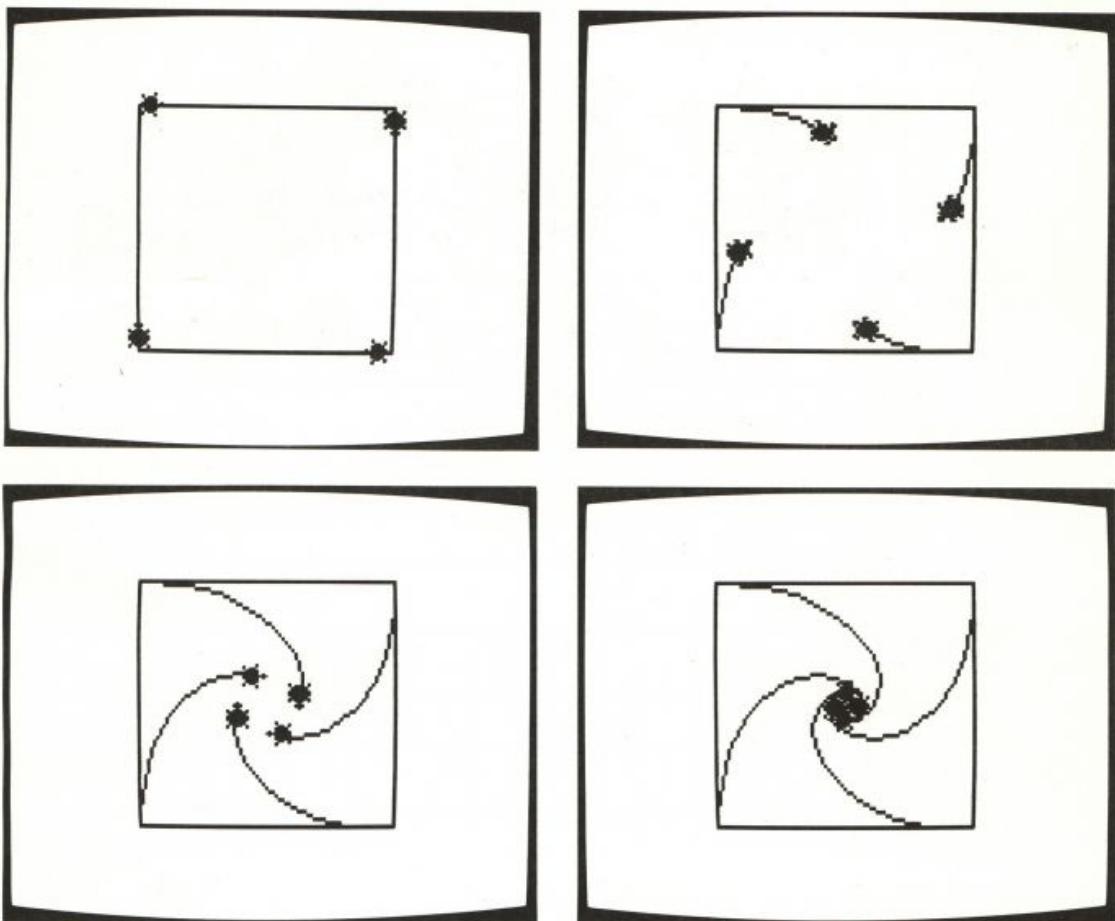
```

Four-Corner Problem

Here is a famous math problem: There are four ants, each at one corner of a square. Each ant faces the next one. They all start walking at the same time. As they walk, each ant turns so that it continues to face the same ant it was facing at the beginning. How far do the ants walk before they all meet at the center of the square?

This Logo program doesn't tell you how far they walk, but it does draw a picture to act out the problem. The only difference is that in this version of the problem we use turtles instead of ants.

By Brian Harvey.



PROGRAM LISTING

This project uses the TOWARDS procedure, which is shown later on in this chapter.

```

TO FOUR
  TELL [0 1 2 3]
  CT CS ST SETSH 0 PU FS
  ASK 0 [SETPOS [-80 -80]]
  ASK 1 [SETPOS [-80 80]]
  ASK 2 [SETPOS [80 80]]
  ASK 3 [SETPOS [80 -80]]
  PD
  ASK 0 [SETH 0 REPEAT 4 [FD 160 RT 90]]
  WAIT 60 SS
  PR [EACH TURTLE KEEPS FACING THE NEXT]
  PR [ONE AS THEY ALL MOVE FORWARD.]
  WAIT 300
  FS
  FOUR.LOOP
END

TO FOUR.LOOP
  EACH [SETH TOWARDS ASK REMAINDER (WHO+1) 4
    [POS]] FD 10
  IF COND TOUCHING 0 1 [STOP]
  FOUR.LOOP
END

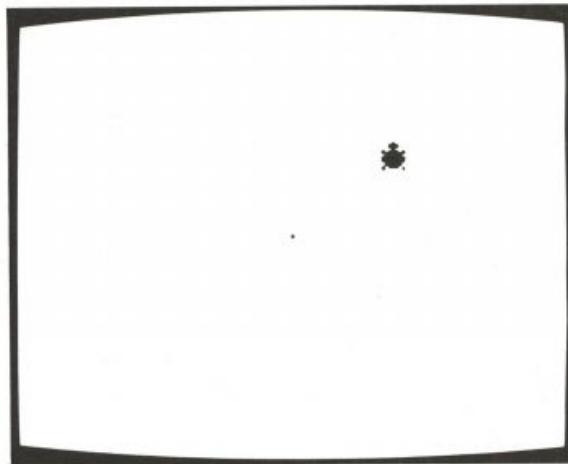
```

Towards and Arctan

TOWARDS is an operation that tells you how to turn the turtle to get it pointing toward a particular position. It takes one input, which is the position toward which you want to turn the turtle (in the form of a list of two coordinates). It outputs the heading to which the turtle should be turned in order to be facing from its current position to the input position. Here is an example. Start out with a clear screen with one dot in the middle.

```
CS  
PD  
FD 0  
PU  
SETPOS [63 27]
```

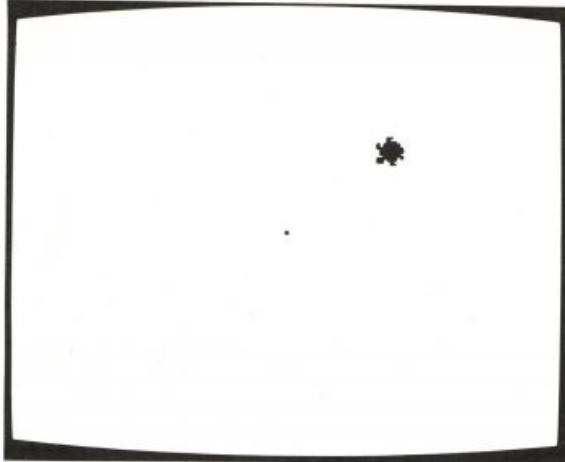
At this point, the turtle is facing north.



```
SETH TOWARDS [0 0]
```

The turtle is now facing the dot we drew at the center of the screen.

This project is based on "Three Computer Mathematics Projects" by Hal Abelson, MIT Logo Laboratory Working Paper No. 16, June 20, 1974; write-up by Brian Harvey.



TOWARDS is defined in terms of the second tool in this package, the ARCTAN procedure. ARCTAN takes a number as input and gives as output the arctangent (in degrees) of that number. The procedure uses an approximation that is good to within about one degree, close enough for graphics!

For those who have studied trigonometry: the TOWARDS procedure computes the differences between the x and y coordinates of the input position and those of the turtle's position, then takes the arctangent of $\Delta y/\Delta x$. The output from ARCTAN is the correct heading, except that attention must be paid to the positive or negative direction of the two differences. (If you haven't studied trig, don't worry about it. You can use TOWARDS without understanding its inner workings.)

PROGRAM LISTING

```

TO TOWARDS :POS
OP TOWARDS1 (FIRST :POS)-XCOR (LAST ▶
:POS)-YCOR
END

TO TOWARDS1 :DX :DY
OP TOWARDS3 :DX :DY TOWARDS2 ABS :DX ▶
ABS :DY
END

TO TOWARDS2 :DX :DY
IF :DX=0 [OP 0]
IF :DY=0 [OP 90]
OP ARCTAN (:DX/:DY)
END

TO TOWARDS3 :DX :DY :ANG
IF :DY<0 [MAKE "ANG 180-:ANG]
IF :DX<0 [MAKE "ANG 360-:ANG]
OP :ANG
END

TO ARCTAN :X
OP 57.3*ARCTAN.RAD :X
END

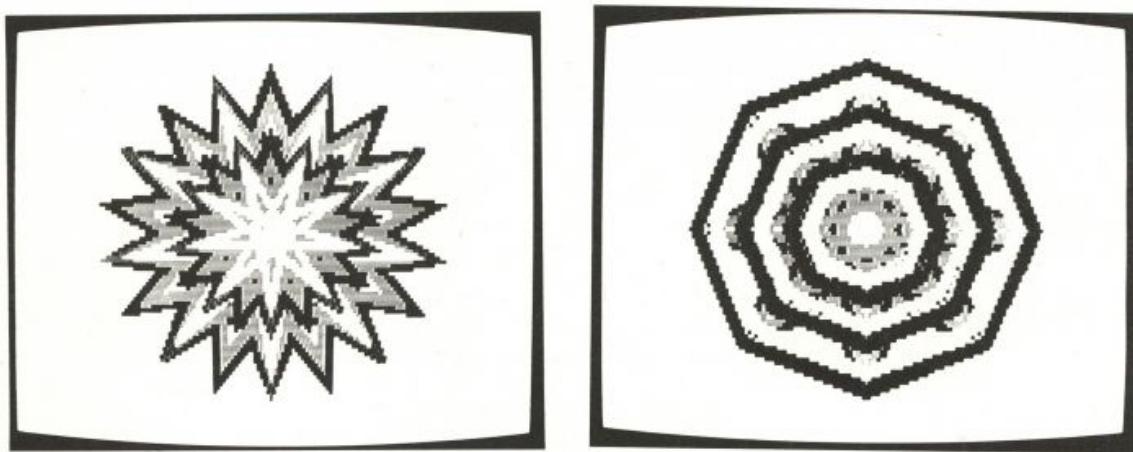
TO ARCTAN.RAD :X
IF :X>1 [OP 1.571-ARCTAN.RAD (1/:X)]
OP :X/(1+0.28*:X*:X)
END

TO ABS :X
OP IF :X<0 [-:X] [:X]
END

```

Gongram: Making Complex Polygon Designs

GONGRAM makes designs like the ones shown below.



To make the first design, type:

```
GONGRAM 14 110 140 160 28 23 0
```

To make the second one, type:

```
GONGRAM 10 110 135 45 23 45 77
```

It takes a long time to make a gongram design since the turtle must draw many lines.

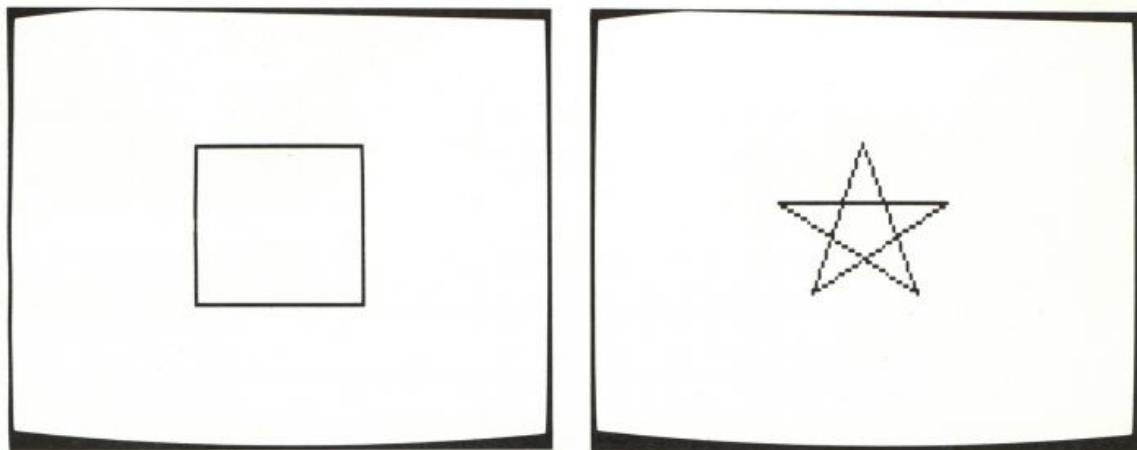
GONGRAM uses a variation of POLY, a procedure that makes a turtle draw polygons of different sizes and shapes. (See *Atari Logo Introduction to Programming Through Turtle Graphics*, p. 138, for a discussion of this procedure.)

```
TO POLY :SIDE :ANGLE
POLY1 :SIDE :ANGLE HEADING
END

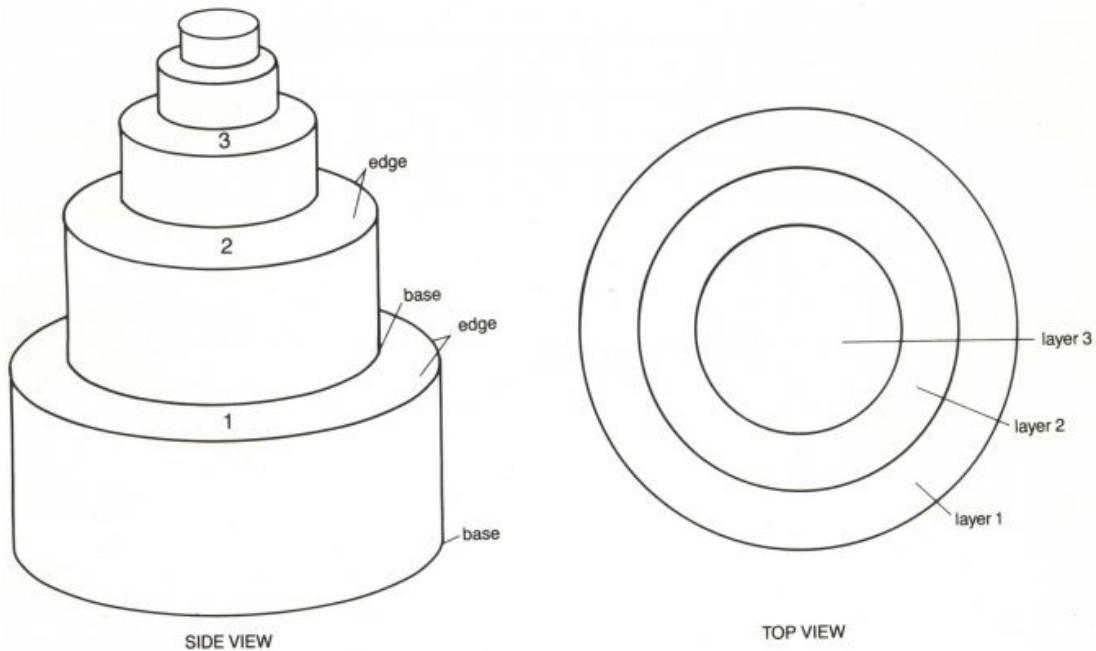
TO POLY1 :SIDE :ANGLE :START
FD :SIDE
RT :ANGLE
IF HEADING = :START [STOP]
POLY1 :SIDE :ANGLE :START
END
```

By Erric Solomon.

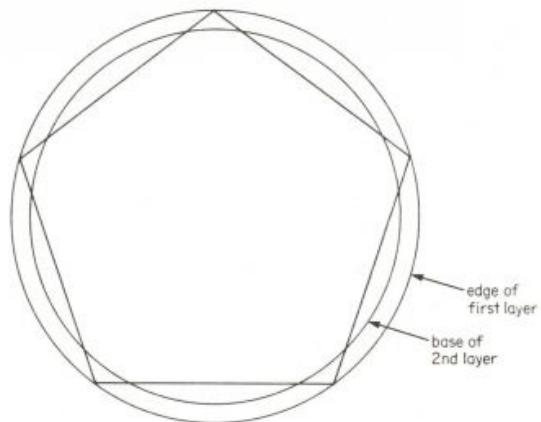
For example, `POLY 50 90` draws a square of side length 50; `POLY 50 144` draws a five-pointed star of side length 50.



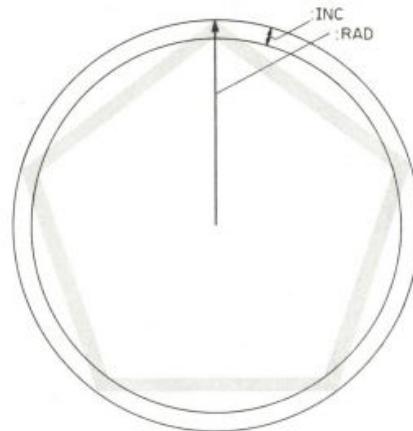
To help in understanding how a gongram design is made, imagine that you are directly above a layer cake.



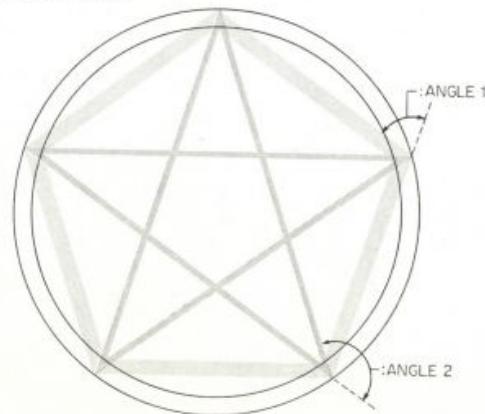
Each layer is slightly smaller than the one below it. Each layer of the cake is transparent, except when we draw on it. At the edge of the bottom layer, which we'll call the first layer, a pentagon is drawn.



Also drawn on the first layer is a smaller pentagon. It digs into the circle formed by the base of the layer above. (The second layer's base rests on the surface of layer 1.) The area between the two is filled in.



The result is a thick POLY shape. In a similar fashion we draw two five-pointed stars and fill the area between them in another color, sharing vertices with the pentagon.



Notice that part of the pentagon is covered by the star. Now we move to the next layer. On the second layer we draw two new pentagons, one at the edge of the second layer, the next one inscribed into the circle formed by the base of the third layer. And as before, we fill in the area between them in a third color.



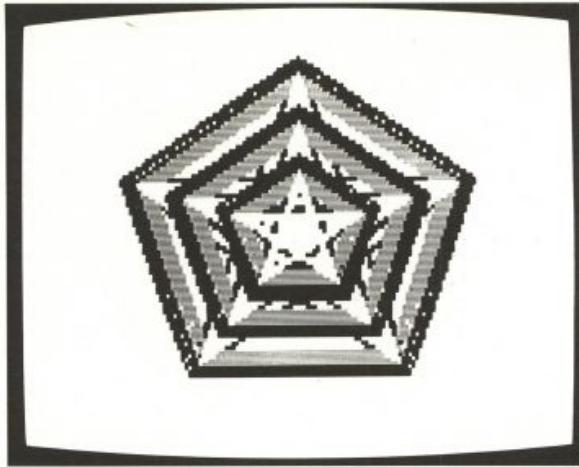
Notice that the view of parts of the star has been obstructed. Two new stars are inscribed in a similar manner, but instead of filling the area between them with a color, we rub an eraser over the area between the stars.



We skip over layer three, and at layer four we pretend that it is the bottom layer and repeat the process.

Of course, we didn't have to use a pentagon or a star. We could have chosen two other POLY-generated shapes. We could choose other colors.

Here is the completed design.



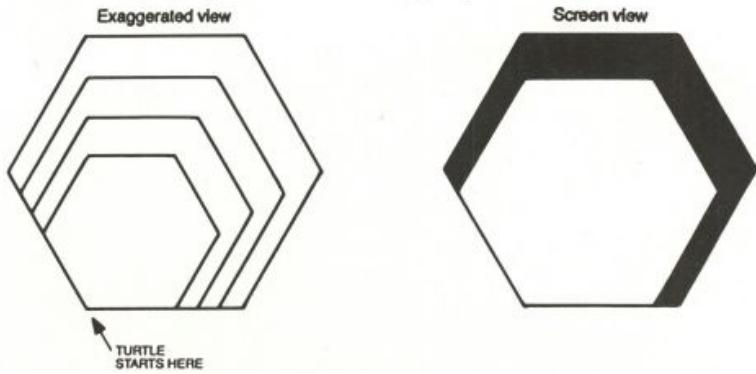
GONGRAM 10 110 72 144 30 62 102

Making a Filled-in POLY

Why not just use POLY several times, with a slightly different first input each time? That would produce several polygons of the same shape but slightly different sizes, one inside the other. For example, we could try this procedure:

```
TO THICK.POLY :SIDE :ANGLE :THICKNESS
IF :THICKNESS=0 [STOP]
POLY :SIDE :ANGLE
THICK.POLY :SIDE - 1 :ANGLE :THICKNESS - 1
END
```

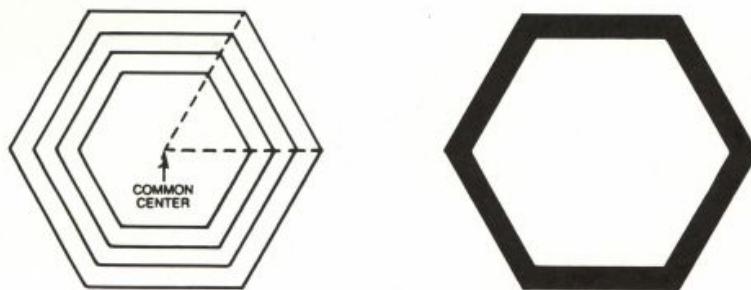
The trouble is that this procedure doesn't equally thicken all the sides.



We could try to solve this problem by moving the turtle in toward the center of the polygon a little before drawing the next polygon. But it's a bit

complicated to figure out exactly how far to move the turtle, and in what direction, between POLYs.

The fundamental problem is that the successive POLYs are "anchored" to one vertex of the polygon, the one where the turtle starts. That vertex is at the same place on the screen for all the POLYs we draw. It would be better if we could anchor the polygons to a common *center* rather than to a common *vertex*.



This is how I approached the problem. First I noted that all the vertices of a POLY design are equidistant from the center of the design. Therefore, all the vertices of the POLY are points on the same circle, and each side of the design is a chord of the circle. Since :SIDE remains constant, each chord is the same length. It is possible, then, to inscribe a POLY into a circle by specifying the radius of the circle and the angle of the POLY design. If you inscribe a series of POLYs into concentric circles, then you get a thick or "filled-in" POLY.

I then wrote a different POLY and called it POLYGON. It places the turtle at points around a circle using the center of the screen as the center of the circle. As the turtle moves from point to point, the pen traces a line along each chord.

```

TO POLYGON :RADIUS :ANGLE
PU
SETPOS LIST :RADIUS * SIN HEADING :RADIUS * COS HEADING
PD
POLYGON1 :RADIUS :ANGLE HEADING
END

TO POLYGON1 :RADIUS :ANGLE :START
RT :ANGLE
SETPOS LIST :RADIUS * SIN HEADING :RADIUS * COS HEADING
IF HEADING = :START [STOP]
POLYGON1 :RADIUS :ANGLE :START
END

```

Another way to look at this procedure is to think of the turtle sitting in the middle of a circle. Each time the turtle turns, it points to a new vertex on the circle. If the turtle makes 90-degree turns each time, it will point to four distinct vertices. If the turtle turns 144 degrees each time, it will point

TURTLE GEOMETRY



to five distinct vertices. These vertices could be connected in several different ways. The order in which the turtle points to them is the order in which they will be connected.

In the earlier version of POLY, the turtle always faces in the direction of the next side to be drawn. After it draws each side, the RIGHT turn points the turtle so that the next FORWARD will draw the next side.

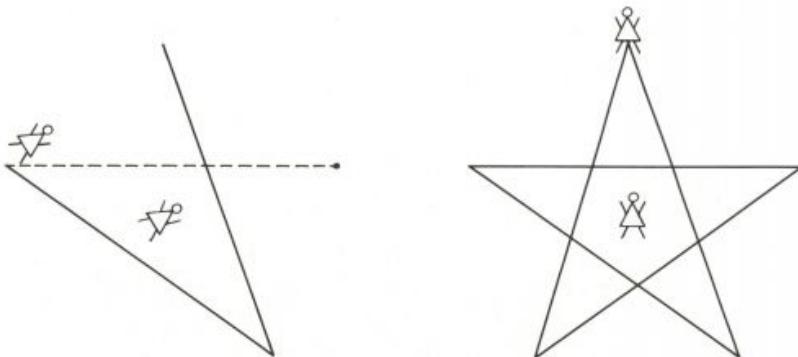
In the new version, the turtle does *not* face in the direction of the next side. That's why the sides are drawn using SETPOS instead of FORWARD; we tell Logo where the next vertex is, instead of telling it the distance and direction. But the turtle's heading is still important in this version of POLY. It is always the heading that a turtle *in the center of the polygon* would face in order to point to the next vertex. To see how this works, watch a version of POLYGON where two turtles are visible. Turtle 0 will actually draw the polygon; turtle 1 will sit in the center of the screen, but will keep turning to retain the same heading as turtle 0.

```

TO VIEWPOLYGON :RADIUS :ANGLE
  TELL [Ø 1] ST
  PU
  ASK Ø [SETPOS LIST :RADIUS*SIN HEADING :RADIUS*COS HEADING]
  PD
  VIEWPOLYGON1 :RADIUS :ANGLE HEADING
  END

TO VIEWPOLYGON1 :RADIUS :ANGLE :START
  RT :ANGLE
  ASK Ø [SETPOS LIST :RADIUS*SIN HEADING :RADIUS*COS HEADING]
  IF HEADING = :START [STOP]
  VIEWPOLYGON1 :RADIUS :ANGLE :START
  END

```



Now that we've made a POLYGON that inscribes the design into a circle, a POLY.FILL is possible.

```

TO POLY.FILL :RADIUS :ANGLE
IF :RADIUS = 0 [STOP]
POLYGON :RADIUS :ANGLE
POLY.FILL :RADIUS - 1 :ANGLE
END

```

But we want to make a POLY.FILL that will make a POLYGON of any thickness.

```

TO POLY.FILL :HI :LO :ANGLE
POLYGON :HI :ANGLE
IF NOT :HI > :LO [STOP]
POLY.FILL :HI - 1 :LO :ANGLE
END

```

And now GONGRAM:

```

TO GONGRAM :INC :RAD :ANGLE1 :ANGLE2 :PC1 :PC2 :PC3
IF :RAD < 31 [STOP]
SETPC 1 :PC1
TELL Ø SETPN 1
POLY.FILL :RAD :RAD+- :INC :ANGLE1
SETPC 2 :PC2 SETPN 2
POLY.FILL :RAD :RAD - :INC :ANGLE2
SETPC Ø :PC3 SETPN Ø
POLY.FILL :RAD - :INC :RAD - 2 * :INC :ANGLE1
ERASE.POLY.FILL :RAD - :INC :RAD - 2 * :INC :ANGLE2
GONGRAM :INC :RAD - 3 * :INC :ANGLE1 :ANGLE2 :PC1 :PC2 :PC3
END

```

GONGRAM takes seven inputs.

- :INC The distance between the edge of a layer and the base of the layer on top of it.
- :RAD The radius of the largest layer.
- :ANGLE1 An angle that dictates the shape of one of the polygons inscribed on the cake. In our example it is 72 (the pentagon).
- :ANGLE2 An angle that dictates the shape of one of the polygons inscribed on the cake. In our example it is 144 for the star.
- :PC1 The pen color for pen 1. In this example it is 30 for red.
- :PC2 The pen color for pen 2. In this example it is 62 for blue.
- :PC3 The pen color for pen 0. In this example it is 102 for green.

ERASE.POLY.FILL is the only procedure I haven't mentioned. It is just like POLY.FILL except that it calls ERASE.POLY. ERASE.POLY is just like POLYGON except that it puts the pen into eraser, or PE, mode.

Here are some nice examples of GONGRAM.

```

GONGRAM 10 110 135 45 23 45 77
GONGRAM 15 110 90 135 23 45 77
GONGRAM 14 110 140 160 28 23 0
GONGRAM 12 110 90 120 45 60 23
GONGRAM 5 110 40 -1000 23 0 45
GONGRAM 15 110 60 120 60 45 23
GONGRAM 15 110 120 60 45 23 77

```

Note: Some of these take a very long time to draw.

PROGRAM LISTING

```

TO GONGRAM :INC :RAD :ANGLE1 :ANGLE2 ▶
    :PC1 :PC2 :PC3
    IF :RAD < 31 [STOP]
    SETPC 1 :PC1
    TELL Ø SETPN 1
    POLY.FILL :RAD :RAD - :INC :ANGLE1
    SETPC 2 :PC2 SETPN 2
    POLY.FILL :RAD :RAD - :INC :ANGLE2
    SETPC Ø :PC3 SETPN Ø
    POLY.FILL :RAD - :INC :RAD - 2 * :INC ▶
        :ANGLE1
    ERASE.POLY.FILL :RAD - :INC :RAD - 2 * ▶
        :INC :ANGLE2
    GONGRAM :INC :RAD - 3 * :INC :ANGLE1 ▶
        :ANGLE2 :PC1 :PC2 :PC3
    END

    TO POLY.FILL :HI :LO :ANGLE
    POLYGON :HI :ANGLE
    IF NOT :HI > :LO [STOP]
    POLY.FILL :HI - 1 :LO :ANGLE
    END

    TO POLYGON :RADIUS :ANGLE
    PU
    SETPOS LIST :RADIUS * SIN HEADING ▶
        :RADIUS * COS HEADING
    PD
    POLYGON1 :RADIUS :ANGLE HEADING
    END

    TO POLYGON1 :RADIUS :ANGLE :START
    RT :ANGLE
    SETPOS LIST :RADIUS * SIN HEADING ▶
        :RADIUS * COS HEADING
    IF HEADING = :START [STOP]
    POLYGON1 :RADIUS :ANGLE :START
    END

    TO ERASE.POLY.FILL :HI :LO :ANGLE
    IF NOT :HI > :LO [STOP]
    ERASE.POLY :LO :ANGLE
    ERASE.POLY.FILL :HI :LO + 1 :ANGLE
    END

    TO ERASE.POLY :RADIUS :ANGLE
    PU SETPOS LIST :RADIUS * SIN HEADING ▶
        :RADIUS * COS HEADING
    PE
    RT :ANGLE
    POLYGON :RADIUS :ANGLE
    END

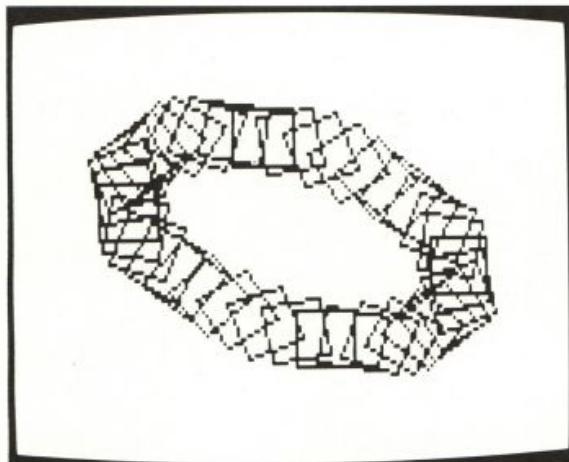
```

Polycirc

POLYCIRC makes designs by drawing polygons or lines around the circumference of an imaginary circle. As the turtle walks around the circumference, its heading changes as well as its position. Thus the polygons are drawn at different angles.

POLYCIRC 35 90 10 80 1

By Erric Solomon.



POLYCIRC takes five inputs: :SIZE, :ANGLE, :INC, :RAD, :TIMES. POLYCIRC calls two procedures: POLY and NEXTPEN.

```
TO POLYCIRC :SIZE :ANGLE :INC :RAD :TIMES
  RT :INC / :TIMES
  PU SETPOS LIST (:RAD * COS :TIMES * HEADING)
    (:RAD * SIN :TIMES * HEADING)
  PD
  NEXTPEN
  POLY :SIZE :ANGLE
  IF HEADING = @ [STOP]
  POLYCIRC :SIZE :ANGLE :INC :RAD :TIMES
  END
```

POLY draws polygons.

```
TO POLY :SIZE :ANGLE
  POLY1 :SIZE :ANGLE HEADING
  END

  TO POLY1 :SIZE :ANGLE :HEAD
    RT :ANGLE
    FD :SIZE
    IF HEADING = :HEAD [STOP]
    POLY1 :SIZE :ANGLE :HEAD
    END
```

NEXTPEN changes the pen each time it is called.

```
TO NEXTPEN
  IF PN = 2 [SETPN @] [SETPN PN + 1]
  END
```

TURTLE GEOMETRY

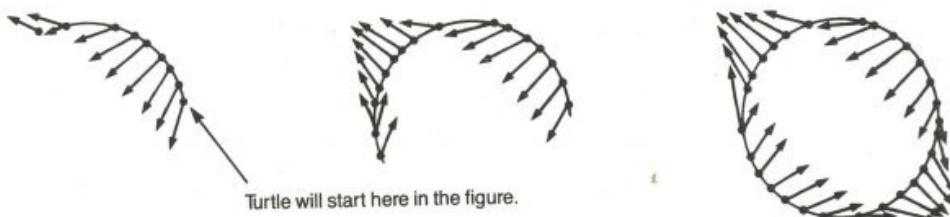
Try:

POLYCIRC 35 180 10 80 1



Notice that this POLYCIRC is similar to the one in the first example, except that it draws a spoke instead of a square.

The following diagram might help you in understanding POLYCIRC.



Two circles are implicit in the figure. One is stationary and has its center in the center of the screen. The other can be thought of as a rolling wheel whose center is always found on the circumference of the stationary circle. This wheel has just one spoke. As the wheel rolls along the circumference of the central circle, this spoke turns. In the figure, the wheel turns one full revolution for every trip around the circle. At the same time, a trace of the spoke is left every 10 degrees around the circle.

To help you understand how the program works, you can make the central circle visible and then watch the spokes being drawn. First draw the central circle this way:

POLYCIRC 2 180 2 50 1

Then type the following to see the spokes being drawn around it.

POLYCIRC 50 180 10 50 1

We have used spokes in this example for their visual clarity, to help you understand how the program places polygons around a circle. A spoke is simply a POLY using an angle of 180 degrees. To draw other kinds of polygons, use a different angle. For example, the square POLYCIRC at the beginning of this section was drawn using an angle of 90 degrees.

Several inputs or parameters of the design can be varied.

:SIZE, the first input, is the radius of the rolling wheel and thus the length of each side of the polygon.

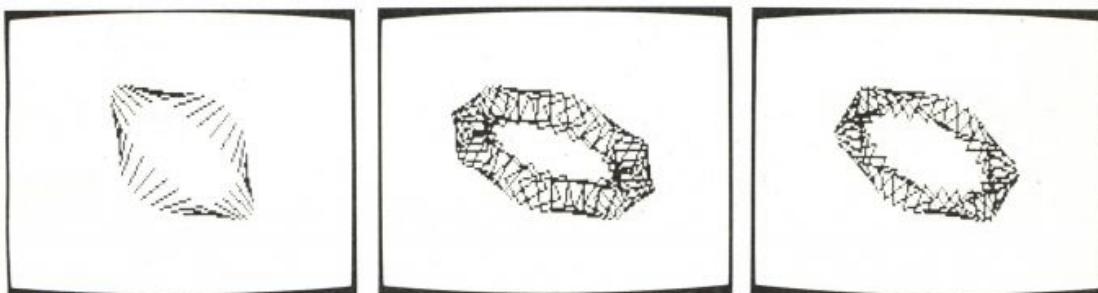


POLYCIRC 30 180 10 60 1

POLYCIRC 10 180 10 60 1

POLYCIRC 60 180 10 60 1

:ANGLE, the second input, is the angle that determines the shape of the polygon. If :ANGLE is 180, just a spoke is drawn.



POLYCIRC 30 180 10 60 1

POLYCIRC 30 90 10 60 1

POLYCIRC 30 120 10 60 1

:INC, the third input, is inversely related to the density of the polygons.



POLYCIRC 30 180 10 60 1

POLYCIRC 30 180 20 60 1

POLYCIRC 30 180 5 60 1

:RAD, the fourth input, is the radius of the center circle.

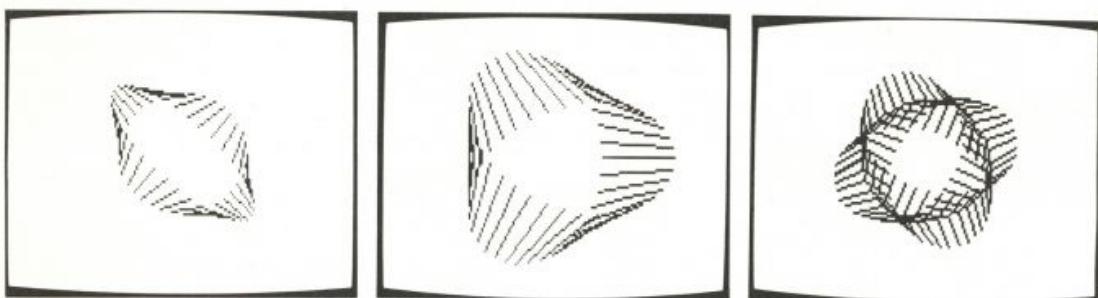


POLYCIRC 30 180 10 60 1

POLYCIRC 30 180 10 15 1

POLYCIRC 30 180 10 30 1

:TIMES, the fifth input, is the number of rotations of the wheel around its own center for each revolution it makes around the central circle. (For example, for each revolution of the earth around the sun, it makes 365 rotations, more or less.)*

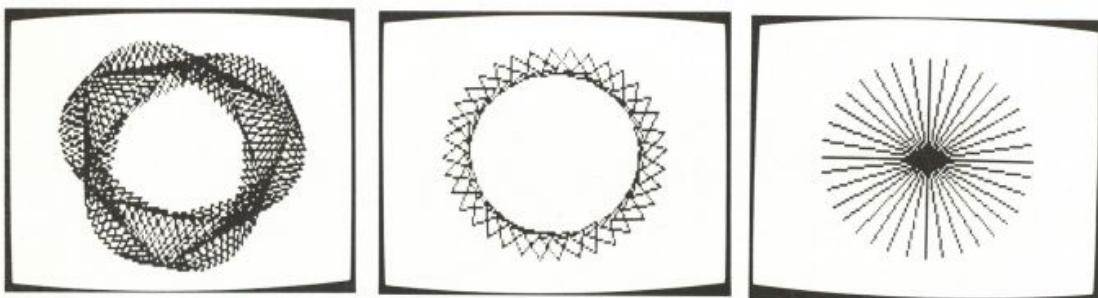


POLYCIRC 30 180 10 60 1

POLYCIRC 30 180 10 60 2

POLYCIRC 30 180 10 60 3

Other inputs for POLYCIRC that you might try are:



POLYCIRC 30 120 5 90 2

POLYCIRC 30 120 10 80 -1

POLYCIRC 100 180 10 0 1

*This input can never be zero.

PROGRAM LISTING

```

TO POLYCIRC :SIZE :ANGLE :INC :RAD ►      TO POLY :SIZE :ANGLE
   :TIMES                                     POLY1 :SIZE :ANGLE HEADING
RT :INC / :TIMES                           END
PU SETPOS LIST (:RAD * COS :TIMES * ►      TO POLY1 :SIZE :ANGLE :HEAD
   HEADING) (:RAD * SIN :TIMES * ►          RT :ANGLE
   HEADING)                                     FD :SIZE
PD                                         IF HEADING = :HEAD [STOP]
NEXTPEN                                     POLY1 :SIZE :ANGLE :HEAD
POLY :SIZE :ANGLE                           END
IF HEADING = @ [STOP]
POLYCIRC :SIZE :ANGLE :INC :RAD :TIMES
END                                         TO NEXTPEN
                                         IF PN = 2 [SETPN @] [SETPN PN + 1]
                                         END

```

Animating Line Drawings

In Atari Logo you can change the color of lines already drawn on the screen. This feature can be used to animate drawings. I will give three examples. In each of them all three pens are used to make a drawing. Then the drawings are transformed from static to moving pictures. This is done by changing pen colors.

The first example is of spinning spokes. The other two examples show how color changes affect designs made by GONGRAM and POLYCIRC, programs that are described in other sections of this book.

Spinning Spokes

STAR draws 36 lines as if they were spokes of a wheel. As it draws lines, the turtle switches from pen 0, to pen 1, to pen 2, to pen 0, and so on until all the spokes are drawn. STAR puts the background's color in pens 1 and 2 so that their lines are not visible to the user while the design is being made. Lines drawn by pen 0 are visible; thus every third spoke is displayed on the screen.

After it draws each spoke, STAR calls NEXTPEN, which changes the pen.

TURTLE GEOMETRY

```

TO STAR
SETPC 0 50
SETPC 1 BG
SETPC 2 BG
REPEAT 36 [FD 100 BK 100 RT 10 NEXTPEN]
END

TO NEXTPEN
IF PN = 2 [SETPN 0] [SETPN PN + 1]
END

```

Now try:

```
CYCLE 200 5
```

CYCLE animates the picture; it displays spoke after spoke by changing the pen colors. The first input to CYCLE is the number of times the animation will be repeated. The second input controls the delay (in sixtieths of a second) between shifts of pen colors. You can think of this time delay as the length of time between frames in the animation.

