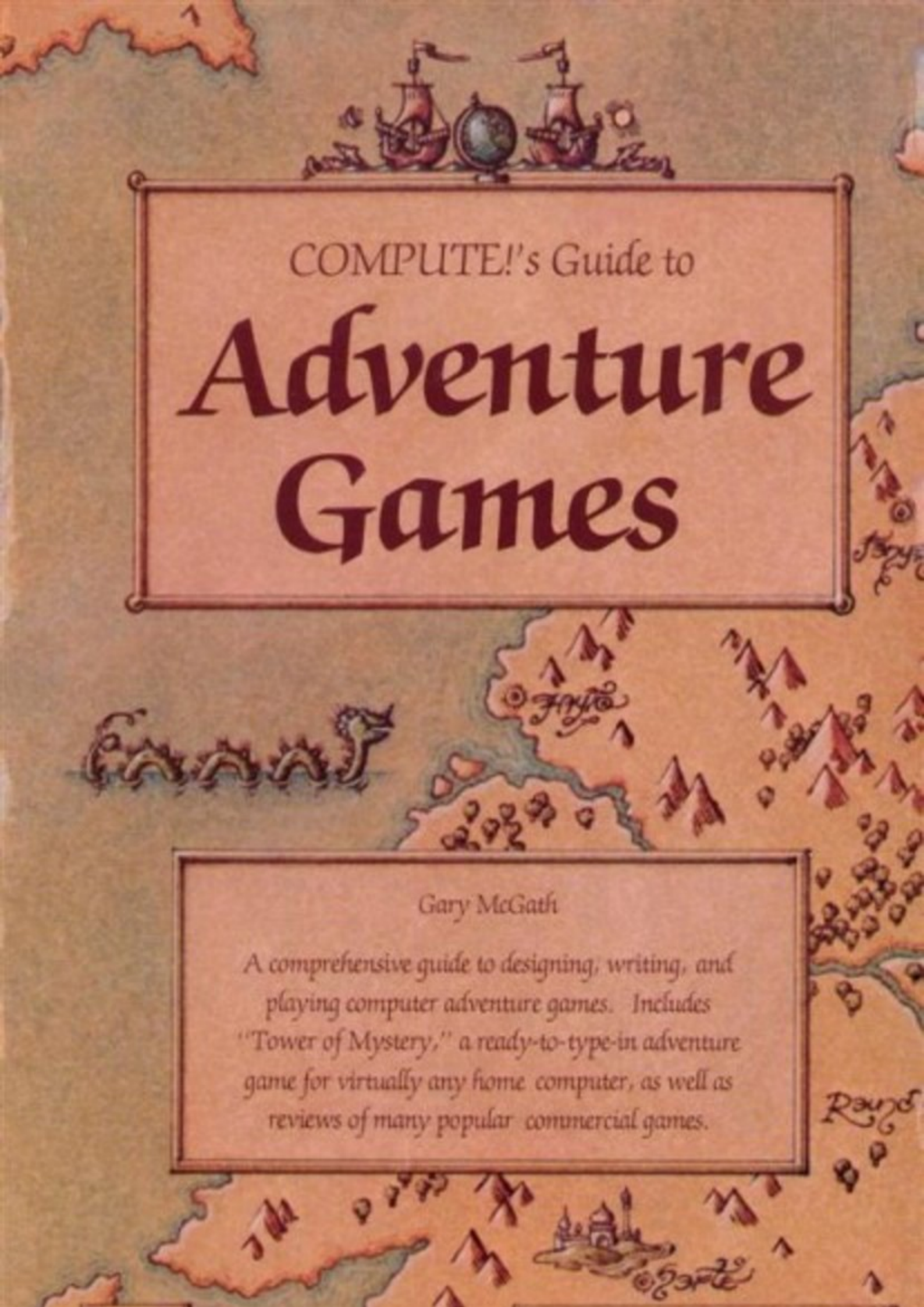


COMPUTE!'s Guide to

Adventure Games

Gary McGath

A comprehensive guide to designing, writing, and playing computer adventure games. Includes "Tower of Mystery," a ready-to-type-in adventure game for virtually any home computer, as well as reviews of many popular commercial games.

COMPUTE!'s Guide to Adventure Games

Gary McGath

COMPUTE! Publications, Inc.
One of the ABC Publishing Companies

Printed in the United States of America

ISBN 0-942386-67-1

10 9 8 7 6 5 4 3

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies and is not associated with any manufacturer of personal computers.

Contents

Foreword

You've come through steamy jungle valleys and over mountains thick with trolls. Across arid prairies and through labyrinthlike cities, you've persevered. You've fought dragons and found treasure and braved all sorts of danger. It's been a long, long trip. Best of all, you've survived.

Is this a story or a game? Both. Adventure games give you a bit of each.

COMPUTE!'s Guide to Adventure Games introduces you to those fascinating worlds that can occupy your computer's memory and your own mind. Can you really find excitement in a game made up mostly of words? Definitely. In fact, of all the games available for home computers, text adventures may well be the most engrossing. Using nothing but words (in some cases supplemented by graphics), they create computer-borne fantasies that excite the imagination in a way that few other games can match.

Experienced adventure gamers will tell you that text adventures force you to think, react, and make judgment calls. You have to reason and make decisions, with real-world results. You actually *experience* what your game character experiences.

There are some excellent text adventures available from commercial software publishers, and most enthusiasts start with one of them. To help you pick the games that are right for you, this book includes reviews of several of the most popular adventure programs.

But it doesn't stop there. You'll find hints on basic game structure and specific tips on programming to help you create text adventures of your own. You'll also observe many of these techniques in action in "Tower of Mystery," a complete text adventure program ready for you to type in and run. Versions are included for virtually every popular computer.

The author has had wide experience with text adventure games. From early experience with *Adventure*, the granddaddy of all text adventures, his infatuation with the genre has grown into an incurable habit. Now, with *COMPUTE!'s Guide to Adventure Games*, you're about to discover why.

Acknowledgments

Thanks are due to many people for providing information and ideas for this book. Special thanks go to John Baker, Kevin Bernier, Denise Bouley, Stu Galley, Scorpia (for more than just the material that appears under her name here), and especially to master adventurer Steve Wright. In addition, I would like to thank the many members of the CompuServe Game SIG who have widened my knowledge of adventuring.

This book is dedicated to all the creators of new worlds.

Introduction

One of the major fringe benefits of working at MIT's Laboratory for Computer Science in 1976 was being close to ARPAnet. ARPAnet was the first nationwide computer network, and some of the most strenuous testing of its capabilities came not from planned experiments but from the volumes of messages and programs that were passed from site to site.

One of the most popular programs that we received over ARPAnet was a new game, by Will Crowther and Don Woods, called *Adventure.* I was hooked from the start and spent many weekends at the lab trying to find my way through the Hall of Mists, past the Troll Bridge, and out of the maze of twisty little passages.

By 1977 I had moved on to a job in industry, but still made regular weekend visits to LCS. On one of those visits, former co-worker Dave Lebling suggested that I try a new game that he and some friends had been working on. This game, which he described as an enhanced successor to *Adventure,* was called *Zork.*

It was fun. I enjoyed being able to type in four-word commands instead of the two-word commands that *Adventure* was limited to. After burgling my way into the house, I spotted a rug and tried lifting it. This earned me only a rebuke from the program, so I wandered off into the woods and remained lost. Dave encouraged me to try the rug again, and the next version of the program explained to rug-lifters that the rug was just too heavy. But *moving* the rug uncovered a trap door and led the way to the riches of the Great Underground Empire.

Knowing the need to save batteries, I carried a lamp down but didn't turn it on yet. A couple of turns later, I was told, "It is pitch dark. You are likely to be eaten by a grue." This, I thought, was the time to see what *Zork* was really good at. So I typed in, WHAT IS A GRUE. The program immediately responded with an explanation of the nasty creatures, including their dietary habits. After that, it was a long time before I gave another thought to *Adventure.*

That was the beginning of an incurable habit, but I had to spend the next couple of years suffering withdrawal symptoms. Personal computers in 1977 were clumsy to use, and mass-market software was a concept no one had yet dreamed up. But the breakthroughs came at last. When the creators of

Zork formed Infocom and started selling their adventures for home computers, I started buying them. Soon I discovered Scott Adams and other adventure creators as well. Since then, I have spent many long evenings trying to find hidden treasures and unmask murderers. Eventually I realized that other adventure enthusiasts could benefit from what I had learned, and the result is this book.

I have tried to cover every major aspect of adventure games: their history, how to play them, how to write them, and where they might lead in the future. I've also reviewed many of the major adventure games.

As I worked on the book, I realized that there was one glaring deficiency in my experience: adventures with graphics. The TRS-80 Model III, which is my mainstay computer, can display only the crudest black-and-white pictures; adventures that involve high-resolution graphics just aren't available for it. To remedy this, I went to a friend whom I have never met: Scorpia, the sysop for the Game Special Interest Group on CompuServe. We had many discussions concerning computer adventures, and her chapter concerning Sierra On-Line's graphic adventures adds greatly to this book.

I have two hopes for this book. First, I trust that readers will finish it knowing more than before about how to enjoy—and perhaps write—computer text adventures. Second, and with unapologetic selfishness, I hope that it will play some part in encouraging software creators to come out with more and better adventures.

Stories in Software

Stories in Software

man has been murdered. Anyone in the household could have done it—or perhaps even someone from outside. There are clues, but only the sharpest detective will be able to spot them and build them into a case against the murderer.

Will the detective succeed? That is up to you—for *you* are the detective. You are, in fact, reading a story; but the story is unfolding on the screen of your computer, in response to your commands. You are playing one of the most engrossing and demanding kinds of computer games, the text adventure.

Computer text adventures fall into the very broad category of *simulation programs.* Anyone who has played video games has dealt with graphic simulations of missile attacks, jungle chases, automobile races, and the like. Text adventures also simulate tense situations, but they interact with the user through words more than through pictures and joystick movements. People may disagree about whether the computer should use pictures (graphics) or words to describe the simulated situation, and many adventures use both. But for telling the computer what *you* want to do, there is no question that words are far more flexible than any joystick or trackball.

You, the player, are the main character in the story that a computer adventure lays out. You are presented with a situation and a goal. Sometimes the goal is a single object, such as catching a murderer or escaping with your life; at other times it's a matter of accumulating as many points as possible by solving problems and picking up treasures along the way.

The program guides you along by giving you a text description of your character's surroundings. It might tell you that you are in a large cave, or a broom closet, or—heaven help you!—a maze of twisting, identical passages. It will single out interesting objects that you might be able to pick up or otherwise use, and it will point out directions in which you can go.