```

TO CYCLE :TIMES :DELAY
REPEAT :TIMES [CYC PC 1 :DELAY]
END

TO CYC :PC :DELAY
SETPC 1 PC 0
SETPC 0 PC 2
SETPC 2 :PC
WAIT :DELAY
END

```

The basic idea in this example is that two pens are always “hidden,” but CYCLE keeps changing which two are hidden. Lines drawn by pen 1 change to the color previously assumed by lines drawn by pen 0. Pen 0’s lines change to the color in pen 2. Lines drawn by pen 2 change to the color that used to be in pen 1. This color is given to CYC as an input.

Color Change with Gongram and Polycirc

CYCLE can work its magic in other situations as well. Let’s try it with GONGRAM. In this example, all three pens have visible colors. (The last three inputs to GONGRAM are the colors for the pens.) Now run CYCLE and watch the result of this color shift.

```
GONGRAM 15 120 72 144 43 77 22
CYCLE 200 5
```

The following creates the same gongram pattern, but with two pens in the background color.

```
GONGRAM 15 130 72 144 43 BG BG
CYCLE 200 5
```

POLYCIRC is also animated by color shifting. Try this:

```
SETPC 1 BG  
SETPC 2 BG  
SETPC 0 55  
POLYCIRC 35 90 10 80 1  
CYCLE 300 5
```

You should see squares moving around in an elliptical path.

PROGRAM LISTING

For a listing of GONGRAM, see page 222; for POLYCIRC, see page 227.

```
TO STAR  
SETPC 0 50  
SETPC 1 BG  
SETPC 2 BG  
REPEAT 36 [FD 100 BK 100 RT 10 ►  
    NEXTPEN]  
END  
  
TO NEXTPEN  
IF PN = 2 [SETPN 0] [SETPN PN + 1]  
END  
  
TO CYCLE :TIMES :DELAY  
REPEAT :TIMES [CYC PC 1 :DELAY]  
END  
  
TO CYC :PC :DELAY  
SETPC 1 PC 0  
SETPC 0 PC 2  
SETPC 2 :PC  
WAIT :DELAY  
END
```

5

Music

Melodies

This section uses Atari Logo to make tunes, to combine them to make bigger ones, and to manipulate melodic elements by playing them backward and transposing them upward and downward. The section concludes with a pitch and rhythm sequencer.

Playing a Tune

TUNE lets you play single-voice melodies. It takes two inputs, a list of notes to play and a duration, and plays each note in the list with that duration.

```
TO TUNE :LIST :DUR
IF EMPTYP :LIST [STOP]
IF (FIRST :LIST)="R [TOOT Ø 15000 Ø :DUR]
    [TOOT Ø PITCH FIRST :LIST 15 :DUR]
TUNE BUTFIRST :LIST :DUR
END
```

The notes are represented as positive or negative integers. The letter R in a list is interpreted as silence (that is, a rest).

Try the following melody. Type:

```
SETENV Ø 1
TUNE [1 1 8 8 1Ø 1Ø 8 R 6 6 5 5 3 3 1 R] 3Ø
```

The melody you just played is “Twinkle, Twinkle Little Star.” The amplitude (loudness) of each of the notes of the melody cannot change.

Duration Using TUNE

TUNE gives the same duration to each note in the list. You might try different durations with the same list of notes. For example:

By Greg Gargarian.

TUNE [1 3 5 1] 40



or

TUNE [1 3 5 1] 20



The difference in these two melodies is that the first is played twice as fast as the second.

Numeric Pitch Representation in TUNE

Here's how to translate between musical notations. Beneath each note is the letter for the note as well as the number that TUNE uses for that note.

A musical staff in G clef with twelve notes. Below the staff is a list of note names and their corresponding TUNE numbers:

A = 1	A♯ = 2	B = 3	C = 4	C♯ = 5	D = 6	D♯ = 7	E = 8	F = 9	F♯ = 10	G = 11	G♯ = 12
-------	--------	-------	-------	--------	-------	--------	-------	-------	---------	--------	---------

and so forth.

The following is a melody using this notation. You see the traditional music notation for "Twinkle, Twinkle Little Star" along with the traditional note names. Underneath is the list of numbers TUNE uses to reproduce that melody.

A musical staff in G clef with a melody. Below the staff is a list of note names and their corresponding TUNE numbers:

A	A	E	E	F♯	F♯	E	R	D	D	C♯	C♯	B	B	A	R
[1	1	8	8	10	10	8	R	6	6	5	5	3	3	1	R]

As we mentioned earlier, R is interpreted as a rest. What TUNE really does when it sees an R is make the frequency of the note so high that you can't hear it! (That's what the high frequency of 15,000 is doing in TUNE.)

Putting Melodies Together

You can give a list of notes a name. For example:

MAKE "TWINKLE1 [1 1 8 8 10 10 8 R 6 6 5 5 3 3 1 R]

Now type:

TUNE :TWINKLE1 30

You can put different melodies together. Here we will stay with "Twinkle, Twinkle" and make another list of notes as a continuation of the melody. Let's give it the name TWINKLE2.

MUSIC

```
MAKE "TWINKLE2 [8 8 6 6 5 5 3 R]
```



Now type:

```
TUNE :TWINKLE2 30
```

Because this part of the melody is normally repeated, it's exactly half as long as TWINKLE1! You can use the REPEAT command to play it twice. Type:

```
REPEAT 2 [TUNE :TWINKLE2 30]
```

The SONG procedure combines TWINKLE1 and TWINKLE2 to make the entire melody.

```
TO SONG :DURATION
TUNE :TWINKLE1 :DURATION
REPEAT 2 [TUNE :TWINKLE2 :DURATION]
TUNE :TWINKLE1 :DURATION
END
```

To hear it, type:

```
SONG 30
```

The *tempo* of SONG (that is, the rate at which the notes follow each other) is determined by its duration input. To play SONG at a faster tempo, type:

```
SONG 20
```

How TUNE Works

TUNE goes through its list of notes, one element at a time, and plays each note at the duration you specified. TUNE calls TOOT for each note.

PITCH converts each note number in the notes list to its corresponding frequency. PITCH uses PITCH1 to help.

```
TO PITCH :NOTENUMB
OP PITCH1 :NOTENUMB 220
END

TO PITCH1 :NOTENUMB :BASE
IF :NOTENUMB = 0 [OP INT :BASE]
IF :NOTENUMB > 12 [OP PITCH1 :NOTENUMB-12 :BASE*2]
IF :NOTENUMB < 0 [OP PITCH1 :NOTENUMB+12 :BASE/2]
[OP PITCH1 :NOTENUMB - 1 :BASE*1.0595]
END
```

The note in this program is A at frequency 220. The A an octave above it has a frequency of 440. In fact, frequencies of notes an octave apart are

always related to each other in this way: going an octave higher doubles the frequency, going an octave lower halves the frequency. The variable :BASE is doubled or halved to perform this octave-changing function.

There are twelve chromatic steps in an octave. Therefore, the frequency of each note in the scale is the twelfth root of two higher than the next. Multiplying a note by 1.0595 gets the next note in the scale. The chromatic steps in between the octaves are determined by multiplying :BASE by 1.0595 *n* times, where *n* is the number of chromatic steps.

TUNE's note numbers start at a frequency of 220. That A is 1, and all the notes in the scale go up or down from there.

Symmetry in Melodies

One way to listen to the characteristics of a melody is to hear it backward. (This is similar to the kind of analysis sometimes done in a painting class. People will often look at a painting sideways or upside down in order to concentrate on the shapes and colors rather than the figures themselves.)

```
TO REVERSE :LIST
IF EMPTYP :LIST [OP []]
OP FPUT LAST :LIST REVERSE BL :LIST
END
```

You use REVERSE to put a list of notes in reverse order. Try it by typing:

```
PRINT REVERSE [1 3 5 6]
```

You should get:

```
6 5 3 1
```

as your result. You can compare the sound of a list of notes forward and backward using TUNE with and without REVERSE. Type and listen to the following:

```
TUNE [1 3 5 6] 30
```

To hear the reverse of it, type:

```
TUNE REVERSE [1 3 5 6] 30
```

Also try:

```
TUNE :TWINKLE1 30
TUNE REVERSE :TWINKLE1 30
```

You can use REVERSE to build a symmetrical melody from a short one. The procedure MUSIC.MIRROR takes a list of notes and plays it in the given order, then plays it in reverse order. (By the way, this reversing process is usually called taking the *retrograde* of the phrase.)

```
TO MUSIC.MIRROR :LIST :DUR
TUNE :LIST :DUR
TUNE REVERSE :LIST :DUR
END
```

MUSIC

Listen to the following examples, the first using our four-note phrase and the second using the tune list :TWINKLE1.

```
MUSIC.MIRROR [1 3 5 6] 30
MUSIC.MIRROR :TWINKLE1 30
```

In the next example, we construct a substantial melody with only two four-note phrases. We use the same four notes played before and make up another melody. Type the following:



Now we put these two melodies together in the procedure SONG1.

```
TO SONG1
REPEAT 2 [MUSIC.MIRROR :TUNE1 35 MUSIC.MIRROR :TUNE2 35]
END
```

Play it by typing:

```
SONG1
```

With eight notes we have been able to construct a twenty-four-note song! Try other melodies of your own design.

Transposing a Melody

A melody has a certain shape or contour that can be preserved regardless of the pitch at which the melody starts. If you add or subtract a musical step (or several steps) from each note in a melody, you don't change its overall shape. This process of raising or lowering all the notes equally is called *transposing*.

Transposing Up

TRANSPOSE.UP transposes all the notes of a phrase up. TRANSPOSE.UP works by adding its second input to each of the numbers in its input list. Try:

```
TUNE TRANSPOSE.UP [1 3 5 6] 2 30
```

This is equivalent to typing:

```
TUNE [3 5 7 8] 30
```

```
TO TRANSPOSE.UP :LIST :INT
IF EMPTYP :LIST [OP []]
OP FPUT (FIRST :LIST) + :INT TRANSPOSE.UP BF :LIST :INT
END
```

Listen to the difference between the list, before and after it has been transposed up.

Type:

TUNE [1 3 5 6] 30



and then

TUNE TRANSPOSE.UP [1 3 5 6] 2 30



An Effect Using TRANSPOSE.UP

CLIMB and CLIMBING use TRANSPOSE.UP to create the effect of a tune climbing. CLIMBING takes a list of notes as its first input and transposes it up step by step as many times as you want. The number of steps is CLIMB's second input.

Try:

```
CLIMB :TUNE1 3
CLIMB :TUNE2 7
CLIMB :TUNE3 5
```

```
TO CLIMB :NOTELIST :TIMES
CLIMBING :NOTELIST :TIMES 0
END
```

```
TO CLIMBING :NOTELIST :TIMES :UP
IF :TIMES = :UP [STOP]
TUNE TRANSPOSE.UP :NOTELIST :UP 35
CLIMBING :NOTELIST :TIMES :UP+1
END
```

Transposing Down

TRANSPOSE.DOWN is similar to TRANSPOSE.UP except that the melody is transposed *down*.

```
TO TRANSPOSE.DOWN :LIST :INT
IF EMPTYP :LIST [OP []]
OP FPUT (FIRST :LIST) - :INT TRANSPOSE.DOWN BF :LIST :INT
END
```

Type:

MAKE "TUNE3 [5 8 6 5]

Then listen to each of the following:

```
TUNE :TUNE3 30
TUNE TRANSPOSE.DOWN :TUNE3 1 30
TUNE TRANSPOSE.DOWN :TUNE3 3 30
```

A Single-Voice Music Sequencer

A music sequencer is an instrument that will repeat a sequence of notes for an indefinite period of time. We can make one by modifying TUNE.

```
TO SEQUENCER :LIST :DUR
IF EMPTYP :LIST [STOP]
IF (FIRST :LIST) = "R [TOOT 0 15000 15 :DUR]
[TOOT 0 PITCH FIRST :LIST 15 :DUR]
SEQUENCER (SE BUTFIRST :LIST FIRST :LIST) :DUR
END
```

Type:



Press BREAK to stop.

The big difference between TUNE and SEQUENCER is that SEQUENCER repeats your tune over and over. It won't stop until you press BREAK. In TUNE, the first note is played, then removed on the recursive call.

Short Durations

If the duration of the notes gets very short, you may want to change the "envelope" of the voice—that is, the rate at which the sound goes to silence (or decays) after its duration has been expended. This prevents a note from "spilling" into the next note.

SETENV's first input determines the voice. Since we're using voice 0 in SEQUENCER, the first input to SETENV should be 0. It's the second input to SETENV that determines the decaying time for the note. Try SETENV 0 1, which is a quick decay.

Type:

```
SETENV 0 1
SEQUENCER [6 5 3 1] 30
```

Try other values. To restore SETENV values, type:

```
SETENV 0 0
```

A Single-Voice Rhythm Sequencer

RHYTHM.SEQ is a single-voice rhythm sequencer that makes bongolike sounds. This procedure expects its list of notes to include only the letters H, M, and L (for *high*, *medium*, and *low*) and R (for *rest*). Type the following two lines and listen to the result, pressing BREAK to stop.

```
SETENV 0 1
RHYTHM.SEQ [H M L L H] 10
```



Press **BREAK** to stop.

The second input is the duration for each of the notes in the list.

```
TO RHYTHM.SEQ :LIST1 :DUR
IF EMPTYP :LIST1 [STOP]
IF EQUALP FIRST :LIST1 "R [TOOT 0 15000 0 :DUR]
  [TOOT 0 BONGO FIRST :LIST1 15 :DUR]
RHYTHM.SEQ (SE BUTFIRST :LIST1 FIRST :LIST1) :DUR
END
```

Traditionally speaking, rhythm usually implies periodic or repeating patterns. The list of elements that you have given **RHYTHM.SEQ** becomes such a pattern as it continues to repeat. For example, type:

```
RHYTHM.SEQ [L M H R R] 20
```

This is a five-beat pattern that gets its rhythm from the sequencing action alone.

A second way to produce rhythmic patterns is to use rests in different ways. Since **RHYTHM.SEQ** doesn't have many notes, you can concentrate on how far apart to space them in time. For example, type:

```
RHYTHM.SEQ [L R M H R M L R M H] 20
```

This has an internal feeling of three (waltzlike), yet it is a ten-beat pattern. Both rhythms seem to coexist.

A third way in which to construct patterns is with the low, medium, and high pitches. They can be used to either reinforce the existing patterns or they can serve as counterpoint to them. For example, type:

```
RHYTHM.SEQ [L M H R L M H L R H] 20
```



This rhythm reinforces the patterns of the previous ten-beat pattern by repeating the low-medium-high sequence almost three times (there's a rest in the middle of one of them) and by adding an additional rest between one of these repetitions to get the ten-beat phrase. Try the previous rhythmic sequences at a faster tempo by typing:

```
RHYTHM.SEQ [L R M H R M L R M H] 10
RHYTHM.SEQ [L M H R L M H L R H] 10
```

How RHYTHM.SEQ Works

RHYTHM.SEQ was designed by modifying **SEQUENCER**. The most conspicuous difference is that **RHYTHM.SEQ** uses a procedure called **BONGO** instead of **PITCH** to produce the **TOOT** frequencies.

```
TO BONGO :NOTE
IF :NOTE = "L [OP (59 + RANDOM 3)]
IF :NOTE = "M [OP (74 + RANDOM 3)]
IF :NOTE = "H [OP (87 + RANDOM 3)] [OP 15000]
END
```

MUSIC

BONGO interprets L, M, and H for RHYTHM. SEQ. It interprets R (and any other character) as a rest by outputting (OP) an inaudible frequency of 15,000.

As in SEQUENCER, if the rhythm is very fast, the second input to SETENV should be very small so that there is separation between notes.

Notice that the frequencies for L, M, and H are small numbers and, thus, relatively low notes. These frequencies will change slightly each time depending on the tiny RANDOM values that are added to them. This has been done to make the sounds more bongolike and less, for example, pianolike—that is, less “pitchy.”

PROGRAM LISTING

```

TO TUNE :LIST :DUR
IF EMPTYP :LIST [STOP]
IF (FIRST :LIST)="R [TOOT 0 15000 0 ▶
:DUR] [TOOT 0 PITCH FIRST :LIST ▶
15 :DUR]
TUNE BUTFIRST :LIST :DUR
END

TO SONG :DURATION
TUNE :TWINKLE1 :DURATION
REPEAT 2 [TUNE :TWINKLE2 :DURATION]
TUNE :TWINKLE1 :DURATION
END

TO PITCH :NOTENUMB
OP PITCH1 :NOTENUMB 220
END

TO PITCH1 :NOTENUMB :BASE
IF :NOTENUMB = 0 [OP INT :BASE]
IF :NOTENUMB > 12 [OP PITCH1 ▶
:NOTENUMB-12 :BASE*2]
IF :NOTENUMB < 0 [OP PITCH1 ▶
:NOTENUMB+12 :BASE/2] [OP PITCH1 ▶
:NOTENUMB - 1 :BASE*1.0595]
END

TO REVERSE :LIST
IF EMPTYP :LIST [OP []]
OP FPUT LAST :LIST REVERSE BL :LIST
END

TO MUSIC.MIRROR :LIST :DUR
TUNE :LIST :DUR
TUNE REVERSE :LIST :DUR
END

TO SONG1
REPEAT 2 [MUSIC.MIRROR :TUNE1 35 ▶
MUSIC.MIRROR :TUNE2 35]
END

TO TRANPOSE.UP :LIST :INT
IF EMPTYP :LIST [OP []]
OP FPUT (FIRST :LIST) + :INT ▶
TRANPOSE.UP BF :LIST :INT
END

TO CLIMB :NOTELIST :TIMES
CLIMBING :NOTELIST :TIMES 0
END

TO CLIMBING :NOTELIST :TIMES :UP
IF :TIMES = :UP [STOP]
TUNE TRANPOSE.UP :NOTELIST :UP 35
CLIMBING :NOTELIST :TIMES :UP+1
END

TO TRANPOSE.DOWN :LIST :INT
IF EMPTYP :LIST [OP []]
OP FPUT (FIRST :LIST) - :INT ▶
TRANPOSE.DOWN BF :LIST :INT
END

TO SEQUENCER :LIST :DUR
IF EMPTYP :LIST [STOP]
IF (FIRST :LIST) = "R [TOOT 0 15000 15 ▶
:DUR] [TOOT 0 PITCH FIRST :LIST ▶
15 :DUR]
SEQUENCER (SE BUTFIRST :LIST FIRST ▶
:LIST) :DUR
END

```

```

TO RHYTHM.SEQ :LIST1 :DUR
IF EMPTYP :LIST1 [STOP]
IF EQUALP FIRST :LIST1 "R [TOOT 0 ▶
 15000 0 :DUR] [TOOT 0 BONGO FIRST ▶
  :LIST1 15 :DUR]
RHYTHM.SEQ (SE BUTFIRST :LIST1 FIRST ▶
  :LIST1) :DUR
END

TO BONGO :NOTE
IF :NOTE = "L [OP (59 + RANDOM 3)]
IF :NOTE = "M [OP (74 + RANDOM 3)]
IF :NOTE = "H [OP (87 + RANDOM 3)] [OP ▶
  15000]
END

```

Ear Training

This project is an interactive music tutorial in ear training. It gives you the opportunity to listen to musical intervals and learn to recognize them.

How to Use the Ear Training Tutorial

The program picks two notes at random to construct a musical interval. Your task is to hear the interval and to select what you think it is. The program will tell you if you are right or give you the correct answer.

The program selects intervals within roughly one octave. They are shown here in traditional musical notation, with a written description, and with the abbreviated notation this program uses.



Running the Program

Those of you who feel bold can run the tutorial without reading this section.
Type:

EAR TRAINING

The following is a step-by-step description of how to use the program.
The program prints out the following instructions.

THIS PROGRAM PLAYS A NOTE
THEN ANOTHER...
...AND ASKS YOU FOR THE PITCH-
INTERVAL BETWEEN THEM.

FOR EXAMPLE, IF THE FIRST NOTE IS:

(a sound here)

AND THE SECOND IS:

(a sound here)

THE INTERVAL BETWEEN THEM IS A PERF.4

THE POSSIBLE INTERVALS ARE:

MINOR.2 MAJOR.2 MINOR.3
MAJOR.3 PERF.4 AUG.4
PERF.5 MINOR.6 MAJOR.6
MINOR.7 MAJOR.7 OCTAVE
MINOR.9 MAJOR.9 MINOR.10
MAJOR.10

HERE WE GO

Then the program gives you two notes and states:

TYPE ? AND RETURN FOR THE INTERVAL,
GIVE YOUR OWN ANSWER OR
PRESS RETURN FOR ANOTHER HEARING

Let's say you type:

MAJOR.6

There are two possibilities: either you are right, in which case it responds:

TRUE

or you are wrong, in which case it responds:

FALSE. THE RIGHT ANSWER IS: *whatever*

If you had typed ? and RETURN the program would have informed you of the interval you had just heard.

The program then gives you instructions.

PRESS RETURN TO CONTINUE, OR
ANY CHARACTER AND RETURN TO STOP.

Pressing RETURN gives you a new interval. If, when you hear an interval, you are not sure of the answer, you may listen a second time. When the program states:

TYPE ? AND RETURN FOR THE INTERVAL,
GIVE YOUR OWN ANSWER OR
PRESS RETURN FOR ANOTHER HEARING

pressing RETURN causes the same interval to be repeated, this time with the notes played together as well as one after the other.

PROGRAM LISTING

```

TO EAR.TRAINING
CT
PRINT [THIS PROGRAM PLAYS A NOTE,]
WAIT 60
PRINT [THEN ANOTHER...]
WAIT 120
PRINT [...AND ASKS YOU FOR THE PITCH-]
PRINT [INTERVAL BETWEEN THEM.]
WAIT 180
PRINT []
PRINT []
PRINT [FOR EXAMPLE, IF THE FIRST NOTE ▶
IS:]
TOOT 0 PITCH 2 15 60
WAIT 120
PRINT [AND THE SECOND IS:]
WAIT 60
TOOT 0 PITCH 7 15 60
WAIT 180
PRINT [THE INTERVAL BETWEEN THEM IS A ▶
PERF.4]
WAIT 100
PRINT []
PRINT []
PRINT [THE POSSIBLE INTERVALS ARE:]
PRINT []
PRINT [MINOR.2 MAJOR.2 MINOR.3]
PRINT [MAJOR.3 PERF.4 AUG.4]
PRINT [PERF.5 MINOR.6 MAJOR.6]
PRINT [MINOR.7 MAJOR.7 OCTAVE]
PRINT [MINOR.9 MAJOR.9 MINOR.10]
PRINT [MAJOR.10]
WAIT 240
PRINT []
PRINT [HERE WE GO]

PRINT []
WAIT 180
EAR.TRAINING2
END

TO EAR.TRAINING2
CT
PRINT [IF THE FIRST NOTE IS:]
MAKE "BASE (RANDOM 6) + 1
WAIT 30
TOOT 0 PITCH :BASE 15 60
WAIT 120
PRINT [AND THE SECOND IS:]
MAKE "INTERVAL (RANDOM 16) + 1
WAIT 30
TOOT 0 PITCH :BASE + :INTERVAL 15 60
WAIT 60
PRINT []
PRINT [TYPE ? AND RETURN FOR THE ▶
INTERVAL,]
PRINT [GIVE YOUR OWN ANSWER OR]
PRINT [PRESS RETURN FOR ANOTHER ▶
HEARING.]
CHECK.ANSWER :BASE :INTERVAL RL
TRY AGAIN
END

TO CHECK.ANSWER :BASE :INTERVAL :ANS
IF EMPTYP :ANS [ANOTHER.HEARING :BASE ▶
:INTERVAL STOP]
IF (FIRST :ANS) = "?" [PRINT (SE [THIS ▶
INTERVAL IS A] INTERVALS ▶
:INTERVAL) WAIT 60] [CHECKLIST ▶
:INTERVAL FIRST :ANS]
END

```

```

TO CHECKLIST :INTERVAL :ANS
IF (:INTERVALS :INTERVAL) = :ANS [PRINT ▶
    TRUE] [PRINT (SE [FALSE. THE ▶
        RIGHT ANSWER IS:] :INTERVALS ▶
        :INTERVAL)
WAIT 60
END

TO INTERVALS :NUMBER
OP ITEM :NUMBER [MINOR.2 MAJOR.2 ▶
    MINOR.3 MAJOR.3 PERF.4 AUG.4 ▶
    PERF.5 MINOR.6 MAJOR.6 MINOR.7 ▶
    MAJOR.7 OCTAVE MINOR.9 MAJOR.9 ▶
    MINOR.10 MAJOR.10]
END

TO ITEM :NUM :LIST
IF :NUM=1 [OP FIRST :LIST]
OP ITEM :NUM-1 BF :LIST
END

TO ANOTHER.HEARING :BASE :INTERVAL
PRINT []
PRINT [TOGETHER, THE NOTES ARE:]
WAIT 60
TOOT 0 PITCH :BASE 15 120
TOOT 1 PITCH :BASE + :INTERVAL 15 90
WAIT 120
PRINT [AGAIN, THE FIRST NOTE IS:]
TOOT 0 PITCH :BASE 15 120
PRINT [AND THE SECOND:]

```

```

TOOT 1 PITCH :BASE + :INTERVAL 15 90
PRINT []
PRINT [TYPE ? AND RETURN FOR THE ▶
    INTERVAL,]
PRINT [GIVE YOUR OWN ANSWER OR]
PRINT [PRESS RETURN FOR ANOTHER ▶
    HEARING.]
CHECK.ANSWER :BASE :INTERVAL RL
END

TO TRY.AGAIN
PRINT []
PRINT [PRESS RETURN TO CONTINUE, OR]
PRINT [ANY CHARACTER AND RETURN TO ▶
    STOP.]
IF EMPTYP RL [EAR.TRAINING2]
END

TO PITCH :NOTENUMB
OP PITCH1 :NOTENUMB 220
END

TO PITCH1 :NOTENUMB :BASE
IF :NOTENUMB = 0 [OP INT :BASE]
IF :NOTENUMB > 12 [OP PITCH1 ▶
    :NOTENUMB-12 :BASE*2]
IF :NOTENUMB < 0 [OP PITCH1 ▶
    :NOTENUMB+12 :BASE/2] [OP PITCH1 ▶
    :NOTENUMB - 1 :BASE*1.0595]
END

```

Sound Effects

This write-up presents a palette of sound effects to give you ideas for using sound in your own projects. We've kept our discussion brief. Instead, we ask you to use your ears. Some of these sound effects are new, others are taken from projects found elsewhere in this book. You might want to try them out and use those effects you like in your own projects, either as they are or in modified form. The procedures in this collection use the Atari Logo music primitives TOOT and SETENV.

A European Ambulance Siren

Typing the following lines results in a European ambulance sound.

By Greg Gargarian and Margaret Minsky; with contributions by Max Behensky.

```
SETENV 0 10
REPEAT 10 [TOOT 0 267 15 40 TOOT 0 200 15 30])
```

Advancing and Retreating Sounds

The procedures ADVANCE and RETREAT make sounds like something is rushing toward you or retreating from you. When sound sources advance toward you, you hear their pitch rising slightly and their volume increasing. When they retreat, you hear a falling pitch and decreasing volume. That is what these procedures try to do.

ADVANCE and RETREAT take two inputs. The first is a starting pitch for their sound and the second is FAST or SLOW for the speed of the advance or retreat. These procedures make sounds in both voice 0 and voice 1.

Here are the procedures for ADVANCE.

```
TO ADVANCE :PCH :DURATION
IF :DURATION = "FAST [ADVANCE1 :PCH 1]
IF :DURATION = "SLOW [ADVANCE2 :PCH 5]
END
```

```
TO ADVANCE1 :PCH :AMP
IF :AMP > 15 [STOP]
TOOT 0 :PCH :AMP 5
TOOT 1 :PCH*1.01 :AMP 5
ADVANCE1 :PCH*1.01 :AMP + 3
END
```

```
TO ADVANCE2 :PCH :AMP
IF :AMP > 15 [STOP]
TOOT 0 :PCH :AMP 5
TOOT 1 :PCH*1.01 :AMP 5
ADVANCE2 :PCH*1.01 :AMP + 1
END
```

Try:

```
SETENV 0 1
ADVANCE 440 "SLOW
ADVANCE 440 "FAST
```

Here are the procedures for RETREAT.

```
TO RETREAT :PCH :DURATION
IF :DURATION = "FAST [RETREAT1 :PCH 15]
IF :DURATION = "SLOW [RETREAT2 :PCH 15]
END
```

```
TO RETREAT1 :PCH :AMP
IF :AMP < 0 [STOP]
TOOT 0 :PCH :AMP 5
TOOT 1 :PCH*.99 :AMP 5
RETREAT1 :PCH*.99 :AMP - 3
END
```

MUSIC

```
TO RETREAT2 :PCH :AMP
IF :AMP < @ [STOP]
TOOT @ :PCH :AMP 5
TOOT 1 :PCH*.99 :AMP 5
RETREAT2 :PCH*.99 :AMP - 1
END
```

Try:

```
RETREAT 440 "FAST
RETREAT 440 "SLOW
```

You might want to try some other inputs.

```
ADVANCE 1000 "FAST
ADVANCE 100 "SLOW
RETREAT 200 "SLOW
```

You can try putting them together.

```
REPEAT 10 [ADVANCE 200"SLOW RETREAT 210 "FAST WAIT 10]
```

This one sounds like a monster snoring.

```
REPEAT 5 [ADVANCE 90 "FAST RETREAT 95 "SLOW WAIT 10]
```

Making Sliding Sounds: Glissandi

The RAMP procedure makes a sound that slides “smoothly” from a starting pitch to an ending pitch.

```
TO RAMP :START :FINISH :RATE
IF :FINISH < :START [REPEAT (:START-:FINISH)/ :RATE
  [TOOT 1 :START 15 2 MAKE "START :START - :RATE]]
IF :START < :FINISH [REPEAT (:FINISH-:START)/ :RATE
  [TOOT 1 :START 15 2 MAKE "START :START + :RATE]]
END
```

Try:

```
RAMP 400 1000 20
RAMP 1000 400 20
RAMP 300 500 70
RAMP 100 800 40
```

You get the idea. The first input is the starting frequency, the second is the ending frequency, and the third determines the rate of the slide. The bigger the third input, the faster the slide.

Try:

```
RAMP 500 700 4
RAMP 500 700 40
```

This one makes a "whooping" sound:

```
REPEAT 5 [RAMP 400 800 40]
```

A Motorcycle Sound

MOTORCYCLE uses RAMP to make the motorcycle warm up and ADVANCE and RETREAT to make it drive away.

```
TO MOTORCYCLE
REPEAT 10 [RAMP 25 120 (RANDOM 18)+1]
ADVANCE 50 "SLOW
RETREAT 50 "SLOW
END
```

A More Continuous Sliding for an Ambulance Siren

RAMP2 is a more complicated procedure that makes a more continuous slide. It can be used to make the sound of an ambulance siren.

```
TO RAMP2 :START :FINISH :RATE
IF :FINISH < :START
  [DOWN :START :FINISH :RATE TOOT 0 :FINISH 12 6]
IF :FINISH > :START
  [UP :START :FINISH :RATE TOOT 0 :FINISH 12 6]
END

TO UP :S :F :R
REPEAT (:F - :S) / :R
  [TOOT 1 :S 15 4 WAIT 1 TOOT 0 :S 12 4 MAKE "S :S + :R]
END

TO DOWN :S :F :R
REPEAT (:S - :F) / :R
  [TOOT 1 :S 15 4 WAIT 1 TOOT 0 :S 12 4 MAKE "S :S - :R]
END
```

Try:

```
REPEAT 10 [RAMP2 1200 1600 25 RAMP 1600 1200 25]
```

A Spaceship Sound

DEPARTURE uses RAMP in voice 1 and holds the starting pitch of the RAMP in voice 0.

```
TO DEPARTURE :FIRST :LAST :RATE
IF 255 < (ABS (:FIRST-:LAST)/:RATE)*2 [TOOT 0 :FIRST 12 255]
  [TOOT 0 :FIRST 12 (ABS (:FIRST-:LAST)/:RATE)*2]
RAMP :FIRST :LAST :RATE
END
```

MUSIC

```
TO ABS :NUM
IF :NUM < 0 [OP -:NUM] [OP :NUM]
END
```

Try DEPARTURE:

```
SETENV 0 8
SETENV 1 8
DEPARTURE 200 50 5
DEPARTURE 500 50 10
```

For a zigzag sound, try the following:

```
REPEAT 2 [DEPARTURE 1000 500 20 DEPARTURE 500 1000 20]
```

For a spaceshiplike sound, try:

```
TO SPACE.SHIP :NUM
REPEAT :NUM [DEPARTURE 1500 2000 20]
END
```

Try:

```
SPACE.SHIP 5
```

or

```
SPACE.SHIP 2
```

*A Boing-*ng*-*ng* Sound*

BOING works best in the low register . . . boings usually do!

BOING's first input is the frequency of the boing and the second input is the duration (in sixtieths of a second).

```
TO BOING :FR :DUR
SETENV 0 0
SETENV 1 0
IF :DUR > 100 [BOING1 15 1.5 :FR :FR/20 (:DUR/10) STOP]
BOING1 15 3 :FR :FR/20 (:DUR/5)
END

TO BOING1 :AMP :INC :FR :FR1 :DUR
IF :AMP < 1 [STOP]
TOOT 0 :FR :AMP :DUR
TOOT 1 :FR - ( :FR / 40 ) :AMP :DUR
BOING1 :AMP - :INC :INC ( :FR + :FR1 ) :FR1 :DUR
END
```

Try:

```
BOING 200 40
REPEAT 5 [BOING 50 30]
```

Trills and Thrills

The TRILL procedure plays a "trill" (a sound made up of alternating sounds). TRILL's first input is a frequency, the second input the interval, and the third is the number of times to alternate.

```
TO TRILL :FREQ :STEPSIZE :TIMES
REPEAT :TIMES [TOOT 0 :FREQ 15 5
                TOOT 0 :FREQ*STEPVALUE :STEPSIZE 15 5]
END

TO STEPVALUE :STEP
IF :STEP<2 [OP 1.0595]
IF :STEP=2 [OP 1.1225]
IF :STEP=3 [OP 1.1893]
IF :STEP=4 [OP 1.26]
IF :STEP=5 [OP 1.335]
IF :STEP=6 [OP 1.4145]
IF :STEP=7 [OP 1.4987]
IF :STEP=8 [OP 1.5878]
IF :STEP=9 [OP 1.6823]
IF :STEP=10 [OP 1.7824]
IF :STEP=11 [OP 1.888]
IF :STEP=12 [OP 2]
OP 2*STEPVALUE :STEP-12
END
```

Try:

```
TRILL 400 3 4
TRILL 400 3 8
TRILL 200 3 8
TRILL 200 2 8
TRILL 200 1 8
```

More with Trill

AGITATION is a procedure that uses TRILL to make trills of decreasing stepsize.

```
TO AGITATION :FREQ :NUMB
IF :NUMB < 1 [TOOT 0 :FREQ 5 30 STOP]
TRILL :FREQ :NUMB 4
AGITATION :FREQ :NUMB-1
END
```

Try:

```
AGITATION 200 4
AGITATION 1000 8
AGITATION 660 48
REPEAT 3 [AGITATION 50 4]
REPEAT 2 [AGITATION 2000 3]
```

MUSIC*Bird Sounds*

BIRDSONG and BIRDSONG1 are two different kinds of bird sounds. BIRDSONG uses RAMP to make a short and upward-gliding sound at a high frequency.

```
TO BIRDSONG
RAMP 1010 1070 10
WAIT (RANDOM 10)+1
END

TO BIRDSONG1
TRILL 1200 2 1
TRILL 1010 2 1
WAIT (RANDOM 10)+1
TRILL 1133 1 1
TRILL 801 2 1
WAIT (RANDOM 10)+1
END
```

Try:

BIRDSONG

BIRDSONG1 uses several short calls of TRILL to make a nervous-jumping bird sound.

Try:

BIRDSONG1

Bird Music

BIRDS makes a birdlike song using BIRDSONG and BIRDSONG1. To create variety BIRDSONG and BIRDSONG1 are played alternately, each a random number of times.

```
TO BIRDS
REPEAT (RANDOM 3)+1 [BIRDSONG]
WAIT (RANDOM 10)+1
REPEAT (RANDOM 3)+1 [BIRDSONG1]
WAIT (RANDOM 10)+1
BIRDS
END
```

Listen to it by typing:

BIRDS

Press the BREAK key to stop!

Sound for Jack and Jill

FANFARE is the music finale to the Jack and Jill project found in this book.

```

TO FANFARE
FANFARE1 55 110 7040
FANFARE1 35 70 7040
FANFARE1 30 60 7040
FANFARE1 25 50 7040
FANFARE1 30 50 7680
FANFARE1 120 200 7680
SETENV 0 15 SETENV 1 15
TOOT 0 240 15 240 TOOT 1 400 15 240
END

TO FANFARE1 :FR0 :FR1 :HIGHFR
IF :FR0 > :HIGHFR [STOP]
TOOT 0 :FR0 15 10 TOOT 1 :FR1 15 7
FANFARE1 :FR0*2 :FR1*2 :HIGHFR
END

```

Try:

FANFARE

Since FANFARE does SETENVs, you might want to restore by saying:

```

SETENV 0 0
SETENV 1 0

```

Playing with SETENV and Amplitudes

BOUNCE makes a ping-ponglike bouncing sound.

BOUNCE's input is for the frequency. As the BOUNCE note gets faster, its amplitude gets quieter and, near the end, its envelope gets shorter.

```

TO BOUNCE :FREQ
SETENV 0 1
BOUNCE1 :FREQ 15 40
END

```

Try:

```

BOUNCE 440
BOUNCE 100
REPEAT 5 [BOUNCE (RANDOM 400)+100]

```

You can shorten the longer durations of the bounce by lowering the duration input to BOUNCE1 (that is, by changing 40 to a smaller number).

```

TO BOUNCE1 :FREQ :AMP :DUR
IF :AMP < 1 [MAKE "AMP ABS :AMP"]
IF :DUR < 1 [FASTBOUNCE :FREQ STOP]
TOOT 0 :FREQ :AMP 10
WAIT :DUR
BOUNCE1 :FREQ :AMP-1 :DUR-(15-:AMP)
END

```

MUSIC

You can shorten the tail of the bounce by lowering the input to REPEAT in FASTBOUNCE.

```
TO FASTBOUNCE :FREQ
SETENV 0 0
REPEAT 10 [TOOT 0 :FREQ 2 5 WAIT 5]
END
```

An Echo Effect

ECHO is similar to BOUNCE, but ECHO doesn't get faster as it gets quieter. It uses SETENV to gradually change the decay of the repeating notes and RANDOM to produce the frequency, starting envelope, and pulsing rate of the echoed note.

```
TO ECHO
ENDING (RANDOM 800)+50 RANDOM 7 (RANDOM 15)+8
ECHO
END

TO ENDING :FR :DECAY :RATE
IF :DECAY=0 [ENDING1 :FR 15 :RATE STOP]
SETENV 0 :DECAY
TOOT 0 :FR 15 :RATE
ENDING :FR :DECAY-1 :RATE
END

TO ENDING1 :FR :AMP :RATE
IF :AMP=0 [STOP]
TOOT 0 :FR :AMP :RATE
ENDING1 :FR :AMP-1 :RATE
END
```

Try:

```
ENDING 400 2 25
ENDING 100 1 10
```

Now type:

ECHO

Press BREAK to stop!

Naming Notes

In the Melodies project, notes are represented by numbers that are later converted to appropriate frequencies. It takes time to do this for each note. If you want to play notes very rapidly one after the other, you can use another technique described in this project. The idea is to *precompute* the frequencies that correspond to the notes. In this project, we do this by giving names to these frequencies. The names we chose in this project come from traditional music notation, for example A#4, which represents the A-sharp in the fourth octave of piano pitches. We might make this a name by doing

```
MAKE "A#4 466
```

Thereafter (as in the Argue program) you can use THING to refer quickly to the frequency of a note. For example, if :NOTE is A#4, then THING :NOTE is 466. Using this scheme, your music procedures would contain lines like

```
TOOT @ THING :NOTE 15 20
```

You might have to name a lot of notes. This means there will be a lot of variables, and they will use up quite a bit of Logo's workspace. Sometimes it is worth it.

This project shows a program that automatically creates note names like A#4 and figures out the right frequencies. It does this for several octaves.*

Naming Notes

This program uses the naming technique just described to allow fast symbolic access to musical notes. The names follow a convention similar to standard music notation where, for example, G3 would be the name for G in the third piano octave.

Some examples:

```
MAKE "G3 392
MAKE "G#3 415
MAKE "A4 440
MAKE "A#4 466
MAKE "B4 493
```

Since we want to have names like these for many notes, we create procedures that automatically calculate and name frequencies.

*It is nice to use names like A#4, but the same technique can work with numbers (or anything else) as the names of variables for the notes. The main advantage of this project's technique is the use of variables for fast access to precomputed values. You could, for example, do MAKE 38 466 to give the name 38 to the frequency 466.

Procedures for Naming Notes

NAMENOTES calls NAMEOCTAVES, which calls NAMEOCTAVE, to name the notes in each octave. NAMENOTES also names the special “note” R so that it represents a rest.

```
TO NAMENOTES
MAKE "R 15000
NAMEOCTAVES [A A# B C C# D D# E F F# G G#] 1 55
END
```

NAMEOCTAVES takes three inputs: a list of prefixes for the names of notes ([A A# B C C# D D# E F F# G G#]), the starting suffix that is the number of the lowest octave for which names are to be created, and the frequency of the lowest pitch in that octave (the A of that octave).

```
TO NAMEOCTAVES :NAMES :OCTAVE :STARTFREQ
IF :OCTAVE > 8 [STOP]
NAMEOCTAVE :NAMES :OCTAVE :STARTFREQ
NAMEOCTAVES :NAMES :OCTAVE + 1 :STARTFREQ * 2
END
```

NAMEOCTAVES calls NAMEOCTAVE to make the variables for each octave. Each note name is created (in NAMEOCTAVE) by making a new word out of the appropriate prefix and the octave number. Then NAMEOCTAVES updates the octave number and the lowest frequency for the next octave. This continues until eight octaves of pitches have been named.

In NAMEOCTAVE the twelfth root of two is used in the following formula to compute the frequency of a note one half-step above another.

$$(frequency\ of\ a\ note) \times (\text{twelfth}\ root\ of\ 2) = (frequency\ of\ next\ note)$$

```
TO NAMEOCTAVE :NAMES :OCTAVE :FREQ
IF EMPTYP :NAMES [STOP]
MAKE ( WORD FIRST :NAMES :OCTAVE ) :FREQ
NAMEOCTAVE BF :NAMES :OCTAVE :FREQ * 1.0595631
END
```

Playing Melodies with Named Notes

Run NAMENOTES to create the note variables. Now you can use a procedure such as PLAYTUNE to play a melody.

```
TO PLAYTUNE :LIST
IF EMPTYP :LIST [STOP]
TOOT @ THING FIRST :LIST 15 15
PLAYTUNE BF :LIST
END
```

You can use PLAYTUNE like this:

```
PLAYTUNE [A2 A#2 B2 C2 C#2 D2 D#2 E2 F2 F#2 G2 G#2 A3]
```

You could name lists that represent phrases of songs and play them with PLAYTUNE. Here are some examples.

```

MAKE "SCALE [C2 D2 E2 F2 G2 A3 B3 C3]
MAKE "SCALE2 [C3 D3 E3 F3 G3 A4 B4 C4]
MAKE "TWINKLE [C2 C2 G2 G2 A3 A3 G2 R
  F2 F2 E2 E2 D2 D2 C2 R]
MAKE "FOLK [D4 C#4 D4 D4 D3 D3 F#3 F#3
  A4 A4 B4 A4 B4 C#4 D4 D4]

```

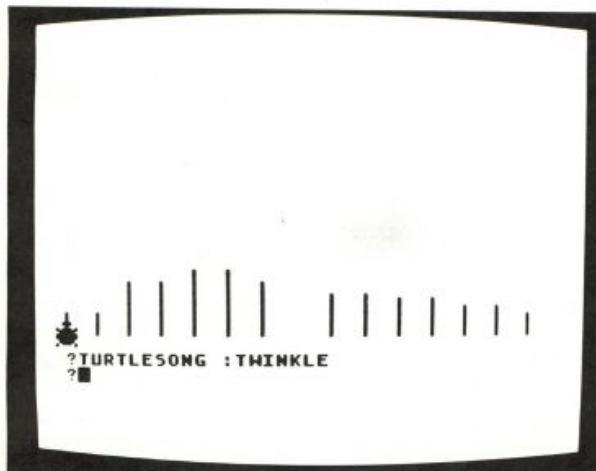
PLAYTUNE :TWINKLE

For more ideas about what you can do with melodies and rhythms, see the Melodies project.

Making Turtles Move to Your Song

Here's a program that makes a turtle show the "ups and downs" of your song. You can use it by trying the procedure TURTLESONG with a list of notes as its input. For example, try

TURTLESONG :TWINKLE



Here are the procedures.

```

TO TURTLESONG :LIST
SETUP.TURTLE
SONG :LIST
END

TO SETUP.TURTLE
CS
TELL 0
PU LT 90 FD 150 RT 90 BK 60 PD
ST
SETPN 0
SETPC 0 40
END

```

MUSIC

```

TO SONG :LIST
IF EMPTYP :LIST [STOP]
MAKE "NOTE FIRST :LIST
IF :NOTE = "R [TOOT 0 THING :NOTE 15 40] [JUMP THING :NOTE]
PU RT 90 FD 20 LT 90 PD
SONG BF :LIST
END

TO JUMP :FREQ
MAKE "INT (:FREQ - 100) / 3
FD :INT
TOOT 0 :FREQ 15 40
WAIT 15
RT 180
FD :INT
RT 180
END

```

These procedures draw lines on the screen whose lengths represent pitches. They fit best for notes in the range from :A2(110) to :D4(640).

Using the Atari Keyboard as a Music Keyboard

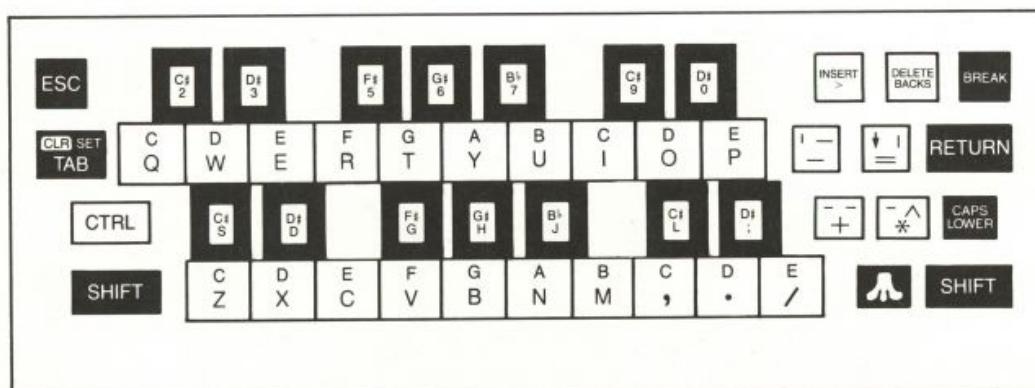
MUSIC is another example of using the precomputed notes; it makes the Atari keyboard act like a music keyboard. Start it up by typing

MUSIC

The program takes a while to set itself up, then it types

READY

Now you can play music by pressing keys. Notes are assigned to the keys according to a layout that is like that of a piano keyboard:



In order to make the layout similar to that of a piano, some of the keys do not play any note. Reminder: Take care not to press the Atari (/ \) key.

The Music Keyboard Procedures

MUSIC also creates variables for fast reference. It creates variables whose names are names of Atari keyboard keys (for example Z, X, C) and whose values are names of notes. This is done by ASSIGNKEYS. For example, :S becomes C#3.

MUSIC then calls KEYBOARD. KEYBOARD waits for you to press keys. It converts the name of the key you pressed to the name of a note and calls NOTE to play it. NOTE converts the name of the note to a frequency and calls the Logo primitive TOOT to make the sound.

There are a couple of fine points in this program. The length of time each note sounds is controlled by the global variable :DURATION, which is set up in the MUSIC procedure and by the SETENV commands in the procedure TEMPO. Also, the VOICE input to KEYBOARD makes it possible for you to press two keys in quick succession and hear both notes. The KEYBOARD procedure calls itself alternating between 0 and 1 as values for :VOICE. This allows the program to alternate the playing of notes between the Atari sound hardware voices 0 and 1. Thus one note keeps sounding in one voice while the program starts a second note sounding in the other voice.

Reminder: After using the MUSIC program, you may want to restore the music envelope decay to its initial state by saying SETENV 0 0 and SETENV 1 0.

```

TO MUSIC
ASSIGNKEYS [[Z C3] [X D3] [C E3] [V F3] [B G3] [N A4] [M B4]
[, C4] [. D4] [/ E4]]
ASSIGNKEYS [[S C#3] [D D#3] [G F#3] [H G#3] [J A#4]
[; L C#4] [; D#4]]
ASSIGNKEYS [[Q C4] [W D4] [E E4] [R F4] [T G4] [Y A5] [U B5]
[I C5] [O D5] [P E5]]
ASSIGNKEYS [[2 C#4] [3 D#4] [5 F#4] [6 G#4] [7 A#5]
[9 C#5] [0 D#5]]
MAKE "DURATION 20
TEMPO :DURATION
PR [READY]
KEYBOARD 0
END

TO ASSIGNKEYS :KEYNOTEPAIRS
IF EMPTYP :KEYNOTEPAIRS [STOP]
MAKE FIRST FIRST :KEYNOTEPAIRS LAST FIRST :KEYNOTEPAIRS
ASSIGNKEYS BF :KEYNOTEPAIRS
END

TO TEMPO :N
SETENV 0 :N / 10
SETENV 1 :N / 10
END

```

```

TO KEYBOARD :VOICE
MAKE "TEMP RC
IF NAMEP :TEMP [NOTE :VOICE THING :TEMP]
KEYBOARD 1 - :VOICE
END

TO NOTE :VOICE :NOTE
TOOT :VOICE THING :NOTE 15 :DURATION
END

```

PROGRAM LISTING

```

TO NAMENOTES
MAKE "R 1500
NAMEOCTAVES [A A# B C C# D D# E F F# G >
G#] 1 55
END

TO NAMEOCTAVES :NAMES :OCTAVE ▶
:STARTFREQ
IF :OCTAVE > 8 [STOP]
NAMEOCTAVE :NAMES :OCTAVE :STARTFREQ
NAMEOCTAVES :NAMES :OCTAVE + 1 ▶
:STARTFREQ * 2
END

TO NAMEOCTAVE :NAMES :OCTAVE :FREQ
IF EMPTYP :NAMES [STOP]
MAKE ( WORD FIRST :NAMES :OCTAVE ) ▶
:FREQ
NAMEOCTAVE BF :NAMES :OCTAVE :FREQ * ▶
1.0595631
END

TO PLAYTUNE :LIST
IF EMPTYP :LIST [STOP]
TOOT Ø THING FIRST :LIST 15 15
PLAYTUNE BF :LIST
END

TO TURTLESONG :LIST
SETUP.TURTLE
SONG :LIST
END

TO SETUP.TURTLE
CS
TELL Ø
PU LT 90 FD 150 RT 90 BK 60 PD
ST
SETPN Ø
SETPC Ø 40
END

TO MUSIC
ASSIGNKEYS [[Z C3] [X D3] [C E3] [V ▶
F3] [B G3] [N A4] [M B4] [, C4] ▶
[, D4] [/ E4]]
ASSIGNKEYS [[S C#3] [D D#3] [G F#3] [H ▶
G#3] [J A#4] [L C#4] [, D#4]]
ASSIGNKEYS [[Q C4] [W D4] [E E4] [R ▶
F4] [T G4] [Y A5] [U B5] [I C5] ▶
[O D5] [P E5]]
ASSIGNKEYS [[2 C#4] [3 D#4] [5 F#4] [6 ▶
G#4] [7 A#5] [9 C#5] [Ø D#5]]
MAKE "DURATION 20
TEMPO :DURATION
PR [READY]
KEYBOARD Ø
END

TO ASSIGNKEYS :KEYNOTEPAIRS
IF EMPTYP :KEYNOTEPAIRS [STOP]
MAKE FIRST FIRST :KEYNOTEPAIRS LAST ▶
FIRST :KEYNOTEPAIRS
ASSIGNKEYS BF :KEYNOTEPAIRS
END

```

```
TO TEMPO :N  
SETENV 0 :N / 10  
SETENV 1 :N / 10  
END
```

```
TO KEYBOARD :VOICE  
MAKE "TEMP RC  
IF NAMEP :TEMP [NOTE :VOICE THING ►  
:TEMP]  
KEYBOARD 1 - :VOICE  
END
```

```
TO NOTE :VOICE :NOTE  
TOOT :VOICE THING :NOTE 15 :DURATION  
END  
  
MAKE "SCALE [C2 D2 E2 F2 G2 A3 B3 C3]  
MAKE "SCALE2 [C3 D3 E3 F3 G3 A4 B4 C4]  
MAKE "TWINKLE [C2 C2 G2 G2 A3 A3 G2 R ►  
F2 F2 E2 E2 D2 D2 C2 R]  
MAKE "FOLK [D4 C#4 D4 D4 D3 D3 F#3 F#3 ►  
A4 A4 B4 A4 B4 C#4 D4 D4]
```

6

Programming Ideas

Adding Numbers

This section is about how to think and talk about the process of making a program. I developed the general approach while introducing elementary school children to computation. But the ideas that are good for children are good for other beginners, and perhaps for some experienced programmers. Variants of the example used here have been used with seventh graders, with college undergraduates, and with teachers. They illustrate a style of programming project, a style of programming language, and a meta-language or style of talking about programming as well as doing it. There is no suggestion that this style is uniquely correct. My message is on a different plane; I mean to assert the importance of paying more attention in the pedagogy of computation to such questions of style.

The problem is very recursive. I want to talk about programming, but I need to invent a way to talk about talking about programming! One way would be to give extracts from real dialog. But this is too cumbersome. Instead I shall condense real dialog into a kind of monolog about developing a program. The monolog gives an impression of one way I know how to think about developing a program. There is nothing very original about this way of thinking. The point I am making is about the technique of getting it out of our heads and into the pedagogy of teaching beginners.

In the discussion I carry with me a computational model in which there are little people, agents, experts in the computer that I can call on to help in thinking about the flow of my program, and, thus, in debugging my program. Keeping this model in mind helps me articulate what jobs need to be done and what procedures I need to get those jobs done. It also helps me figure out how these procedures interact with one another, how they report back what they have found out or constructed. Furthermore, as I debug my program and its individual procedures I talk again to these little people and get them to act out each procedure step by step, instruction by instruction.

The Project

We pretend the computer is ignorant of arithmetic and create an operation that will add two integers. No Logo arithmetic operations may be used. An apparent exception might seem to be EQUALP (=), but it is used to compare

By Cynthia Solomon.

whether two Logo words or letters are the same. (It is an identity operator.) So `+`, `<`, `>`, `COUNT`, `*`, `/`, and `REMAINDER` are prohibited. Two implications arise from posing this project. One is that an addition operation can be decomposed into smaller procedures. The other is that numbers are really just words asked to play special roles.

This project generates interesting discussions. It really frees one's thinking about numbers and operations and primitiveness. It is true that arithmetic is a very necessary part of any computer's hardware, but the hardware is made up of "logical units" that are based on the same ideas we will investigate. How do computers really add? It's in their hardware. It's built into the system. It's hardwired. Is addition "hardwired" into *our* system? Are we like computers and so if a wire is loose we can't do it? What about addition among children? Is it really a built-in capacity, or are there pieces of knowledge that are acquired? Maybe we are so familiar with addition that we forget its components. In fact, addition must rely on lots of procedures.

Let's look at this project. Try to situate this particular task into a familiar environment. We have to imagine that there are no arithmetic operators available to us and that there are no arithmetic experts already existing in Logo. We want to make up an addition operation so that we can say

```
PRINT ADD 16 532
```

and the computer will say

548

Yes, addition is a familiar operation and it's easy for us to hand-simulate its job. But what if we had to tell a little person in the computer how to add? Where do we start? We might ask ourselves if we know of a similar experience. What we have to do is "teach the computer" to add—just as we might teach a person! Well, now, teachers teach kids to add; we were once those kids. How did we learn—can we give ourselves some tips? (But I thought it was hardwired and teacher just . . .)

At this point in past discussions two suggestions emerge. Teachers say we have to teach the computer the "number facts" and computerists say we have to build a 10×10 table. Great, I say, a beginning. I ask teachers how we teach the number facts and what are they and how many of them there are. I ask computerists if a 10×10 table is large enough and how we organize it. The teachers will face these issues too. After all, making a table is a way of "teaching" number facts.

What kind of table and what are number facts? A table of the sums of the first 100 numbers is very limited, and building a larger table is still very limited. Is that what I have in my head? Isn't there a key idea or two that I could build on without exhausting the computer's memory?

Do children learn "number facts" like $16 + 20 = 36$ as a primitive notion, or is there a more fundamental idea underlying it all? What do kids learn about numbers? They learn their relationship to each other. They learn to order them. *Sesame Street* teaches kids to count from 1 to 20. Kids learn to recognize the digits and their order. They learn that one is the name of 1 and eleven is the name of 11 and one hundred eleven is the name of 111. They learn that 11 is different from 2; they learn that 10 has been added to 1. But there is another way of discussing that change. Let's say 1

PROGRAMMING IDEAS

is a special word. We can create a new word by putting it together with another. So WORD 1 1 is 11, or eleven. Concatenating is a way of changing numbers.

Let's return to learning to recognize digits and ordering them. That indeed is what we have to tell the computer and build upon. You might say we want to teach the computer to count. On the other hand, it does us no good to see the computer spew out numbers from 1 to 500. We want the computer to know *how* to count. Think of what's involved in counting. How many symbols are there? In one sense there are ten, 0 1 2 3 4 5 6 7 8 9; but there are many constructions like 13 or 444; then there are also funny changes such as from 9 to 10, from 19 to 20, from 29 to 30, and so forth.

We want to teach the computer that 7 comes after 6 and 10 follows 9 and so on. Some of it is tricky. But look, the only elements used in a base-10 number system are 0 1 2 3 4 5 6 7 8 9. If kids learn how to use those ten symbols in thousands of different ways, surely we can teach the computer. There must be some rules that specify what to do to produce the "next number in sequence."

That's what we have to do. That is our plan of attack. Tell the computer what the basic elements (our data base) are. Then develop rules of behavior so that we can make the computer give us our number plus 1, that is, the next number. If the computer can do that, it knows how to count.

What is knowing how to count? Here's a computerist model: There is "in the head" a collection of little people, experts capable of doing a whole bunch of things like spewing numbers out, but also capable of conceiving questions like what comes after this or before that. The computer, like children, learns to recognize the digits, how to order them, and then how to use them to make other numbers.

Okay, let's make a procedure that knows about digits. For example, if it receives the input 3, it will output 4. It will add 1 (in some mysterious way) to its input.

```
ADD1 3 ---> 4
ADD1 7 ---> 8
```

We Make ADD1

There are a couple of ways (at least) to do this. People who suggested "teaching number facts" or making tables, of course, had the right idea. There are different ways of constructing tables. For example:

```
TO DIGITTABLE :DIGIT
IF :DIGIT = 0 [OP 1]
IF :DIGIT = 1 [OP 2]
IF :DIGIT = 2 [OP 3]
IF :DIGIT = 3 [OP 4]
IF :DIGIT = 4 [OP 5]
IF :DIGIT = 5 [OP 6]
IF :DIGIT = 6 [OP 7]
IF :DIGIT = 7 [OP 8]
IF :DIGIT = 8 [OP 9]
IF :DIGIT = 9 [OP 10]
END
```

We can also look at the ordered list of digits [0 1 2 3 4 5 6 7 8 9] as another representation that has the same effect if we have a NEXT type of operation.

```
NEXT 0 [0 1 2 3 4 5 6 7 8 9] ---> 1
```

NEXT will output the next element in the list after the one specified.

```
TO ADD1 :DIGIT
OP NEXT :DIGIT [0 1 2 3 4 5 6 7 8 9]
END
```

Why do I suggest this way? It is a more general method. This process will work for any base; all that needs to be changed are the elements of the list!

We Design NEXT

NEXT must supply ADD1 with a word. ADD1 will then send the word out as its answer. From the example of NEXT at work, we see that NEXT is given two inputs, a word like 0 and a list of words. NEXT tells its helpers to look for the word in the list. They send back the word following it in the list.

```
IF :WD = FIRST :LIST [OP FIRST BF :LIST]
```

If :WD doesn't match with :LIST's first word, one of NEXT's helpers just crosses the first word off the list and turns the job over to someone else.

```
OP NEXT :WD BUTFIRST :LIST
```

Check this procedure out.

```
TO NEXT :WD :LIST
IF :WD = FIRST :LIST [OP FIRST BF :LIST]
OP NEXT :WD BF :LIST
END
```

Notice there is a potential bug. What if :WD is not in :LIST? Let's remember the bug, but postpone dealing with it for the moment.

Let's try ADD1 now. Give it a thorough testing. You could exhaustively try each digit because there are only ten. Another strategy is to choose extremes like 0 and 9.

```
PR ADD1 0
1
PR ADD1 1
2
PR ADD1 2
3
PR ADD1 3
```

```
PR ADD1 9
FIRST DOESN'T LIKE [] AS INPUT IN NEXT
```

PROGRAMMING IDEAS

It's logical that there is a bug. After all, 9 is the last element of the list. So there is more work to be done; we have to teach ADD1 that $9 + 1 = 10$.

ADD1 will work on any number that doesn't end in 9 if we make one small change! Look, all numbers not ending in 9 behave like digits when you add 1 to them.

```
123 ---> 124
13 ---> 14
```

The only digit that changes is the LAST, so ADD1 merely makes up a new word by replacing LAST :DIGIT with NEXT LAST :DIGIT. Let's be opportunistic—seize the chance, change ADD1 and call its input NUMBER.

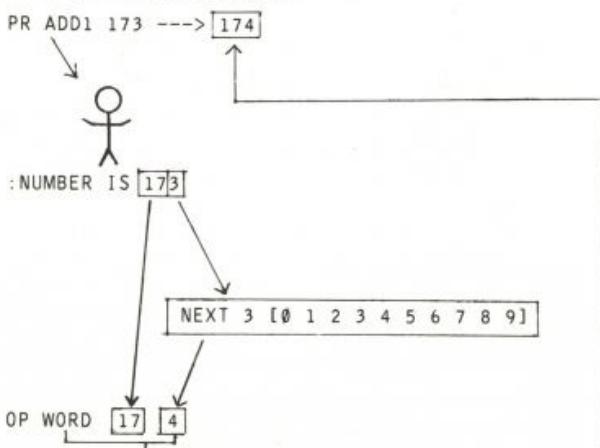
```
TO ADD1 :NUMBER
```

```
OP WORD [BL :NUMBER]
```

```
NEXT [LAST :NUMBER] [0 1 2 3 4 5 6 7 8 9]
```

```
END
```

Now we trace through this procedure using the little person metaphor. As a reminder, I draw a stick figure.



(So we thought ADD1 was only good for nine inputs. Suddenly we see it's good for how many—millions? infinitely many? nine-tenths of all the numbers?)

Now ADD1 works on all numbers that don't end in 9. Would it work if we pretend 10 is a digit and add it to the list given NEXT—that is, [0 1 2 3 4 5 6 7 8 9 10]? Then

```
PR ADD1 9
10
```

but

```
PR ADD1 19
110
```

instead of 20!

So putting 10 in the list did not really help. This *nines* bug is not cured so quickly. This issue is really about what to do with the "carry" when adding numbers. If a number is 9 then the answer is 10, but if a number ends in 9 we want to carry one to add it to the next digit of the number. Now, how can wishful thinking help? How can we make use of what we just did? Let's see how we do it. Try 179:

```
179 + 1 ---> 180
```

We turn the 9 into a 0 and add the 1 to the 17. We get 18... and don't forget to glue the 18 and the 0 back together.

Make a special check for LAST :NUMBER being 9. Then replace the 9 by 0 and ADD1 to BL :NUMBER.

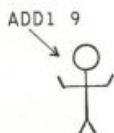
```
IF 9 = LAST :NUMBER [OP WORD ADD1 BL :NUMBER 0]
```

So

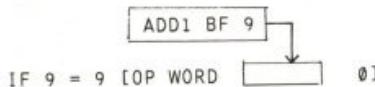
```
TO ADD1 :NUMBER
IF 9 = LAST :NUMBER [OP WORD ADD1 BL :NUMBER 0]
OP WORD BL :NUMBER NEXT LAST :NUMBER [0 1 2 3 4 5 6 7 8 9]
END
```

Now we try ADD1 with "little people."

```
PRINT ADD1 9
```



:NUMBER IS 9



but BF 9 ---> "

We can fix this bug by making another special test

```
IF 9 = :NUMBER [OP 10]
```

as the first instruction in ADD1. Now

```
PR ADD1 179
180
```

and

```
PR ADD1 9999
10000
```

What luck! Perhaps you thought that the first 9 on the left would give trouble. But we lucked out (or were super smart!).

Adding Two Numbers

Now that we can add 1 to any number, we can really add any number to any other.

It's simple if we think of the kinds of procedures we know about. Some procedures operate on their inputs until they are empty or until a thing has been found. Other procedures do a job for a specified number of times. We can think of the next stage in our project as *adding one* to an input for a declared number of times.

6 + 4 is ADD1 ADD1 ADD1 ADD1 6.

Typically, counter procedures count down to 0 and then they know the job is done. But they use subtraction, and we are trying to invent addition without using any of Logo's built-in arithmetic operations. We can teach the computer to *subtract one*.

If we had a SUB1 procedure, then

```
TO ADDUP :NUM1 :NUM2
IF :NUM2 = 0 [OP :NUM1]
OP ADD1 ADDUP :NUM1 SUB1 :NUM2
END
```

Making a procedure for subtracting 1 is really easy because we have already thrashed through the difficulties encountered in ADD1. How can we use what we know about ADD1 to describe a SUB1? Let's look at a concrete situation.

```
PR SUB1 2
1
PR SUB1 9
8
PR SUB1 1
0
```

Can SUB1 use NEXT?

If we want NEXT 1 [. . .] to be 0, how should the list be ordered? If we leave the list as [0 1 2 . . . 9], then NEXT 1 would output 2. It should output 0. Reverse the list. Then NEXT 1 [9 8 7 6 5 4 3 2 1 0] outputs 0.

So

```
TO SUB1 :NUMBER
OP WORD BL :NUMBER NEXT LAST :NUMBER [ 9 8 7 6 5 4 3 2 1 0 ]
END
```

Try SUB1.

It works! As long as the numbers don't end in what? Nine is okay. Why? The digit that is the LAST position of the list given to NEXT is the problem digit. That is when a "carry" or a "borrow" takes place. So SUB1 must take special measures when LAST :NUMBER is 0.

```
TO SUB1 :NUMBER
IF 0 = LAST :NUMBER [OP WORD SUB1 BL :NUMBER 9]
OP WORD BL :NUMBER NEXT LAST :NUMBER [ 9 8 7 6 5 4 3 2 1 0 ]
END
```

Now ADDUP works but very slowly and sometimes it needs too many people to complete the job. Look, ADDUP 9999 9999 requires 9999 little people.

Is there a shortcut? Yes. Let's treat the numbers as words and add the LAST digit of each number to ADDUP until :N1 and :N2 have been added together.

```
TO ADD :N1 :N2
IF EMPTYP BL :N1 [OP ADDUP :N2 :N1]
IF EMPTYP BL :N2 [OP ADDUP :N1 :N2]
OP WORD ADD BL :N1 BL :N2 ADDUP LAST :N1 LAST :N2
END
```

This is ideal but won't work very often. Do you know when it works?