In response to each description, you type in a verbal-style command indicating the action you want your character to

take. Adventure programs vary tremendously in the vocabulary and complexity of commands they will accept. Most adventures are restricted to simple verb-object commands, such as TAKE BOOK, ATTACK DRAGON, or GO NORTH. However, the most sophisticated programs allow commands with adjectives, multiple objects, and modifying phrases, such as TAKE ALL BOXES EXCEPT THE RED ONE.

Your character can carry objects. When you give a command to TAKE an object, the program checks whether you can in fact pick it up, then adds it to your inventory if you can. Having the right object at the right time (for instance, a sword if an unfriendly troll shows up) can be vital to your success. There are limits, of course, on what you can carry; the program will let you carry only so many objects (or, if it's clever enough, only so much weight), and there are some objects that are just too big to pick up.

An adventure consists of a series of puzzles to solve. The most common challenge is simply figuring out how to get somewhere. You may have to find a secret passage, open a lock, get past a guardian, or answer a riddle. Then, after getting to your destination, you might find more surprises. You could encounter a locked box, an enigmatic machine, or a dragon sitting on a treasure. In each case, the solution lies in having the right tools and thinking of the right action.

Unlike video arcade games or realtime action adventures, some text adventures (which I'll refer to from now on simply as adventures) invite you to sit back and think instead of calling on you to make quick reflex actions. Most adventures let you take as long as you want to decide on a command; game time doesn't pass while you're making up your mind. The better ones even have a save feature that lets you save the current state of the game on disk or tape, turn off your computer, and come back later to continue. But adventures aren't for the passive-minded. You have to be observant and inventive to get through one.

At its best, an adventure is a literary form that is unique in requiring the reader's participation. In fact, some books have been written in imitation of adventure games. These ask the reader to select at intervals one of several possible courses of action and then to turn to a corresponding page. But they don't really demand active participation; the only choices are the ones listed. The adventure game doesn't enumerate your

alternatives in advance the way an adventure book does; the right thing to do may be one that doesn't occur to you for a long time. The thrill of finally thinking of that solution and finding that it works is the special pleasure of adventures.

Some people seem to think that enjoyment means turning off their minds and watching something passively or engaging in meaningless activity. But for others, enjoyment means giving their minds a change of pace and scene while still staying alert and curious. Adventure games offer this kind of change of pace—the chance to experience a different kind of world and try to solve the problems that it offers, as well as the chance to take risks without the unpleasant consequences of real-life failure. It is escapism, perhaps, in the sense of an escape from the routine of daily life. But it's not an escape from thinking.

In the Beginning

Will Crowther created the first adventure game—the preliminary version of *Adventure*—in 1975. It was written in FORTRAN on a Digital Equipment Corporation PDP-10 computer, and its popularity spread quickly as Crowther made it available nationally through ARPAnet. (ARPAnet was the first nationwide computer network, linking many computer centers at universities and research facilities around the country.)

Don Woods of Stanford University made some major enhancements to Crowther's program, and the result was the program that is still regarded as the standard version of *Adventure*. Enhancements and adaptations have gone on, though, and today the game is available in numerous versions for every sort of computer.

Adventure has all the features that are now traditional parts of the genre. The player gives two-word commands to direct the character's actions (through an underground complex). The more treasure the character finds and brings to the surface, the more points he gets. And if he solves the entire labyrinth, he is finally carried away on the shoulders of the cheering elves.

Crowther's inspiration for *Adventure*, with its mixed bag of mythologies and its environment of mysterious underground passages, came largely from a noncomputer game, which was as revolutionary in its own field as *Adventure* was

in computer games: the role-playing game called Dungeons and Dragons.

Tactical Studies Rules (now known as TSR Hobbies, Inc.) published the first edition of Dungeons and Dragons in 1974. The rules in that edition were badly written and often grossly ambiguous, but players were thrilled by the concept. D&D is an open-ended game in which the players assume the roles of characters in a story and can have those characters attempt any action whatsoever. The game is controlled by a gamesmaster, who uses tables, dice, and personal judgment to decide on the effect of a character's efforts.

In more traditional games, including the most sophisticated war games, the player's choices are set out in advance. If a hero piece moves next to an ogre piece, the hero can attack the ogre, run away from it, or even befriend it, if a specific rule is included. The only options are the ones spelled out in the rules. But in Dungeons and Dragons, a player can try anything he might think of. He might try to sneak past the ogre, intimidate it by waving a weapon, or offer it a reward. The gamesmaster has to consider the character's specified abilities and the nature of ogres to assign probabilities to the various outcomes; then he rolls the dice (with those probabilities in mind) to determine what actually happens.

The open-endedness of role-playing games can produce all kinds of surprises—and as often as not, it's the gamesmaster who is surprised. Consider the following dialogue from a hypothetical D&D game:

Gamesmaster: You advance north 50 feet and come to the end of the corridor. There's a wooden door there.

Garmon (a fighter): I open the door.

(Gamesmaster rolls dice to fool the players, even though he knows that the door is locked.)

Gamesmaster: It doesn't even budge.

Melkar (a thief): I'm checking the door to see if it's locked or anything.

Gamesmaster: It's locked, all right. You see a bolt going into the door frame. There's also a keyhole.

Melkar: You said the door's made of wood?

Gamesmaster (anticipating): It looks like very tough wood, probably too tough for an ax.

Melkar (to the other players): Well, we've got that bottle from the alchemist, which is supposed to warp any wooden

object entirely out of shape. Shall we try it on the door?

(The gamesmaster is suddenly worried but endeavors to look impassive. A dialogue ensues in which the players agree to use the bottle.)

Melkar: I'm going to pour the stuff onto the door, starting close to the lock side, about a foot above the lock. I'm pouring slowly, to make sure it soaks nicely into the wood.

(The gamesmaster considers. Is this wood so tough that the chemical will just run off it? Might it just warp the door more firmly into the lock, so that even when they find the key they were supposed to use, they won't be able to open it? Or will it warp the door out of the lock, as the players hope? He assigns a range on the possible dice rolls to each of these alternatives, then rolls.)

Gamesmaster (glumly): You pour the contents of the bottle onto the door. It starts to buckle, and the lock snaps loose from the frame. (He smiles as he has another thought, then rolls the dice.) But some of the potion splashed onto your hand, Melkar. You took four points of damage.

Melkar: No big deal. Now let's open the door and see what's in here.

(The players have now entered the central chamber without having met the Wizard who was supposed to have provided them with the key to the door. This sets up a whole new situation, which the gamesmaster must now handle.)

Obviously no computer game can be as completely open-ended as a game managed by a human being. But *Adventure* did capture some of the spirit of D&D by not laying out a set of usable commands before starting the game. The player can try any command at all, and it just might work, or at least elicit an amusing response.

The second milestone in computer adventures was a program called *Zork*, designed by Dave Lebling, Marc Blank, Tim Anderson, and Bruce Daniels. *Zork* also used a D&D-type milieu, but its authors had several advantages over *Adventure*'s creators. First, the language they used was MDL (humorously called MUDDLE) a language similar to LISP that has much better facilities for expressing data relationships than FORTRAN does. Second, they were working just downstairs from MIT's Artificial Intelligence Laboratory, so they had ready access to previous work in handling English-language input.

Zork's creators gave it a command-handling facility that

went far beyond *Adventure*. They added prepositional phrases, indirect objects, adjectives, multiple direct objects, and compound commands. Prepositional phrases allow specifying an implement to be used in an action (ATTACK DEMON WITH TOOTHPICK). Adjectives distinguish between objects of the same category (OPEN LARGE BOX). Multiple direct objects are convenient when grabbing or discarding several things at once (TAKE DIAMOND AND AX AND HAT). Multiple objects are implied in the command DROP ALL, and further refinements come from combining adjectives, prepositions, and multiple objects (TAKE ALL GEMS EXCEPT THE BROKEN ONE).

Zork also expanded the scope of the nonplayer characters controlled by the program. *Adventure* has several dwarves and a pirate who move around randomly and take specific actions, such as throwing knives at the adventurer; it also has some characters in fixed locations who perform specialized functions, such as blocking a bridge. But in *Zork*, the pirate is upgraded to Thief, a character who performs a number of unpleasant functions besides taking treasure away from you. For instance, he may pick up objects and drop them off in another place to confuse you. There is also a robot, who can move at your command and who might be able to do jobs that a human could not manage. Other creatures (including a troll and a Cyclops) do not move, but their capabilities are more varied than their counterparts in *Adventure*.

Combat with these creatures is especially interesting, for the battle is not decided by a single blow. Instead, each fighter accumulates wounds, and several wounds are usually needed to kill. In each turn during a combat, both the adventurer and his adversary can attack; the effectiveness of the attack is affected by the attacker's health and his choice of weapon, as well as by a random factor. One side will usually get the upper hand and then finish off his wounded opponent; but if the attacker is disarmed at a crucial moment, he can suddenly find himself in trouble.

Another feature found in *Zork* is the use of vehicles, which can take the adventurer to places unreachable on foot. You must BOARD the vehicle before using it, then issue an appropriate command, such as LAUNCH, to make it carry you. You don't necessarily have full control of the vehicle; this can be inconvenient if you are in a boat rushing toward a waterfall.

The original version of *Zork* was created in 1977; the name incidentally, is just a nonsense word of the sort popular in MIT's computer labs for spur-of-the-moment generic names. A couple of years later, its creators implemented the Z-machine. The idea of the Z-machine was to write *Zork* for one imaginary computer and then write programs for real computers that would let them run as if they were Z-machines. (This is the same concept as the P-code system that lets compiled Pascal programs be moved from one computer to another.) The Z-machine software included a high-level, LISP-like language called ZIL (*Zork* Implementation Language) and an assembly language called ZAP (*Zork* Assembly Program). The first Z-machine emulator was created for the TRS-80 Model I, followed shortly by an Apple version. The original *Zork* was reduced to a smaller program, with the intention (since fulfilled) of releasing the remaining portions as parts II and III.

Zork's creators formed Infocom to sell their adventure programs. Originally the marketing was done through Personal Software; later, Infocom set up its own marketing and direct sales operations.

Since releasing *Zork*, Infocom has continued to enhance the command-handling capability of its new adventures. In addition, starting with *Deadline*, it has introduced real plots into its adventures. True, *Zork* has some rudimentary elements of plotting, including the adventurer's ongoing war with the Thief. But *Deadline* and subsequent Infocom adventures have tied all the events into a single, overriding purpose—catching the murderer, stopping a plague, or whatever.

As microcomputers grew in popularity, Scott Adams broke ground in a different direction with his Adventure International series. Many hobby computers have no more than 16 kilobytes of storage and can input programs only from an ordinary audio-cassette recorder. This puts programs like *Zork* and even *Adventure* out of their reach. But Adams showed that with clever programming techniques, enough material can be crammed into that amount of storage to make an exciting adventure program.

Adams's first adventures used an interpreter program, written in BASIC, that would read a data file and spin out the adventure in accordance with the data in that file and the player's input. This method provided much more effective use

of data storage than straightforward programming of the adventure in BASIC would have provided.

Since then, Adams has rewritten his interpreter to run in the machine language of a variety of computers. This rewriting has enhanced the speed and compactness of the adventures while greatly reducing the amount of time they take to load from cassette tape. For those interested, the original BASIC version of Adams's *Pirate's Adventure* was published in the December 1980 issue of *Byte*.

All 12 programs in Adams's series use a standard screen format for presenting information. The top part shows the character's current locations ("I am in...."), followed by a list of "visible items." The remainder of the screen, which may vary in size depending on the size of the items list, is a scrolling area for commands and responses.

The descriptions and responses are terse; Adams prefers having more rooms and a larger command vocabulary to giving lengthy descriptions. The same phrase will often be incorporated into several different responses, at a savings in memory over having each response be different. For instance, the phrase "won't let me" might appear in the reactions of several different guardians.

As memory becomes cheaper and home computers with disk drives become more common, the restrictions that affected Adams's earliest adventures are becoming less of a problem. Adventure publishers are looking for ways to take advantage of the additional capability of expanded memory. One path that several publishers are pursuing is the *Zork* approach, which has more descriptions, a larger vocabulary, and more complex command handling. Such adventures as *Cyborg* and *Empire of the Over-Mind* have gone this way to varying degrees. Another popular route, though it has not always been put to the best possible use, is the addition of graphics.

Graphics, when used to complement the text and not just illustrate it, can add a great deal to an adventure. One of the best examples is *Prisoner 2*, in which the situation is presented largely through graphics, and commands directly affect the picture. Other adventures provide animated sequences to get from one location to another. Too often, though, graphics are used just to illustrate the text with still pictures. Since the best home-computer graphics today fail to live up to the quality of a Saturday-morning cartoon, this is a rather ineffective use of

computer storage. If the author of an adventure really wants to provide illustrations, he or she would be better off to include a booklet and have the program say, "Look at page 23" (though this does have the disadvantage of letting players peek before they've gotten to their destination). Avalon Hill's *G.F.S. Sorceress* makes good use of this method.

Complicated command handlers do take a lot of programming effort, and the publisher has to be able to justify the cost. On the other side of the ledger, too, computer-game publishers may be hesitant to make their product too complex. Still, several publishers have been contributing to the slow but steady growth of more sophisticated adventure programs. This growth suggests that there *are* computer gamers who can handle the English language and who regard a game as a chance to think about something different rather than to stop thinking.

Origins

The English-language-handling capabilities of adventure programs, and especially of *Zork* and its descendants, owe a great deal to the work that was done in artificial intelligence (AI) in the 1960s and '70s. The researchers in AI put a great deal of work into handling natural language input, meaning input in English or other human languages, and some of the results are remarkable.

The best known of these AI programs is *Eliza* by Professor Joseph Weizenbaum at MIT. *Eliza,* named for the heroine of Shaw's *Pygmalion,* accepts input text in English and produces responses based on a script. The best-known *Eliza* script simulates a psychotherapist, and the program that uses this script is often known as *Doctor.*

(There are several programs available for microcomputers that are called *Eliza,* and that emulate the original program's *Doctor* script to a greater or lesser extent. But most of these programs lack any script-handling capability, or at least don't let the user know how to create scripts. Thus, they miss the major point of the original *Eliza.*)

According to Weizenbaum, who described the program in "*Eliza*—A Computer Program for the Study of Natural Language Communication Between Man and Machine" (*Communications of the ACM,* Volumn 9, No. 1, January 1966), "The gross procedure of the program is quite simple; the text is read

and inspected for the presence of a *keyword*. If such a word is found, the sentence is transformed according to a *rule* associated with the keyword, if not a content-free remark or, under certain conditions, an earlier transformation is retrieved. The text so computed or retrieved is then printed out."

The program might turn I HATE YOU into YOU HATE ME, for instance, or cue on a word like BROTHER or MOTHER to evoke a standard question about the user's family. It is easy enough for the user to provide input that drives the program to unintelligible responses, but some sessions have let the program seem surprisingly human. A piece of sample dialogue, presented by Weizenbaum, follows, with the program's responses in uppercase:

I need some help, that much seems certain.
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
Perhaps I could learn to get along with my mother.
TELL ME SOME MORE ABOUT YOUR FAMILY
My mother takes care of me.
WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU
My father.
YOUR FATHER
You are like my father in some ways.
WHAT RESEMBLANCE DO YOU SEE

Eliza showed that a computer program could scan English-language input and produce interesting results; however, its approach is entirely different from the method needed for any sort of command input. Rather than trying to exhaustively analyze the syntax of each input line, *Eliza* simply looks for key words and phrases and makes the best of those. It will never admit to not understanding a sentence. Weizenbaum has been somewhat embarrassed by the "aura of magic" that the program has acquired and stresses that it is "merely a translating processor."

The program's strategy of concealing its failures is appropriate to generating that aura.

"But to encourage its conversational partner to offer inputs from which it can select remedial information," Weizenbaum notes, "it must *reveal* its misunderstanding." A program that does this successfully—that, for instance, can tell the user it doesn't understand CLOBBER THE DIRTY CROOK and indicate the exact point of failure—has to take quite a different approach from *Eliza*.

In addition, *Eliza* retains only a little information from previous input. It does not change its model of the situation in response to every input line. Such information as it does retain has very little structure to it; it simply allows the program to make occasional references back to earlier parts of the dialogue. For instance, some time after the user mentions an illness he once had, the program might attempt to liven the dialogue with the line, AWHILE AGO YOU MENTIONED YOUR ILLNESS. WHAT ELSE DOES THAT SUGGEST TO YOU? Even the most elementary adventure program has to do more than *Eliza* in this regard.

A more direct ancestor to adventure games, particularly to *Zork*, was T. Winograd's *SHRDLU*. This program, created as part of his 1971 Ph.D. thesis, accepts complex commands for moving blocks of various shapes around. The program kept track of the blocks only in an internal representation; but with today's robotic technology, if not that of 1971, it would be feasible to have a mechanical arm that actually obeyed the commands.

You might, for instance, tell *SHRDLU* to pick up the big red block. This causes it to perform a whole sequence of actions to carry out the command. First it must determine which object is the big red block; then it must check if any objects are sitting on the block and move them to the "table" that supports all objects. After that, it can designate the indicated block as being picked up.

Patrick H. Winston, in *Artificial Intelligence* (Addison-Wesley, 1977) cites four categories that are significant in *SHRDLU*'s "blocks world." These are *objects* (such as specific blocks and pyramids); *relations between objects* (such as IN-FRONT-OF and SUPPORTED-BY); *relations between objects and intrinsic properties* (such as color and size); and *actions* (such as PICK-UP, PUT-ON, and STACK).

These capabilities are necessary to any simulation that involves manipulating physical objects in a realistic way. In an adventure program, you need to know whether the dragon (object) is on the pedestal (relation between objects), which pedestal you are talking about (its intrinsic properties), and whether it is possible to open the pedestal (action) without doing something else first.

SHRDLU, in its restricted context, has capabilities that go beyond any adventure program yet created. Its syntax

handling allows very complex commands. Relations between intrinsic properties, such as "taller than," are within its capacity, as are alternatives such as "either a green cube or a pyramid."

On the other hand, *SHRDLU* deals with a very small class of objects, which occupy only one "room" and which always remain passive except when manipulated by a command. A block will not explode two turns after you put it in the box, nor will a second robot walk in and steal it. There is no question of what implement to use for an action, since the robot has only one hand. *SHRDLU* is an expert in a very narrow field; adventure programs must be generalists by comparison.

Computer Fantasies

When talking about *Eliza, SHRDLU,* or adventure programs, it is important to realize that these programs *do not think.* Each of these programs is set up to deal with its input and generate output according to certain rules that cannot be broken.

Some artificial intelligence programs, such as *Eliza,* have made a great impression on naive users, who have thought that the computer really "understood" what they were telling it. Weizenbaum has been honest enough to discourage this impression; unfortunately, creators of some other computer programs seem to seriously believe that what the programs are doing is "thinking" or "understanding."

Working with these programs is a good way to dispel any illusions about what they are doing. Enter commands which go beyond their designated capabilities, and the programs either return a stock confession of ignorance or issue nonsensical responses.

Deliberately playing an adventure in this way—not trying to solve it, but to probe the limits of its present capabilities—can be fun as well as help to break down the illusion that the computer "knows" something. With a two-word command handler and a tiny vocabulary, overcoming the program is so easy that it doesn't provide any satisfaction. But if the program is powerful enough that you can't be sure whether a command really is beyond the program, then testing it can become an enjoyable game in itself.

Player: Drop dead
Program: I DON'T SEE ANY DEAD HERE

Granted, even the best adventure programs don't match the complicated conversational programs coming out of artificial intelligence labs. Still, the pattern is there. The program handles only a certain amount of subject matter and can handle it only according to certain rules. When the input touches on a borderline or unusual case of the subject matter, the success of the program depends on how complete the rules are in that area. If the subject matter is well delimited—a chess game or the manipulation of blocks, for instance—then the programmer doesn't have to worry very much about the borderline cases. But when it is wide open, as it is when the program is trying to simulate the actions of a human adventurer, then there will always be cases that the program can't handle. Discovering this truth helps to take much of the mystique away from complex computer programs.

An adventure game can be played on a number of levels. You can treat it simply as a game of logic and try to solve the puzzles. You can look at it as an unfolding story in which your actions shape your character's fate. Or you can offer it your own challenge and see how far you can push it before it reaches its limits.

The games also promote a special sort of interaction among people, even though they are played solitaire. In playing an adventure, you may find yourself stuck on some point; if you've tried your best, there's nothing wrong with asking a friend who's already played the game to offer you a hint. Experienced players delight in coming up with subtle, devious hints in response to inquiries. A good hint still lets the player solve the puzzle mostly on his or her own; it just gives another perspective on the situation. *Pac-Man*, on the other hand, doesn't lend itself to much conversation beyond asking what fruit is in the tenth maze.