```
PR ADD 34 21
55
PR ADD 2468 321
2789
```

but

```
PR ADD 19 19
218
```

The *carry* bug has to be dealt with. How can ADD tell if there is a carry? A carry means that ADDUP will send back two digits (1 and something). That makes it easy. ADD needs to test whether the result from ADDUP is one or two digits long. ADD uses ADDIT to help and now looks like:

PROGRAMMING IDEAS

```

TO ADD :N1 :N2
IF EMPTYP BL :N1 [OP ADDUP :N2 :N1]
IF EMPTYP BL :N2 [OP ADDUP :N1 :N2]
OP ADDIT ADDUP LAST :N1 LAST :N2 BL :N1 BL :N2
END

TO ADDIT :SUM :N1 :N2
IF EMPTYP BF :SUM [OP WORD ADD :N1 :N2 :SUM]
OP WORD ADD :N1 ADD1 :N2 BF :SUM
END

```

In some sense this project is completed. We have constructed an addition operation, and it works on positive integers. There are many extensions we could pursue. For example, handling negative numbers would probably necessitate making a subtract operation.

EXTENSIONS

In discussing setting up the table at the start, I mentioned the possibility of generalizing this scheme so that the operation would add numbers of other bases. What about fractions or decimals? But what about looking at a more general question? There are many arithmetic operations like MULTIPLY, DIVIDE, EXPONENTIATION, REMAINDER, BASE, CONVERSION, FACTORIAL. There are also others, like the Logo operation COUNT that outputs the length of a word or a list, and the predicates > (greater) and < (less). Any of these could be implemented as extensions to this project.

Although we might be able to write procedures to perform many of these operations, the process would probably be uncomfortably slow. This leads to the question: Are there some arithmetic operations that we couldn't define without special hardware or without special software? What operations are primitive? Imagine writing WORD or LIST or FIRST or BUTFIRST. What would be required? Is the derivation too clumsy? The answers to these questions will undoubtedly change as the contexts in which they arise change.

PROGRAM LISTING

<pre> TO ADD :N1 :N2 IF EMPTYP BL :N1 [OP ADDUP :N2 :N1] IF EMPTYP BL :N2 [OP ADDUP :N1 :N2] OP ADDIT ADDUP LAST :N1 LAST :N2 BL ▶ :N1 BL :N2 END </pre>	<pre> TO ADDUP :NUM1 :NUM2 IF :NUM2 = ∅ [OP :NUM1] OP ADD1 ADDUP :NUM1 SUB1 :NUM2 END </pre>
<pre> TO ADDIT :SUM :N1 :N2 IF EMPTYP BF :SUM [OP WORD ADD :N1 :N2 ▶ :SUM] OP WORD ADD :N1 ADD1 :N2 BF :SUM END </pre>	<pre> TO ADD1 :NUMBER IF 9 = :NUMBER [OP 10] IF 9 = LAST :NUMBER [OP WORD ADD1 BL ▶ :NUMBER ∅] OP WORD BL :NUMBER NEXT LAST :NUMBER ▶ [∅ 1 2 3 4 5 6 7 8 9] END </pre>

```

TO NEXT :WD :LIST
IF :WD = FIRST :LIST [OP FIRST BF ▶
:LIST]
OP NEXT :WD BF :LIST
END

```

```

TO SUB1 :NUMBER
IF Ø = LAST :NUMBER [OP WORD SUB1 BL ▶
:NUMBER 9]
OP WORD BL :NUMBER NEXT LAST :NUMBER ▶
[9 8 7 6 5 4 3 2 1 Ø]
END

```

Fill

FILL is a program to fill in solid areas on the graphics screen.

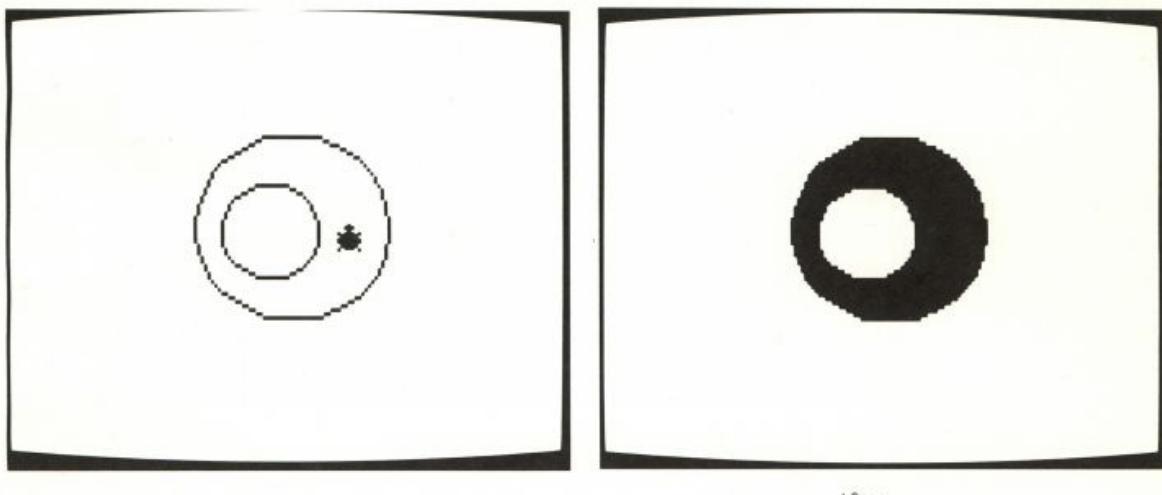


Figure 1

To use FILL, position the turtle inside the area you want to fill. Then type the command FILL with no inputs. The area the program will fill is bounded by lines drawn with any pen.* For example, try this:

```

CS
REPEAT 4 [FD 80 RT 90]
PU
SETPOS [20 20]
FILL

```

*If the screen dot at the turtle's position was already drawn with one of the pens, then FILL treats that pen as the background color for filling. So if you have a filled-in area on the screen, you can draw a picture within that area and fill the inside of the picture using another color.

PROGRAMMING IDEAS

to draw a solid, filled-in square. The `SETPOS` instruction is necessary to position the turtle inside the square, rather than on its edge, before using `FILL`.

Note: If you have a 16K Atari computer, you should use the number 8192 instead of 16384 in procedure `POSADDR`.

How It Works: Overview

Figure 2 shows a sort of eccentric doughnut with the turtle positioned between the two circles, so that the doughnut shape will be filled. The program begins by filling horizontally from the turtle's initial position, in both directions (figure 3). It remembers how far it got, to set left and right limits for what comes later. Then it starts moving up (figure 4), filling horizontally at each level.

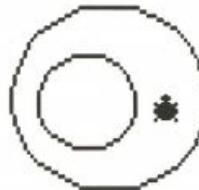


Figure 2

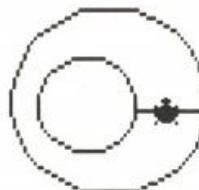


Figure 3

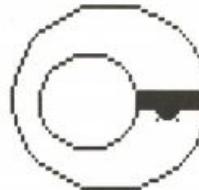


Figure 4

But when a newly filled line extends beyond the previous line (as illustrated by the left edge of the filled area in figure 4), the program also checks for an unfilled space below the new horizontal stretch. If it finds one, it starts filling downward in that new area (figure 5). This search for new areas works from left to right on each line, so (figure 6) the program continues moving downward below the inner hole until it reaches the bottom (figure 7).



Figure 5



Figure 6



Figure 7

Then it starts moving up into the newly discovered area to the right of the hole (figure 8), and when that area is filled, the program continues its interrupted upward filling of the top area (figure 9). The final result is shown in figure 10.



Figure 8



Figure 9



Figure 10

Screen Coordinates and Turtle Steps

The graphics screen consists of about 15,000 small dots, in a rectangular array of 96 rows and 160 columns. Logo draws lines on the screen by "turning on" some of these dots. To fill an area, we must also turn on dots in this array.*

When you use the FORWARD command, the distance measured in "turtle steps" is not the same as the number of screen dots (or *pixels*) through which the turtle passes. There are two reasons for this difference. The first reason is that the distance between two vertically adjacent pixels is greater than the distance between two horizontally adjacent pixels. If Logo measured distances in pixels, squares would come out looking like tall rectangles. Instead, Logo uses the *aspect ratio* (the ratio of a horizontal pixel distance to a vertical pixel distance) as a scale factor for vertical turtle steps. The second reason is that both vertical and horizontal turtle steps are scaled by a factor of two, so that 100 turtle steps is a reasonable distance on the screen.

The reason this scaling of distances is important for the FILL project is that we're going to have to think in terms of pixels, not in terms of turtle steps. Remember that the overall task of the program is to move along the screen looking for the border of the region we want to fill. In other words, the program must look at a position on the screen to see if that position is in the background color. If so, the program should fill in that position and move on to the next. Suppose we wrote the program in terms of turtle steps. (We'd then use FORWARD 1 to move from one position to the next.) Since a turtle step is smaller than the distance between pixels, two consecutive turtle positions will often occupy *the same pixel* on the screen! After filling in the first position, we'd move on to the next position and think we'd

*For more details about the screen array, see the Savepict and Loadpict project.

hit the border, because the screen dot would no longer be in the background color.

The approach I took in writing `FILL` is to think about positions in terms of screen pixel coordinates, rather than turtle coordinates. The top-level procedure `FILL` computes the pixel coordinates corresponding to the turtle's position, and those pixel coordinates are used as inputs to the lower-level procedures which do the real work. Figure 11 shows the screen coordinate system used in `FILL`. The origin of this system (the point with horizontal and vertical coordinates zero) is in the top left corner of the screen. `XCOR` (the horizontal coordinate) gets bigger as you move to the right. `YCOR` (the vertical coordinate) gets bigger as you move *down* the screen; compare this with Logo's turtle-step `YCOR`, which gets bigger as you move up the screen.

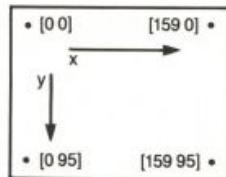


Figure 11

Because `FILL` uses screen coordinates instead of turtle coordinates, we can't use the usual Logo graphics procedures like `FORWARD` or `XCOR`. Instead, we have to write our own tools for examining and modifying screen pixels. Two important procedures in this project are `COLOR.AT`, which examines the color of a pixel, and `DOT`, which fills in a pixel.

One final point about the screen array is that each byte of computer memory contains the color information for four pixels. Logo's `.EXAMINE` procedure lets us look at an entire byte at a time, not just one pixel. Therefore, the program is more efficient if we can design it to examine four pixels at once. You'll see how we do that when we get to the description of the `FILL.RAY` procedure.

Initialization

Procedures `FILL`, `FILL1`, and `FILL2` are invoked just once each time you use `FILL`. They set up certain information that is needed throughout the program. Here are the procedures, followed by a list of their important variables.

```

TO FILL
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
PU
FILL1 79+INT (XCOR/2) 48-(INT (YCOR*:SCRUNCH/2))
    IF (PEN="PE") [0] [PN + 1]
END

- TO FILL1 :XCOR :YCOR :PEN
FILL2 COLOR.AT :XCOR :YCOR 0 0
END

```

```

TO FILL2 :BG :BGBYTE :PENBYTE
MAKE "BGBYTE 85*:BG
MAKE "PENBYTE 85*:PEN
FILL.BOTH FILL.LINE 0 0
END

```

SCRUNCH	The aspect ratio. This ratio is 0.8 unless you have changed it by using Logo's .SETSCR command. There is no direct way for FILL to find out the current aspect ratio, so it simply assumes a value of 0.8 unless you provide a different value in the global variable named SCRUNCH before you use FILL. This information is used in the procedure FILL to help convert the current turtle position into screen pixel coordinates.
XCOR	The turtle's current horizontal position, in pixels. Note that the <i>variable</i> XCOR is different from the Logo <i>procedure</i> named XCOR, which operates in turtle steps. Note also that the name XCOR is used for other variables in several sub-procedures to hold local position information.
YCOR	The turtle's current vertical position, in pixels. The same notes apply as for XCOR.
PEN	The pen we should use for filling. Since one of the possibilities is to fill by erasing (setting pixels to the background color), we don't use exactly the same numbers that Logo uses for pens. Instead, Logo's pens 0 to 2 are represented in this variable with the numbers 1 to 3, while the number 0 represents the background color. We use the background color if the turtle is in penetrase (PE) when you give the FILL command. Representing the background as 0 and the three pens as 1 to 3 is convenient in this program, because those numbers are the ones that are actually stored in the screen memory in the Atari computer.
BG	The pen number that is the background of the region we should fill. This is not necessarily <i>the</i> background color of the screen. When you give the FILL command, FILL1 uses sub-procedure COLOR.AT to find out whether the particular pixel at the turtle's position is in the background color or in one of the three pens. Whichever is true of that pixel, the corresponding color is what we look for to determine the region we're supposed to fill. The value of BG is coded like that of PEN: 0 for background, 1 to 3 for the three pens.
BGBYTE	FILL2 sets this variable to the value of BG multiplied by 85. This has the effect of reproducing the value of BG four times in a byte.* A memory byte that contains this number represents four consecutive BG-colored pixels.
PENBYTE	This is PEN reproduced four times in a byte, and it represents four consecutive PEN-colored pixels.

*If you understand how numbers are represented in binary in the computer's memory, you'll want to know that 85 is 01010101 binary. Multiplying a two-bit code (the possible values are 0 to 3) by this number has the desired effect of reproducing it four times in the eight-bit byte. If you don't know about binary representation, don't worry about it.

PROGRAMMING IDEAS

There is a trick in the way FILL1 calls FILL2. FILL2 has three inputs, named BG, BGBYTE, and PENBYTE. FILL1 provides the real value for the first input (BG), but it uses zero as the values for the others.

```
FILL2 (COLOR.AT :XCOR :YCOR) 0 0
```

FILL2 starts by assigning new values to these input variables. The reason for this trick is to make BGBYTE and PENBYTE *local* variables of FILL2 instead of global variables. Using local variables avoids leaving clutter around when FILL is finished. Actually, the use of local variables isn't terribly important in this particular example, but the same trick is used in some procedures we'll see later (most notably FILL.UP1) where it really is essential.

FILL2 begins the real work of filling an area with the instruction

```
FILL.BOTH FILL.LINE 0 0
```

FILL.LINE fills horizontally, on the line where the turtle is when you give the FILL command. Then FILL.BOTH uses information output by FILL.LINE to handle the vertical part of the filling. We'll discuss these procedures in more detail in the following sections.

Filling a Line

Here is the definition of FILL.LINE.

```
TO FILL.LINE :LEFT :RIGHT
MAKE "LEFT FILL.RAY :XCOR :YCOR (-1)
MAKE "RIGHT FILL.RAY :XCOR+1 :YCOR 1
OP (SE :YCOR :LEFT :RIGHT)
END
```

This procedure uses the same trick as FILL2 to create local variables LEFT and RIGHT. Although they're defined as inputs to FILL.LINE, these variables really get their values within FILL.LINE itself.

Most Logo procedures are either *commands*, which do something visible like move a turtle, or *operations*, which have no visible effect but instead output a value, like the arithmetic operations. FILL.LINE has both an effect and an output. Its effect is to fill the line on which the turtle starts. (Turn back to figure 3 to see FILL.LINE at work.) Its output is a list of coordinates, indicating how far to the left and right it was able to fill.

The turtle starts out somewhere in the middle of the area we want to fill. To fill the line containing the turtle's position, we have to start from that position and fill both to the left and to the right. FILL.LINE invokes FILL.RAY twice, first to fill toward the left and then to fill toward the right. FILL.RAY knows which direction to use because of its third input, which is -1 to fill leftward or 1 to fill rightward.

Filling in One Direction

`FILL.RAY` does all of the actual filling in of dots in the entire `FILL` program. The other procedures simply figure out where to tell `FILL.RAY` to go to work.

Because of the importance of `FILL.RAY`, I put a lot of effort into trying to make it fast. Unfortunately, the cost of speed is complexity. Let's start by examining a version of `FILL.RAY` that doesn't yet have all of the efficiency features added.

```
TO FILL.RAY :XCOR :YCOR :DELTA
IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
DOT :XCOR :YCOR
OP FILL.RAY :XCOR+:DELTA :YCOR :DELTA
END
```

`FILL.RAY` has three inputs. The first two are the horizontal (*x*) and vertical (*y*) screen coordinates of the pixel at which we want to start filling. The third input tells `FILL.RAY` the direction in which to fill.*

The strategy of `FILL.RAY` is this:

1. Look at a pixel to see if it's in our background color.[†]
2. If it's not in our background color, it is a border for the area we're filling. Output the *x* coordinate of the last pixel we actually filled—the one before this one.
3. If it is in our background color, fill it and move on to the next pixel in the desired direction, left or right.

To implement this strategy, `FILL.RAY` uses two subprocedures. The first, `EDGE`, is a predicate that outputs `TRUE` if the pixel it examines is in something *other than* the background color. The second subprocedure, `DOT`, fills in the pixel at the coordinates you give it as inputs. We'll look at those procedures later. For now, the important point is to understand how they're used by `FILL.RAY`.

Filling Vertically

We have seen how the `FILL` program fills one horizontal line, the one containing the turtle's position. What remains is to fill more lines, above and below that first one. This task is entrusted to `FILL.BOTH`.

```
TO FILL.BOTH :RANGE
FILL.UP :RANGE (-1)
FILL.UP :RANGE 1
END
```

*The word *delta* is the name of a Greek letter (Δ) that is often used in mathematics to represent a *change* in something. In this case, `:DELTA` is added to `:XCOR` each time a dot is filled in. If `:DELTA` is positive, the new *x* coordinate is to the right of the old one. If `:DELTA` is negative, the new coordinate is to the left.

[†]As explained earlier, this may or may not be the background color of the screen.

PROGRAMMING IDEAS

The name `FILL.BOTH` indicates that it must fill both above and below the line we've already filled. Just as `FILL.LINE` invokes `FILL.RAY` twice, `FILL.BOTH` invokes a subprocedure called `FILL.UP` twice.

`FILL.BOTH`, you'll remember, is invoked by `FILL2`. The input to `FILL.BOTH` is the output from `FILL.LINE`. This output is a list of three numbers: the vertical (y) coordinate of the line we've filled, and the leftmost and rightmost horizontal (x) coordinates of the line.* See figure 12 for a pictorial representation of this information.

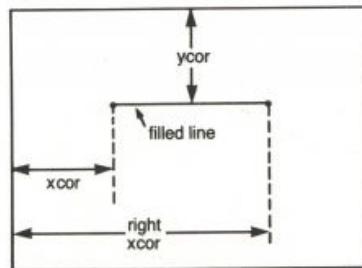


Figure 12

`FILL.BOTH` gives two inputs to `FILL.UP`. The first input is the range list. The second input tells `FILL.UP` the direction (up or down) in which to fill. This second input is either 1 or -1, just like the similar direction input to `FILL.RAY`.

Here is the definition of `FILL.UP`.

```
TO FILL.UP :RANGE :DELTA
  FILL.UP1 (:DELTA+FIRST :RANGE)
    FIRST BF :RANGE LAST :RANGE :DELTA 0 0
  END
```

All it does is to invoke `FILL.UP1`, with six inputs. The first three inputs are the three members of the range list, except that the vertical coordinate is offset by one. (The reason is this: the range list output by `FILL.LINE` contains the vertical coordinate of the line it just filled. We now want to fill a new line, just above or just below that line. The first input to `FILL.UP1` is the vertical coordinate of the line we should fill next.) The fourth input to `FILL.UP1` is the direction indicator, 1 or -1. The fifth and sixth inputs are given as zero. They're really used as local variables within `FILL.UP1`.

The Smart Procedure

`FILL.UP1` really contains all the geometric knowledge of this program. `FILL.UP1` has to know how to fill an area above or below a given line. This task would be very easy if areas were always pleasantly shaped. In fact, though, the filling job may have to "double back" because of irregularities in the area we're filling. This complication is illustrated in figures 4 and 5

*If you want to be picky, of course, what we've filled is a line *segment*, not a line.

(reproduced here). In figure 4, we are filling upward. This process continues straightforwardly until we get above the "hole" in the center of the region. At that point, the program is able to extend the filled area farther to the left. It then discovers a new, unfilled region below the new line. Figure 5 shows that the program has reversed its direction; it's filling downward to take care of the area to the left of the central hole.

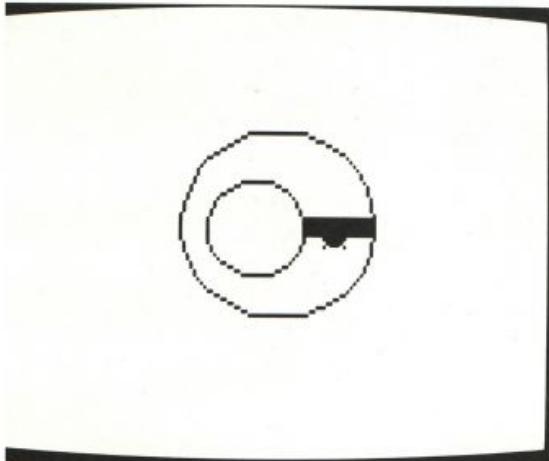


Figure 4

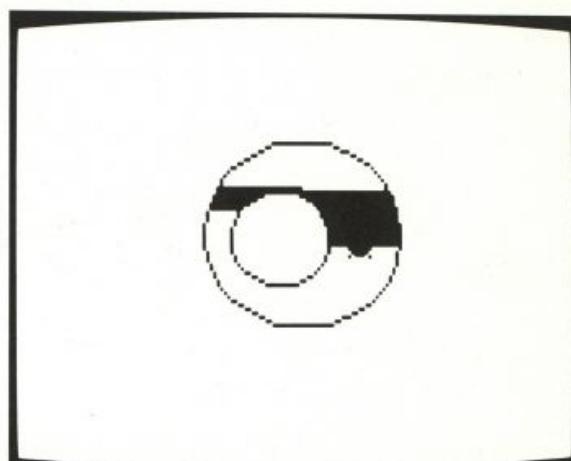


Figure 5

The strategy of FILL.UP1 is quite complicated, but it's made up of two kinds of parts: using FILL.RAY, and using FILL.UP1 recursively.

1. Use FILL.RAY to fill at the current vertical position.
2. Compare the horizontal extent of FILL.RAY's work to the horizontal extent of the previous line.
3. If we've gone farther on this line than on the previous line, invoke FILL.UP1 recursively to deal with the area newly exposed.
4. Also invoke FILL.UP1 recursively to continue with the same region we were already filling.

Since the procedure is complicated, we'll show its definition with the instruction lines numbered. In the discussion that follows we'll refer to particular lines by number.

```
[1] TO FILL.UP1 :YCOR :LEFT :RIGHT :DELTA :NEWL :NEWR
[2] MAKE "NEWL FILL.RAY ::LEFT :YCOR (-1)
[3] IF :NEWL<:LEFT
    [FILL.UP1 :YCOR-:DELTA :NEWL :LEFT (-:DELTA) 1 0]
[4] MAKE "NEWR
    IF :NEWL>:RIGHT [:NEWL-1] [FILL.RAY :LEFT+1 :YCOR 1]
[5] IF :NEWL<:NEWR+1
    [FILL.UP1 :YCOR+:DELTA :NEWL :NEWR :DELTA 2 0]
[6] IF :NEWR>:RIGHT
    [FILL.UP1 :YCOR-:DELTA :RIGHT :NEWR (-:DELTA) 3 0]
[7] MAKE "NEWL FIND.BG :NEWR :YCOR :RIGHT
[8] IF WORDP :NEWL [FILL.UP1 :YCOR :NEWL :RIGHT :DELTA 4 0]
[9] END
```

PROGRAMMING IDEAS

Refer to figure 13 for a picture of what happens in `FILL.UP1`'s work. The solid horizontal line in that picture was filled earlier, either by `FILL.LINE` or by the previous invocation of `FILL.UP1`. The dashed horizontal line above is the one that will be filled by the current invocation of `FILL.UP1`.

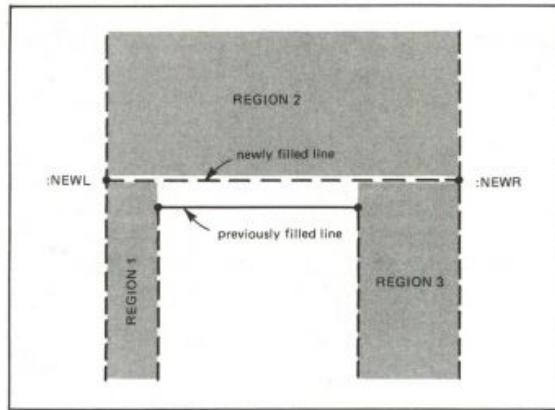


Figure 13

Here is a list of the variables used in `FILL.UP1`.

<code>YCOR</code>	The vertical coordinate of the dashed line, the one being filled by this invocation of <code>FILL.UP1</code> .
<code>LEFT</code>	The leftmost horizontal coordinate of the <i>solid</i> line, the one previously filled.
<code>RIGHT</code>	The rightmost horizontal coordinate of the solid, previously filled line.
<code>DELTA</code>	The direction indicator. Its value will be 1 if the new (dashed) line is above the old (solid) line, or -1 if the new line is below the old line.
<code>NEWL</code>	The leftmost horizontal coordinate of the new (dashed) line.
<code>NEWR</code>	The rightmost horizontal coordinate of the new line.

Each invocation of `FILL.UP1` actually fills only one line. This filling is done by using `FILL.RAY` twice, on lines 2 and 4 of the procedure. Line 2 fills to the left of `:LEFT`, and line 4 fills to the right of `:LEFT`. The variables `NEWL` and `NEWR` are given as values the *x* coordinates of the endpoints of the newly filled line.

When we're filling vertically, the most obvious thing is that after filling one line, we must continue filling vertically in the same direction. Referring to figure 13, after filling the dashed line we must continue upward, filling region 2 in the figure. (Of course, we don't know yet what the exact shape of that region will be. In the figure, it's shown as extending straight up, but the edges might really be curved.) This continuation in the same vertical direction is done in line 5 of the procedure.

How do we know when to stop? The answer is that if on *this* level we didn't manage to fill anything (because we ran into borders right away), then we shouldn't continue to the next level up. That's why line 5 compares `:NEWL` to `:NEWR`. If they're equal, we didn't fill anything on this level.

There are two possible cases of "doubling back": one if the newly filled line extends farther to the left than the old line, and one if the new line extends farther to the right. In figure 13, both of these situations have arisen.

We know that the new line has extended farther to the left than the old line if :NEWL is less than :LEFT. This is the situation at the transition from figure 4 to figure 5, which we've discussed earlier. Line 3 of the procedure checks for this situation. If the condition is met, then FILL.UP1 is recursively invoked to fill what is labeled region 1 in figure 13.

Similarly, we must double back on the right (into region 3 of figure 13) if :NEWR is greater than :RIGHT. Line 6 of FILL.UP1 takes care of this case. An example of this situation is at the transition between figure 7 and figure 8 (reproduced here). In figures 6 and 7, the program was filling downward. When the lower boundary of the region is reached, in figure 7, the program doubles back and starts filling upward in figure 8.



Figure 6



Figure 7



Figure 8

By the way, the doubling back into region 1 happens *before* the continued filling of region 2. But the doubling back into region 3 happens *after* region 2 is filled. That's because lines 3, 5, and 6 happen to be in the order they are. If line 3 were moved below line 5, the program would always complete one direction of filling before starting in the other direction.

There is one more complication in FILL.UP1. The line that is filled in lines 2 and 4 of the procedure extends to both sides of :LEFT, the leftmost end of the previously filled line. Suppose that a border is reached above the old line, before its rightmost end. This situation is shown in figure 14. Since we want to fill all of the area above the previously filled line, it's not enough to fill the area above the dashed line in the figure. We must also fill what is labeled as region 4.

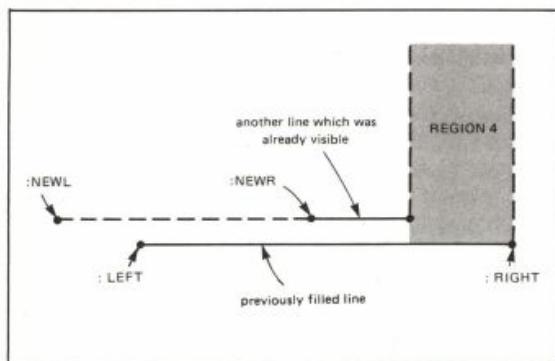


Figure 14

PROGRAMMING IDEAS

How do we know when this situation arises? First of all, :NEWR must be less than :RIGHT. Second, if we look to the right of :NEWR, we must find another patch of background color before reaching :RIGHT. This search is conducted by FIND.BG, which is used on line 7 of FILL.UP1. FIND.BG outputs the empty list if it does not find a suitable background pixel. If it does find one, FIND.BG outputs the *x* coordinate of that pixel. This coordinate is the left edge of region 4. Line 8 of FILL.UP1 checks to see if FIND.BG found a background pixel. If so, it invokes FILL.UP1 once more to fill region 4.

Examining a Screen Pixel

The real core of this program is the strategy FILL.UP1 uses to explore the nooks and crannies of irregular shapes. What remains for us to consider are the utility procedures that actually manipulate individual pixels. For example, FILL.RAY relies on EDGE to find out whether a particular pixel is a border of the area.

```

TO EDGE :XCOR :YCOR
OP NOT EQUALP :BG COLOR.AT :XCOR :YCOR
END

TO COLOR.AT :XCOR :YCOR
OP PIXEL (.EXAMINE POSADDR :XCOR :YCOR) REMAINDER :XCOR 4
END

TO POSADDR :XCOR :YCOR
OP 16384 + 40*:YCOR + INT (:XCOR/4)
      Use 8192 instead of 16384 for 16K Atari.
END

TO PIXEL :BYTE :XCOR
IF :XCOR=0 [OP INT (:BYTE/64)]
IF :XCOR=1 [OP REMAINDER INT (:BYTE/16) 4]
IF :XCOR=2 [OP REMAINDER INT (:BYTE/4) 4]
OP REMAINDER :BYTE 4
END

```

EDGE compares the color* of a particular pixel with our background color. It outputs TRUE if the two are different. That is, EDGE outputs TRUE if the pixel it's examining is on an edge of the area we're filling.

COLOR.AT outputs the color status of a pixel. Remember that each byte of screen memory contains this information for four pixels. So COLOR.AT must read a byte of screen memory and extract from that byte the particular pixel we're interested in.

POSADDR translates from the *x* and *y* coordinates of a pixel to the byte address in screen memory that contains that pixel. If you want to know about how these addresses are calculated, read the Savepic and Loadpic project.

*Actually, not the color number, but the pen number, in the form discussed earlier in the description of the PEN and BG variables.

PIXEL extracts one pixel from a byte. It takes two inputs. The first input is a byte of screen memory. The second input is a number from 0 to 3, specifying which pixel we want within that byte.

Filling One Pixel

FILL.RAY uses the procedure DOT to fill each pixel. DOT takes the coordinates of the pixel as inputs. Here it is.

```
TO DOT :XCOR :YCOR
DOTA POSADDR :XCOR :YCOR
END

TO DOTA :ADDR
RUN SE (WORD "DOT REMAINDER :XCOR 4") .EXAMINE :ADDR
END

TO DOT0 :BYTE
.DEPOSIT :ADDR SUM (REMAINDER :BYTE 64) 64*:PEN
END

TO DOT1 :BYTE
.DEPOSIT :ADDR (SUM (64*INT (:BYTE/64))
(16*:PEN) (REMAINDER :BYTE 16))
END

TO DOT2 :BYTE
.DEPOSIT :ADDR (SUM (16*INT (:BYTE/16))
(4*:PEN) (REMAINDER :BYTE 4))
END

TO DOT3 :BYTE
.DEPOSIT :ADDR SUM (4*INT (:BYTE/4)) :PEN
END
```

DOT must change the color of one pixel in a byte, leaving the other three pixels of that byte unchanged. Since Logo's .DEPOSIT command can only change an entire byte of memory at once, DOT has to combine the new color of one pixel with the old colors of the three other pixels. Precisely how to do this depends on which pixel in the byte we want to change, so DOT has a subprocedure for each possibility. These subprocedures are named DOT0 through DOT3.

Making FILL.RAY More Efficient

Earlier we looked at a simplified version of FILL.RAY, which examines and fills one pixel at a time. It's faster if we can examine an entire byte full of pixels at once. Here is the modified FILL.RAY, which does that, along with some new subprocedures.

PROGRAMMING IDEAS

```

TO FILL.RAY :XCOR :YCOR :DELTA
IF BYTEPOS :XCOR :DELTA
  [IF :BGBYTE=.EXAMINE POSADDR :XCOR :YCOR
   [OP FILL.CHUNK :XCOR :YCOR
    POSADDR :XCOR :YCOR :DELTA] ]
  IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
  DOT :XCOR :YCOR
  OP FILL.RAY :XCOR+DELTA :YCOR :DELTA
END

TO FILL.CHUNK :XCOR :YCOR :ADDR :DELTA
.DEPPOSIT :ADDR :PENBYTE
IF :BGBYTE=.EXAMINE :ADDR+:DELTA
  [OP FILL.CHUNK :XCOR+4*:DELTA :YCOR
   :ADDR+:DELTA :DELTA]
OP FILL.RAY :XCOR+4*:DELTA :YCOR :DELTA
END

TO BYTEPOS :XCOR :DELTA
IF :DELTA>0 [OP 0=REMAINDER :XCOR 4]
OP 3=REMAINDER :XCOR 4
END

```

FILL.RAY can only examine a complete byte of four pixels if the pixel it's ready to examine next is the first one in a byte. The predicate BYTEPOS outputs TRUE if that is the case. If not, FILL.RAY does the same things it did in the simpler version.

If BYTEPOS is TRUE, FILL.RAY examines the entire byte containing the pixel of interest. If that byte contains four pixels all in background color, we can fill all four at once. The variable BGBYTE contains the byte value that represents four background pixels.

If FILL.RAY does find a byte full of background pixels, it uses FILL.CHUNK to fill all four at once. FILL.CHUNK then examines the next byte to see if it, too, contains four background pixels. Once FILL.CHUNK reaches a byte that is not entirely background, it reverts to the use of FILL.RAY to check individual pixels.

Finding Region 4

The procedure FIND.BG, which is used to detect the appearance of a fourth region to fill, is very much like FILL.RAY, with two exceptions. First, FIND.BG passes over nonbackground pixels and stops when it reaches a background pixel. Second, FIND.BG just examines the pixels, whereas FILL.RAY fills them also.

```

TO FIND.BG :XCOR :YCOR :LIMIT
IF :XCOR>:LIMIT [OP []]
IF BYTEPOS :XCOR 1
  [IF :PENBYTE=.EXAMINE POSADDR :XCOR :YCOR
   [OP FIND.BG :XCOR+4 :YCOR :LIMIT] ]
IF NOT EDGE :XCOR :YCOR [OP :XCOR]
OP FIND.BG :XCOR+1 :YCOR :LIMIT
END

```

PROGRAM LISTING

```

TO FILL
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ▶
    0.8]
PU
FILL1 79+INT (XCOR/2) 48-(INT ▶
    (YCOR*:SCRUNCH/2)) IF (PEN="PE") ▶
    [0] [PN + 1]
END

TO FILL1 :XCOR :YCOR :PEN
FILL2 COLOR.AT :XCOR :YCOR 0 0
END

TO FILL2 :BG :BGBYTE :PENBYTE
MAKE "BGBYTE 85*:BG
MAKE "PENBYTE 85*:PEN
FILL.BOTH FILL.LINE 0 0
END

TO FILL.LINE :LEFT :RIGHT
MAKE "LEFT FILL.RAY :XCOR :YCOR (-1)
MAKE "RIGHT FILL.RAY :XCOR+1 :YCOR 1
OP (SE :YCOR :LEFT :RIGHT)
END

TO FILL.BOTH :RANGE
FILL.UP :RANGE (-1)
FILL.UP :RANGE 1
END

TO FILL.UP :RANGE :DELTA
FILL.UP1 (:DELTA+FIRST :RANGE) FIRST ▶
    BF :RANGE LAST :RANGE :DELTA 0 0
END

TO FILL.UP1 :YCOR :LEFT :RIGHT :DELTA ▶
    :NEWL :NEWR
MAKE "NEWL FILL.RAY :LEFT :YCOR (-1)
IF :NEWL<:LEFT [FILL.UP1 :YCOR-:DELTA ▶
    :NEWL :LEFT (-:DELTA) 1 0]
MAKE "NEWR IF :NEWL>:RIGHT [:NEWL-1] ▶
    [FILL.RAY :LEFT+1 :YCOR 1]
IF :NEWL<:NEWR+1 [FILL.UP1 ▶
    :YCOR+:DELTA :NEWL :NEWR :DELTA 2 ▶
    0]
IF :NEWR>:RIGHT [FILL.UP1 :YCOR-:DELTA ▶
    :RIGHT :NEWR (-:DELTA) 3 0]
MAKE "NEWL FIND.BG :NEWR :YCOR :RIGHT
IF WORDP :NEWL [FILL.UP1 :YCOR :NEWL ▶
    :RIGHT :DELTA 4 0]
END

TO EDGE :XCOR :YCOR
OP NOT EQUALP :BG COLOR.AT :XCOR :YCOR
END

TO COLOR.AT :XCOR :YCOR
OP PIXEL (.EXAMINE POSADDR :XCOR ▶
    :YCOR) REMAINDER :XCOR 4
END

TO POSADDR :XCOR :YCOR
OP 16384 + 40*:YCOR + INT (:XCOR/4)
END

TO PIXEL :BYTE :XCOR
IF :XCOR=0 [OP INT (:BYTE/64)]
IF :XCOR=1 [OP REMAINDER INT ▶
    (:BYTE/16) 4]
IF :XCOR=2 [OP REMAINDER INT (:BYTE/4) ▶
    4]
OP REMAINDER :BYTE 4
END

TO DOT :XCOR :YCOR
DOTA POSADDR :XCOR :YCOR
END

TO DOTA :ADDR
RUN SE (WORD "DOT REMAINDER :XCOR 4) ▶
    .EXAMINE :ADDR
END

TO DOT0 :BYTE
.DEPPOSIT :ADDR SUM (REMAINDER :BYTE ▶
    64) 64*:PEN
END

TO DOT1 :BYTE
.DEPPOSIT :ADDR (SUM (64*INT ▶
    (:BYTE/64)) (16*:PEN) (REMAINDER ▶
    :BYTE 16))
END

TO DOT2 :BYTE
.DEPPOSIT :ADDR (SUM (16*INT ▶
    (:BYTE/16)) (4*:PEN) (REMAINDER ▶
    :BYTE 4))
END

TO DOT3 :BYTE
.DEPPOSIT :ADDR SUM (4*INT (:BYTE/4)) ▶
    :PEN
END

```

```

TO FILL.RAY :XCOR :YCOR :DELTA
IF BYTEPOS :XCOR :DELTA [IF ▶
  :BGBYTE=.EXAMINE POSADDR :XCOR ▶
  :YCOR [OP FILL.CHUNK :XCOR :YCOR ▶
    POSADDR :XCOR :YCOR :DELTA] ]
IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
DOT :XCOR :YCOR
OP FILL.RAY :XCOR+:DELTA :YCOR :DELTA
END

TO FILL.CHUNK :XCOR :YCOR :ADDR :DELTA
.DEPPOSIT :ADDR :PENBYTE
IF :BGBYTE=.EXAMINE :ADDR+:DELTA [OP ▶
  FILL.CHUNK :XCOR+4*:DELTA :YCOR ▶
  :ADDR+:DELTA :DELTA]
OP FILL.RAY :XCOR+4*:DELTA :YCOR ▶
  :DELTA
END

TO BYTEPOS :XCOR :DELTA
IF :DELTA>0 [OP 0=REMAINDER :XCOR 4]
OP 3=REMAINDER :XCOR 4
END

TO FIND.BG :XCOR :YCOR :LIMIT
IF :XCOR>:LIMIT [OP []]
IF BYTEPOS :XCOR 1 [IF ▶
  :PENBYTE=.EXAMINE POSADDR :XCOR ▶
  :YCOR [OP FIND.BG :XCOR+4 :YCOR ▶
    :LIMIT] ]
IF NOT EDGE :XCOR :YCOR [OP :XCOR]
OP FIND.BG :XCOR+1 :YCOR :LIMIT
END

```

Savepict and Loadpict

When you've drawn a complicated picture, it's useful to be able to save the picture itself in a disk file, so that you can later restore it to the screen without going through the procedures that drew the picture again. For example, suppose you're writing a video adventure game in which characters in the story are drawn against a backdrop showing a forest, dungeon, or whatever. The backdrop could be saved as a picture file and then loaded onto the screen for each scene before drawing in the actors.

In this project, you'll see three different sets of Logo programs for saving and loading pictures. The three versions differ in how fast they can load a picture and also differ somewhat in flexibility. The last version, for example, allows a small picture to be "stamped" on the screen in different positions. One thing to learn from this project is how using different *data representations* can affect the efficiency of a program.

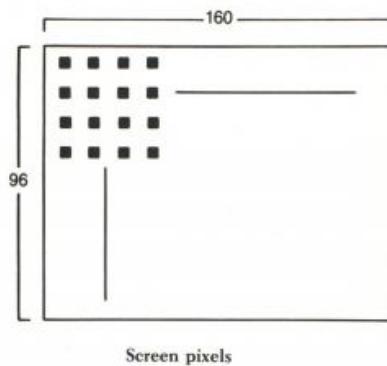
There are two ways to approach this project. If you just want to use these procedures as a tool to save and load pictures for some other project of your own, you don't have to understand some of the details explained here about how pictures are stored. On the other hand, by studying how the project works, you can learn about the important idea of data representation.

Note: If you have a 16K Atari computer, you should use the number 8192 instead of 16384 in procedures `SAVEPICT`, `LOADPICT`, and `PICTLOC`. (`PICTLOC` appears only in the third version of the project.) With a 16K machine, you don't have a disk drive, but you could save pictures on cassette.

By Brian Harvey.

How a Picture Is Stored

In order to save and load pictures, we have to know something about how a picture is represented in the Atari computer. In this project we are concerned only with the pictures drawn with pens, not with the turtle shapes. The lines you draw are represented as a pattern of dots (called *pixels*) on the screen. There are 96 rows and 160 columns of dots on the screen:



Screen pixels

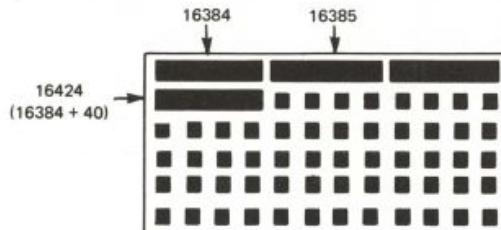
The reason that a diagonal line comes out jagged on the screen is that it isn't actually drawn as a smooth line, but simply by filling in certain dots on the screen. Each pixel can be in one of four conditions: it can be empty (that is, it can be in the background color) or it can be filled in with one of the three possible pens.

By the way, the length of a "turtle step" is not the same as the distance between pixels. That is, when you type the command FORWARD 100 the turtle does not move 100 pixels on the screen. How many pixels it actually does move depends on the direction. If you're moving horizontally (heading 90, for example), then FORWARD 100 moves through 50 pixels. If you're moving vertically, the distance depends on the *aspect ratio*, which is controlled by the .SETSCR command. The usual aspect ratio is 0.8, in which case FORWARD 100 moves 40 pixels (50 times 0.8). In this project, since we're interested in saving a picture that is already on the screen rather than drawing a picture with turtle commands, we have to think in terms of pixels, not in terms of turtle steps.

I said that each pixel can be in any of four conditions (background or three pens). Therefore, each pixel can be represented in the computer's memory using two *bits*, or binary digits. Each bit can be either zero or one. The four conditions are represented this way:

```
0 0 background
0 1 pen 0
1 0 pen 1
1 1 pen 2
```

Memory is grouped into *bytes* of eight bits. So each byte represents four pixels. There are 96 times 160, or 15,360, pixels altogether on the screen. The memory required is one fourth of that, or 3840 bytes. It happens that the first byte of Logo's screen memory is at memory location number 16384. So the picture memory is arranged something like this:



Picture memory

Characters (letters, digits, spaces, and so on) are represented in the computer's memory by a number that is stored in one byte. For example, the letter A is represented by a byte containing the number 65. Most of the time you don't have to worry about this, but if you remember this fact, it'll help you understand the process of storing information in disk files.

Representing the Screen in a Disk File

The most straightforward way to represent a screen picture in a disk file is simply to write each of the 3840 bytes into the file. To find out what is in each byte, we use the .EXAMINE operation, which outputs a number representing the byte at whatever memory location is used as its input. For example:

```
PRINT .EXAMINE 16384
```

will print the number in the first byte of Logo's screen memory. This byte represents the first four pixels in the upper left corner of the screen. (For Atari computers with 16K of RAM, the first byte of screen memory is in location 8192 instead of 16384.)

It would be possible to save a picture in a file, then, with a program like this:

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
PRINT .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END
```

Each byte of the picture memory would be represented in the file by a line containing the digits in the number in that byte. That is, if a particular byte happened to contain the number 125, that byte would be stored in the file as the three digits 1, 2, 5, just as it is typed on the screen by a PRINT command. Each digit takes up one byte in the file. Therefore, using this scheme, it takes three bytes in the file to represent one byte in the picture!