Can an adventure game be played only once? This is like asking if a novel can be read only once. Usually you would rather go on to a new book than reread an existing one; but if a particular book contains more subtleties than you could enjoy in a single reading, you may well want to read it over and over again. The same is true, in a somewhat different way, of a good adventure game. Even after you've reached the highest score, you can go back and try courses you hadn't taken before just to see what might happen. If you got through a situation by luck, you might want to look again in order to

see whether the author planted some subtle clue you could have used.

No other class of computer games offers such a multi-faceted source of excitement. Chess sharpens your analytic powers, war games test your sense of strategy, word games stretch your ability to recognize words, and arcade shoot-'em-ups challenge your coordination and reactions. But none of these offers the range of satisfactions to be found in a top-quality adventure.

Welcome, then, to the dungeon, to the scene of the crime, and to other worlds.

What Makes a Good Adventure?

What Makes a Good Adventure?

In judging the quality of any home computer software, you should keep one point in mind: publishers of this kind of software suffer a theft, or piracy, rate that would frighten any store owner. It's hard for someone to justify large development costs when he knows that a large percentage of the people who take advantage of his efforts won't be paying him anything. Under the circumstances, we should be grateful that people keep producing new home computer programs at all, instead of moving into a more financially secure area.

Evaluating adventures is trickier than evaluating most other kinds of software. A strictly practical program, such as an accounting package or word processor, is judged by how easily it gets the job done. Even with an ordinary game program, the critic expects the commands or controls to be as easy to use as possible. And in either case, the documentation is supposed to be complete; if you aren't told how to print a letter or shoot at the invaders, you have every reason to complain.

With an adventure, on the other hand, figuring out how the program works is part of playing the game. You don't know what all the commands are or whether they'll always work. The program might even go into a special mode without warning, requiring you to use an entirely different set of commands until you get back into the normal situation.

Even so, there are definite standards for a good adventure, just as there are standards for a good novel. First of all, the difficulties in an adventure should lie in the situation it presents, not in getting the program to understand what you mean. There is nothing more frustrating than knowing exactly what you want to do but being unable to convey your intent to the program. To avoid this frustration, the program should have a powerful command parser, a large vocabulary, and understandable error messages. As the authors of *Zork* put it, "the game should simulate the real world sufficiently well so that the player is able to spend most of his time solving the

What Makes a Good Adventure?

In a game with a good parser, you can express your actions directly. In an adventure that accepts only two-word commands, it is difficult to express an action like putting a coal into a fire. The program must either arbitrarily assume that DROP COAL puts it into the fire or ask for a second command to specify the destination of the coal. Being able to type DROP COAL IN FIRE is much more convenient.

A large vocabulary can increase both the number of available actions and the number of ways the player can express those actions. After all, kicking an ogre *might* do something useful; why not let the player try it? Having a small vocabulary encourages playing the program rather than playing the adventure, since it lets the player zero in on the correct actions just by noticing that certain words are in the game's vocabulary. If the vocabulary is large, the mere fact that a word is acceptable doesn't mean that it's necessary to solving the puzzle.

Good error handling makes it easier to figure out just where the program's limits lie. If the response is simply "I don't understand," you can't tell whether one of the words you used isn't in the program's vocabulary (or *which* word) or whether the way you combined the words made no sense to it. This means you have to rephrase a command several times before you can figure out why you weren't getting through.

The program's responses should be clear and grammatical. Exits and objects should be clearly noted, unless they are supposed to be difficult to find. Descriptions should convey a sense of your actually being there, just like descriptive passages in any other work of fiction.

They should also follow the rules of the language on grammar and spelling. Sadly, a number of adventure publishers have allowed errors to slip into their games—errors that even the sloppiest magazine editor would catch at a glance. The trouble is that nearly all adventure authors are programmers first and writers second.

Responses shouldn't be misleading. It's acceptable for the program to misdirect the player, to disguise the nature of things, but it shouldn't produce a response that flatly contradicts the way the adventure world is intended to be. That may sound obvious, but it's tricky to accomplish because of all the possible actions that a player can attempt. The first version of *Zork*, for instance, responded to attempts to LIFT the rug by

saying that fiddling with it wasn't useful, when in fact there was something hidden under it. The program handled the intended verb, MOVE, correctly, but the writers just hadn't considered the possibility that someone would try LIFT.

If the adventure includes graphics, they should do more than just illustrate what the text says. They should supply additional facts or at least expand on the ones the text mentions. Many graphic adventures do this by including items in the graphics but not mentioning them in the list of items present. In many cases, the graphics can also help to clarify the geographic layout. Some adventures, however, add graphics only as an afterthought. In those cases, they only slow things down and leave less room on the screen for text.

The speed at which the program runs is important. If you have to wait ten or fifteen seconds after each command, it's easy to get bored with the adventure. This can be a problem if the program is constantly bringing in graphics or text overlays from the disk, especially on systems with slow disk drives. The language in which the adventure is written can affect its speed; any program that runs under a BASIC interpreter is going to be slower than an equivalent machine language program.

Finally, the quality of the theme, story (if any), and associated puzzles is all-important. Not all adventure programs spin out a story; some just present an assortment of puzzles in a common setting. But in either case the situation should hold the player's interest, or the program degenerates into a dry exercise in problem solving. In *Adventure,* much of the pleasure comes from the sense of going deeper and deeper into the cave and discovering unexpected passages. Monsters and treasures aside, it conveys the feeling of exploring a spectacular area. In *Witness* and other mystery adventures, it comes from interacting with the characters and discovering their motives.

To have this quality, the adventure should fit together as a self-consistent world. This means that the puzzles should play fair, but it includes much more. Elements that are foreign to the game's milieu shouldn't intrude. If the setting is based on science fiction, a puzzle shouldn't arbitrarily throw in magical elements. If it follows the world of Greek mythology, Wotan and Brunhilde shouldn't appear without a good reason.

Deliberately mixing milieus can be the basis of an amusing story, as in Poul Anderson's novel, *Operation Chaos,* which gleefully combines magic and science fiction. But the mixing

should be intentional and well prepared, or the player will get the feeling that the author is just trying to confuse him.

The difficulties that the player's character encounters should make sense in terms of the situation. It makes sense, for example, that an elevator button might be gimmicked so that trespassers will have trouble getting to restricted areas. However, it doesn't make sense to have the elevator button at the other end of the building. The player should be able to recognize patterns and follow them through logically. If he's reduced to trying every possible action at random, then the adventure (or at least that particular puzzle) is a failure.

The adventure should also create the illusion that the player's character is a clever person who anticipates difficulties, senses dangers, and deals with them successfully on his way to victory. When the player has finally solved the adventure and can play it from beginning to triumphant end, the actions he takes along the way should seem reasonable to someone looking over his shoulder.

This means that every danger should be preceded by signs that, at least in hindsight, can be recognized as a warning. It means that the character shouldn't have to take actions that are motivated only by information he gained in a "previous life."

Some adventures give the character a limited number of resurrections; this makes it more likely that he will learn from fatal mistakes. However, such an approach makes sense only in a capricious world like that of *Adventure* or *Zork*, or in one in which science, magic, or mythology permits a return to life, such as *Lords of Karma*. Adventures with more natural environments don't have this recourse.

Consider, for example, the case of a room full of poison gas. The way to get through the room is to give the command HOLD BREATH before entering. If the character has no reason for holding his breath except that he choked to death in that room the last time he played, his actions become illogical.

However, things can be kept reasonable if the description of the previous room states that wisps of green mist are coming from under the door. Giving the command SMELL MIST might elicit a stronger warning, and then it would make sense that the character should take precautions. The point isn't that a really good player should be able to get through the adven-

ture on the first try, but that the character should stay within the bounds of the game's reality.

So what is a good adventure? A good adventure is one in which the player can vicariously experience the sense of *being* the character in a story. To the greatest extent possible, the player shouldn't have to worry about the right phrasing. He should be able to feel that his opposition is the situation itself, rather than a programmer who's trying to confuse him. The adventures currently on the market vary tremendously in how closely they approach this ideal, and even the best have a long way to go.

Infocom Adventures

Infocom Adventures

In the world of adventure programs, certain families stand out enough to deserve overall discussions in addition to reviews of their individual members. Among the most noteworthy of these is Infocom's Interlogic adventure series.

Infocom's adventures use text without graphics, a fact in which the company's advertisements take great pride. A typical Infocom ad states, "We draw our graphics from the limitless imagery of your imagination—a technology so powerful, it makes any picture that's ever come out of a screen look like graffiti by comparison." Sticking to text has another advantage: it makes it much easier to put the same adventure out on many different computers.

In the realm of text, Infocom is considered by many clearly to be the industry leader. Its products are unmatched in command handling, story construction, and literary quality. Commands can use adjectives, prepositions, and multiple objects. PUT THE RED CARD ON THE TABLE, GIVE THE RING TO THE WEASEL-LIKE ALIEN, and TAKE ALL BOXES EXCEPT THE WOODEN BOX are typical of what the programs can handle. Each program "understands" more than 600 different words.

The programs make use of the context in interpreting commands. Suppose, for instance, that you want to attack a troll with your sword. Then you could type the whole command, ATTACK TROLL WITH SWORD. Aternatively, you can type only ATTACK TROLL. If you have only one weapon (a sword, for instance), the program will add the message (with the sword) to let you know what assumption it has made. If you have more than one weapon, it will ask what you want to attack the troll with. You don't have to retype the whole command at this point; you can just type SWORD or KNIFE or whatever your choice of weapon might be.

Another way to initiate an attack against the troll would be to type SWING SWORD. The program will then look for a possible target. If the troll is the most reasonable target, it will complete the command and let you attack the troll.

There are hazards, though. In an early version of *Zork I*, if

there were no better target around, the program would give the message (at the you) in response to SWING SWORD—in fact, to swinging anything, however harmless—and cheerfully let you commit suicide. The complexities of a program often lead to surprises, even to the people who wrote it.

Infocom's developers have been learning from experience with such unexpected results, and they have incorporated a few improvements into each new product. Recently the company went back to *Zork I* and re-released it with many of the enhancements that they had created for more recent games.

Each of the adventures has a similar format; the text scrolls up the screen, and a status line at the top displays such information as the name of the currently occupied room and the player's score. Many of the descriptions are long, and the player is prompted to hit a key for MORE if the text is so lengthy that it would scroll off the screen before he could read it. A standard feature is the ability to save the state of the game on any return and later restore it. The number of available save files varies from one computer system to another.

Many of the adventures are populated by characters who are busily moving about, even when the player's character isn't around to see them. It is possible to address these characters; for example, you could type FRED, TELL ME ABOUT MR. SMITH. The limitations of the technology show up more in talking to characters than in straight command handling; there are a lot of things you'd like to ask the characters that you can't. The format is still basically verb, object, and prepositional phrase; this excludes a lot of probing questions.

Time is a major factor. Game time passes only when the player enters a move, but then any number of things may happen. In several of the adventures, the player's character must eat, drink, and sleep after a certain amount of game time has passed. Characters do things at scheduled times, and natural events may happen after a certain amount of time has passed. In the murder mysteries *Deadline* and *Witness*, the time of day is so critical that it is displayed on the status line.

Although *Zork* was written to challenge MIT students and staff members, not for the general computer-gaming community, Infocom's more recent adventures, starting with *Witness*, have been at an easier level than the earlier ones. This move was intended to appeal to a wider market. Let's hope that the

company will continue to put out really challenging adventures along with the easier ones.

Most of the negative points about Infocom's adventures come at a level that other companies haven't even reached yet. Occasionally a puzzle has an arbitrary solution or requires tedious trial and error. A more significant point is that the tongue-in-cheek attitude that runs through all the adventures has kept them from developing into stories of real worth. The plot of *Deadline*, for instance, is worthy of Agatha Christie in its basic ideas, but it has many touches that remind the player that it's "only a game." For instance, in one of the rooms is a novelization of *Deadline* itself. If you read the ending of the book, you learn that the detective kills himself in disgust—and then you kill yourself in disgust.

There's nothing wrong with humor, but the writers at Infocom seem afraid to be taken seriously. Perhaps they're too self-conscious about the limitations of the medium or too diffident about their literary abilities.

But it's hard to complain about a company that has accomplished so much more than anyone else. If a competitor arises with adventures of equal technical quality but different strengths, then serious criticisms will be more meaningful.

Hints are available from Infocom in the form of the "InvisiClues" booklets. These illustrated booklets give hints printed in invisible ink, which can be exposed with the developing marker that is included with the game. One booklet and a map are packaged together for each game. Besides providing an aid to perplexed players, they make a good souvenir of a completed game. Finally, a newsletter called *The New Zork Times* is available to anyone who wants to be put on Infocom's mailing list.

Now let's take a look at Infocom's adventures, one at a time. The reviews are presented here in the order in which the programs were released.

Zork I

Zork I: The Great Underground Empire, by Marc Blank and Dave Lebling, is essentially a cut-back version of the original *Zork*, with very little new material added. The reduction is only in the number of rooms and accompanying puzzles; the command handling, descriptions of the surviving rooms, and general capabilities are retained in full.

Infocom Adventures

Zork I borrows a lot from *Adventure*. There is a house on the surface near the Great Underground Empire; there is "a maze of twisty little passages, all alike"; and there is a Thief who is an enhanced version of *Adventure*'s pirate. The map and puzzles, however, are completely different from those of its predecessor. The Great Underground Empire gives the impression of a world created entirely out of magic, in contrast with the natural feel of *Adventure*'s Colossal Cave.

Nothing resembling a plot is to be found. Your goal is simply to go down into the dungeon, recover treasures, and put them in the trophy case in the house. Not all the treasures are in the dungeon; a few are to be found in the surrounding woods and along the Frigid River, which stretches from Flood Control Dam #3 to Aragain Falls. When you have stored all the treasures away, you are rewarded with an ending that serves as a lead-in to *Zork II*.

Light is necessary in the dungeon, since there are hungry creatures called grues roaming anywhere it is dark. Only a light will keep them at a distance. Grues have become Infocom's standard way of enforcing the need for light, lurking even in the dark passageways of science-fiction adventures.

Consistency obviously wasn't a goal. Some parts are technological, such as the dam and the machine in the coal mine, and even the brass lantern (like the one in *Adventure*) runs on batteries. Elsewhere there are elements of pure magic, such as the rainbow bridge. A Cyclops, who turns out to be the son of Polyphemus from the *Odyssey*, has somehow wandered in. The torch has an inextinguishable flame that seems magical, but it also has some realistic physical drawbacks. Some of the puzzles require strict logic; others need wild flashes of intuition.

There isn't necessarily a single solution to a problem. When you first meet the Cyclops, for instance, he may seem impossible to defeat, but there are actually two different ways to handle him.

Just getting your treasures back to the house is a problem in itself; someone always locks the door to the dungeon after you go in, so you have to find another way back out. There are several such exits, though some of them may not be wide enough to let you carry everything out.

An assortment of weapons is available to you. Some are better than others, and some may be positively suicidal in use.

If your weapons fail you and all seems lost, prayer may help.

This adventure is a good introduction to Infocom's line, since it awards points for performing certain tasks and for acquiring treasure. This gives the beginner a sense of making progress. Some other Infocom titles, such as *Deadline* and *Witness*, give no points for partial success; these are better suited for experienced players who can hope to finish the whole thing. Others, including *Enchanter* and *Planetfall*, give points for making progress but are really built around a single goal. *Zork I* is the only Infocom adventure that gives the less experienced player a chance to pick up a few of the treasures, leave with them, and get a sense of accomplishment.

Zork II

Zork II: The Wizard of Frobozz, by Marc Blank and Dave Lebling, continues where *Zork I* leaves off. Before you are more underground passages to explore. You discover such creatures as a dragon and a unicorn, which are no more than you expected in such a place. But soon you find that you have a more persistent foe—the Wizard of Frobozz—who continually throws an assortment of spells at you. His specialty is spells beginning with *F*, such as Fear, Falling, and Fierceness. Most of these spells are just nuisances, but the wrong one at the wrong time might make you launch a kamikaze attack on the dragon or fall fatally off a cliff. In the end you can defeat the Wizard, but the cost will be high.

As in *Zork I*, you accumulate treasures and score points for them. This time, however, there is no concept of going home with some of the treasures. You can leave the dungeon only when you have solved everything, and then you will exit to a passage that leads to *Zork III*.

One of the most difficult features here is a maze that seems entirely random. There is an odd pattern of clues to the maze, including diamond-shaped windows, a club autographed by Babe Flathead, and the Wizard's voice, mocking your inability to get to first base. But even with this information, it may be a long time before you discover the maze's secret, and it may be even longer before you pass the obstacle that lies beyond the maze.

There is a rather unlikely explosive device in this game; when you find that you might need an explosive, you may

have to start lighting various objects until you do yourself in. Then you can figure out how to use it correctly.

Zork III

Zork III: The Dungeon Master, by Marc Blank and Dave Lebling, takes on a different flavor from the first two. Your object is not to acquire treasures, but to gain entry to the sanctum of the Dungeon Master. What you'll need in order to accomplish that is learned only through careful observation. As in the first two installments, you face an adversary; however, the obvious response to his challenge isn't necessarily the right one. The encounter may leave you wondering whether George Lucas played this game before creating *Return of the Jedi.*

Instead of having a point system for accumulating treasures, *Zork III* gives you a "potential" rating from 0 to 7, based primarily on your reaching certain locations in the labyrinth. This score isn't very informative, since you may have reached all the right places but still not have performed the right actions or obtained the items you need.

A more useful indicator is the Dungeon Master's response each time you seek him out. He will express near-despair if you haven't found any of the objects you need, and he will get more encouraging as you accumulate more of them. But there is one deed that you must perform before you even get to see him.

In most adventures, the map remains constant except for passages that you open and close by your actions. Here, though, an earthquake intervenes part way through the adventure to destroy existing passages and open new ones. This makes being in the right place at the right time very important. Elsewhere in the adventure, being in the right time at the right place is what counts...but I'll leave that one for you to puzzle over.

In one room you will find a portal that lets you see and briefly enter scenes labeled I, II, III, and IV. Two of these take you into parts of the earlier *Zorks,* and the third takes you to another room in the current adventure. The one labeled IV takes you to a scene that has since become part of *Enchanter;* your stay there is brief but permanent.

Deadline

In *Deadline*, by Marc Blank, you are the detective and have 12 hours to solve a murder. The clock ticks, not in real time, but according to how much time it takes your character to perform each action you specify. The status line at the top of the screen tells you exactly what time it is, a fact which is important for anticipating scheduled events as well as for knowing how much time is left.

The adventure is peopled by characters who are constantly moving about the house, engaged in routine tasks (or perhaps trying to destroy evidence). You can explore the house, examine possible evidence, have Sergeant Duffy take items to the lab for analysis, and eventually arrest a suspect when you think you have enough evidence. To bring the adventure to a successful conclusion, you must not only arrest the correct suspect but have enough evidence to get a conviction.

The dead man, Marshall Robner, has apparently committed suicide in a locked room by taking an overdose of a prescription drug. But there is a suspicious circumstance. Just a few days before his death, Robner contacted his lawyer about changing his will. Could someone have poisoned Robner? And if so, how?

As with any good mystery, there are subplots and red herrings. As you make progress, you may be convinced several times that you have found the real murderer, only to discover that you have to recheck the facts and let the situation develop a little more. But if you wait too long, the murderer may strike again to get rid of someone who knows too much—and it may be you!

The variety of possible outcomes is very impressive. Aside from the one you would like—arresting and convicting the murderer—you can arrest a suspect and not get a conviction, or get killed yourself, or run out of time. If you don't get a conviction, you receive a letter explaining just why the jury was dissatisfied with your case.

The package comes with a sealed folder containing a letter describing your assignment, along with various memos and items of evidence. These show the same attention to detail as the game itself.

Infocom Adventures

Starcross

Starcross, by Dave Lebling, casts you as a space miner whose luck is running low until your mass detector reports a previously uncharted asteroid in the area. But when you go to investigate it, your delight changes to astonishment—the object is not an asteroid, but a gigantic alien spaceship! You have little choice at that point, as a tractor beam brings you into a docking area; you might as well see what's in the artifact.

The first trick is to get in. You find a bunch of oddly arranged bumps on the airlock, and pressing one is apt to make them all disappear. But look again; does that arrangement remind you of anything? This motif of having to recognize arrangements of dots or bumps occurs several times in the adventure.

The makers of the artifact stay out of sight, but you discover that an assortment of species from other worlds is on board. There is a tribe of weasel-like aliens, pathetically degenerated from their once high technological level. There is a lone intelligent spider, who is content to stand by and observe things. Worst of all, the makers have visited the world of Zork and brought grues aboard; these nasty creatures lurk in dark places, ready to devour you if you don't stay in the light.

Before long, you realize that the makers are subjecting you to an intelligence test and that your life is forfeit if you don't pass. Success, on the other hand, will make you a hero, since you will be able to bring a product of unbelievably advanced technology to Earth.

The key to success is finding the various colored rods around the artifact, which are used to activate machines and open doors. You have to make progress here quickly, since you find that the place is running out of breathable air.