(Actually, another byte is used to represent the end-of-line code.) This leads to very large files.

Instead, it would be better to use only one byte in the file to represent each byte in the picture. This can be done by using the operation CHAR. This procedure takes a number as its input and outputs the single character that corresponds to that number. For example, CHAR 65 outputs the letter A. Using this procedure, we can write the program as follows:

Savepict/Loadpict, Version 1

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
TYPE CHAR .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 3840
SETREAD []
END

TO LOADPICT1 :LOC :NUM
IF :NUM=0 [STOP]
.DEPPOSIT :LOC ASCII RC
LOADPICT1 :LOC+1 :NUM-1
END
```

To use the SAVEPICT procedure, you first draw a picture on the screen using the usual turtle commands. Then you say

SAVEPICT "D:PICTFILE

or whatever you want to name the file. The program writes 3840 bytes into this file. Later, you can restore the picture to the screen by typing

LOADPICT "D:PICTFILE

The operation ASCII, which is used in LOADPICT1, is the inverse of CHAR. It takes a single character as input and outputs the number that represents that character. So ASCII "A" outputs 65.

Experiment with these procedures. You'll find that both saving and loading pictures are quite slow. This is because the procedures SAVEPICT1 and LOADPICT1 are invoked 3840 times, once for each byte of screen memory, even if nothing is drawn in that part of the screen. Also, the files written by this version of SAVEPICT are rather large (3840 bytes), so you can't fit very many on a diskette.

Sparse Data Representations

A typical turtle graphics picture is *sparse*. This means that most of the pixels on the screen are unused (background color), which means that most of the bytes of picture memory are zero. It seems silly to write a file that is mostly full of zeros. By using a cleverer representation of the picture, we can write smaller files and make the loading of a picture file much faster.

The idea is this: as we look through the picture memory, we'll find a bunch of zero bytes, and then a nonzero one, and then a bunch more zero bytes, and so on. To make this more specific, consider this sample fragment of a picture memory:

```
0 0 0 0 0 0 0 0 23 0 0 0 0 47 0 0 0 0 0 0 0 0 15
```

In the first version of the program, we'd represent these twenty-four bytes of screen memory as twenty-four bytes in the file. But instead, we can think of this as 9 zeros, 23, 4 zeros, 47, 8 zeros, 15. We could store this information in a file in this form:

```
9 23 4 47 8 15
```

In other words, we have decided that odd-numbered bytes in the file represent how many consecutive zero bytes are in the picture, while even-numbered bytes represent actual picture data. By representing the picture in this way, we've reduced twenty-four bytes of picture to six bytes in the file. We'll find that it is also much faster to load a picture stored in this form.

In practice, there may be several hundred consecutive zero bytes in a picture. This poses a slight problem: the largest number that can be represented in a single byte is 255. Therefore, if there are more than that many consecutive zeros, the new `SAVEPICT` procedure writes the sequence 255 0 in the file for each group of 256 zeros.

A second minor detail is that there must be a way for `LOADPICT` to know when the end of the file has been reached. This isn't a problem in the first version of the program because there all picture files are the same length, 3840 bytes. But in the new version, the length of the file depends on the number of pixels that are drawn in a nonbackground color. To solve this problem, `SAVEPICT` writes the sequence 0 0 at the end of the file. This sequence can't be part of real picture data.

Savepict/Loadpict, Version 2

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 .EXAMINE :LOC :NULL
END
```

```
TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 ASCII RC ASCII RC
SETREAD []
END

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
.DEPPOSIT :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII RC
END
```

Experiment with this version of the program. You'll notice that `SAVEPICT` isn't any faster, but `LOADPICT` is usually very much faster. The reason is that `SAVEPICT` must still examine every byte of picture memory, because it doesn't know ahead of time where you've drawn lines. But `LOADPICT` only has to deposit information into the bytes in picture memory that actually correspond to lines in the saved picture file.

Snapshots

In the second version, `LOADPICT` doesn't change the parts of picture memory that aren't used in the picture file you're loading. This suggests that it should be possible to *merge* two pictures. (In the first version, loading a picture file completely replaced whatever might have been on the screen before you invoked `LOADPICT`.) Try drawing a picture, saving it with `SAVEPICT`, clearing the screen, drawing another picture, and then using `LOADPICT` to restore the first picture. Make sure that the two pictures aren't in exactly the same part of the screen, so you can see whether the old picture remains intact.

What you'll find is that this merging of two pictures works pretty well, but not perfectly. The problem comes up if the two pictures use pixels that are right next to each other, so that a pixel in one picture is part of the same byte of memory as a pixel of the other picture. (Remember that each byte contains four pixels.) Loading a new number into that byte eliminates the pixel that used to be there. Still, this technique works perfectly if the two pictures are widely separated, and it works pretty well in most cases.

It would be handy to take advantage of this merging capability by using a picture file as a kind of rubber stamp that could be drawn in different positions on the screen. The scheme is this: you draw a small picture near the center of the screen. Then you use a version of `SAVEPICT` to make a "snapshot" of this picture. You can then use a version of `LOADPICT` to "stamp" the saved picture anywhere on the screen, depending on the turtle position.

PROGRAMMING IDEAS

To make this work, the picture file must include information about where the turtle was when the picture was taken. `SAVEPICT` must be modified to write this information in the file. Then `LOADPICT` must be modified to compare the current position of the turtle to the one stored in the file. If the two positions are different, the picture should be loaded into a different part of the screen memory.

This third version of the program is quite a bit more complicated than the others. The main reason for this is that it has to deal with the difference between pixels and turtle steps. To know where to "stamp" the saved picture in memory, we have to think in terms of pixels. But Logo tells us the turtle's position in turtle steps. This position has to be rounded off to the nearest pixel. Also, as explained earlier, the conversion between steps and pixels depends on the aspect ratio. There is no easy way for a Logo procedure to find out what this ratio is. The solution used in this program is that it looks for a variable named `SCRUNCH` in the workspace. If there is such a variable, its value should be the aspect ratio. If not, the standard value of 0.8 is assumed.

Another complication is that if the picture is being loaded into a position that is different from where it came from, part of the picture may extend beyond the edge of the screen. The procedure `PUTBYTE` in the following program is used like `.DEPOSIT`, but it checks to be sure that you are trying to deposit into the part of memory that contains the picture.

Savepict/Loadpict, Version 3

```

TO SAVEPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
SETWRITE :FILE
TYPE CHAR (XCOR+160)/2
TYPE CHAR (120-YCOR)*:SCRUNCH/2
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 .EXAMINE :LOC :NULL
END

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
SETREAD :FILE
LOADPICT1 PICTLOC ASCII RC ASCII RC
SETREAD []
END

```

```

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
PUTBYTE :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII RC
END

TO PICTLOC
OP 16384+((XDIFF ASCII RC)+160*YDIFF ASCII RC)/4
END

TO PUTBYTE :LOC :BYTE
IF (AND :LOC>16383 :LOC<20224 :BYTE>0) [.DEPOSIT :LOC :BYTE]
END

```

Note: If you have a 16K Atari computer, you should use the following:

```

IF (AND :LOC>8191 :LOC<12032 :BYTE>0) [.DEPOSIT :LOC :BYTE]

TO XDIFF :XLOC
OP INT (XCOR+160)/2-:XLOC
END

TO YDIFF :YLOC
OP INT (120-YCOR)*:SCRUNCH/2-:YLOC
END

```

To experiment with this program, try something like this:

```

CS
REPEAT 4 [FD 40 RT 90]
SAVEPICT "D:SQSNAP"
PU
SETPOS [80 60]
LOADPICT "D:SQSNAP"
SETPOS [-70 20]
LOADPICT "D:SQSNAP"

```

In practice, you wouldn't bother making a snapshot of something as simple as a square, because it's easier to draw another square than to load it from a disk file. But if you draw more complicated pictures, in multiple colors, this technique can really be worthwhile.

Suggestion: Run-Length Encoding

What if you filled in the screen completely with some pen color and tried to save that in a picture file? Using the first version of the program, of course, it doesn't matter what's on the screen; the file ends up with 3840 data bytes. But with the two later versions, something else happens. The picture memory is completely filled with bytes that represent the same number, but not zero. For example, if you fill the screen with pen 0, the picture memory will be

85 85 85 85 ...

In the sparse encoding scheme we've been using, this is thought of this way: 0 zero bytes, 85, 0 zero bytes, 85, and so on. What ends up in the picture file is

```
0 85 0 85 0 85 0 85 0 85 ...
```

The picture file is twice as big as the screen memory! This isn't a very good result. The smart LOADPICT will be slower for this picture than the stupid one. A sparse representation only works well if the picture is, in fact, sparse.

This is an extreme, unlikely example. But it isn't unlikely for *part* of the screen to be filled in solidly. For example, if you're drawing a picture of a farm, the background might be blue to represent the sky, and there might be a large solid green area at the bottom of the screen to represent grass.

Still, although that green area isn't *empty*, it is *uniform*. The bytes representing that area in screen memory are mostly all the same, even if not all zero. We could use a slightly more complicated data representation called *run-length encoding*, which would handle this case well. Here's how it works. Instead of a two-byte sequence representing the number of zero bytes and then the value of a data byte, we can use a sequence representing the value of a data byte and the number of consecutive bytes containing that value. For example, suppose the screen memory looks like this:

```
85 85 85 85 85 1 0 0 0 0 0 0 43 85 85 85 85 ...
```

We would represent that in the picture file this way:

```
85 5 1 1 0 7 43 1 85 4 ...
```

In this example, the version in the file is only a little smaller than the screen memory. But in real situations, the run lengths would often be several hundred bytes, not just five or seven.

This run-length technique is often used in serious computer graphics work. It's especially efficient for black-and-white pictures, because there are only two possible values for the data. You can just alternate them and leave them out of the file. You only store the run lengths. That is, the odd-numbered bytes of the file would contain the numbers of consecutive black pixels and the even-numbered bytes would contain the numbers of consecutive white pixels.

On the other hand, for a color picture that really is sparse, the representation we've been using is somewhat more efficient than the run-length representation. The moral is that before you choose a data representation for any problem, you should think hard about different possibilities!

PROGRAM LISTING

VERSION 1

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END
```

```
TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
TYPE CHAR .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END
```

```

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 3840
SETREAD []
END

TO LOADPICT1 :LOC :NUM
IF :NUM=0 [STOP]
.DEPPOSIT :LOC ASCII RC
LOADPICT1 :LOC+1 :NUM-1
END

VERSION 2

TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 ▶
    .EXAMINE :LOC :NULL
END

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ▶
    0.8]
SETREAD :FILE
LOADPICT1 PICTLOC ASCII RC ASCII RC
SETREAD []
END

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
PUTBYTE :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII ▶
    RC
END

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 ASCII RC ASCII RC
SETREAD []
END

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
.DEPPOSIT :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII ▶
    RC
END

VERSION 3

TO SAVEPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ▶
    0.8]
SETWRITE :FILE
TYPE CHAR (XCOR+160)/2
TYPE CHAR (120-YCOR)*:SCRUNCH/2
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 ▶
    .EXAMINE :LOC :NULL
END

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ▶
    0.8]
SETREAD :FILE
LOADPICT1 PICTLOC ASCII RC ASCII RC
SETREAD []
END

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
PUTBYTE :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII ▶
    RC
END

TO PICTLOC
OP 16384+((XDIFF ASCII RC)+160*YDIFF ▶
    ASCII RC)/4
END

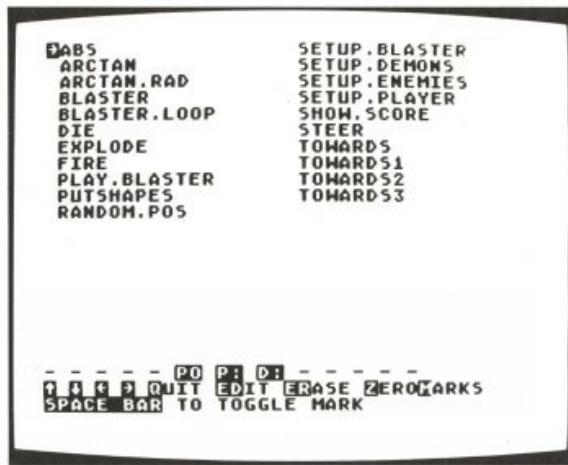
TO PUTBYTE :LOC :BYTE
IF (AND :LOC>16383 :LOC<20224 :BYTE>0) ▶
    [.DEPOSIT :LOC :BYTE]
END

TO XDIFF :XLOC
OP INT (XCOR+160)/2-:XLOC
END

TO YDIFF :YLOC
OP INT (120-YCOR)*:SCRUNCH/2-:YLOC
END

```

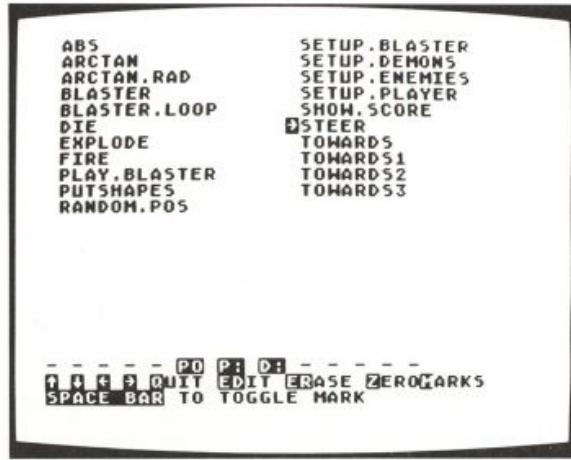
Display Workspace Manager



The Display Workspace Manager (DWM) is a tool that helps you manage projects that involve large numbers of procedures. The program lists all your procedures on the screen. You can move a pointer around, marking particular procedures. Then you can edit, erase, print, or save the marked procedures.

DWM divides the screen into two parts. The top part is used to list the names of procedures. The bottom few lines remind you of the commands you can type to DWM. (For example, you can type ER to *erase* procedures.)

In the figure above, DWM is being used to examine Blaster, a project in this book. The arrow points to the word STEER on the screen. STEER is the name of one of the procedures in Blaster. The pointer arrow can be moved from one procedure name to another by using the arrow keys on the Atari keyboard.



By Brian Harvey.

In the next figure, the user has typed the P0 command to DWM. DWM's P0 command tells it to print out the definition of the procedure at which the arrow points, in this case STEER.



```

TO STEER :WHERE
IF :WHERE < 0 [STOP]
SETH 45 * :WHERE
SETSH 1 + :WHERE
END

TYPE A SPACE WHEN READY

```

In the following figure, seven procedures have been *marked* with asterisks on the screen.



*ABS	SETUP.BLASTER
*ARCTAN	SETUP.DEMONS
*ARCTAN.RAD	SETUP.ENEMIES
BLASTER	SETUP.PLAYER
BLASTER.LOOP	SHOW.SCORE
DIE	STEER
EXPLODE	*TOWARDS
FIRE	*TOWARDS1
PLAY.BLASTER	*TOWARDS2
PUTSHAPES	*TOWARDS3
RANDOM.POS	

----- PO P: D: -----
 ↑ ↓ ← → QUIT EDIT ERASE ZERO MARKS
 SPACE BAR TO TOGGLE MARK

In the next figure the user has typed the command ER, which means to erase all the marked procedures. DWM has printed "Really erase 7 proce-

PROGRAMMING IDEAS

dures?" on the bottom line of the screen. It asks this question to make it harder for someone to erase many procedures accidentally.

```

*ABS          SETUP.BLASTER
*ARCTAN       SETUP.DEMONS
*ARCTAN.RAD   SETUP.ENEMIES
BLASTER      SETUP.PLAYER
BLASTER.LOOP  SHOW.SCORE
DIE          STEER
EXPLODE     *TOWARDS
FIRE         *TOWARDS1
PLAY.BLASTER *TOWARDS2
PUTSHAPES   *TOWARDS3
RANDOM.POS

```

PO P: D: - - -
 ↑ ↓ ← → QUIT EDIT ERASE ZEROARKS
 SPACE BAR TO TOGGLE MARK
 REALLY ERASE 7 PROCEDURES?

In the next figure, the user has typed Y for yes, and DWM has erased the marked procedures. It now displays a shorter list of the remaining procedures.

```

BLASTER
BLASTER.LOOP
DIE
EXPLODE
FIRE
PLAY.BLASTER
PUTSHAPES
RANDOM.POS
SETUP.BLASTER
SETUP.DEMONS
SETUP.ENEMIES
SETUP.PLAYER
SHOW.SCORE
STEER

```

PO P: D: - - -
 ↑ ↓ ← → QUIT EDIT ERASE ZEROARKS
 SPACE BAR TO TOGGLE MARK

This is a large project. I won't attempt a complete explanation of every detail of the program. Instead, I'll indicate the most important parts to understand.

Creating the List of Procedures

In order for DWM to work, it must have a list of the names of all the procedures in your project. This list must be in a global variable named

ALL.PROCEDURES. If this list doesn't already exist, the first thing DWM does is to call DWM.PROCLIST to create the list. This automatic creation of the list requires a disk drive with a writeable disk in it! DWM.PROCLIST works by doing a POTS command while writing to the disk, then rereading the results to find the names of your procedures. When the list is created automatically, it is sorted alphabetically. The sorting process is quite slow, because it's done simply rather than cleverly. (Read the Mergesort project for another sorting technique.) The automatically generated list omits all procedures whose names start "DWM" so that the procedures in the DWM program itself won't clutter up your list.

If you want to save time when starting up DWM, or if you want the procedures in your project listed in some order other than alphabetical, you can create the variable ALL.PROCEDURES yourself and make it part of the workspace file.

How DWM Arranges the Display

Once the list of procedures exists, DWM lists them on the screen. This is done by two main procedures, DWM.SIZE.MENU and DWM.DRAW.MENU. The first of these figures out how the names should be arranged on the screen, given the number of procedures you have in your list. The more procedures, the more columns on the screen will be required to list them all. The more columns, the less wide each column can be. This limits the length of a procedure name that can be displayed. Therefore, the program uses the smallest number of columns that will fit your list. Then the DWM.DRAW.MENU procedure uses this information to draw the display.

If the name of a procedure is too long to fit in a screen column, an inverse video plus sign (+) is shown at the end of the truncated name.

Reading DWM Commands

The procedure DWM.MAIN.LOOP reads and processes the commands you type to the program. Commands are either one or two characters long. Here is a list of the commands.

- | | |
|-----------|---|
| arrows | Move the pointer up, down, left, or right. You can type the arrow keys either with or without the CTRL key held down. |
| space bar | Mark the procedure where the pointer is, if it's not marked already, or unmark it if it is. An asterisk is displayed next to the name of marked procedures. |
| ED | Edit the marked procedures in the Logo editor. |
| ER | Erase all the marked procedures. This command first tells you how many procedures are marked and insists that you type Y to confirm that you really want to erase the procedures. |
| PO | Print out on the screen the single procedure whose name is pointed to by the arrow. |
| P: | List all marked procedures on the printer. |
| D: | Save all marked procedures on the disk. This command prompts for a filename to be used for the saved procedures. Notice that these save files do not contain the values of varia- |

bles! But the procedures in them can be loaded with the LOAD command.

- ZM Zero Marks. Unmark all procedures.
- Q Quit. Exits from DWM.

If there are no marked procedures, the commands that normally apply to marked procedures apply instead to all procedures in the display. Be careful about erasing!

Possible Extensions

DWM takes up just under 2000 nodes, somewhat more than half the available space. This limits the size of the programs you can use it on. (This is particularly unfortunate since it's the big projects that most need this sort of help.)

If there were space, this project could be the basis for implementing workspace management tools like PACKAGE and BURY, which are found in some other versions of Logo. The technique would be to have several lists of procedures instead of just one ALL.PROCEDURES list.

PROGRAM LISTING

In the program listing that follows, characters that are underlined represent inverse-video characters on the Atari.

```
TO DWM
DWM.1 [] [] [] [] []
END

TO DWM.1 :PROCS :COLUMNS :CHARS :ROWS :TABS :MARKED
IF NOT NAMEP "ALL.PROCEDURES [DWM.PROCLIST]
DWM.SIZE.MENU
DWM.DRAW.MENU
SETCURSOR [Ø Ø]
DWM.SHOW.CURSOR 1
DWM.MAIN.LOOP RC Ø 1 1
END
```

CREATING THE ALL.PROCEDURES LIST

```
TO DWM.PROCLIST
PR [ONE MOMENT, I'M LISTING...]
SETWRITE "D:DWM.TMP
POTS
SETWRITE []
SETREAD "D:DWM.TMP
PR [HANG ON A BIT LONGER...]
MAKE "ALL.PROCEDURES []
DWM.READ.TITLE RL
SETREAD []
ERF "D:DWM.TMP
END
```

```

TO DWM.READ.TITLE :LINE
IF EMPTYP :LINE [STOP]
DWM.READ.TITLE1 FIRST BF :LINE
DWM.READ.TITLE RL
END

TO DWM.READ.TITLE1 :NAME
IF EQUALP "DWM DWM.FIRSTPART :NAME 3 [STOP]
MAKE "ALL.PROCEDURES DWM.INSERT :NAME :ALL.PROCEDURES
END

TO DWM.INSERT :WORD :LIST
IF EMPTYP :LIST [OP FPUT :WORD []]
IF DWM.BEFORE :WORD FIRST :LIST [OP FPUT :WORD :LIST]
OP FPUT FIRST :LIST DWM.INSERT :WORD BF :LIST
END

TO DWM.BEFORE :NEW :OLD
IF EMPTYP :NEW [OP "TRUE"]
IF EMPTYP :OLD [OP "FALSE"]
IF (ASCII FIRST :NEW) < (ASCII FIRST :OLD) [OP "TRUE"]
IF (ASCII FIRST :NEW) > (ASCII FIRST :OLD) [OP "FALSE"]
OP DWM.BEFORE BF :NEW BF :OLD
END

TO DWM.FIRSTPART :WORD :NUM
IF EMPTYP :WORD [OP "1"]
IF EQUALP :NUM 1 [OP FIRST :WORD]
OP WORD FIRST :WORD DWM.FIRSTPART BF :WORD :NUM-1
END

```

PRINTING THE MENU

```

TO DWM.SIZE.MENU
MAKE "PROCS COUNT :ALL.PROCEDURES
MAKE "COLUMNS 1+INT ((:PROCS-1)/20)
MAKE "CHARS (INT 37/:COLUMNS)-2
MAKE "ROWS 1+INT ((:PROCS-1)/:COLUMNS)
MAKE "TABS DWM.SIZE.TABS 1 :CHARS+2 :COLUMNS
END

TO DWM.SIZE.TABS :COL :CHARS :COLS
IF :COLS = 0 [OP [99]]
OP FPUT :COL DWM.SIZE.TABS :COL+:CHARS :CHARS :COLS-1
END

TO DWM.DRAW.MENU
TS CT
DWM.DRAW.M1 :ALL.PROCEDURES @ 1 BF :TABS 1
SETCURSOR [0 20]
PR [- - - - P0 P: D: - - - -]
(PR CHAR 156 CHAR 157 CHAR 158 CHAR 159 ►
 [QUIT EDIT ERASE ZEROMARKS])
PR [SPACE_BAR TO TOGGLE MARK]
END

```

PROGRAMMING IDEAS

```

TO DWM.DRAW.M1 :PROCS :ROW :COL :TABS :INDEX
IF EMPTYP :PROCS [STOP]
IF MEMBERP :INDEX :MARKED [DWM.STAR]
SETCURSOR LIST :COL :ROW
TYPE DWM.SHORT FIRST :PROCS :CHARS
IF EQUALP :ROW :ROWS-1    ▶
    [DWM.DRAW.M1 BF :PROCS    ▶
        @ FIRST :TABS BF :TABS :INDEX+1]    ▶
    [DWM.DRAW.M1 BF :PROCS :ROW+1 :COL :TABS :INDEX+1]
END

TO DWM.STAR
SETCURSOR LIST :COL-1 :ROW
TYPE "*"
END

TO DWM.SHORT :NAME :CHARS
IF (COUNT :NAME)<(:CHARS+1) [OP :NAME]
OP DWM.SHORT1 :NAME :CHARS
END

TO DWM.SHORT1 :NAME :CHARS
IF :CHARS=1 [OP CHAR 171]
OP WORD FIRST :NAME DWM.SHORT BF :NAME :CHARS-1
END

```

READING COMMANDS FROM THE KEYBOARD

```

TO DWM.MAIN.LOOP :CMD :ROW :COL :INDEX
DWM.SET.CURSOR
IF :CMD = "- [DWM.UP]
IF :CMD = CHAR 28 [DWM.UP]
IF :CMD = "=" [DWM.DOWN]
IF :CMD = CHAR 29 [DWM.DOWN]
IF :CMD = "+" [DWM.LEFT]
IF :CMD = CHAR 30 [DWM.LEFT]
IF :CMD = "*" [DWM.RIGHT]
IF :CMD = CHAR 31 [DWM.RIGHT]
IF :CMD = CHAR 32 [DWM.TOGGLE.MARK]
IF :CMD = "E [DWM.CMD.2 "E [[D DWM.EDIT] [R DWM.ERASE]]]
IF :CMD = "Z [DWM.CMD.2 "Z [[M DWM.FLUSH.MARKS]]]
IF :CMD = "P    ▶
    [DWM.CMD.2 "P [[O DWM.PRINTOUT] [: DWM.PRINTER]]]
IF :CMD = "D [DWM.CMD.2 "D [[: DWM.DISKSAVE []]]
IF :CMD = "Q [SETCURSOR [Ø 23] STOP]
DWM.HIDE.CURSOR
SETCURSOR LIST (DWM.ITEM :COL :TABS)-1 :ROW
MAKE "INDEX (:COL-1)*:ROWS+:ROW+1
DWM.SHOW.CURSOR :INDEX
DWM.MAIN.LOOP RC :ROW :COL :INDEX
END

TO DWM.ITEM :NUM :LIST
IF :NUM=1 [OP FIRST :LIST]
OP DWM.ITEM :NUM-1 BF :LIST
END

```

```
TO DWM.CMD.2 :LETTER :LIST
DWM.PROMPT :LETTER
DWM.CMD.21 RC :LIST
DWM.SET.CURSOR
END

TO DWM.CMD.21 :CHAR :LIST
DWM.PROMPT CHAR 32
IF EMPTYP :LIST [TOOT 0 400 10 10 STOP]
IF EQUALP :CHAR FIRST FIRST :LIST [RUN BF FIRST :LIST STOP]
DWM.CMD.21 :CHAR BF :LIST
END

TO DWM.PROMPT :LETTER
SETCURSOR [0 23]
TYPE :LETTER
END

TO DWM.SHOW.CURSOR :INDEX
TYPE CHAR IF MEMBERP :INDEX :MARKED [170] [159]
END

TO DWM.HIDE.CURSOR
TYPE CHAR IF MEMBERP :INDEX :MARKED [42] [32]
END

TO DWM.SET.CURSOR
SETCURSOR LIST (DWM.ITEM :COL :TABS)-1 :ROW
END
```

MOVING THE POINTER

```
TO DWM.RIGHT
IF (:COL*:ROWS+:ROW+1) > :PROCS [TOOT 0 400 10 10 STOP]
MAKE "COL :COL+1
END

TO DWM.LEFT
IF :COL=1 [TOOT 0 400 10 10 STOP]
MAKE "COL :COL-1
END

TO DWM.DOWN
IF :INDEX+1 > :PROCS [TOOT 0 400 10 10 STOP]
IF :ROWS > :ROW+1 [MAKE "ROW :ROW+1 STOP]
MAKE "ROW 0
MAKE "COL :COL+1
END

TO DWM.UP
IF :ROW>0 [MAKE "ROW :ROW-1 STOP]
IF :COL=1 [TOOT 0 400 10 10 STOP]
MAKE "ROW :ROWS-1
MAKE "COL :COL-1
END
```

PROGRAMMING IDEAS***SETTING AND CLEARING MARKS***

```
TO DWM.TOGGLE.MARK
IF MEMBERP :INDEX :MARKED [DWM.UNMARK] [DWM.MARK]
END
```

```
TO DWM.MARK
MAKE "MARKED FPUT :INDEX :MARKED
END
```

```
TO DWM.UNMARK
MAKE "MARKED DWM.REMOVE :INDEX :MARKED
END
```

```
TO DWM.REMOVE :THING :LIST
IF EMPTYP :LIST [OP []]
IF EQUALP :THING FIRST :LIST [OP BF :LIST]
OP FPUT FIRST :LIST DWM.REMOVE :THING BF :LIST
END
```

```
TO DWM.FLUSH.MARKS
MAKE "MARKED []
DWM.DRAW.MENU
END
```

EDIT

```
TO DWM.EDIT
EDIT DWM.MARKLIST
DWM.DRAW.MENU
END
```

```
TO DWM.MARKLIST
IF EMPTYP :MARKED [OP :ALL.PROCEDURES]
OP DWM.MARKLIST1 :ALL.PROCEDURES 1
END
```

```
TO DWM.MARKLIST1 :LIST :NUM
IF EMPTYP :LIST [OP []]
IF MEMBERP :NUM :MARKED [OP FPUT FIRST :LIST ▶
    DWM.MARKLIST1 BF :LIST :NUM+1]
OP DWM.MARKLIST1 BF :LIST :NUM+1
END
```

PRINTOUT

```
TO DWM.PRINTOUT
CT
PO DWM.ITEM :INDEX :ALL.PROCEDURES
TYPE [TYPE A SPACE WHEN READY]
DWM.IGNORE RC
DWM.DRAW.MENU
END
```

```
TO DWM.IGNORE :CHAR
END
```

ERASE

```
TO DWM.ERASE
DWM.PROMPT (SE [REALLY ERASE] ►
    COUNT DWM.MARKLIST [PROCEDURES?])
IF NOT EQUALP RC "Y [DWM.CLEAR.PROMPT STOP]
DWM.CLEAR.PROMPT
ERASE DWM.MARKLIST
DWM.ERASE1 DWM.MARKLIST
MAKE "MARKED []
DWM.SIZE.MENU
DWM.DRAW.MENU
MAKE "ROW 0
MAKE "COL 1
END

TO DWM.ERASE1 :LIST
IF EMPTYP :LIST [STOP]
MAKE "ALL.PROCEDURES DWM.REMOVE FIRST :LIST :ALL.PROCEDURES
DWM.ERASE1 BF :LIST
END

TO DWM.CLEAR.PROMPT
DWM.PROMPT CHAR 32
REPEAT 35 [TYPE CHAR 32]
END
```

SAVE TO D: OR P:

```
TO DWM.DISKSAVE :FILE
DWM.PROMPT "FILE:
MAKE "FILE FIRST RL
CT
SETWRITE WORD "D: :FILE
DWM.SAVE DWM.MARKLIST
SETWRITE []
DWM.DRAW.MENU
END

TO DWM.PRINTER
CT
SETWRITE "P:
DWM.SAVE DWM.MARKLIST
SETWRITE []
DWM.DRAW.MENU
END

TO DWM.SAVE :LIST
IF EMPTYP :LIST [STOP]
PO FIRST :LIST
DWM.SAVE BF :LIST
END
```

A Logo Interpreter

Introduction

Suppose you were marooned on a desert island, with only your Atari and an assembler/editor cartridge. If you wanted to use Logo, you would have to write it yourself. How would you go about writing a computer language? You would have to write a program that runs the language. It is possible to write such a program, with some simplifications, in Logo itself.

Logo is an *interpreted* language. When you run your programs, Logo reads through them one instruction line at a time and executes each instruction in the line before proceeding to the next line. This is called interpreting a program.

Once you grasp the basic principles of interpreter design and operation, you could write an interpreter for any computer language, not just Logo. And you could write your interpreter in another language, like assembly language.

This project is about writing an interpreter for Logo in Atari Logo. We'll call this interpreter MLogo(for micro-Logo), to distinguish it from Atari Logo, which is an interpreter written in Atari machine language.

How to Use MLogo

To use MLogo, first initialize it (INIT), then start it (LOGO).

MLogo will prompt you for input with a ? in inverse video.

MLogo has fewer primitives than Atari Logo; among them are some list and arithmetic operations and some turtle commands.

```
?PRINT SUM 3 4
?
?PRINT FIRST BF "WALLABEE
A
?FD 100
?
```

You can write procedures in MLogo.

```
?TO POLY :SIDE :ANGLE
?FD :SIDE
?RT :ANGLE
?POLY :SIDE :ANGLE
?END
POLY DEFINED
```

MLogo is different from Atari Logo in some ways. MLogo doesn't care whether a line outputs or not. If you type:

```
SUM 3 4
```

By Jim Davis and Ed Hardebeck. An earlier version of this project was written by Henry Minsky.

the value 7 is just ignored. In Atari Logo you would get the error message:

YOU DON'T SAY WHAT TO DO WITH 7

MLogo doesn't have the STOP primitive. Every line of a user procedure is executed. It also doesn't have OP. The value of a user procedure is the value of the last line in it.

```
?TO GREET :WHO
>SE "HELLO :WHO
>END
GREET DEFINED
?PRINT GREET "ARTHUR
HELLO ARTHUR
```

If you typed this to Atari Logo, you'd get an error:

YOU DON'T SAY WHAT TO DO WITH [HELLO ARTHUR] IN GREET

Another difference is that all variables start with the empty list as their value.

```
?SHOW :NOVAL
[]
```

In Atari Logo, you'd get an error.

NOVAL HAS NO VALUE

If you try to use an undefined procedure in MLogo, you get a mysterious error message, then MLogo "crashes."