Among the entertaining touches that are scattered throughout the artifact is a computer that stands in need of repair. If you forget to take certain basic precautions, you end up filling the air with the delightful smell of fried computer. Another clever idea is the use of the ray gun. You can use it to wreak all kinds of violence—but then will you have it when you really need it? And is killing the only thing a ray gun is good for?

The package, which resembles a Frisbee, is clever (but not

very practical for stacking on a shelf). Don't lose the map that comes with it; without it, you'll never get anywhere. The package also includes a sheet explaining how to give co-ordinates to your ship's computer.

Suspended

Suspended was created by Michael Berlyn. Like the author's earlier *Cyborg*, *Suspended* presents a situation that is especially appropriate to the adventure game format. You are the master of a complex that controls monitoring systems necessary to the world's survival, but you have no power to do anything yourself since you are in suspended animation. To control the complex, you issue orders to six robots, each with different capabilities. Iris can see, Whiz can get information from computers, Auda can listen, Poet can diagnose objects by touch and utter strange sayings, Sensa has an assortment of special sensory apparatus, and Waldo specializes in carrying and manipulating things. You talk to the robots through the Filtering Computers, which understand only a limited range of commands.

You control a robot by addressing it. For instance, you can say, WALDO, TAKE THE 16-INCH CABLE. For convenience in moving the six robots around, you can give a robot a destination (AUDA, GO TO LIBRARY CORE), and the robot will compute a path and tell you when it has arrived.

This adventure comes with a map of the entire complex, as well as pieces to keep track of the robots' locations. Even so, you will have to send your robots exploring to find out what the various chambers contain.

This game is best approached slowly, since it takes time to get used to looking at the world from six different perspectives and then adding the information up. Hard as it may be to ignore the fact that millions of people are dying out there, you should probably spend the first few sessions just discovering what the robots can and cannot do and what equipment they have to work with.

At the start of the game, weather control is going bad. In addition, seismic shocks are disrupting parts of your own complex. When you start assessing the damage, you soon discover another problem: Iris has gone blind. You can fix that, but soon afterward the *serious* trouble begins.

Unlike most adventures, this one continues to present a

challenge even after you've solved it. First, your score depends on how good a job you've done. After solving it the first time, you can keep refining your strategy to get the best score. And once you've perfected your performance, you can select the advanced scenario, in which problems hit you more quickly. You can even configure the game to challenge yourself or a friend.

My one complaint is that the messages you get for finishing the adventure with a mediocre score are unnecessarily insulting, considering that you have just saved the world.

Once you've completed the advanced and configured versions, you can type IMPOSSIBLE for the toughest challenge of all. Infocom promises that "anyone successful in completing the Impossible version of *Suspended* will win an all-expenses-paid trip to Contra, there to be immediately installed as Central Mentality for the next 10,000 years."

Believe me, they're on very safe ground.

Witness

Witness, by Stu Galley, is Infocom's second murder mystery. It is set in the 1930s. Freeman Linder has asked you to come to his house because of a threat on his life, and before you have been there very long, he is killed before your eyes! A few minutes later, Sergeant Duffy arrives, accompanying a man in handcuffs. He explains that he had seen the suspect running from the house right after the shot was fired. What's more, the suspect is the very man who Linder had said was threatening him. An open-and-shut case, right? Well, maybe not....

Witness is significantly easier than any previous Infocom adventure, perhaps too easy for some people. But it is rich in detail, and the way the crime was actually committed is an ingenious piece of plotting.

Your choice of whom to arrest is limited to three characters: Stiles, who apparently shot Linder; Phong, the Oriental butler; and Monica, Linder's daughter. This is a smaller number than in *Deadline*, but before you can make a successful arrest, you have to dig out the evidence. The smaller number of characters leaves room for a greater variety of actions; for instance, asking a suspect a particular question may provoke different responses depending on what you have already done.

As in *Deadline,* Sergeant Duffy is on call to analyze things for you at headquarters. Of course, since this mystery is set in the 1930s, don't count on the same sophisticated lab equipment that's available to Duffy in the other adventure.

In the early part of the adventure, what happens is mostly predetermined. Phong answers the door and brings you to the living room; Linder puts off your questions until he finishes his drink; you accompany him to his office; Monica goes out; Linder is shot; Stiles is arrested. You can take actions that change this sequence—for instance, just staying outside the house for an hour—but they either lead back to the main plot line or turn out to be dead ends. After Stiles is handcuffed to the couch, you have many more options.

In spite of its relative simplicity, *Witness* probably would not make a good introduction to Infocom games, since it doesn't offer any partial victories. However, it makes a good second step before going on to the more difficult ones.

Planetfall

Planetfall is by Steve Meretzky. You start as an Ensign Seventh Class mopping floors on a Stellar Patrol spaceship. But then the ship explodes, and your escape pod lands on an apparently deserted planet. Your first order of business is to find enough food to keep yourself alive; then you can start figuring out the purpose of the scientific complex on the island. Your time is limited by the food supply, disease, and the fact that the surrounding ocean is rising and threatening to engulf the island. *Planetfall* is set in the same universe as *Starcross,* and the events of *Starcross* are cited as the source of the knowledge that makes interstellar travel possible in *Planetfall*'s time. This planet also has the nasty, darkness-loving creatures called grues, which first came to earth (that is, to Zork) from the starship discovered in *Starcross.*

A lighter touch is evident here than in most of Infocom's adventures, particularly in the treatment of a robot named Floyd. This robot is at first an amusing companion, later possibly a little boring as his antics become repetitious, but ultimately he is indispensable to your purpose. He can come to a tragic end, though, and the hideously maudlin description of his death must be intended to punish the player for sending a brave robot to its doom.

There are two island complexes that you must explore; you can attempt to move between them by train, helicopter, or teleporter. Using any of these conveyances requires finding the appropriate magnetic access card. And do be careful about the handling of these cards! Operating the train is an interesting exercise; it can accelerate or decelerate by a certain amount each turn, and it may take a few tries before you can bring it into the station without crashing into the wall.

Not all of the problems in *Planetfall* are soluble; in fact, some of the dead ends are quite elaborate. This helps to make the world of the adventure seem more complete, since many things besides the elements of the actual problem are present. These dead ends can be frustrating to the player who assumes that every interesting object and place must be useful or attainable.

The packaging includes a Stellar Patrol ID card, three picture postcards from different planets, and a few pages of a diary. The ID card includes a magnetic strip and a signature strip, and they all look impressive. However, these materials don't provide the kind of vital information that the items packaged with most of the recent Infocom games do.

Enchanter

In *Enchanter*, by Dave Lebling and Marc Blank, the warlock Krill is at large—and the Circle of Enchanters must find a way to prevent his evil power from spreading over the land. But sending a powerful mage against him won't work, though, because Krill would be alerted by his coming. So the Enchanters choose you, an apprentice at magic, to penetrate Krill's castle and find some way to defeat him.

This adventure returns to the magical world of *Zork*, with several enhancements. For example, you now have the use of magic spells. You start out with a spell book that contains a few spells; as you proceed, you will find scrolls containing spells that can be added to your repertoire. As in Dungeons and Dragons, a spell fades from your mind after you cast it; you must relearn it from the spell book afterward. You also forget all spells when you go to sleep.

Another enhancement is that the game pays attention to the basic necessities of life. Before you enter the castle, you must provide yourself with food and water. Cycles of day and night occur, and you must sleep periodically. As well as giving

you rest, sleep brings you dreams, which may contain hints from the gods—or rather, the Implementers.

Your score advances as you overcome obstacles between yourself and Krill. If you defeat him, you earn a score of 400 and a seat at the Circle of Enchanters. On the other hand, there is an evil even worse than Krill that you may carelessly unleash. If you do this, you get a score of −10 and a rating of Menace to Society. Achieving your goal while avoiding this catastrophe requires solving a nice logical puzzle (which is not too difficult).

One of the most amusing touches in this game is the appearance of an adventurer bearing a brass lantern and an Elvish sword of great antiquity. Anyone who has played *Zork* will recognize this character, who engages in the kinds of strange actions that frustrated *Zork* adventurers often attempt. The adventurer does serve a purpose, as do several other creatures found in or around the castle.

Like most of Infocom's more recent adventures, *Enchanter* is considerably easier to solve than the *Zork*s. Still, it provides a satisfying assortment of difficulties and a sense of growing excitement as you move toward your final duel with Krill.

Finally, the package is one of the most beautiful ever created for a computer game. Anyone considering accepting a pirated disk instead of the legitimate packaged product should keep in mind what he or she will be missing.

Sorcerer

Sorcerer, by Steve Meretzky, is a sequel to *Enchanter* and starts off at a fever pitch. As the adventure begins, a hellhound is racing toward you, baring its teeth—and you have just one turn to do something about it!

Your character is the same one who, in *Enchanter*, defeated Krill and was rewarded with membership in the Circle of Enchanters. In this adventure, Belboz, the leader of the Circle, has disappeared; since all the other Enchanters are away, it's up to you to find and rescue him.

The first part of the action (not counting that little business with the hellhound) takes place in the Hall of the Circle of Enchanters. You must find out what has happened to Belboz, gather any items that might be useful, and discover how to get to his location. Don't be shy about taking things

that belong to your fellow Enchanters; under the circumstances, they'll understand.

You then transport yourself to an entirely different area to search for Belboz. There is no going back to the Hall once you've done this, so make sure you take everything you might need. You have a large area to explore, including wilderness, caverns, and an assortment of manufactured structures, before you can find him.

As in *Enchanter*, spells are the key to success. You have a larger initial repertoire this time, but you still need to find many spell scrolls in order to accomplish your quest. One of the spells allows you to provide for your own resurrection and lets you return to life any number of times. It is not useful toward the end of the game, however, since some places can be entered only once. A new source of magic—powerful potions—is available as well.

This adventure contains some excellent descriptive writing. There is excitement when you go on a wild roller-coaster ride, humor as you rout some monsters that have disguised themselves, bafflement as you come face-to-face with your own double, and sheer horror if you find Belboz but blunder in your attempt to save him. The handling of the time loop when you meet your double is especially striking.

Still, *Enchanter* is a difficult act to follow, and *Sorcerer* does not really break new ground. It does not provide as consistent a universe as its predecessor; the amusement park (which includes the roller coaster) is a gross incongruity in a medieval world of magic. Players who succeeded at *Enchanter* might well want a greater challenge in the sequel, but both adventures are relatively easy. Another problem is that *Sorcerer* sometimes requires "knowledge from a previous life." For example, one part of the adventure involves exploring a coal-mine maze on a limited air supply; performing the necessary actions in time is impossible unless the maze has already been mapped.

Included with the adventure is a copy of *Popular Enchanting*, which provides background information as well as explaining how to play, and an "infotater" wheel with illustrated descriptions of a dozen monster species. Another unusual packaging element is the musty smell that is apparent when the wrapper is opened.

Sorcerer offers no radical departures from Infocom's previous adventures, but it is an enjoyable adventure that makes good use of the company's game experience.

Infidel

Infidel, by Michael Berlyn, is the most literary of Infocom's adventures to date. The trouble is that the hero of *Infidel* imitates an anti-hero with whom few people would care to identify.

You are an archeologist who has antagonized your entire Egyptian work crew into quitting and running off (taking most of your equipment). This leaves you with barely enough supplies to set out on your own and find the pyramid that only you know the location of. Once you find the pyramid, you must commit further acts of vandalism in order to achieve the ultimate goal of finding and opening the sarcophagus.

As you, the player, push this character toward his fate, you may find yourself wondering whether it's really worth it. Even the messages you get in response to silly commands have a nastier edge to them than usual and rub in the character's worthlessness. Your doubts will only be doubled when you finish the game with a perfect score, for there is no real victory. The best possible outcome is to find the sarcophagus, open it, drool over its riches—and then die. You have no choice about the character's development; in success or failure, he's equally rotten. This aspect stands in contrast especially to *Zork III*, in which you can make choices about your character's development and thereby affect the outcome of his quest.

In an adventure that builds toward a single goal, the quality of the final puzzle is very important. *Infidel*'s finale only adds to the disappointment, since it requires performing actions in every possible order until you just chance on the right combination. Considering that this fiddling around occurs in the burial chamber, it can get quite depressing.

The command handling and text descriptions are up to Infocom's usual high standard. The puzzles are strictly logical, perhaps more so than in any other Infocom adventure. Other than that, the major plus in the game is the use of "hieroglyphics," made of combinations of ASCII characters, found in various places. A student of languages would turn pale at seeing them presented as ideographs with an English-like syntax,

but they provide a novel way of getting clues about the pyramid. The best approach to these hieroglyphs isn't to try to translate whole sentences, but to notice when the same symbol occurs in different places.

Infidel's apparent goal of moving in a realistic and literary direction is a commendable one. But there is more to reality than self-destructiveness, and more to literature than anti-heroes. Of all Infocom's adventures, this has the least to recommend it.

Scott Adams Adventures

Scott Adams Adventures

cott Adams wrote his adventures to run on a wide variety of computers, using just 16 kilobytes of memory and a tape recorder to input the program. The memory limitations make the vocabulary and map small in comparison to many disk-based adventures, but all the games are strong on imagination and filled with clever puzzles. Their main weakness is rather poor English usage and spelling.

While Infocom's adventures are, by most measures, greatly superior to Adams's, they are also much more expensive and time-consuming. The player who doesn't want to spend 40 to 60 dollars on an adventure, and weeks on solving it, can find a satisfying alternative in this series. The real adventure fan will, of course, want to explore both series.

This series approaches things differently from *Adventure*. Instead of treating you, the player, as the character in the story, the Adams programs let you give commands to a more or less obedient "puppet." Instead of saying, "You are at the top of a tree" (or wherever your character might be), the program says, "I am at the top of a tree." Both viewpoints (first and second person) have proven popular in adventures written since then.

Beyond a few lines of promotional material, no background material is provided in any of the adventures. Your character just appears in a situation and you may never find out why. In several of the adventures, this means that you will have to spend some time just figuring out what's going on.

Adams makes good use of repeated message fragments to pack more messages in. A given phrase, such as "won't let me," may appear in several different messages while taking up memory space only once.

Adventure International is in the process of releasing all these adventures in illustrated versions, called the S.A.G.A. (Scott Adams Graphic Adventure) series. No significant changes have been made in the text or problems; the only difference is in the display format, which now includes one

illustration per location. Some of the solutions are more obvious than in the standard edition, since the pictures make the situation clearer. The graphic quality is good, but putting up and taking down pictures makes the program run more slowly. The INVENTORY command produces a sequence of objects being emptied from a bag; this is cute but tedious.

For frustrated explorers, Adventure International provides a "Book of Hints" that covers all 12 Scott Adams adventures.

Adventureland

Adventureland, by Scott Adams, is the first of the Scott Adams series and the closest in spirit to *Adventure*.

You start off wandering around in an area filled with such hazards as quicksand, nasty chiggers, and a sleeping dragon with a keen sense of smell. Fortunately, there is also an item that can be put to a couple of good uses, one of which lets you get into an underground complex. From there, you have to face an assortment of hostile creatures, including a bear and a swarm of killer bees, in order to pick up the treasures they guard. There is also a helpful being around—but he can be pushed too far.

Even if you get killed, you're offered another chance. You find yourself in Limbo; if you can find the exit, you live again. If you go the wrong way, though, the comment that accompanies the next description puts your situation quite succinctly.

Hints are liberally sprinkled throughout the game. The descriptions are concise and often humorous, if not elegant. On the whole, this is one of the easiest adventures in the series.

Pirate's Adventure

In *Pirate's Adventure*, by Scott Adams, finding a copy of *Treasure Island* literally sends you back to the days of Long John Silver, where you have the opportunity to go sailing and find some treasure for yourself. As with several of the series games, various critters are around to help or hinder you. The parrot, for example, can give useful advice and help you in other ways (a gimmick from *Adventure* is reused here), but only if you can keep it from flying away. The mongoose may be able to help you—or then again, it might not. Crocodiles and snakes can make life difficult or short.

Since the island where you arrive isn't the one with the

matter of bringing all the necessary materials together; once you've got the materials and the plans, the ship goes together with amazing speed. Gathering these materials makes up a large part of the adventure..

A lot of the fun involves the pirate, a tough-looking character who is helpful enough if you play on his weaknesses. He turns up in some outrageously unexpected places, and he's ultimately indispensable to your getting to Treasure Island. After all, you can't sail a ship without a crew.

There are only two treasures in this adventure, so it's very nearly an all-or-nothing proposition. However, finding them is easy enough to provide a good introduction for new adventure players. This is one of the easiest of the Scott Adams series, but the puzzles are still clever.

A full listing of the BASIC version is given in the December 1980 issue of *Byte*.

Adventure #3

In *Adventure #3* (formerly *Mission Impossible*), by Scott Adams, your mission—should you decide to accept it—is to stop a saboteur from blowing up a nuclear reactor.

The major difficulty is in getting through the reactor's security system. A number of doors are locked and will open only if you show appropriate authorization to the automated camera above the door. You were originally provided with that authorization, but the saboteur has run off with some of your materials. That leaves you on your own to get the security badges you need. This system of using badges to get through various areas anticipates *Planetfall*'s use of access cards. In this case, though, the two-word format shows its limits. You may eventually accumulate several different badges (called pictures), in which case it becomes impossible to tell the program which picture you want to take or drop. Fortunately, the command SHOW PICTURE always selects the right one.

While you have to be clever to work your way through this security system, there are also times when brute force is needed. Think of the objects that are available and consider which one might do the job.

The danger of the bomb blowing up adds to the excitement. Several actions that you have to perform will arm the

you); then you have to drop whatever you were doing and run to disarm it. If you haven't learned how to disarm it—well, your family will get a nice pension.

When you finally do find the bomb, timing becomes even more critical. You must take exactly the right actions and perform them in the right place, or you will set off a disaster of your own making, even as you prevent the one the saboteur intended.

This adventure shows how different people may perceive the difficulty of an adventure and shows why I haven't tried to assign difficulty ratings to the adventures in this book. The Adventure International catalogue rates this one as advanced in difficulty, going so far as to say, "If you survive this challenging mission, consider yourself a true Adventurer!" However, I had less difficulty with this one than with any other adventure in the series. Personal difficulty is very much a matter of how well your mind meshes with the situation.

Voodoo Castle

"Count Cristo's been CURSED! There's one way for him to flee! Find it, and he'll go FREE!" Not very good poetry, but there are some fine puzzles to solve in *Voodoo Castle*, by Alexis Adams. What is the strange meaning from the chimney? How do you get through a passage that's too small for you? How do you survive a room in which chemical tubes are constantly exploding? What significance does the Ju-Ju Man statue have?

Eventually the pieces start coming together, letting you discover the magic ritual for breaking the curse. Different kinds of magic are thrown together freely; the spell uses a couple of European good-luck charms along with voodoo's traditional doll stuck with pins. At the climax, there are lots of dramatic effects and thunder. The only catch is that along the way, you've shrunk yourself to four feet in height, and you seem doomed to live the rest of your life that size. Oh, well, at least you'll be able to get through low doorways in your next adventure.

This adventure isn't very careful about making you specify which tool to use for a job. As a result, it's possible to go around with a double handful of items and get things open without really solving the problem, or even realizing that there ever was one.

In a couple of places, items are listed in capitals to call attention to them. This is rather uncalled for; hints should be something you look for, not something that the adventure's reality throws in your face. It's not that hard to realize that two sapphires have something to do with each other without being told that the door has "a SAPPHIRE in it."

Voodoo Castle is fun and fairly easy. It would be a good first step up from such simple games as *Pirate's Adventure.*

The Count

In *The Count,* by Scott Adams, your enemy is Count Dracula himself. Your object is to destroy him from within his house. If you don't succeed in time, you will turn into a vampire, too. But if you overcome the obstacles, you have the satisfaction of typing KILL DRACULA as the last command of the game.

The Count predates Infocom's *Planetfall* and *Enchanter* in providing a cycle of day and night. The adventure runs over a period of days and you have to sleep. This causes problems, since Dracula takes that opportunity to do all sorts of unpleasant things. When you sleep, you have strange dreams, which appear in the form of a screenful of messages flashed for a fraction of a second; if you're quick, you may get a clue from them. There are some things that you have to do at night. In order to get them done, you need something to overcome your tendency to doze off right after sunset.

The way you find the coffin is a bit unusual, to say the least. Let's just say that where there are coffin nails, a coffin can't be far away.

There doesn't seem to be a guaranteed route to success, since on several occasions you have to engage in an activity with a certain probability of a fatal outcome. This makes it a good idea to save the game at several points as you progress.

The lack of documentation is irritating in this game, as in some of the other Scott Adams adventures. When you first start playing, you have no idea where you are or what you are supposed to do; only by exploring the house (and perhaps getting killed by angry villagers if you try to leave prematurely) do you learn what your quest really is. It doesn't take very long to figure out what's going on, but a little information on paper would have made for a more well-rounded adventure.