```
?ZIPPER 3
$ZIPPER HAS NO VALUE IN FSYMEVAL
```

After MLogo crashes, you are once again talking to Atari Logo. You must restart MLogo.

You may notice other differences as well. The reason for these differences is that it's difficult to implement Logo completely.

Now that you've had a chance to use MLogo and know what it does, we'll explain how it does it. The discussion, however, omits many details about interpreters.* Throughout this explanation we use the technical terms usually used by Logo implementors for describing Logo interpreters.

Interpretation Happens a Line at a Time

The structure of MLogo resembles that of Atari Logo. The normal action of Logo is to repeatedly type a prompt (?), read a line from the keyboard,

*For more information see *Structure and Interpretation of Computer Programs* by Gerald J. Sussman and Harold Abelson, MIT Press and McGraw-Hill, 1984.

and *evaluate* it. The top-level loop of MLogo is:

```
TO LOGOLOOP
  IGNORE EVLINE GET.LINE
  LOGOLOOP
END
```

The output of GET.LINE is a list of what the user typed.

EVLINE accepts a line as its input and carries out whatever instructions the line contains (for example, moves the turtle, prints a sentence, and so on).

```
TO EVLINE :LINE
  OP EVLINE1 "$NOVALUE
END

TO EVLINE1 :VALUE
  IF EMPTYP :LINE [OP :VALUE]
  OP EVLINE1 EVAL NEXT.ITEM
END
```

EVLINE1 does the actual evaluation of the instructions of a line. The easiest way to understand it is to look first at its last line.

The operation NEXT.ITEM removes the first item from the variable LINE and outputs it. Each call to NEXT.ITEM removes one item and outputs it. In this way each item is inspected in turn.

```
TO NEXT.ITEM
  OP NEXT.ITEM1 FIRST :LINE
END

TO NEXT.ITEM1 :FIRST
  MAKE "LINE BF :LINE
  OP :FIRST
END
```

EVAL takes an item, decides what kind of thing it is, and evaluates it to get its value. The value output from this call to EVAL is the input to EVLINE1 when it recurses. If there's nothing left on the line, this is the value to output. Otherwise, there's another instruction on the line.

When EVLINE calls EVLINE1, none of the line has been evaluated. If it should turn out that the line has no instructions (a blank line), then there is no value to output. \$NOVALUE is just a default value.

Here's an example of how this recursion works. Suppose you type

```
FD 60 RT 90
```

to MLogo. LOGOLOOP calls EVLINE with the list [FD 60 RT 90] as input. NEXT.ITEM outputs FD, and :LINE is [60 RT 90]. FD is passed to EVAL for evaluation.

In evaluating FD, the number 60 would be removed from the line (it is an input to FD). When EVAL stops, the value of :LINE is [RT 90]. Since this is not an empty list, evaluation would continue.

The Rules of EVAL

The value of an item is determined by these rules.

- The value of a list is just the list.
- The value of a number is just the number.
- The value of a quoted word is the word itself without the quote.
- The value of a word prefaced with a colon (called "dots") is the value of the variable.
- Otherwise the word is the name of a procedure to call. Inputs to the procedure appear after the name of the procedure.

EVAL looks at an item to see what type it is, then carries out the appropriate rule.

How EVAL Carries Out Its Rules

```
TO EVAL :ITEM
IF LISTP :ITEM [OUTPUT :ITEM]
IF NUMBERP :ITEM [OUTPUT :ITEM]
IF QUOTED? :ITEM [OUTPUT UNQUOTE :ITEM]
IF DOTTED? :ITEM [OUTPUT GET.VARIABLE.VALUE UNDOT :ITEM]
OUTPUT EVAL.CALL FSYMEVAL :ITEM
END
```

EVAL's first test is for a list. If the item isn't a list, it must be a word. The remaining tests all assume the item is some kind of word and don't include WORDP as part of the test.

The predicate QUOTED? tests whether its input is a quoted word.

```
TO QUOTED? :WORD
OUTPUT EQUALP FIRST :WORD ""
END
```

The operation UNQUOTE removes the quote and outputs the word.

```
TO UNQUOTE :WORD
OUTPUT BF :WORD
END
```

In Logo a colon ("dots") before a word is a request for the value of a variable. The predicate DOTTED? checks this case.

```
TO DOTTED? :WORD
OUTPUT EQUALP FIRST :WORD ":"
END
```

The procedure UNDOT outputs the word with the dots removed.

```
TO UNDOT :WORD
OUTPUT BF :WORD
END
```

GET.VARIABLE.VALUE outputs the value of a variable.

PROGRAMMING IDEAS

```
TO GET.VARIABLE.VALUE :WORD
IF BOUND? :WORD [OUTPUT SYMEVAL :WORD]
OUTPUT []
END
```

The predicate BOUND? checks whether the word has been assigned a value. If it has one, SYMEVAL outputs it, otherwise the value is [] . We'll explain more about this later on.

Evaluating a Procedure Call

To evaluate a procedure call, we have to know some things about the procedure being called, such as how many inputs it has and whether it's a primitive or a user procedure. Information about a procedure is kept in the *definition* of the procedure. We'll describe definitions in detail later. For now, we'll just say that the operation FSYMEVAL outputs the definition and leave it at that.

When EVAL wishes to evaluate a procedure, it passes the definition of the procedure to EVAL.CALL.

```
TO EVAL.CALL :DEFINITION
OP APPLY :DEFINITION EVAL.ARGS NARGS :DEFINITION
END
```

APPLY actually runs the procedure. It takes two inputs. The first is the definition of a procedure, the second is a list of values for the inputs. It causes the procedure to "do its thing," whatever that is, and APPLY outputs whatever the procedure outputs.

Before we can run the procedure, we have to get the values of its inputs. We usually refer to inputs as *arguments* (or *args*, for short).

The operation NARGS outputs the number of arguments this procedure expects. NARGS extracts this from the definition. (We'll see NARGS later.) This number is the input to EVAL.ARGS. EVAL.ARGS takes these inputs from the line being evaluated and evaluates each one, returning a list of the values.

To simplify MLogo, everything outputs. If commands were allowed in MLogo, as they are in Atari Logo, EVAL.CALL would have to know if an output was expected and make the proper complaint if an output was missing or an unexpected output showed up.

Inputs Require Recursive Evaluation

```
TO EVAL.ARGS :NARGS
IF EQUALP :NARGS 0 [OP []]
OUTPUT FPUT EVAL NEXT.ITEM EVAL.ARGS :NARGS - 1
END
```

EVAL.ARGS calls NEXT.ITEM to get the next item in the line being evaluated and makes a recursive call to EVAL to evaluate this item.

Here's an example. Suppose we type:

```
MAKE "DOGS 3
PRINT :DOGS
```

to MLogo. Our example begins after the MAKE, when evaluating the call to PRINT.
 EVLINE gets [PRINT :DOGS].
 NEXT.ITEM outputs PRINT.
 EVAL gets PRINT and decides it's the first word of a procedure call,
 so it calls EVAL.CALL with the definition.
 EVAL.CALL calls NARGS to get the number of arguments that PRINT
 wants, and passes this to EVAL.ARGS.
 EVAL.ARGS gets 1 as input, so it calls NEXT.ITEM, which outputs
 :DOGS. EVAL.ARGS calls EVAL to evaluate it.
 EVAL gets :DOGS as input. This is a dotted word so
 GET.VARIABLE.VALUE is called with :DOGS. It outputs the value
 DOGS, which is 3. EVAL outputs 3.
 EVAL.ARGS recurses. :NARGS - 1 is 0.
 EVAL.ARGS is called with 0, so it outputs the empty list.
 EVAL.ARGS outputs FPUT 3 [] to EVAL.CALL.
 EVAL.CALL calls APPLY with the definition and the list of values
 returned by EVAL.ARGS, in this case [3].
 APPLY invokes PRINT with an input of 3. Whatever APPLY outputs is
 what EVAL.CALL outputs.
 EVAL.CALL returns to EVAL, which returns to EVLINE1.
 Before we show how APPLY works, we'll give some details of procedure
 definitions.

Procedure Definitions

Both primitives and user procedures have definitions. Their definitions have some features in common and some differences.

In the remainder of this discussion, we refer to a primitive as an *sfun* (System FUNction), pronounced “ess-fun.” Likewise we refer to a user procedure as a *ufun* (User FUNction), pronounced “you-fun.” These are the terms usually used by Logo implementors.

Procedure definitions are kept in lists.

The first item in the list is the word SFUN or UFUN. The predicate SFUN? distinguishes sfuns from ufuncs by inspecting this item.

```
TO SFUN? :DEFINITION
OP EQUALP FIRST :DEFINITION "SFUN
END
```

The second item is the number of inputs the procedure expects (this may be zero). The operation NARGS outputs this number.

```
TO NARGS :DEFINITION
OP FIRST BF :DEFINITION
END
```

The remaining items of the list differ for the two types of procedures. We'll show you the rest of an sfun definition now and take up ufuncs later.

PROGRAMMING IDEAS

The third and final item in an sfun definition is the name of the Atari Logo procedure that implements the MLogo primitive.

The operation SFUN.FUNC outputs this procedure.

```
TO SFUN.FUNC :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END
```

If you print the names in the Logo workspace, you'll see definitions for all the MLogo primitives. All the definitions are in words beginning with \$.

```
?SHOW :$PRINT
[SFUN 1 %PRINT]
?SHOW NARGS :$PRINT
1
?SHOW SFUN.FUNC :$PRINT
%PRINT
?SHOW :$SUM
[SFUN 2 SUM]
```

Sometimes an MLogo primitive is implemented directly by an Atari Logo primitive (for example, SUM), and sometimes by a procedure (%PRINT).

The operation MAKE.SFUN.DEF makes a definition for an sfun.

```
TO MAKE.SFUN.DEF :NARGS :FUNC
OUTPUT (SE "SFUN :NARGS :FUNC")
END
```

Now we can finish discussing the evaluation of a procedure call.

APPLY *Evaluates a Procedure Call*

The first input to APPLY is the definition of a procedure to evaluate. The second input is a list of input values for that procedure. Sfuns and ufuncs are evaluated differently.

```
TO APPLY :DEFINITION :VALUES
IF SFUN? :DEFINITION [OP APPLY.SFUN :DEFINITION :VALUES]
OP APPLY.UFUN :DEFINITION :VALUES
END
```

The command APPLY.SFUN applies an sfun to its inputs by building a list as input for RUN.

```
TO APPLY.SFUN :DEF :VALUES
OUTPUT RUN SE SFUN.FUNC :DEF ( QUOTIFY :VALUES )
END
```

Suppose you typed the following to MLogo:

```
MAKE "WHO "LOWELL
PRINT :WHO
```

While evaluating the call to PRINT, APPLY.SFUN would get the inputs [SFUN 1 %PRINT] (the definition of PRINT) and LOWELL (the value of the variable WHO).

Recall that SFUN.FUNC outputs the procedure that implements the sfun. In our example it will output %PRINT.

APPLY.SFUN would call RUN with the input [%PRINT "LOWELL"]. RUN would call %PRINT on behalf of MLogo and output whatever it output.

Here's the MLogo sfun PRINT.

```
TO %PRINT :ARG
PRINT :ARG
OUTPUT :ARG
END
```

The operation QUOTIFY puts a quote in front of words that need it.

```
TO QUOTIFY :VALS
IF EMPTYP :VALS [OUTPUT []]
OUTPUT FPUT QUOTIFY1 FIRST :VALS QUOTIFY BF :VALS
END

TO QUOTIFY1 :VAL
IF NUMBERP :VAL [OP :VAL]
IF WORDP :VAL [OP WORD "" :VAL]
OUTPUT :VAL
END
```

Variable Values

The values of MLogo variables are stored in Atari Logo variables with slightly "funny" names. (This is useful for learning about how MLogo works. You can stop it and print out names. You can easily spot all MLogo variables by their names.)

The operation VSYM makes these names by adding a # to the front of the name. (The name VSYM stands for Variable SYMbol.)

```
TO VSYM :WORD
OP WORD "# :WORD
END
```

The command SET sets the value of an MLogo word, and SYMEVAL gets the value of an MLogo word. They both use VSYM to get the name of the word to use. VSYM *translates* from an MLogo name to an Atari Logo name.

```
TO SET :SYM :VAL
MAKE VSYM :SYM :VAL
END

TO SYMEVAL :SYM
OP THING VSYM :SYM
END
```

The predicate BOUND? tells whether there is a value for the word.

```
TO BOUND? :WORD
OP NAMEP VSYM :WORD
END
```

A second reason to use "funny" names for MLogo variables is that otherwise an MLogo user might set a variable with the same name as one used in the MLogo program itself. The results would be very strange. Adding the character guarantees that the names will never be the same.

Using a scheme like the one for variables, the definition of a procedure is kept in a variable whose name is the name of the procedure with a "\$" prefix. The operation FSYM (Function SYMbol) outputs the Logo variable for the definition of the MLogo procedure.

```
TO FSYM :WORD
OP WORD "$ :WORD
END
```

The command FSET sets the definition of a procedure, and the operation FSYMEVAL outputs the definition of a procedure.

```
TO FSET :SYMBOL :DEF
MAKE FSYM :SYMBOL :DEF
END

TO FSYMEVAL :NAME
OUTPUT THING FSYM :NAME
END
```

How Sfuns Are Defined

The primitives we implemented are all very similar to familiar Logo primitives. In some cases we could call Logo primitives directly. But because every MLogo sfun must output, we had to write small Atari Logo procedures for those that don't output. These procedures call the sfun, then output a value. The value may just be TRUE.

```
TO %PRINT :ARG
PRINT :ARG
OUTPUT :ARG
END

TO %MAKE :SYM :VAL
SET.VARIABLE.VALUE :SYM :VAL
OUTPUT :VAL
END

TO %FD :N
FD :N
OP "TRUE
END
```

`DEF.SFUN` defines an sfun, that is, it associates the name of an sfun with the definition.

```
TO DEF.SFUN :NAME :NARGS :FUNC
FSET :NAME MAKE.SFUN.DEF :NARGS :FUNC
END
```

All sfun procedures' names begin with a percent sign to distinguish them from procedures that are part of MLogo itself. This makes it easy to spot all the MLogo sfuns in the workspace (except those implemented directly by Atari Logo primitives).

Ufun Definitions Include Arglist and Body

Like sfuns, ufuncs have a definition, but the definition is slightly different. A user procedure consists of an *arglist* and a *body*. The arglist is a list of the input variables for the ufunc. The body is a list of the lines of the procedure. Like sfuns, ufunc definitions are lists.

If we had defined `SQUARE` by

```
TO SQUARE :N
PRINT :N
PRODUCT :N :N
END
```

... then the definition would be

```
?SHOW FSYMEVAL "SQUARE
[EUFUN 1 [N] [[PRINT :N][PRODUCT :N :N]]]
```

The arglist is `[N]`, the body is `[[PRINT :N] [PRODUCT :N :N]]`. Remember that MLogo ufunc definitions are stored by the interpreter as Atari Logo *variables*, not as Atari Logo *procedures*.

The operation `MAKE.UFUN.DEF` makes a definition for a ufunc. The operation `UFUN.ARGLIST` outputs the arglist from the definition, and the operation `UFUN.BODY` extracts the ufunc body from the definition.

```
TO MAKE.UFUN.DEF :ARGS :BODY
OP (SE "UFUN COUNT :ARGS LIST :ARGS :BODY)
END

TO UFUN.ARGLIST :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END

TO UFUN.BODY :DEFINITION
OUTPUT FIRST BF BF BF :DEFINITION
END
```

PROGRAMMING IDEAS***Evaluating a Ufun Means Evaluating the Lines of Its Body***

An ufun is a primitive, but a ufun body is a collection of lines, each requiring evaluation itself.

```
TO EVAL.BODY :LINES
OP EVAL.BODY1 "$NOVALUE :LINES
END

TO EVAL.BODY1 :VALUE :LINES
IF EMPTYP :LINES [OP :VALUE]
OP EVAL.BODY1 EVLINE FIRST :LINES BF :LINES
END
```

EVAL.BODY1 does the actual evaluation of the lines of the body. The value of the ufun is the value of the last line evaluated.

EVAL.BODY1 recurses in the same way EVLINE does. To understand it, look at the recursive call first. Each time EVAL.BODY1 recurses its first input is the value from evaluating the previous line. When the last line is evaluated, this is the value to output.

When EVAL.BODY1 is first called (from EVAL.BODY), it is passed \$NOVALUE as a first input. When first called, EVAL.BODY1 has yet to evaluate a line, so there is no value to output from the ufun. If the ufun body is empty, then \$NOVALUE is output. Otherwise there is at least one line to evaluate. EVAL.BODY1 evaluates the first line in the body, and recurses with this value and the remainder of the lines.

Ufuns Have Inputs with Names

Ufuns can have inputs. The title line (and therefore the arglist) of a ufun lists a set of variables that hold the inputs to the ufun. While a ufun is being evaluated, it can find its inputs in these variables.

For example, if you have the procedure:

```
TO GREET :WHO
PRINT SE "HELLO :WHO
END
```

and you type:

GREET "PHIL

Logo (either Atari Logo or MLogo) responds:

HELLO PHIL

Logo acts as if the value of :WHO had been set by MAKE before any of the instructions were evaluated. The effect is like what you could get by

```
?MAKE "WHO "PHIL
?PRINT SE "HELLO :PHIL
```

The difference is that after the ufun GREET is finished, the variable WHO has the same value it had before. Try it yourself if you don't already know this.

```
?MAKE "WHO [BAKED HAM]
?GREET "BOB
HELLO BOB
?SHOW :WHO
[BAKED HAM]
?MAKE "WHO "BOB
?PRINT SE "HELLO :WHO
HELLO BOB
?SHOW :WHO
BOB
```

Before EVAL.BODY can do its work, the previous values of certain variables must be saved before those variables receive new values. The input variables hold their values only for the duration of evaluation of the ufun, and then they have their old values restored.

The process of setting and restoring of values is referred to as *binding* the variables. Making binding work properly is one of the most difficult parts of writing an interpreter.

APPLY.UFUN is called by APPLY to evaluate a ufun. See APPLY.SFUN for comparison.

```
TO APPLY.UFUN :DEF :VALUES
BIND.ARGS UFUN.ARGLIST :DEF :VALUES
OP CLEANUP EVAL.BODY UFUN.BODY :DEF
END
```

BIND.ARGS saves the old values of variables in the arglist, then sets the new values.

EVAL.BODY evaluates the forms of the body and outputs a value.

CLEANUP restores variables to their previous values. It outputs the value of the ufun.

How Binding Is Implemented

BIND.ARGS does two things. It saves old values and it sets new ones. It cooperates with CLEANUP, which restores the old values. These two procedures cooperate through the global variable BIND.STACK, which is where BIND.ARGS saves the values and CLEANUP finds them.

```
TO BIND.ARGS :ARGLIST :VALUES
PUSH "BIND.STACK BIND.FRAME :ARGLIST
SET.ARGS :ARGLIST :VALUES
END
```

The saved values are referred to collectively as a *bind frame*. The operation BIND.FRAME builds a bind frame for the variables in the arglist. BIND.ARGS uses PUSH to save this frame in the shared variable BIND.STACK, then calls SET.ARGS to set the new values.

```

TO SET.ARGS :NAMES :VALUES
IF EMPTYP :NAMES [STOP]
SET FIRST :NAMES FIRST :VALUES
SET.ARGS BF :NAMES BF :VALUES
END

```

SET.ARGS simply recurses through the argument list and the values. There is a one-to-one correspondence between the argument list and the values list. For each input there is a value.

After a ufun is evaluated it outputs a value, and this is the value that APPLY.UFUN should output as the value of the ufun it was asked to apply. But first the bound variables must be unbound. This is the purpose of CLEANUP.

```

TO CLEANUP :VALUE
UNBIND
OP :VALUE
END

```

CLEANUP's input is the output of the ufun. It holds onto this value while UNBIND undoes the binding, then returns the held value. UNBIND is explained later.

A bind frame enables the interpreter to restore the values of the input variables of a single ufun call. But a ufun can call other ufuns. We need one bind frame for each ufun call. There will be as many bind frames as the depth of calling. Bind frames are created as calls occur and cleaned up as the call returns. The most recently added frame is always the one to clean up.

We need to keep track of all these bind frames and ensure we bind and unbind in the same order calls and returns are made. To do this, we use a *stack*.

The Concept of a Stack

A stack is a method of arranging data. You can think of it as a pile of papers on a desk. Only the topmost sheet is visible (if the stack is neat) because it covers the others. If you add another sheet to the pile, it becomes the topmost. You can only touch the top sheet. If you remove it, a new top sheet is exposed.

This order of accessing is sometimes referred to as "Last In, First Out," because the last item added to the stack is the first one that can be removed.

Stacks Are Implemented by Lists

Most computers have machine instructions to implement stacks. But since we wrote MLogo in Logo and not in machine language, we had to implement stacks. We decided to use Logo lists to hold stacks, and to put the top of the stack at the front of the list so that we could use FIRST to get the top item on the stack and FPUT to add a new one.

PUSH puts something on the top of a stack. It takes two inputs. The first is the name of the word containing the stack, the second is the item to add to the stack.

```
TO PUSH :STACK :ITEM
MAKE :STACK FPUT :ITEM THING :STACK
END
```

PUSH makes a new list by adding the item to the old contents of the stack. This new list is assigned to the variable holding the stack.

The operation POP outputs the top value on the stack. This value is removed from the stack.

```
TO POP :STACK
OP POP1 :STACK THING :STACK
END

TO POP1 :STACK :LIST
MAKE :STACK BF :LIST
OP FIRST :LIST
END
```

The input to the operation POP is the variable holding the stack. THING of this variable outputs its value—the list holding the stack. The first item in the list is the item to output. Before outputting it, POP1 sets the stack variable to hold the BF of the list, thus removing the top item from the stack.

This example shows how stacks work.

```
?MAKE "STACK []
?PUSH "STACK 9
?SHOW :STACK
[9]
?PUSH "STACK 5
?PUSH "STACK 2
?SHOW :STACK
[2 5 9]
?PRINT POP "STACK
2
?SHOW :STACK
[5 9]
```

Bind Frame in Detail

A bind frame is a list of bindings. Each binding is a list of a name and a value. The name is the name of a variable that must be saved, and the value is the value it had at the time it was saved.

A typical bind frame might be

```
[[A 3][NAME [JAMES ALLEN]]]
```

This bind frame is holding two variable bindings, for A and NAME.

BIND.FRAME makes a bind frame. Its input is the argument list of a ufun. Each input is a variable whose value must be saved.

```
TO BIND.FRAME :ARGLIST
IF EMPTYP :ARGLIST [OP []]
OP FPUT BIND.ARGS FIRST :ARGLIST BIND.FRAME BF :ARGLIST
END
```

PROGRAMMING IDEAS

BIND.FRAME recurses through the list, collecting one binding for each variable. The operation BIND.ARG makes a binding for a variable. It outputs a list of the variable name and the current value of the variable.

```
TO BIND.ARG :NAME
OP LIST :NAME GET.VARIABLE.VALUE :NAME
END
```

UNBIND pops a single frame off the stack and passes it to UNBIND.ARGS, which acts like BIND.FRAME in reverse, restoring each saved value in the frame.

```
TO UNBIND
UNBIND.ARGS POP "BIND.STACK
END

TO UNBIND.ARGS :FRAME
IF EMPTYP :FRAME [STOP]
UNBIND.ARG FIRST :FRAME
UNBIND.ARGS BF :FRAME
END

TO UNBIND.ARG :PAIR
SET FIRST :PAIR FIRST BF :PAIR
END
```

The interpreter's top-level procedure, LOGO, initializes BIND.STACK to hold an empty list.

```
TO LOGO
MAKE "BIND.STACK []
LOGOLOOP
END
```

The Sfun TO Is Harder Than Others

In Logo the primitive TO treats its inputs differently from all other sfuns. It does not evaluate them. The first input to TO is the name of the procedure to define. The rest of the inputs are the names of the inputs of the procedure being defined. These are written with dots to remind you that they are the inputs.

```
?TO SQUARE :A
≥PRODUCT :A :A
≥END
SQUARE DEFINED
```

TO manages the trick of not evaluating its inputs by lying to the evaluator about its number of inputs. It says it takes none but then goes and takes them off LINE (where the current line is kept) by itself. EVAL.ARGS evaluates the arguments as it collects them. This trick also lets TO take as many arguments as are present on the line.

The TO definition is:

```
[SFUN Ø %TO]
```

The procedures that implement it are

```
TO %TO
OP T01 NEXT.ITEM GATHER.ARGS
END

TO T01 :NAME :ARGLIST
DEF.UFUN :NAME :ARGLIST READ.BODY
PRINT (SE :NAME "DEFINED")
OP "TRUE
END
```

The GATHER.ARGS operation pops the input names directly off LINE and removes the dots.

```
TO GATHER.ARGS
IF EMPTYP :LINE [OP []]
OP FPUT UNDOT NEXT.ITEM GATHER.ARGS
END
```

DEF.UFUN takes as inputs the name of the procedure to define, a list of its arguments, and its body, which is a list of the lines that make up the procedure.

```
TO DEF.UFUN :NAME :ARGS :BODY
FSET :NAME MAKE.UFUN.DEF :ARGS :BODY
END
```

MAKE.UFUN.DEF makes the actual definition. We have already seen it. READ.BODY reads an entire ufun body, prompting with > before reading each line.

```
TO READ.BODY
OP READ.BODY1 READ.LINE
END

TO READ.BODY1 :LINE
IF EQUALP :LINE [END] [OP []]
OP FPUT :LINE READ.BODY1 READ.LINE
END
```

READ.BODY1 recurses, reading a line each time, until it gets a line END.

Reading Things You Type

Both TO and the LOGOLOOP need to get typein from the user. They don't want empty lines as input. Each has its own prompt character. They can both share INPUT.LINE.

```
TO GET.LINE
OP INPUT.LINE "?"
END
```

PROGRAMMING IDEAS

```

TO READ.LINE
OP INPUT.LINE "≥
END

TO INPUT.LINE :PROMPT
TYPE :PROMPT
OP INPUT.LINE1 RL
END

TO INPUT.LINE1 :INPUT
IF NOT EMPTYP :INPUT [OP :INPUT]
OP INPUT.LINE :PROMPT
END

```

INPUT.LINE1 recurses until the user types a line that isn't empty.

Some Improvements

Here are some modifications to MLogo to make it more like Atari Logo. We didn't include them in MLogo because we wanted to keep it simple to explain. If you want to have these extra features, you can type in the following procedures.

First, a synonym for PRINT.

```

DEF.SFUN "PR 1 "%PRINT

Here's the sfun P0:

TO %P0 :NAME
TYPE SE "TO :NAME
P0.ARGS UFUN.ARGLIST FSYMEVAL :NAME
P0.BODY UFUN.BODY FSYMEVAL :NAME
OP :NAME
END

TO P0.ARGS :ARGLIST
IF EMPTYP :ARGLIST [PRINT [] STOP]
TYPE "\"
TYPE ":" 
TYPE FIRST :ARGLIST
P0.ARGS BF :ARGLIST
END

TO P0.BODY :LINES
IF EMPTYP :LINES [PR "END STOP"]
PRINT FIRST :LINES
P0.BODY BF :LINES
END

```

P0 is by far the longest sfun yet, because there is no useful Atari Logo primitive for it. The Atari Logo primitive P0 prints out Atari Logo user procedures, which are not stored like MLogo user procedures.

The procedure P0.ARGS prints each word in the argument list,

preceded by a space and a colon. (In the second line of PO.ARGS, a space appears after the backslash even though you can't tell from this listing.)

To add OP to MLogo we have to change EVAL.BODY and EVAL.BODY1. As is, each line is always evaluated. By adding a flag variable OP? we can cause evaluation to stop.

Here's the sfun OP.

```
TO %OP :VALUE
MAKE "OP :VALUE
MAKE "OP? "TRUE
OP "TRUE
END

DEF.SFUN "OP 1 "%OP
```

The input to OP is the value to output. This value is stored in the variable OP for reference by the evaluator. %OP sets the flag OP?, which causes the evaluation of the body to stop.

We have to modify the evaluator to check these flags.

```
TO EVAL.BODY :LINES
OP EVAL.BODY1 :LINES "FALSE
END

TO EVAL.BODY1 :LINES :OP?
IF EMPTYP :LINES [OP "$NOVALUE"]
IGNORE EVLINE FIRST :LINES
IF :OP? [OP :OP]
OP EVAL.BODY1 BF :LINES "FALSE
```

PROGRAM LISTING

TO LOGO	TO NEXT.ITEM1 :FIRST
MAKE "BIND.STACK []	MAKE "LINE BF :LINE
LOGOLOOP	OP :FIRST
END	END
TO LOGOLOOP	TO EVAL :ITEM
IGNORE EVLINE GET.LINE	IF LISTP :ITEM [OUTPUT :ITEM]
LOGOLOOP	IF NUMBERP :ITEM [OUTPUT :ITEM]
END	IF QUOTED? :ITEM [OUTPUT UNQUOTE ►
TO EVLINE :LINE	:ITEM]
OP EVLINE1 "\$NOVALUE	IF DOTTED? :ITEM [OUTPUT ►
END	GET.VARIABLE.VALUE UNDOT :ITEM]
TO EVLINE1 :VALUE	OUTPUT EVAL.CALL FSYMEVAL :ITEM
IF EMPTYP :LINE [OP :VALUE]	END
OP EVLINE1 EVAL NEXT.ITEM	TO QUOTED? :WORD
END	OUTPUT EQUALP FIRST :WORD ""
TO NEXT.ITEM	END
OP NEXT.ITEM1 FIRST :LINE	TO UNQUOTE :WORD
END	OUTPUT BF :WORD
	END

```

TO DOTTED? :WORD
OUTPUT EQUALP FIRST :WORD ":
END

TO UNDOT :WORD
OUTPUT BF :WORD
END

TO GET.VARIABLE.VALUE :WORD
IF BOUND? :WORD [OUTPUT SYMEVAL :WORD]
OUTPUT []
END

TO VSYM :WORD
OP WORD "# :WORD
END

TO SET :SYM :VAL
MAKE VSYM :SYM :VAL
END

TO SYMEVAL :SYM
OP THING VSYM :SYM
END

TO BOUND? :WORD
OP NAMEP VSYM :WORD
END

TO FSYM :WORD
OP WORD "$ :WORD
END

TO FSYMEVAL :NAME
OUTPUT THING FSYM :NAME
END

TO FSET :SYMBOL :DEF
MAKE FSYM :SYMBOL :DEF
END

TO EVAL.CALL :DEFINITION
OP APPLY :DEFINITION EVAL.ARGS NARGS ▶
:DEFINITION
END

TO MAKE.SFUN.DEF :NARGS :FUNC
OUTPUT (SE "SFUN :NARGS :FUNC)
END

TO MAKE.UFUN.DEF :ARGS :BODY
OP (SE "UFUN COUNT :ARGS LIST :ARGS ▶
:BODY)
END

TO SFUN? :DEFINITION
OP EQUALP FIRST :DEFINITION "SFUN
END

TO NARGS :DEFINITION
OP FIRST BF :DEFINITION
END

TO SFUN.FUNC :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END

TO UFUN.ARGLIST :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END

TO UFUN.BODY :DEFINITION
OUTPUT FIRST BF BF BF :DEFINITION
END

TO APPLY :DEFINITION :VALUES
IF SFUN? :DEFINITION [OP APPLY.SFUN ▶
:DEFINITION :VALUES]
OP APPLY.UFUN :DEFINITION :VALUES
END

TO APPLY.SFUN :DEF :VALUES
OUTPUT RUN SE SFUN.FUNC :DEF ( QUOTIFY ▶
:VALUES )
END

TO QUOTIFY :VALS
IF EMPTYP :VALS [OUTPUT []]
OUTPUT FPUT QUOTIFY1 FIRST :VALS ▶
QUOTIFY BF :VALS
END

TO QUOTIFY1 :VAL
IF NUMBERP :VAL [OP :VAL]
IF WORDP :VAL [OP WORD "" :VAL]
OUTPUT :VAL
END

TO EVAL.ARGS :NARGS
IF EQUALP :NARGS 0 [OP []]
OUTPUT FPUT EVAL NEXT.ITEM EVAL.ARGS ▶
:NARGS - 1
END

TO APPLY.UFUN :DEF :VALUES
BIND.ARGS UFUN.ARGLIST :DEF :VALUES
OP CLEANUP EVAL.BODY UFUN.BODY :DEF
END

```

```

TO BIND.ARGS :ARGLIST :VALUES
PUSH "BIND.STACK BIND.FRAME :ARGLIST
SET.ARGS :ARGLIST :VALUES
END

TO SET.ARGS :NAMES :VALUES
IF EMPTYP :NAMES [STOP]
SET FIRST :NAMES FIRST :VALUES
SET.ARGS BF :NAMES BF :VALUES
END

TO CLEANUP :VALUE
UNBIND
OP :VALUE
END

TO BIND.FRAME :ARGLIST
IF EMPTYP :ARGLIST [OP []]
OP FPUT BIND.ARGS FIRST :ARGLIST ▶
BIND.FRAME BF :ARGLIST
END

TO BIND.ARG :NAME
OP LIST :NAME GET.VARIABLE.VALUE :NAME
END

TO UNBIND
UNBIND.ARGS POP "BIND.STACK
END

TO UNBIND.ARGS :FRAME
IF EMPTYP :FRAME [STOP]
UNBIND.ARGS FIRST :FRAME
UNBIND.ARGS BF :FRAME
END

TO UNBIND.ARG :PAIR
SET FIRST :PAIR FIRST BF :PAIR
END

TO EVAL.BODY :LINES
OP EVAL.BODY1 "$NOVALUE :LINES
END

TO EVAL.BODY1 :VALUE :LINES
IF EMPTYP :LINES [OP :VALUE]
OP EVAL.BODY1 EVLINE FIRST :LINES BF ▶
:LINES
END

TO %PRINT :ARG
PRINT :ARG
OUTPUT :ARG
END

TO %IF :PRED :C1 :C2
IF :PRED [OP EVLINE :C1] [OP EVLINE ▶
:C2]
END

TO %MAKE :SYM :VAL
SET :SYM :VAL
OUTPUT :VAL
END

TO %FD :N
FD :N OP "TRUE
END

TO %RT :N
RT :N OP "TRUE
END

TO %CS
CS OP "TRUE
END

TO %TO
OP T01 NEXT.ITEM GATHER.ARGS
END

TO T01 :NAME :ARGLIST
DEF.UFUN :NAME :ARGLIST READ.BODY
PRINT (SE :NAME "DEFINED")
OP "TRUE
END

TO GATHER.ARGS
IF EMPTYP :LINE [OP []]
OP FPUT UNDOT NEXT.ITEM GATHER.ARGS
END

TO DEF.SFUN :NAME :NARGS :FUNC
FSET :NAME MAKE.SFUN.DEF :NARGS :FUNC
END

TO DEF.UFUN :NAME :ARGS :BODY
FSET :NAME MAKE.UFUN.DEF :ARGS :BODY
END

TO INIT
INITPRIMS
END

TO DEF.SSFUN :NAME :NARGS
DEF.SFUN :NAME :NARGS :NAME
END

```

```

TO INITPRIMS
DEF.SSFUN "SUM 2
DEF.SSFUN "PRODUCT 2
DEF.SSFUN "EMPTYP 1
DEF.SSFUN "EQUALP 2
DEF.SSFUN "LIST 2
DEF.SSFUN "FIRST 1
DEF.SSFUN "BF 1
DEF.SSFUN "SE 2
DEF.SSFUN "WORD 2
DEF.SFUN "RT 1 "%RT
DEF.SFUN "FD 1 "%FD
DEF.SFUN "CS Ø "%CS
DEF.SFUN "THING 1 "GET.VARIABLE.VALUE
DEF.SFUN "MAKE 2 "%MAKE
DEF.SFUN "PRINT 1 "%PRINT
DEF.SFUN "IF 3 "%IF
DEF.SFUN "TO Ø "%TO
END

TO READ.BODY
OP READ.BODY1 READ.LINE
END

TO READ.BODY1 :LINE
IF EQUALP :LINE [END] [OP []]
OP FPUT :LINE READ.BODY1 READ.LINE
END

TO GET.LINE
OP INPUT.LINE "?"
END

TO READ.LINE
OP INPUT.LINE ">"
END

TO INPUT.LINE :PROMPT
TYPE :PROMPT
OP INPUT.LINE1 RL
END

TO INPUT.LINE1 :INPUT
IF NOT EMPTYP :INPUT [OP :INPUT]
OP INPUT.LINE :PROMPT
END

TO PUSH :STACK :ITEM
MAKE :STACK FPUT :ITEM THING :STACK
END

TO POP :STACK
OP POP1 :STACK THING :STACK
END

TO POP1 :STACK :LIST
MAKE :STACK BF :LIST
OP FIRST :LIST
END

TO IGNORE :X
END

```

Map

Have you ever written a procedure like this:

```

TO LINEPRINT :LIST
IF EMPTYP :LIST [STOP]
PRINT FIRST :LIST
LINEPRINT BF :LIST
END

```

By Brian Harvey.

Or like this:

```
TO TUNE :NOTES
IF EMPTYP :NOTES [STOP]
TOOT Ø FIRST :NOTES 15 3Ø
TUNE BF :NOTES
END
```

Or like this:

```
TO FLASH :COLORS
IF EMPTYP :COLORS [STOP]
WAIT 6Ø
SETBG FIRST :COLORS
FLASH BF :COLORS
END
```

All of these procedures have a common pattern. They go through a list, doing something with each member of the list, and then stop when they get to the end of the list. The procedures *differ* in what they do with the members of their input list. In one case it's a list of things to print; in the second it's a list of frequencies of musical notes; in the third it's a list of color numbers. But they all share this structure:

```
TO procedure.name :LIST
IF EMPTYP :LIST [STOP]
do.something.with FIRST :LIST
procedure.name BF :LIST
END
```

You can think of this skeleton procedure as a template for many procedures that do similar work for you.

Mapping Commands

You can write a single procedure that does all these things. What's special about it is that it is a *general* tool that can apply *any* procedure to each member of a list. This general process is called *mapping* the procedure over the list, so we call this general procedure MAP. Here are some examples.

```
?MAP [PRINT] [VANILLA CHOCOLATE GINGER LEMON]
VANILLA
CHOCOLATE
GINGER
LEMON
?

?MAP [PRINT FIRST] [VANILLA CHOCOLATE GINGER LEMON]
V
C
G
L
?
```

PROGRAMMING IDEAS

```
?MAP [TYPE FIRST] [EVERY GOOD BOY DOES FINE]
EGBDF?
```

The first example of using MAP is equivalent to the procedure LINEPRINT with which we started this discussion. The first input to MAP says what you want to do to each member of the input list (in this example, PRINT it). The second input is the list over which you are mapping. So the instruction

```
MAP [PRINT] [THIS IS A LIST]
```

is equivalent to

```
LINEPRINT [THIS IS A LIST]
```

Here are the procedure definitions.

```
TO MAP :TEMPLATE :LIST
IF EMPTYP :LIST [STOP]
RUN LPUT QUOTED FIRST :LIST :TEMPLATE
MAP :TEMPLATE BF :LIST
END

TO QUOTED :THING
IF LISTP :THING [OP :THING]
OP WORD "" :THING
END
```

You can use MAP with more complicated instructions than just PRINT. In the second example, the first input to MAP is the list [PRINT FIRST]. This example works as if we'd written a special procedure like this:

```
TO FIRST.PRINT :LIST
IF EMPTYP :LIST [STOP]
PRINT FIRST FIRST :LIST
FIRST.PRINT BF :LIST
END
```

We can use MAP to obtain the same effect as the FLASH procedure we showed earlier.

```
MAP [WAIT 60 SETBG] [0 88 74 7]
```

To get the same effect as our TUNE procedure, we have to work a little harder. The problem is that the frequency input to TOOT comes in the middle of the instruction, like this:

```
TOOT @ FIRST :NOTES 15 30
```

MAP expects to put each member of the list at the end of an instruction, not in the middle. What we have to do is write an auxiliary procedure that takes the frequency as a single input:

```
TO NOTE :FREQ
TOOT @ :FREQ 15 30
END
```

Now we can use MAP to get the same effect as TUNE:

```
MAP [NOTE] [440 880 220 440]
```

How It Works

What makes it possible for MAP to be a general-purpose tool instead of a procedure for a specific purpose is its use of Logo's RUN command. This replaces the specific commands like PRINT or TOOT or SETBG in the earlier examples. The input to RUN is a Logo instruction that is assembled out of two parts: the *template*, which is the first input to MAP, and one member of the list, which is MAP's second input.

Let's look at an example. If we say

```
MAP [PRINT] [VANILLA CHOCOLATE GINGER LEMON]
```

then MAP has to carry out these four instructions:

```
PRINT "VANILLA  
PRINT "CHOCOLATE  
PRINT "GINGER  
PRINT "LEMON
```

Each of these four instructions is made by combining the template [PRINT] with one member of [VANILLA CHOCOLATE GINGER LEMON]. The combination is made using LPUT, which adds the list member at the end of the template. For example, the expression

```
LPUT ""VANILLA [PRINT]
```

outputs the list

```
[PRINT "VANILLA]
```

The procedure MAP itself has much the same pattern as the examples at the beginning of this discussion. The first instruction inside MAP is the IF EMPTYP stop rule; the last instruction is the recursive use of MAP with the BUTFIRST of the input list. Compare MAP with LINEPRINT, for example:

```
TO LINEPRINT :LIST  
IF EMPTYP :LIST [STOP]  
PRINT FIRST :LIST  
LINEPRINT BUTFIRST :LIST  
END  
  
TO MAP :TEMPLATE :LIST  
IF EMPTYP :LIST [STOP]  
RUN LPUT QUOTED FIRST :LIST :TEMPLATE  
MAP :TEMPLATE BUTFIRST :LIST  
END
```

PROGRAMMING IDEAS

One possibly confusing detail in MAP has to do with quotation marks. Notice that if you want Logo to print the word VANILLA, you can't say

PRINT VANILLA		Wrong!
---------------	--	--------

but must quote the input to PRINT:

```
PRINT "VANILLA
```

To assemble this instruction, the first input to LPUT must be the word "VANILLA, including the quotation mark as part of the word. The procedure QUOTED is used by MAP to supply the needed quotation marks.

Mapping Operations

So far, the templates we've used have been *commands*. That is, they have been Logo procedures that do something external, like print something, make a sound, or change the color of the screen. An even more powerful facility is to map *operations* over a list, producing (outputting) a new list of the results. Perhaps an example will make this clearer.

```
?SHOW MAP.LIST [FIRST] [THIS IS A LIST]
[T I A L]
?

?SHOW MAP.LIST [SQRT] [1 2 3 4]
[1 1.414214 1.732051 2]
?
```

Like MAP, MAP.LIST generalizes a common pattern of Logo procedures. The examples here could have been written as special-purpose procedures this way:

```
TO EVERY.FIRST :LIST
IF EMPTYP :LIST [OP []]
OP FPUT (FIRST FIRST :LIST) (EVERY.FIRST BF :LIST)
END

TO EVERY.SQRT :LIST
IF EMPTYP :LIST [OP []]
OP FPUT (SQRT FIRST :LIST) (EVERY.SQRT BF :LIST)
END
```

MAP.LIST is an operation. Its output is a list of the same length as its second input. Each member of the output list is the result of applying the template to a member of the input list.

MAP.LIST itself follows the same pattern it generalizes.

```
TO MAP.LIST :TEMPLATE :LIST
IF EMPTYP :LIST [OP []]
OP FPUT (RUN LPUT QUOTED FIRST :LIST :TEMPLATE)
        (MAP.LIST :TEMPLATE BF :LIST)
END
```

Here the first input to FPUT is the same expression that was used to assemble the instructions in MAP.

An example of using MAP.LIST to apply a procedure to each word of a sentence is this program to translate a sentence into Pig Latin.

```
TO PIGLATIN :WORD
IF MEMBERP FIRST :WORD [A E I O U Y] [OP WORD :WORD "AY"]
OP PIGLATIN WORD BF :WORD FIRST :WORD
END

?PRINT PIGLATIN 'HELLO
ELLOHAY
?PRINT MAP.LIST [PIGLATIN] [THIS IS GREEK TO ME]
ISTHAY ISAY EEKGRAY OTAY EMAY
?
```

Mapping Over Words

In Logo, we can assemble letters into words, just as we can assemble words into lists. We can extend the idea of mapping to apply a procedure to each letter of a word.

```
TO MAP.WORD :TEMPLATE :WORD
IF EMPTYP :WORD [OP ""]
OP WORD (RUN LPUT QUOTED FIRST :WORD :TEMPLATE)
        (MAP.WORD :TEMPLATE BF :WORD)
END
```

MAP.WORD is the same as MAP.LIST, except that it uses WORD instead of FPUT as the combining operation, and it builds onto an empty word instead of an empty list.

Here is an example of how to use MAP.WORD. Suppose you want to print a word in inverse video (black on white). On the Atari computer, to print any character in inverse video, you must add 128 to the code that represents that character.

```
?PRINT MAP.WORD [CHAR 128+ASCII] 'HELLO
```

```
HELLO
?
```

PROGRAMMING IDEAS

If we put this into a procedure, we can print an entire sentence with each word inverted by combining MAP.WORD and MAP.LIST.

```
TO INVERT :WORD
OP MAP.WORD [CHAR 128+ASCII] :WORD
END

?PRINT MAP.LIST [INVERT] [THIS IS A TEST.]
```

*List Reduction*

There is one more way in which an operation can be applied to the members of a list. Consider an operation with two inputs, like SUM or PRODUCT. It is often convenient to be able to add up all the numbers in a list, or multiply them together. Of course, as in the earlier situations, we could write special-purpose procedures.

```
TO ADD :LIST
IF EMPTYP :LIST [OP Ø]
OP SUM (FIRST :LIST) (ADD BF :LIST)
END

TO MULTIPLY :LIST
IF EMPTYP :LIST [OP 1]
OP PRODUCT (FIRST :LIST) (MULTIPLY BF :LIST)

?PR ADD [1 2 3 4]
10
?PR MULTIPLY [1 2 3 4]
24
?
```

What we'd like to do is produce a general tool for these situations.

```
?PR REDUCE [SUM] [1 2 3 4]
10
```

```
?PR REDUCE [PRODUCT] [1 2 3 4]
24
?
```

There is one slight complication that prevents REDUCE from following exactly the pattern of ADD and MULTIPLY. The problem is that each of those procedures knows about the *identity element* for the corresponding operation. The identity element is the value to start with when the input list is empty: 0 for SUM, 1 for PRODUCT. To make REDUCE a general tool, we want to avoid building this kind of information into it. The solution is to apply REDUCE recursively only down to the point where there are *two* members remaining in the input list, then just apply the template to those two. The resulting procedure is a little messy, but if you go through it carefully you'll see that it's really much like the mapping procedures we've used before.

```
TO REDUCE :TEMPLATE :LIST
IF EMPTYP BF :LIST [OP FIRST :LIST]
IF EMPTYP BF BF :LIST [OP RUN SE :TEMPLATE
  LIST (QUOTED FIRST :LIST) (QUOTED FIRST BF :LIST)]
OP RUN SE :TEMPLATE LIST (QUOTED FIRST :LIST)
  (QUOTED REDUCE :TEMPLATE BF :LIST)
END
```

Here are more examples of how REDUCE can be used.

```
?PRINT REDUCE [WORD] [A B C D]
ABCD
?

TO REVERSE :LIST
OP REDUCE [LPUT] LPUT [] :LIST
END

?SHOW REVERSE [A B C D]
[D C B A]
?
```

SUGGESTIONS

- You could modify these procedures so that the list members could be inserted anywhere in the template, instead of only at the end. For example, the music example that earlier required writing an auxiliary procedure NOTE could instead be written

```
MAP [TOOT 0 ? 15 30] [440 880 220 440]
```

where the question mark indicates the position in the template into which the members of the input list are placed.

- The general name for doing something over and over is *iteration*. Mapping is a particular kind of iteration, based on using the members of a list, one after the other. Other kinds of iteration can also be

invented using the RUN primitive. For example, here is an iteration procedure that tests a predicate to control the repetition.

```
TO WHILE :PREDICATE :COMMAND
IF NOT RUN :PREDICATE [STOP]
RUN :COMMAND
WHILE :PREDICATE :COMMAND
END

?CS
?WHILE [HEADING < 270] [FD 10 RT 10]
?
```

You might try to write a procedure to create numeric iteration.

```
?STEP "NUM 3 7 [PRINT :NUM ° :NUM]
9
16
25
36
49
?
```

- Use MAP.WORD and MAP.LIST to implement a *substitution cipher*. A cipher is a technique for protecting secret messages by transforming each letter into some other form. (Ciphers are sometimes called *codes*, but, strictly speaking, a code is a technique that transforms a word by looking it up in a dictionary, rather than by manipulating it letter by letter. A foreign language is like a code.) Write a procedure that takes a single letter as input and outputs some secret representation of the input letter. Then you can encipher a word by applying MAP.WORD to it, and you can encipher a sentence by applying MAP.LIST to encipher each word. The example of inverse video works like a cipher, although of course the result isn't very secret.
- MAP.LIST uses FPUT to accumulate the results for each member of the input list, and MAP.WORD uses WORD to accumulate its results. Logo has other accumulating operations: SE, LIST, and LPUT. Try writing versions of MAP.LIST that use each of these. Are any of them useful?
- Here is a tricky example.

```
TO FLATTEN :LIST
IF WORDP :LIST [OP :LIST]
OP REDUCE [SE] MAP.LIST [FLATTEN] :LIST
END

?SHOW FLATTEN [[THIS IS] [A [LIST]]]
[THIS IS A LIST]
?
```

FLATTEN combines iteration over a list, list reduction, and recursion, since the template input to MAP.LIST uses FLATTEN itself. The procedure converts any list into a *flat list*, one that has only words as

members. Can you see why both REDUCE and MAP.LIST must be used? Compare the result of FLATTEN to these:

```
SHOW REDUCE [SE] [[THIS IS] [A [LIST]]]
SHOW MAP.LIST [FLATTEN] [[THIS IS] [A [LIST]]]
```

PROGRAM LISTING

```
TO MAP :TEMPLATE :LIST
IF EMPTYP :LIST [STOP]
RUN LPUT QUOTED FIRST :LIST :TEMPLATE
MAP :TEMPLATE BF :LIST
END

TO QUOTED :THING
IF LISTP :THING [OP :THING]
OP WORD "" :THING
END

TO MAP.LIST :TEMPLATE :LIST
IF EMPTYP :LIST [OP []]
OP FPUT (RUN LPUT QUOTED FIRST :LIST ▶
:TEMPLATE) (MAP.LIST :TEMPLATE BF ▶
:LIST)
END

TO MAP.WORD :TEMPLATE :WORD
IF EMPTYP :WORD [OP "]
OP WORD (RUN LPUT QUOTED FIRST :WORD ▶
:TEMPLATE) (MAP.WORD :TEMPLATE BF ▶
:WORD)
END

TO REDUCE :TEMPLATE :LIST
IF EMPTYP BF :LIST [OP FIRST :LIST]
IF EMPTYP BF BF :LIST [OP RUN SE ▶
:TEMPLATE LIST (QUOTED FIRST ▶
:LIST) (QUOTED FIRST BF :LIST)]
OP RUN SE :TEMPLATE LIST (QUOTED FIRST ▶
:LIST) (QUOTED REDUCE :TEMPLATE ▶
BF :LIST)
END
```

Mergesort

People often want to use computers to *sort* information of various kinds. For example, you may want to list your friends' addresses in alphabetical order, or you may want the same information arranged in order of their birthdays to remind you when to send cards. Programmers have invented many different techniques to solve the sorting problem. Generally, the methods that are easy to understand tend to run slowly, while the faster methods are rather complicated. Here is a method that is medium-fast and medium-tricky. Its name is *mergesort*.

In Logo, we'll represent the information we want to sort as a list of items. The general strategy is this:

1. Divide the list into two smaller parts.
2. Sort each part separately.
3. Merge the two sorted lists into one big sorted list.

This may not seem like much of a strategy, because we are still left with the problem of sorting the smaller lists in the second step. But the clever part

is that if we keep applying the strategy to the smaller lists, eventually we get lists with just one member, and we can simply declare these lists sorted.

Here's a specific example. To make it easy to read, we'll sort a list of numbers in size order. Start with this list:

[14 3 27 1 10 5]

Divide it into two smaller lists.

[14 3 27] [1 10 5]

Now sort the first of the smaller lists. To do that, divide it into two smaller lists.

[14 3] [27]

Now sort the first of *these* lists, again by dividing it into two smaller lists.

[14] [3]

Each of these lists has only one member, so each is already sorted. Now we merge them to get

[3 14]

Now we can merge this list with its "partner," which is the list [27]. The result is

[3 14 27]

The next step is to sort the "partner" of *this* list, namely the list [1 10 5]. This also involves dividing it into smaller lists, as before. To make this example shorter, we'll skip the steps of sorting the list [1 10 5]. Finally we are left with two sorted lists:

[3 14 27] [1 5 10]

The last step is to merge these:

[1 3 5 10 14 27]

Dividing a List into Two Parts

The first step in the sorting process is to divide a list into two parts. To do that, we can use procedures FIRST.PART and LAST.PART.

```
TO FIRST.PART :LIST
OP FIRST.N (INT (COUNT :LIST)/2) :LIST
END

TO FIRST.N :NUMBER :LIST
IF :NUMBER=0 [OP :LIST]
OP FIRST.N :NUMBER-1 BL :LIST
END

TO LAST.PART :LIST
OP LAST.N (INT (1+COUNT :LIST)/2) :LIST
END
```

```
TO LAST.N :NUMBER :LIST
IF :NUMBER=0 [OP :LIST]
OP LAST.N :NUMBER-1 BF :LIST
END
```

You may notice that `LAST.PART` refers to `1 + COUNT :LIST` instead of `COUNT :LIST`. The reason for this difference is that if the input list has an odd number of members, we must divide the list into two pieces that differ in length by one. For example, if the input list has five members, `FIRST.PART` will output the first three members of the list and `LAST.PART` will output the last two members.

```
?SHOW FIRST.PART [14 3 27 1 10]
[14 3 27]
?SHOW LAST.PART [14 3 27 1 10]
[1 10]
?
```

Merging Two Ordered Lists

The last step of the sorting procedure is to *merge* two lists. The `MERGE` procedure assumes that each of the two lists is already in the correct order. `MERGE` takes two inputs, namely, the two lists.

`MERGE` compares the first member of one input list with the first member of the other list. One of these becomes the first member of the final merged list; `MERGE` is applied recursively to the remaining members of the input lists.

```
TO MERGE :A :B
IF EMPTYP :A [OP :B]
IF EMPTYP :B [OP :A]
IF COMPARE FIRST :A FIRST :B
  [OP FPUT FIRST :A MERGE BF :A :B]
OP FPUT FIRST :B MERGE :A BF :B
END
```

`MERGE` uses a subprocessure, `COMPARE`, which tells whether one item should come before or after another. `COMPARE` takes two inputs. It outputs the word `TRUE` if the first input comes before the second input, or `FALSE` otherwise.

You can write different versions of `COMPARE` depending on what ordering you want to use for your sorted lists. If you are sorting numbers by size, as in the earlier example, you can use this version:

```
TO COMPARE :A :B
OUTPUT :A < :B
END
```

If you want to sort words alphabetically, or use some other ordering, you need a more complicated version of `COMPARE`. We'll show an example later.

Putting It All Together

We've written the easy parts of this sorting method. The hard part is putting it all together. The main procedure SORT does this. It takes one input, which must be a list. It outputs the same list, but with its members in sorted order.

```
TO SORT :A
IF EMPTYP :A [OP []]
IF EMPTYP BF :A [OP :A]
OP MERGE (SORT FIRST.PART :A) (SORT LAST.PART :A)
END
```

If the input list is empty, or has only one member, then the list is already sorted. SORT outputs the list unchanged. For larger lists, SORT goes through the steps we described at the beginning.

1. It uses FIRST.PART and LAST.PART to divide the list in two.
2. It uses SORT to sort each of these smaller lists.
3. It uses MERGE to combine the resulting ordered lists.

Alphabetical Order

Sometimes we want to deal with information composed of words or sentences, rather than numbers. Here are procedures to alphabetize lists of words.

```
TO COLLATE.BEFORE :A :B
IF EMPTYP :A [OP "TRUE"]
IF EMPTYP :B [OP "FALSE"]
IF COLLATE.BEFORE.WORD FIRST :A FIRST :B [OP "TRUE"]
IF NOT EQUALP FIRST :A FIRST :B [OP "FALSE"]
OP COLLATE.BEFORE BF :A BF :B
END

TO COLLATE.BEFORE.WORD :A :B
IF EMPTYP :A [OP "TRUE"]
IF EMPTYP :B [OP "FALSE"]
IF (ASCII FIRST :A) < (ASCII FIRST :B) [OP "TRUE"]
IF NOT EQUALP FIRST :A FIRST :B [OP "FALSE"]
OP COLLATE.BEFORE.WORD BF :A BF :B
END
```

COLLATE.BEFORE takes two inputs. Each input is a sentence (in other words, a list of words). It outputs the word TRUE if the first input comes before the second alphabetically.

COLLATE.BEFORE.WORD is similar to COLLATE.BEFORE, except that its two inputs are single words instead of lists of words.

An Example

Here is a list of the greatest songs of all time.

```
MAKE "RECORDS [
  [[SHE LOVES YOU] [BEATLES]]
  [[SHE'S NOT THERE] [ZOMBIES]]
  [[WATERLOO SUNSET] [KINKS]]
  [[FLYING ON THE GROUND IS WRONG]
  [BUFFALO SPRINGFIELD]]
  [[MY GENERATION] [WHO]] ]
```

(To type in a long list like this, you have to use the Logo editor. When you are typing directly to the ? prompt in Atari Logo, there is a limit to how long a line you can type.)

This list contains five items. Each item is itself a list with two members, the title and artist of a record. This is a simple example of a *data structure*. That is, instead of having a list of words or a list of numbers, we have a list of more complicated things, each of which is itself made up of smaller parts.

Suppose we want to sort these songs by title. We can define a COMPARE procedure to do that.

```
TO COMPARE :A :B
OP COLLATE BEFORE FIRST :A FIRST :B
END
```

The FIRST of each song is a list containing its title, so this version of COMPARE sees which title comes first alphabetically.

```
?SHOW SORT :RECORDS
[[[FLYING ON THE GROUND IS WRONG] [BU->
FFALO SPRINGFIELD]] [[MY GENERATION] ->
[WHO]] [[SHE LOVES YOU] [BEATLES]] [[->
SHE'S NOT THERE] [ZOMBIES]] [[WATERLO->
O SUNSET] [KINKS]]]
```

We can make this prettier by using a formatting procedure to print each record on a separate line.

```
TO FORMAT :LIST
IF EMPTYP :LIST [STOP]
PRINT SE "TITLE: FIRST FIRST :LIST
PRINT SE "...ARTIST: LAST FIRST :LIST
FORMAT BF :LIST
END
```

```
?FORMAT SORT :RECORDS
TITLE: FLYING ON THE GROUND IS WRONG
...ARTIST: BUFFALO SPRINGFIELD
TITLE: MY GENERATION
...ARTIST: WHO
TITLE: SHE LOVES YOU
...ARTIST: BEATLES
TITLE: SHE'S NOT THERE
...ARTIST: ZOMBIES
TITLE: WATERLOO SUNSET
...ARTIST: KINKS
?
```

PROGRAMMING IDEAS

Now suppose we want to sort the same list of records, this time by artist. To do this, we replace the COMPARE procedure with one that uses the LAST of each item instead of the FIRST.

```
TO COMPARE :A :B
OP COLLATE BEFORE LAST :A LAST :B
END
```

The LAST of each song is a list containing the name of the group that performed it.

```
?FORMAT SORT :RECORDS
TITLE: SHE LOVES YOU
...ARTIST: BEATLES
TITLE: FLYING ON THE GROUND IS WRONG
...ARTIST: BUFFALO SPRINGFIELD
TITLE: WATERLOO SUNSET
...ARTIST: KINKS
TITLE: MY GENERATION
...ARTIST: WHO
TITLE: SHE'S NOT THERE
...ARTIST: ZOMBIES
?
```

SUGGESTIONS

If you are interested in learning about other ways to write sorting programs, the standard reference book on this subject is *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, by Donald E. Knuth (Reading, Mass.: Addison-Wesley, 1973).

PROGRAM LISTING

Note: There are three different versions of COMPARE in the write-up. The one here is the first version. COMPARE2 and COMPARE3 are the other two versions and can be substituted for COMPARE in MERGE.

<pre>TO FIRST.PART :LIST OP FIRST.N (INT (COUNT :LIST)/2) :LIST END</pre>	<pre>TO LAST.N :NUMBER :LIST IF :NUMBER=0 [OP :LIST] OP LAST.N :NUMBER-1 BF :LIST END</pre>
<pre>TO FIRST.N :NUMBER :LIST IF :NUMBER=0 [OP :LIST] OP FIRST.N :NUMBER-1 BL :LIST END</pre>	<pre>TO MERGE :A :B IF EMPTYP :A [OP :B] IF EMPTYP :B [OP :A] IF COMPARE FIRST :A FIRST :B [OP FPUT ▶ FIRST :A MERGE BF :A :B] OP FPUT FIRST :B MERGE :A BF :B END</pre>
<pre>TO LAST.PART :LIST OP LAST.N (INT (1+COUNT :LIST)/2) ▶ :LIST END</pre>	

```

TO COMPARE :A :B
OUTPUT :A < :B
END

TO SORT :A
IF EMPTYP :A [OP []]
IF EMPTYP BF :A [OP :A]
OP MERGE (SORT FIRST.PART :A) (SORT ▶
LAST.PART :A)
END

TO COLLATE.BEFORE :A :B
IF EMPTYP :A [OP "TRUE"]
IF EMPTYP :B [OP "FALSE"]
IF COLLATE.BEFORE.WORD FIRST :A FIRST ▶
:B [OP "TRUE"]
IF NOT EQUALP FIRST :A FIRST :B [OP ▶
"FALSE"]
OP COLLATE.BEFORE BF :A BF :B
END

TO COLLATE.BEFORE.WORD :A :B
IF EMPTYP :A [OP "TRUE"]
IF EMPTYP :B [OP "FALSE"]
IF (ASCII FIRST :A) < (ASCII FIRST :B) ▶
[OP "TRUE"]

IF NOT EQUALP FIRST :A FIRST :B [OP ▶
"FALSE"]
OP COLLATE.BEFORE WORD BF :A BF :B
END

IF NOT EQUALP FIRST :A FIRST :B [OP ▶
"FALSE"]
OP COLLATE.BEFORE WORD BF :A BF :B
END

TO COMPARE2 :A :B
OP COLLATE.BEFORE FIRST :A FIRST :B
END

TO FORMAT :LIST
IF EMPTYP :LIST [STOP]
PRINT SE "TITLE: FIRST FIRST :LIST
PRINT SE "...ARTIST: LAST FIRST :LIST
FORMAT BF :LIST
END

TO COMPARE3 :A :B
OP COLLATE.BEFORE LAST :A LAST :B
END

MAKE "RECORDS [[SHE LOVES YOU] ▶
[BEATLES]] [[SHE'S NOT THERE] ▶
[ZOMBIES]] [[WATERLOO SUNSET] ▶
[KINKS]] [[FLYING ON THE GROUND ▶
IS WRONG] [BUFFALO SPRINGFIELD]] ▶
[[MY GENERATION] [WHO]] ]

```

Bestline

Bestline is a Logo project that draws the “best-fitting” straight line on a Cartesian graph of some data points. It is a strategy commonly used among scientists to predict the value of some quantity based on another.

An Example: A Scientific Experiment

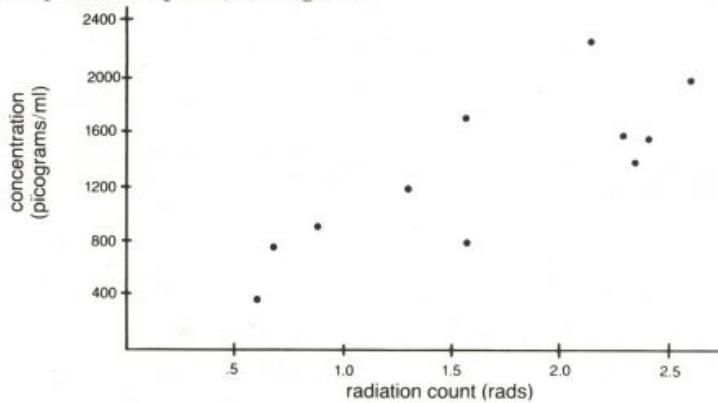
I got the idea for this project while helping a friend interpret data from a laboratory experiment. The purpose of the experiment was to find the concentration of antibodies in each of a large number of test tubes. This is done by adding radioactive iodine to the antibodies. A certain amount of the iodine bonds to the antibody and the rest is removed. The concentration of antibodies can be determined by measuring how much iodine bonded to them. Since the iodine is radioactive, you can run it through a machine that measures how much radiation is emitted by each test tube. In this experiment, the radiation (rad) counts were collected and processed by a computer in my friend’s lab. I thought I could write a Logo program that could generate a “best-fit” line for this data and for samples of other data.

By Julie Minsky.

*In statistics, this kind of plot is called a scattergram.

Making a Graph of the Data

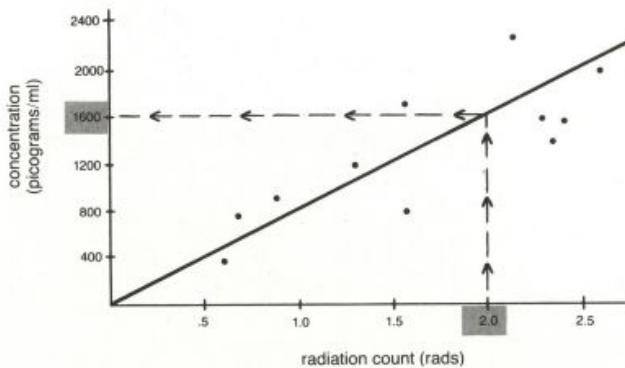
In the antibody experiment, we take samples of known antibody concentrations, measure their radiation counts, and plot them on a graph. For each known concentration, we plot the corresponding radiation count.* When we plot all the points, we might see:



We can use this plot for looking at the data from our samples of known concentrations. How can we use this data to estimate the *unknown* concentrations of our experimental samples?

We know that for this kind of experiment, the radiation count of a sample is proportional to the concentration of antibodies in it. That is, when we double the concentration, the radiation emitted will be doubled. This relationship suggests that the graph of radiation versus concentration is a *line*. We need to find a line on which we can look up an estimate of the concentration of a sample once we have experimentally found its radiation count.

Looking Things Up on a Graph



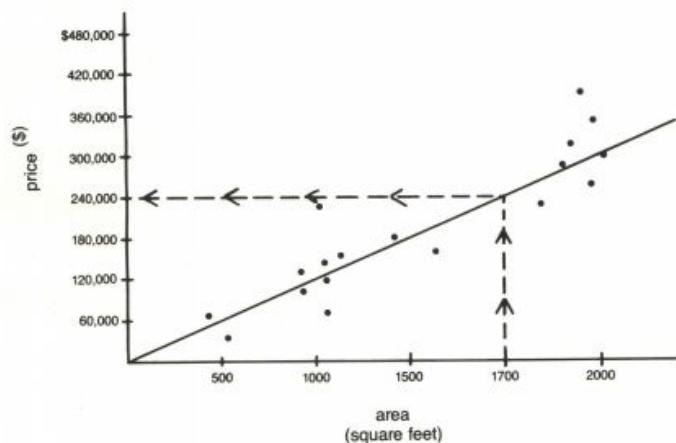
Let's look at the graph above. It shows a regression line plotted for the data points in this experiment.* Once we know how much radiation is emitted,

*The "best-fitting" line through a sample of data points is called a "least squares," or regression, line and is calculated from the data points.

we can estimate the concentration of antibodies present. For example, if the radiation count is 2, then the concentration is estimated to be 1600 picograms/ml.

This line was already calculated for this particular experiment; someone plotted the data points and determined the line. Different samples of data generate different graphs and regression lines.

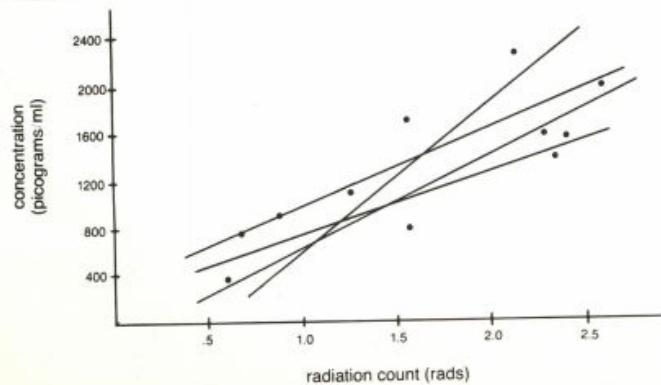
For example, a realtor selling office space might want to know how much to charge for a 1700-square-foot building. Let's say the realtor called other realtors who sold office space in the same community and asked them how much they charged for buildings of different square footages. A helpful graph would be price plotted against square footage. While the realtor might consider other factors in setting the price (for example, property location, condition of the building), she is able to estimate the market price for the office space.



Possible Lines

Many different lines might be drawn through the known sample points. How can we find a line that goes through all the measured points and makes sense for our data?

Since this is a real-world experiment, the sample points don't all lie exactly on a straight line. We could draw lots of lines near these data points. We would like to find the one line that goes as close as possible to *all* of them.



Moving the line closer to some points will increase its distance from others. Some of the lines fit the data so poorly that we wouldn't even consider them. Others would seem to be pretty good fits to the data. We need to find a way of determining the line with the best fit, the line that comes closest to all the points. How can we choose which line is the best fit?

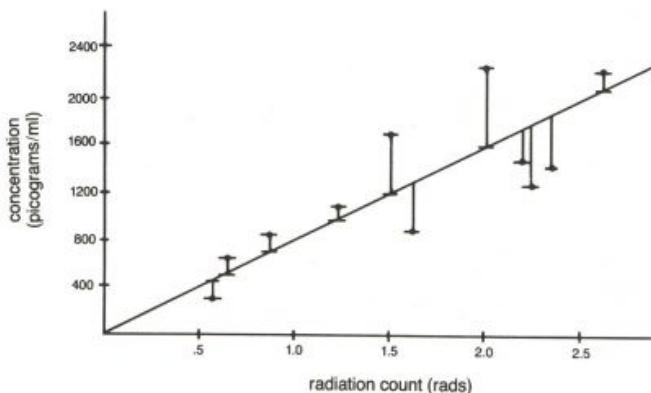
A Technique for Finding the Best Line

There are two things you need to know to plot a line: its slope (m) and its y -intercept (b). The standard equation for a line is

$$y = mx + b$$

Once you know m and b , you can use the equation to find the y coordinate for any x .

A method usually used to find the best-fit line is called "least squares." The least squares line is that which minimizes the sum of the squares of the vertical distances between the line and the data points. It is a neat way of solving the problem when the real-world data points are not exactly on the line (this is usually called minimizing the error). Let's look at the following graph.



Not all the data points fall on the regression line. The amount of "error" of the regression line is the sum of the squares of the vertical distance of each point from the line. If all the data points fall on the best-fit line the error would be 0. This is the ideal; most real-world data do not behave so neatly. The method of least squares is used to calculate a line that *minimizes* the error.

Given a set of points, you can find the best-fit least squares line by solving two equations: one to calculate the slope of the best-fit line and one to calculate its y -intercept.

In our experiment, we have the x and y coordinates of N points. We can use the coordinates and these two equations to find m and b :

$$m = \frac{N\Sigma xy - \Sigma x}{N\Sigma x^2 - (\Sigma x)^2}$$

$$b = \frac{\Sigma y - m\Sigma x}{N}$$

The Greek letter sigma, Σ , is called summation notation; it means that you add a set of numbers. Σx means add up all the x coordinates from the set of points. Σxy means you should multiply the x and y coordinates of each point and add up all the products.

Using the Program

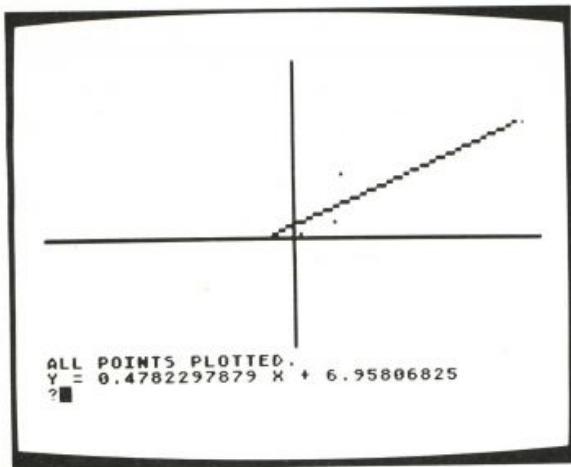
Here is an example of how to use BESTLINE. The text that is boldface is what you type. Say your points are (28, 39), (25, 10), (140, 72), and (5, 2).

```
BESTLINE
PLEASE TYPE YOUR POINTS: X Y X Y ...
28 39 25 10 140 72 5 2
```

Before plotting the line, BESTLINE prints:

```
SLOPE = 0.4756966082
Y-INTERCEPT = 7.203018
Y = 0.4756966882 X + 7.203018
PRESS ANY KEY TO CONTINUE
```

When BESTLINE continues, it plots your points and draws the line that best fits them.



At the bottom of the screen BESTLINE prints:

```
ALL POINTS PLOTTED.
THE X FOR 2 = -10.93768151
THE X FOR 72 = 136.214933
```

PROGRAMMING IDEAS

To find the y coordinate on the best fit line for a certain x -100, for example) type:

```
SOLVE.Y 100
THE Y FOR 100 = 54.77267882
```

Similarly, you can use a procedure called SOLVE.X to find the x value for a certain y .

How the Program Works

Overview

BESTLINE is the top-level procedure. It sets up a global variable, POINTLIST, to contain the list of points the user types in. The list of x and y coordinates the user types is converted into a list of lists by PAIRUP. Each sublist contains the x and y coordinates for each point. In our example, :POINTLIST is [[28 39] [25 10] [140 72] [5 2]]. BESTLINE calls LINE.EQUATION to find the slope and the y -intercept of the line that best fits these points. BESTLINE then calls PLOTLINE to plot the points and draw the best-fit line.

```
TO BESTLINE
HT TS CT
PR [PLEASE TYPE YOUR POINTS: X Y X Y ...]
MAKE "POINTLIST PAIRUP RL
LINE.EQUATION :POINTLIST
PR [PRESS ANY KEY TO CONTINUE]
IGNORE RC
WINDOW
PLOTLINE :POINTLIST
END

TO IGNORE :THING
END

LINE.EQUATION creates two global variables, M and B. :M is the slope
of the line and is computed by LEAST.SQUARES.SLOPE. :B is the  $y$ -
intercept and is computed by YINTERCEPT.

TO LINE.EQUATION :POINTLIST
MAKE "M LEAST.SQUARES.SLOPE :POINTLIST
PR SE [SLOPE IS] :M
MAKE "B YINTERCEPT :POINTLIST
PR SE [Y]-INTERCEPT =] :B
PR ( SE [Y =] :M [X +] :B )
END

TO LEAST.SQUARES.SLOPE :POINTS
MAKE "SUMX BIGE :POINTS "JUSTX
OP ( ( COUNT :POINTS ) * ( BIGE :POINTS "XTIMESY )
    - ( :SUMX * BIGE :POINTS "JUSTY ) )
    / ( ( COUNT :POINTS ) * ( BIGE :POINTS "XSQUARED )
    - ( :SUMX * :SUMX ) )
END
```

```

TO YINTERCEPT :POINTS
OP ( ( BIGE :POINTS "JUSTY")
      - ( :M * BIGE :POINTS "JUSTX" ) ) / COUNT :POINTS )
END

```

Both LEAST.SQUARES.SLOPE and YINTERCEPT rely on a collection of procedures used by BIGE.

BIGE

BIGE takes two inputs, a list of points and the name of another procedure. It sums the result of applying that procedure to each point in the list.* (This procedure is called BIGE, pronounced "big-ee," because the Greek letter Σ , used as the summation symbol, looks like an upper-case "E." For example, if you type BIGE :POINTS "JUSTX", JUSTX will extract just the x coordinate from each point in the list and BIGE will end up adding up just the x 's! BIGE :POINTS "XTIMESY" adds up the products of the x and y coordinates for each point. The procedures used with BIGE in the formulas are JUSTX, JUSTY, XTIMESY, and XSQUARED.

```

TO BIGE :LIST :PROC
IF EMPTYP :LIST [OP 0]
OP ( SUM ( RUN LIST :PROC FIRST :LIST )
          BIGE BF :LIST :PROC )
END

TO JUSTX :POINT
OP FIRST :POINT
END

TO JUSTY :POINT
OP FIRST BF :POINT
END

TO XTIMESY :POINT
OP ( JUSTX :POINT ) * ( JUSTY :POINT )
END

TO XSQUARED :POINT
OP ( JUSTX :POINT ) * ( JUSTX :POINT )
END

```

Graphing

After the equation of the best-fit line is determined, PLOTLINE plots your points and the line. PLOTLINE first uses PLOT.POINTS to draw your points.

*BIGE is a mapping procedure. See Brian Harvey's Map project (p.322) for more about mapping. ↵

PROGRAMMING IDEAS

```

TO PLOTLINE :LIST
SS
PLOT.AXES
PLOT.POINTS :LIST
WAIT 60
RANGE :LIST
IF :M = 0 [PLOT.HORIZ STOP]
PU
SETPOS LIST XVALUE :MINY :MINY
PD
SETPOS LIST XVALUE :MAXY :MAXY
PRINT (SE [Y = ] :M [X + ] :B)
END

TO PLOT.AXES
MAKE "PN PN
IF :PN = 2 [SETPN 0] [SETPN PN + 1]
SETPC PN 0
PU SETPOS [-150 0]
PD SETPOS [150 0]
PU SETPOS [0 -110]
PD SETPOS [0 110]
PU SETPN :PN HOME
END

TO PLOT.POINTS :LIST
IF EMPTYP :LIST [PR [ALL POINTS PLOTTED.] STOP]
PU
SETPOS FIRST :LIST
PD FD 0
PLOT.POINTS BF :LIST
END

TO SOLVE.Y :X
PR ( SE [THE Y FOR] :X "= ( :M * :X + :B ) )
END

TO SOLVE.X :Y
IF :M = 0 [PRINT [NO SOLUTION, M=0] STOP]
PRINT (SE [THE X FOR] :Y "= XVALUE :Y)
END

TO XVALUE :Y
OP (:Y - :B) / :M
END

```

PLOTLINE then draws the best-fit line. If the slope (:M) is 0, then PLOT.HORIZ draws the line. The procedure RANGE finds the smallest and largest x and y coordinates for your set of points. SOLVE.X finds the best-fit line's x coordinate for the minimum y and maximum y computed by RANGE. These are the procedures for finding and plotting the endpoints of the best-fit line.

```

TO RANGE :PLIST
MAKE "MINX LEAST.NUM XLIST :PLIST
MAKE "MINY LEAST.NUM YLIST :PLIST
MAKE "MAXX GREATEST.NUM XLIST :PLIST
MAKE "MAXY GREATEST.NUM YLIST :PLIST
END

TO LEAST.NUM :NUMS
IF EMPTYP BF :NUMS [OP FIRST :NUMS]
IF ( FIRST :NUMS ) < ( FIRST BF :NUMS )
    [OP LEAST.NUM SE BF BF :NUMS FIRST :NUMS]
OP LEAST.NUM BF :NUMS
END

TO GREATEST.NUM :NUMS
IF EMPTYP BF :NUMS [OP FIRST :NUMS]
IF ( FIRST BF :NUMS ) > FIRST :NUMS
    [OP GREATEST.NUM BF :NUMS]
OP GREATEST.NUM SE BF BF :NUMS FIRST :NUMS
END

TO XLIST :POINTLIST
IF EMPTYP :POINTLIST [OP []]
OP FPUT JUSTX FIRST :POINTLIST XLIST BF :POINTLIST
END

TO YLIST :POINTLIST
IF EMPTYP :POINTLIST [OP []]
OP FPUT JUSTY FIRST :POINTLIST YLIST BF :POINTLIST
END

```

PLOT.HORIZ is used in the special case when the slope of the line is 0.

```

TO PLOT.HORIZ
PU SETPOS LIST :MINX :B
PD SETPOS LIST :MAXX :B
END

```

PAIRUP and PAIRS are used by BESTLINE to convert a list of coordinates typed by the user into a list of points that the program can use.

```

TO PAIRUP :LIST
IF 1 = REMAINDER COUNT :LIST 2 [MAKE "LIST BL :LIST]
OP PAIRS :LIST
END

TO PAIRS :LIST
IF EMPTYP :LIST [OP []]
OP FPUT SE FIRST :LIST FIRST BF :LIST PAIRS BF BF :LIST
END

```

PROGRAM LISTING

```

TO BESTLINE
HT TS CT
PR [PLEASE TYPE YOUR POINTS: X Y X Y ▶
...]
MAKE "POINTLIST PAIRUP RL
LINE.EQUATION :POINTLIST
PR [PRESS ANY KEY TO CONTINUE]
IGNORE RC
WINDOW
PLOTLINE :POINTLIST
END

TO IGNORE :THING
END

TO LINE.EQUATION :POINTLIST
MAKE "M LEAST.SQUARES.SLOPE :POINTLIST
PR SE [SLOPE IS] :M
MAKE "B YINTERCEPT :POINTLIST
PR SE [Y]-INTERCEPT =] :B
PR ( SE [Y =] :M [X +] :B)
END

TO LEAST.SQUARES.SLOPE :POINTS
MAKE "SUMX BIGE :POINTS "JUSTX
OP ( ( COUNT :POINTS ) * ( BIGE ▶
:POINTS "XTIMESY ) - ( :SUMX * ▶
BIGE :POINTS "JUSTY ) ) / ( ( ▶
COUNT :POINTS ) * ( BIGE :POINTS ▶
"XSQUARED ) - ( :SUMX * :SUMX ) )
END

TO YINTERCEPT :POINTS
OP ( ( ( BIGE :POINTS "JUSTY ) - ( :M ▶
* BIGE :POINTS "JUSTX ) ) / COUNT ▶
:POINTS )
END

TO BIGE :LIST :PROC
IF EMPTYP :LIST [OP Ø]
OP ( SUM ( RUN LIST :PROC FIRST :LIST ▶
) BIGE BF :LIST :PROC )
END

TO JUSTX :POINT
OP FIRST :POINT
END

TO JUSTY :POINT
OP FIRST BF :POINT
END

TO XTIMESY :POINT
OP ( JUSTX :POINT ) * ( JUSTY :POINT )
END

TO XSQUARED :POINT
OP ( JUSTX :POINT ) * ( JUSTX :POINT )
END

TO PLOTLINE :LIST
SS
PLOT.AXES
PLOT.POINTS :LIST
WAIT 60
RANGE :LIST
IF :M = Ø [PLOT.HORIZ STOP]
PU
SETPOS LIST XVALUE :MINY :MINY
PD
SETPOS LIST XVALUE :MAXY :MAXY
PRINT (SE [Y =] :M [X +] :B)
END

TO PLOT.AXES
MAKE "PN PN
IF :PN = 2 [SETPN Ø] [SETPN PN + 1]
SETPC PN Ø
PU SETPOS [-150 Ø]
PD SETPOS [150 Ø]
PU SETPOS [Ø -110]
PD SETPOS [Ø 110]
PU SETPN :PN HOME
END

TO PLOT.POINTS :LIST
IF EMPTYP :LIST [PR [ALL POINTS ▶
PLOTTED.] STOP]
PU
SETPOS FIRST :LIST
PD FD Ø
PLOT.POINTS BF :LIST
END

TO SOLVE.Y :X
PR ( SE [THE Y FOR] :X "= ( :M * :X + ▶
:B ) )
END

TO SOLVE.X :Y
IF :M = Ø [PRINT [NO SOLUTION, M=Ø] ▶
STOP]
PRINT (SE [THE X FOR] :Y "= XVALUE :Y)
END

```

```

TO XVALUE :Y
OP (:Y - :B) / :M
END

TO RANGE :PLIST
MAKE "MINX LEAST.NUM XLIST :PLIST
MAKE "MINY LEAST.NUM YLIST :PLIST
MAKE "MAXX GREATEST.NUM XLIST :PLIST
MAKE "MAXY GREATEST.NUM YLIST :PLIST
END

TO LEAST.NUM :NUMS
IF EMPTYP BF :NUMS [OP FIRST :NUMS]
IF ( FIRST :NUMS ) < ( FIRST BF :NUMS ▶
    ) [OP LEAST.NUM SE BF BF :NUMS ▶
    FIRST :NUMS]
OP LEAST.NUM BF :NUMS
END

TO GREATEST.NUM :NUMS
IF EMPTYP BF :NUMS [OP FIRST :NUMS]
IF ( FIRST BF :NUMS ) > FIRST :NUMS ▶
    [OP GREATEST.NUM BF :NUMS]
OP GREATEST.NUM SE BF BF :NUMS FIRST ▶
    :NUMS
END

```

```

TO XLIST :POINTLIST
IF EMPTYP :POINTLIST [OP []]
OP FPUT JUSTX FIRST :POINTLIST XLIST ▶
    BF :POINTLIST
END

TO YLIST :POINTLIST
IF EMPTYP :POINTLIST [OP []]
OP FPUT JUSTY FIRST :POINTLIST YLIST ▶
    BF :POINTLIST
END

TO PLOT.HORIZ
PU SETPOS LIST :MINX :B
PD SETPOS LIST :MAXX :B
END

TO PAIRUP :LIST
IF 1 = REMAINDER COUNT :LIST 2 [MAKE ▶
    "LIST BL :LIST]
OP PAIRS :LIST
END

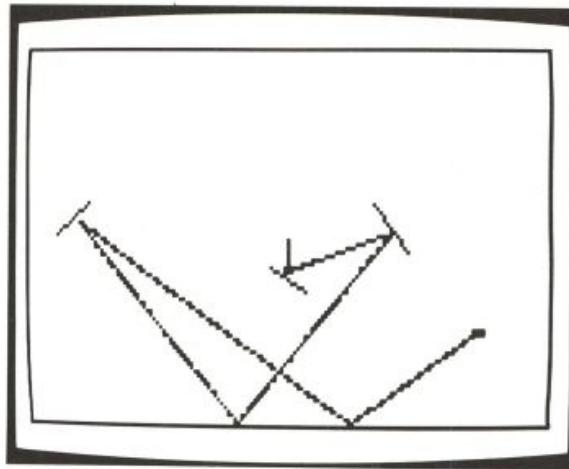
TO PAIRS :LIST
IF EMPTYP :LIST [OP []]
OP FPUT SE FIRST :LIST FIRST BF :LIST ▶
    PAIRS BF BF :LIST
END

```

Lines and Mirrors

This program was designed to simulate a beam of light bouncing off mirrors or a ball bouncing off walls. The user enters the coordinates of endpoints of lines. The program then draws the lines and starts the turtle going in a random direction. When the turtle hits one of those lines, it will bounce off at the same angle at which it came in. The turtle draws its path as it goes. You can think of the turtle's path as a beam of light and the lines as mirrors.

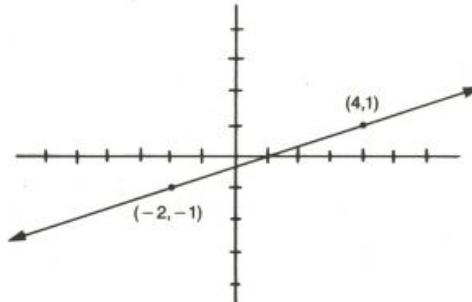
In this write-up there are three main sections: first, how the program calculates the angle at which the turtle should bounce after hitting a line; second, how the information about the lines is remembered; and third, the detailed structure of the program.



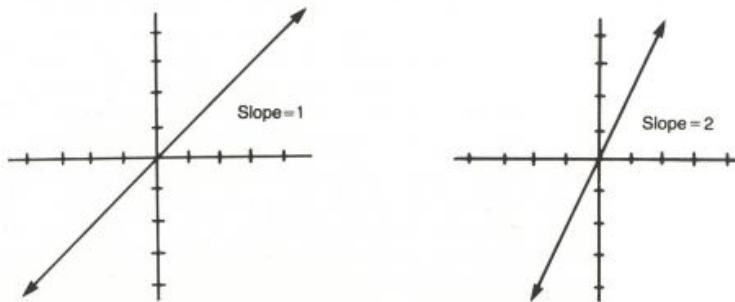
Bouncing Off a Line

What Is a Line?

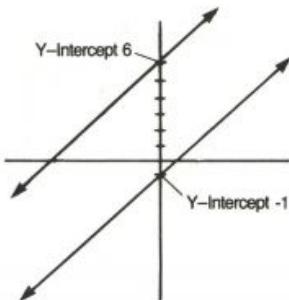
You probably know that two points determine a line, as in the following illustration.



Another way to determine a line is by its slope (m) and y -intercept (b). The slope is the steepness of the line. In the following figure the line on the right rises twice as fast as the line on the left.



There may be many different lines with the same slope. A way to distinguish these lines is by their y -intercept. The y -intercept is the point at which the line crosses the y axis.



Calculating the Turning Angle

For our purposes we need a representation that tells us *where* the line is and what its *orientation* is. (The orientation is particularly important because the problem we are trying to solve is about directions of motion.) These aspects of lines are reminiscent of the major components of the state of a turtle: position and heading. This suggests that the best way to represent the orientation of a line is by the heading that a turtle would take to draw it.

It is important for us to know the heading of a line in order to figure out how the turtle should bounce off it. The angle at which the turtle comes in (angle of incidence) should be the same as the angle at which it bounces out (angle of reflection).



The angle of incidence is the amount the turtle must turn to get from its initial heading to the heading of the line.



PROGRAMMING IDEAS

So we get

$$(line's\ heading) - (turtle's\ heading)$$

as the angle of incidence. The angle of reflection should also be

$$(line's\ heading) - (turtle's\ heading)$$

The total amount through which the turtle turns is therefore

$$2 \times |(line's\ heading) - (turtle's\ heading)|$$



Figuring Out the Heading

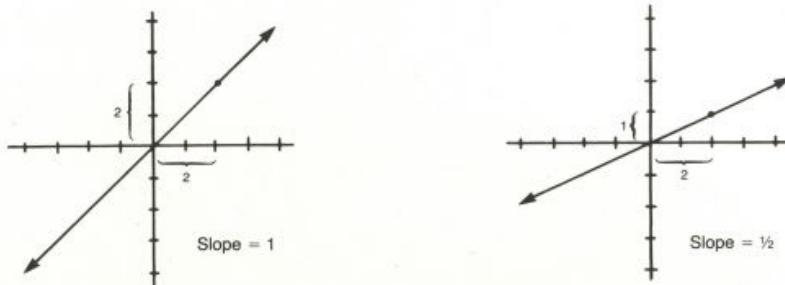
In fact, we are not given the heading or the position of a line. All we are given are its two endpoints. With the two endpoints we can figure out the slope. Then we can use the ARCTAN procedure, described in the Towards and Arctan project, to figure out the heading. The procedure to figure out the heading is as follows.

```
TO FIGH :LINE
IF ( DX :LINE ) = 0 [OP 0]
OP 90 - ARCTAN SLOPE :LINE
END
```

(It is $90 - \text{ARCTAN}$ because Logo headings are clockwise from north, not counterclockwise from east as in algebra.)

Figuring Out the Slope

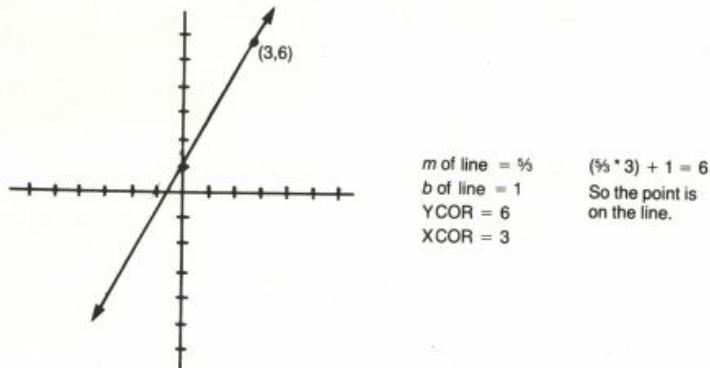
The slope is the difference between the y coordinates of any two points on the line divided by the difference between the x coordinates of those points ($\Delta y / \Delta x$, where Δ stands for “difference in”).



Traditionally, a line is represented by the equation $y = mx + b$. Given m and b , we can tell whether a particular point is on a particular line. For example, if

$$\text{YCOR} = (m \times \text{XCOR}) + b$$

then we know that the turtle's position is on the line.



We use $\Delta y / \Delta x$ to figure out the slope. But if the line is vertical (the two x coordinates are the same), then the slope is infinite, so the procedure to figure out the slope has to treat that case in a special way. The procedures to figure out the slope (m) and y -intercept (b) of the line are as follows.

```

TO FIGM :LINE
IF ( DX :LINE ) = 0 [OP []] [OP SLOPE :LINE]
END

TO SLOPE :LINE
OP ( DY :LINE ) / DX :LINE
END

TO DY :LINE
OP ( LAST POINT1 :LINE ) - ( LAST POINT2 :LINE )
END

TO DX :LINE
OP ( FIRST POINT1 :LINE ) - ( FIRST POINT2 :LINE )
END

TO FIGB :LINE
IF (M :LINE) = [] [OP []]
OP (LAST POINT1 :LINE) - ((M :LINE) * (FIRST POINT1 :LINE))
END

```

Information We Need About a Line

What information does this program need about a line?

It needs the heading for use in calculating the turning angle when the turtle hits the line.

It uses slope and y -intercept to figure out if a position is on a line, with the equation $y = mx + b$.

It also needs the endpoints. Why? So far we have been talking about *lines*, which are infinitely long. Really the program has to deal with line *segments*, which have two endpoints.

So we finally need five pieces of information to represent a line segment: two endpoints, heading, slope, and y -intercept.

Storing the Lines

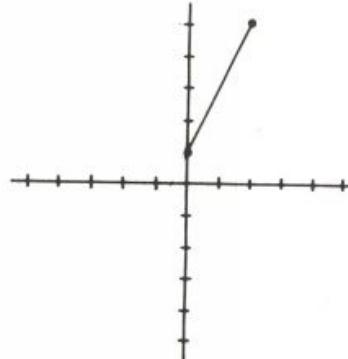
What Each Line Looks Like

The only thing that the user gives the program is the endpoints of lines. From this the slope, heading, and y -intercept are figured out. The program has to have a way of storing all this information in some kind of organized structure. It stores the five pieces of information about the line in a list of the following format:

```
[[x1 y1] [x2 y2] slope heading y-intercept]
```

Here is a sample line and the list that represents it.

```
[[0 1] [2 5] 2 26.5651 1]
```



Retrieving Information About Lines

Throughout the program we need to recall information about lines. We could do it in a messy way. For example, to retrieve the slope of a line we could say

```
FIRST BF BF :LINE
```

The program would get very ugly and confusing if we used that approach. A much clearer and neater way is to have a procedure that extracts one piece of information about a line. So we could say `M :LINE` to retrieve the slope, or `POINT2 :LINE` to retrieve the coordinates of the second endpoint. By using these procedures, other parts of the program do not have to know the detailed structure of a line list. The procedures also make

the program much easier to read and understand. Here are the information retrieving procedures.

```

TO POINT1 :LINE
OP ITEM 1 :LINE
END

TO POINT2 :LINE
OP ITEM 2 :LINE
END

TO M :LINE
OP ITEM 3 :LINE
END

TO D :LINE
OP ITEM 4 :LINE
END

TO B :LINE
OP ITEM 5 :LINE
END

TO ITEM :INUM :LIST
IF :INUM = 1 [OP FIRST :LIST]
OP ITEM :INUM - 1 BUTFIRST :LIST
END

```

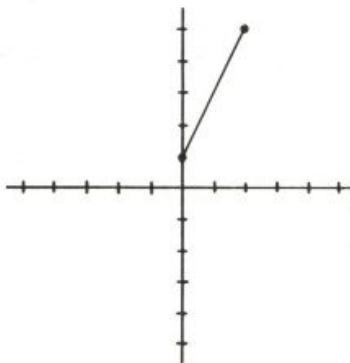
The List of Lines

Since the program has to keep track of a lot of lines, it stores them all in a list called `LINES`. The elements of `:LINES` are themselves lists, each representing a line. For example, the border lines and the line shown earlier would be represented as follows:

```

[ [[-158 -119] [-158 120] [] 0 []]
[[161 -119] [161 120] [] 0 []]
[[-158 120] [161 120] 0 90 120]
[[-158 -119] [161 -119] 0 90 -119]
[[0 1] [2 5] 2 26.5651 1] ]

```



Program Structure

The top-level procedure is BOUNCE. It has four tasks. First it creates the list of lines. Then it sets up the initial position and shape of the turtle. The third task is to draw the lines in the list. Finally it starts the turtle moving and prepares to turn the turtle when it bounces off a wall. There is a subprocedure for each of these tasks.

```
TO BOUNCE
  LEARN.LINES
  SETUP.GRAPHICS
  DRAW.LINES
  START.TURTLE
END
```

Creating the List of Lines

When the program starts up, LEARN.LINES creates the list of lines. It calls INFO, to remember the lines which the user enters. It also calls BORDER, which remembers the border lines.

```
TO LEARN.LINES
  MAKE "LINES []
  INFO
  BORDER
END
```

INFO lets the user enter lines. It calls GETLINE to get each line and calls REMEMBER to add each line to the list :LINES.

```
TO INFO
  REMEMBER GETLINE
  PRINT [ANOTHER LINE?]
  IF EQUALP RL [YES] [INFO]
END
```

GETLINE asks the user to type in the endpoints of a line segment. It calls FIGLINE to calculate the other information about the line. The output from GETLINE is the list representing the line.

```
TO GETLINE
  TYPE [X AND Y OF FIRST POINT?]
  MAKE "FP RL
  TYPE [X AND Y OF SECOND POINT?]
  MAKE "SP RL
  MAKE "LINE LIST :FP :SP
  OP FIGLINE :LINE
END
```

FIGLINE takes the endpoints of a line segment as its input. It calls FIGM, FIGH, and FIGB to compute the slope, heading, and *y*-intercept of the line.

```
TO FIGLINE :LINE
MAKE "LINE LPUT FIGM :LINE :LINE
MAKE "LINE LPUT FIGH :LINE :LINE
MAKE "LINE LPUT FIGB :LINE :LINE
OP :LINE
END
```

REMEMBER adds a line to the list of lines (:LINES).

```
TO REMEMBER :LINE
MAKE "LINES FPUT :LINE :LINES
END
```

BORDER remembers the four lines making up the border of the screen.

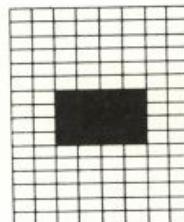
```
TO BORDER
REMEMBER FIGLINE [[-158 -119] [-158 120]]
REMEMBER FIGLINE [[161 -119] [161 120]]
REMEMBER FIGLINE [[-158 120] [161 120]]
REMEMBER FIGLINE [[-158 -119] [161 -119]]
END
```

Setting Up Graphics

SETUP.GRAPHICS selects turtle 0 and changes its shape. We don't use the normal turtle shape because later on we will need to know the precise position of the turtle. The normal turtle shape is big enough that its edges are at a very different position from the position of the center, which XCOR and YCOR output. Instead, we use a small square dot shape.

```
TO SETUP.GRAPHICS
ASK [1 2 3] [HT]
TELL 0
PUTSH 1 :SHAPE1
SETSH 1
CS
ST
FS
END

MAKE "SHAPE1 [0 0 0 0 0 60 60 60 60 0 0 0 0 0]
```



Drawing the Lines

DRAW.LINES draws the lines in :LINES on the screen. The lines are drawn with pen 1. Later, when the turtle is moving, its trajectory is drawn with pen 0. Using a different pen for the walls allows the demon to notice collisions with the walls and not notice collisions between the turtle and its own earlier path.

```
TO DRAW.LINES
SETPN 1
DRAW :LINES
END
```

PROGRAMMING IDEAS

```
TO DRAW :LIST
IF EMPTYP :LIST [STOP]
PU
SETPOS POINT1 FIRST :LIST
PD
SETPOS POINT2 FIRST :LIST
DRAW BUTFIRST :LIST
END
```

Starting the Turtle

`START.TURTLE` positions the turtle in the center of the screen, points it in a randomly chosen direction, and starts it moving. It also creates the demon that waits for collisions with lines. Finally, `START.TURTLE` calls `LOOP`, which is explained next.

```
TO START.TURTLE
SETPN 0
PU
SETPOS [0 0]
PD
SETH RANDOM 360
SETSP 20
WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 []]
LOOP
END
```

Knowing When a Line Is Hit

While the turtle is moving, `LOOP` continually checks if it has hit a line. `LOOP` knows the turtle has hit a line when its speed becomes zero.

```
TO LOOP
IF SPEED = 0 [NEWHEAD]
LOOP
END
```

How does the speed become zero? There is a demon, created by `START.TURTLE`, whose instructions include `SETSP 0`.

```
WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 []]
```

Setting the turtle's speed to zero is a convenient way for the demon to signal to `LOOP` that the turtle has hit a line. We could have changed something else as the signal, but we had to stop the turtle anyway. Otherwise the turtle would go through the line. Using the speed as the signal solves two problems at once.

When the speed is zero, `LOOP` calls `NEWHEAD` to figure out which line was hit and how much the turtle should turn.

Which Line Is Being Hit?

When a line is hit, NEWHEAD calls SEARCH to go through the list of lines, finding the one that was hit. Then NEWHEAD uses that line as the input to FIGTURN, which figures out how much the turtle should turn. Finally, NEWHEAD restarts the turtle and the demon.

```
TO NEWHEAD
MAKE "L SEARCH :LINES
IF NOT EMPTYP :L [RIGHT FIGTURN :L]
SETSP 20
WHEN OVER Ø 1 [SETSP Ø WHEN OVER Ø 1 []]
END
```

SEARCH goes through the list of lines, looking for the one the turtle hit. It calls the predicate CHECK for each line in the list. If CHECK outputs TRUE, SEARCH outputs the line that has been found.

```
TO SEARCH :LINES
IF EMPTYP :LINES [OP []]
IF CHECK FIRST :LINES [OP FIRST :LINES]
OP SEARCH BF :LINES
END
```

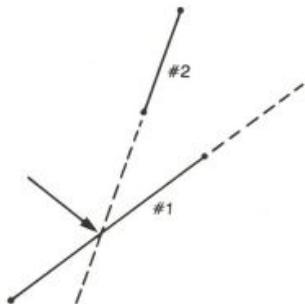
How to Check a Line

The straightforward way to check if the turtle hit a certain line is to use the equation $y = mx + b$, substituting the XCOR and YCOR of the turtle for x and y . If the equation holds true, then the turtle hit that line.

```
TO CHECK1 :LINE
OP YCOR = SOLVE :LINE XCOR
END

TO SOLVE :LINE :X
OP ( ( M :LINE ) * :X ) + B :LINE
END
```

There are three problems. The first problem has to do with the fact that we are using line *segments* and not lines. Look at the following picture.



PROGRAMMING IDEAS

The turtle has hit line segment #1. The turtle's coordinates satisfy the equation $y = mx + b$ for the line containing that segment. The turtle has *not* hit line segment #2. However, the line containing that segment happens to pass through the turtle's position. Therefore, the equation $y = mx + b$ for *that* line is also satisfied. `CHECK` must also check to see if the turtle is *between* the two endpoints of the line segment.

```

TO CHECK2 :LINE
IF NOT BETWEEN XCOR (FIRST POINT1 :LINE)
  (FIRST POINT2 :LINE) [OP FALSE]
OP YCOR = SOLVE :LINE XCOR
END

TO BETWEEN :THING :LIMIT1 :LIMIT2
IF :LIMIT1 > :LIMIT2 [OP AND ( GE :LIMIT1 :THING )
  ( GE :THING :LIMIT2 )]
  [OP AND ( GE :LIMIT2 :THING ) ( GE :THING :LIMIT1 )]
END

TO GE :A :B
OP NOT :A < :B
END

```

The second problem is that the turtle isn't actually one point; it is slightly bigger. This means that when the edge of the turtle hits a line, the turtle's coordinates won't match up exactly with the line's, because the turtle's coordinates are those of its center, not those of its edge. In this program the turtle has a square shape. If its `YCOR` is within seven units of the line's y value for the turtle's `XCOR`, we consider the turtle to be on the line. The number seven worked out best experimentally. Larger numbers lead to false hits. Smaller ones lead to not finding any hits at all. Our updated version of `CHECK` looks like this.

```

TO CHECK3 :LINE
IF NOT BETWEEN XCOR (FIRST POINT1 :LINE)
  (FIRST POINT2 :LINE) [OP FALSE]
OP ( ABS (SOLVE :LINE XCOR) - YCOR ) < 7
END

TO ABS :NUMBER
OP IF :NUMBER < 0 [- :NUMBER] [:NUMBER]
END

```

The last problem is that vertical lines have an infinite slope (Δx is zero in $\Delta y / \Delta x$). `SOLVE` won't work for a vertical line, because it needs a numeric slope. Also, we can't check to see if the turtle is between the x values of the endpoints of the line, because the x values are the same; there is no "be-

tween.”* So, for a vertical segment, we have to see if the YCOR of the turtle is between the y values of the two endpoints.

Instead of calling SOLVE, we see if the XCOR of the turtle is within seven units of the x value of one of the endpoints. Our final version of CHECK looks like this.

```
TO CHECK :LINE
OP IF EMPTYP M :LINE [CHECK.VERT :LINE] [CHECK.SLANT :LINE]
END

TO CHECK.SLANT :LINE
IF NOT BETWEEN XCOR ( FIRST POINT1 :LINE )
  ( FIRST POINT2 :LINE ) [OP "FALSE"]
OP ( ABS ( SOLVE :LINE XCOR ) - YCOR ) < 7
END

TO CHECK.VERT :LINE
IF NOT BETWEEN YCOR ( LAST POINT1 :LINE )
  ( LAST POINT2 :LINE ) [OP "FALSE"]
OP ( ABS XCOR - FIRST POINT1 :LINE ) < 7
END
```

Turning the Turtle

Once NEWHEAD knows which line was hit, it can figure out how much to turn the turtle. The turtle’s original heading is provided by the primitive procedure HEADING. The heading of the line is provided by

D :LINE

Recall that the angle through which the turtle should turn is therefore

$2 * ((D :LINE) - \text{HEADING})$

NEWHEAD calls FIGTURN to figure out how much the turtle should turn:

```
TO FIGTURN :L
OP 2 * ( ( D :L ) - \text{HEADING} )
END
```

*This is not exactly true. If the turtle’s XCOR is *equal* to the x values of the line, then in a sense the turtle is between the x values. This doesn’t mean the turtle is on the line segment. The problem is that for a vertical line segment, the turtle’s YCOR might not be between the y values of the endpoints of the line segment, even though the XCOR is in the right range. For diagonal lines, if one coordinate is in range, the other must also be in range.

PROGRAM LISTING

```

TO FIGH :LINE
IF ( DX :LINE ) = 0 [OP 0]
OP 90 - ARCTAN SLOPE :LINE
END

TO FIGM :LINE
IF ( DX :LINE ) = 0 [OP []] [OP SLOPE ▶
    :LINE]
END

TO SLOPE :LINE
OP ( DY :LINE ) / DX :LINE
END

TO DY :LINE
OP ( LAST POINT1 :LINE ) - ( LAST ▶
    POINT2 :LINE )
END

TO DX :LINE
OP ( FIRST POINT1 :LINE ) - ( FIRST ▶
    POINT2 :LINE )
END

TO FIGB :LINE
IF ( M :LINE ) = [] [OP []]
OP (LAST POINT1 :LINE) - ( ( M :LINE) ▶
    * ( FIRST POINT1 :LINE) )
END

TO POINT1 :LINE
OP ITEM 1 :LINE
END

TO POINT2 :LINE
OP ITEM 2 :LINE
END

TO M :LINE
OP ITEM 3 :LINE
END

TO D :LINE
OP ITEM 4 :LINE
END

TO B :LINE
OP ITEM 5 :LINE
END

TO ITEM :INUM :LIST
IF :INUM = 1 [OP FIRST :LIST]
OP ITEM :INUM - 1 BUTFIRST :LIST
END

TO BOUNCE
LEARN.LINES
SETUP.GRAPHICS
DRAW.LINES
START.TURTLE
END

TO LEARN.LINES
MAKE "LINES []
INFO
BORDER
END

TO INFO
REMEMBER GETLINE
PRINT [ANOTHER LINE?]
IF EQUALP RL [YES] [INFO]
END

TO GETLINE
TYPE [X AND Y OF FIRST POINT?]
MAKE "FP RL
TYPE [X AND Y OF SECOND POINT?]
MAKE "SP RL
MAKE "LINE LIST :FP :SP
OP FIGLINE :LINE
END

TO FIGLINE :LINE
MAKE "LINE LPUT FIGM :LINE :LINE
MAKE "LINE LPUT FIGH :LINE :LINE
MAKE "LINE LPUT FIGB :LINE :LINE
OP :LINE
END

TO REMEMBER :LINE
MAKE "LINES FPUT :LINE :LINES
END

TO BORDER
REMEMBER FIGLINE [[-158 -119] [-158 ▶
    120]]
REMEMBER FIGLINE [[161 -119] [161 ▶
    120]]
REMEMBER FIGLINE [[-158 120] [161 ▶
    120]]
REMEMBER FIGLINE [[-158 -119] [161 ▶
    -119]]
END

```

```

TO SETUP.GRAPHICS
ASK [1 2 3] [HT]
TELL Ø
PUTSH 1 :SHAPE1
SETSH 1
CS
ST
FS
END

TO DRAW.LINES
SETPN 1
DRAW :LINES
END

TO DRAW :LIST
IF EMPTYP :LIST [STOP]
PU
SETPOS POINT1 FIRST :LIST
PD
SETPOS POINT2 FIRST :LIST
DRAW BUTFIRST :LIST
END

TO START.TURTLE
SETPN Ø
PU
SETPOS [Ø Ø]
PD
SETH RANDOM 360
SETSP 20
WHEN OVER Ø 1 [SETSP Ø WHEN OVER Ø 1 ▶
  []]
LOOP
END

TO LOOP
IF SPEED = Ø [NEWHEAD]
LOOP
END

TO NEWHEAD
MAKE "L SEARCH :LINES
IF NOT EMPTYP :L [RIGHT FIGTURN :L]
SETSP 20
WHEN OVER Ø 1 [SETSP Ø WHEN OVER Ø 1 ▶
  []]
END

TO SEARCH :LINES
IF EMPTYP :LINES [OP []]
IF CHECK FIRST :LINES [OP FIRST ▶
  :LINES]
OP SEARCH BF :LINES
END

TO SOLVE :LINE :X
OP ( ( M :LINE ) * :X ) + B :LINE
END

TO BETWEEN :THING :LIMIT1 :LIMIT2
IF :LIMIT1 > :LIMIT2 [OP AND ( GE ▶
  :LIMIT1 :THING ) ( GE :THING ▶
  :LIMIT2 )] [OP AND ( GE :LIMIT2 ▶
  :THING ) ( GE :THING :LIMIT1 )]
END

TO GE :A :B
OP NOT :A < :B
END

TO ABS :NUMBER
OP IF :NUMBER < Ø [- :NUMBER] ▶
  [:NUMBER]
END

TO CHECK :LINE
OP IF EMPTYP M :LINE [CHECK.VERT ▶
  :LINE] [CHECK.SLANT :LINE]
END

TO CHECK.SLANT :LINE
IF NOT BETWEEN XCOR ( FIRST POINT1 ▶
  :LINE ) ( FIRST POINT2 :LINE ) ▶
  [OP "FALSE"]
OP ( ABS ( SOLVE :LINE XCOR ) - YCOR ) ▶
  < 7
END

TO CHECK.VERT :LINE
IF NOT BETWEEN YCOR ( LAST POINT1 ▶
  :LINE ) ( LAST POINT2 :LINE ) [OP ▶
  "FALSE"]
OP ( ABS XCOR - FIRST POINT1 :LINE ) < ▶
  7
END

TO FIGTURN :L
OP 2 * ( ( D :L ) - HEADING )
END

TO ARCTAN :X
OP 57.3*ARCTAN.RAD :X
END

TO ARCTAN.RAD :X
IF :X>1 [OP 1.571-ARCTAN.RAD (1/:X)]
OP :X/(1+Ø.28*:X*:X)
END

MAKE "SHAPE1 [Ø Ø Ø Ø Ø Ø 60 60 60 60 ▶
  Ø Ø Ø Ø Ø Ø]

```