Scott Adams Adventures

Strange Odyssey

Strange Odyssey, by Scott Adams, puts you aboard a spaceship that has apparently suffered from a bad landing, since you can't blast off again. All that you can do is explore the planetoid in the hope of finding something that may help you. Fortunately, this is a rather unusual planetoid. A little exploring leads you to a hexagonal room, apparently constructed by aliens, with a portal to several different worlds. With all these options, you should be able not only to repair your ship but to return home richer than you were before.

There is an obvious *Star Trek* influence in this adventure. Your weapon is a *phaser*, with *stun* and *destroy* settings. One of the treasures is a bottle of Saurian brandy. The reason you can't take off is that the *crystals* in your ship are damaged.

The events also show an interest in the physical requirements of space travel. The space suit has a limited air supply and should be used carefully. Where there's breathable air, you don't need to wear it, but many areas have either no atmosphere or one without oxygen. Gravity on the planetoid is light, a fact that makes it possible for you to get started; but in another place, the gravity is so strong that it will kill you unless you have special equipment.

A number of details are included that help you develop an idea of what the aliens who created the gateway are like. The ease with which the controls move suggests that they are a physically weak race compared with humans; the nature of the treasures suggests that they have strong artistic tendencies, but that their eyes see in a different spectrum from ours. Their technological advancement goes without saying.

This attention to detail makes *Strange Odyssey* one of Scott Adams's best. It's a pity he hasn't done more science-fiction adventures.

Mystery Fun House

In *Mystery Fun House*, by Scott Adams, you explore a very strange fun house in which a secret is hidden. This one is short on motivation but long on fun, even though it's quite difficult to solve. First, you have to get into the place, and the program has some fun at your expense before you find the right kind of admission payment. Next you have to figure out why you went there; this requires you to pay some attention

to oddities about yourself. As usual, there is no documentation; you just appear without any idea of who you are or what you're supposed to do.

The areas that are open to patrons don't pose too many hazards. The fun-house format gives Adams an opportunity to bring in a lot of intriguing situations. There are machines to operate, knobs to fiddle with, a merry-go-round to ride, clay pigeons to shoot, and a small maze to wander through. One area provides an entertaining take-off on the Loud Room in *Zork I;* a calliope is playing so loudly that your puppet has trouble hearing your commands! Fortunately, this puzzle has a more logical solution than its counterpart in *Zork.*

As long as you carry your ticket and keep your shoes on, you can't get into too much trouble in these areas. But if you try anything violent, a bouncer will promptly escort you to the exit. The house rules are something of a nuisance, since the shoes and ticket cut deeply into your carrying capacity. Add a watch and a stick of chewing gum, and you're bent over with the load—although you can carry a trampoline as easily as the gum. This adventure illustrates the shortcomings of treating all items as being equally burdensome.

Luckily, the bouncer doesn't enforce the "Authorized Personnel Only" signs, which you have to ignore in order to find the secret. Once you venture beyond these signs, you can find yourself in real danger; but being a well-trained adventurer, you can often get out of a tight spot with your athletic abilities.

One of the toughest adventures in the series, this will provide experienced players with a good workout.

Pyramid of Doom

Pyramid of Doom, by Scott Adams, puts you in an Egyptian pyramid where you hope to find the treasure of the Pharaohs—but you have your work cut out just to get in alive in the first place. In the course of searching the nearby desert and oasis for something that will get you through the entrance, a small desert nomad may appear. Be careful of him; he might be waiting for the first opportunity to do you in. Once you enter, you can start your treasure hunt. But how do you get past the hostile mummy? What can you do about the mundane but equally dangerous rats? Can you get the oyster

to provide you with a pearl? Oyster? What's an oyster doing in a pyramid?

One of the puzzles is figuring out what to do to get credit for the treasures. This requires putting some information together, and you can't get any points at all until you've figured it out. It's vital to SEARCH everything in order to get the necessary hints. Better yet, SEARCH everything twice. Being so thorough will, perversely, get you killed in a couple of cases, but there's nothing to do about that except save the game frequently and go back to where you left off. This adventure, like many others, has the flaw of requiring knowledge from a "previous life" in order to anticipate traps.

A flashlight is necessary for exploring the pyramid's passages, but one room is so filled with mirrors that your flashlight becomes worse than useless. Here, you must explore in the dark—carefully! Be sure to turn your light back on at the first opportunity.

Once you get past these early problems, you find your real foe: an iron statue of Pharaoh that comes to life when you approach it and ends yours unless you do something fast. Only by learning how Pharaoh's heart has grown dark with evil can you find the way to defeat this juggernaut. Even then your job is not done; poisoned traps, disease, or the dreaded purple worm may kill you before you can reach the top of the pyramid and gather all the treasures.

Words are abbreviated to three letters, which causes confusion in at least one case. You can FEED creatures and FEEL some objects, but both commands are stored as FEE. FEEing may transfer food to the object or let you know what it feels like. It may also leave you wondering why the nomad thinks you're weird when you offer him food.

This is not one of the series' easiest adventures, but the puzzles are logical enough so that it shouldn't be impossible.

Ghost Town

Ghost Town, by Scott Adams, puts you in an old western ghost town that might still contain valuables. This particular town also has a lively population of late lamented residents, who are about as frightening as Disneyland's "Grim Grinning Ghosts." They can often help you. The puzzles are up to Adams's usual standard of ingeniousness, and there is even more than the usual dose of humor.

This isn't to say that the area is entirely safe. A rattlesnake endangers you at one point, and you are likely to blow yourself up trying to solve one of the puzzles. Besides, the game goes through a cycle of day and night, and stumbling around in the dark is as hazardous here as in any adventure. In fact, the nighttime is when the ghosts are likely to be most active, so think twice before dropping off to sleep!

Your spirits might start to decline when a ghostly voice keeps whispering "Vain..." in your ear. But that voice isn't necessarily talking about your efforts. There is a way to get the town's only other living being to do something for you. And there is a complex solution to the problem of how to get the safe open. *Star Trek* fans may have a sense of déjà vu as they discover this solution.

It's especially important to make sure you go to every place that can be reached. This requires getting over natural obstacles, taking a ride, and even changing some of the natural features. In some cases you need the right equipment, but elsewhere simple know-how is all that is necessary.

Pyramid of Doom, the preceding adventure in the series, featured a purple worm that had the nasty habit of eating adventurers. In this adventure, you can get a satisfying revenge on the worm.

An interesting feature is that the program keeps two scores: one is the usual score for accumulating treasures; the other is a measure of how fast you complete the adventure. This means that even after you've solved it, you can keep working on getting the highest possible speed score. Getting that score to be a positive number is a difficult trick.

Savage Island

Savage Island (Part 1, #10, and Part 2, #11), by Scott Adams, is sold in two separate parts, but you must solve Part 1 completely in order to learn the password that will let you play Part 2. When you finish Part 2, you are given a code. By using it, you can decipher a card that provides the conclusion to the adventure. Without a doubt, this is the toughest of the series.

At the beginning, this seems to be a straightforward wilderness survival adventure—though surviving is anything but straightforward. You need shelter from a hurricane that soon strikes, but the only available cave is occupied by a bear

there makes you sweat, and that increases the bear's interest in you. Even if you manage to wash the perspiration off, being around a bear makes you nervous, and you start sweating again. But it's possible to get the bear interested in something besides you.

You have to cross a lake to get to some places, but you aren't an especially good swimmer. If you're carrying anything at all, you find yourself in imminent danger of drowning. In fact, you are so inept in the water that the program assumes that you don't know enough to hold your breath before diving beneath the surface; you die unless you first type HOLD BREATH. Fortunately, help for YMCA dropouts is available.

If you survive the hurricane and don't drown, you can start gathering the materials you need to explore further. Only after you do this do you find out just how strange this island is. Why are there antennae coming out of the pirate's head? And why does the cave drawing show a spacecraft landing among dinosaurs? Never mind the spacecraft, when did any cave dweller ever see a live dinosaur?

At this point, you have to so something with an artifact you find. The solution to this one borders on the ridiculous; so it's only fair to advise trying every possible action on the artifact with every possible object. After this, things get a little easier, provided your transportation doesn't fail you. It has become obvious by now, though, that this is not your everyday jungle island.

When you get to Part 2, a sign advises you: "Compared to what you're about to go through you'll think Part I was a piece of cake!" In fact, you're missing a few things by now: your clothes, some of your possessions, and air to breathe. There is more than one way into Part 2, and not all of them give you any chance of success. So don't sell your copy of Part 1 yet; you may want to replay it to a more useful conclusion.

Much of the difficulty comes from the fact that even if you do everything right, at several points you could still get killed by bad luck. This means that you will have to save and restore the game frequently, which is rather tedious if you have the cassette version. To make matters worse, the procedure for saving the game is disabled during one crucial stretch. You're on your own to survive there!

This is one adventure in which a better command parser would have helped greatly. In several cases, you have to do

something with a specific object. If you give a command that requires such an object, the program asks "How?" You must then give an appropriate answer, such as WITH SHOVEL. This method isn't really enough to handle the complex actions required, so in some cases you wind up "playing the program" rather that the situation. Still, Adams deserves credit for having packed an amazingly complex adventure into 16 kilobytes.

Any adventurer who has zipped through everything else and wants a real challenge should give *Savage Island* a try. It sets up its difficulties without being wildly illogical, even if some of the puzzles are a bit arbitrary. No one can claim to be a master adventurer without having solved this one; I hope to solve it myself someday.

Golden Voyage

Golden Voyage, by William Demas and Scott Adams, could very well be Sinbad's eighth voyage, as a dying king commands you to find the remedy that will save his life. He provides you with plenty of gold, so you have no problem buying a ship and supplying it. You must search the neighboring islands for a cure, and there are problems enough to overcome. An animated statue with a sharp sword hinders your access to a temple; a mysterious fountain provides a liquid which may or may not be useful; broken stones contain fragmentary markings to be deciphered.

The ship is fully automatic and you have no trouble sailing where you want to go, even without a crew. However, you don't have a map of the islands, so you have to be sure you've found them all by thorough exploration. The ship includes a crow's nest to facilitate spotting land masses, as well as a cot, which makes a good refuge when it gets dark. But time does pass, and you have only three days to save the king's life.

A staircase on one of the islands presents a problem, since the UP and DOWN commands don't work on it for some reason. The commands to use here are WALK UP and WALK DOWN. It's nice to know this in advance, since you're likely to get attacked the first time you step onto those stairs.

There isn't quite as much flavor here as in many of Adams's adventures. Only a few situations give the sense of

Scott Adams Adventures

Middle Eastern adventure that you'd expect. The dark passages, animated statues, scorpions, buried implements, and secret entrances give a sense of reused formulas. Perhaps Adams realized he was running out of ideas when he closed off his adventure series with number 12.

Sierra On-Line Adventures

Sierra On-Line Adventures By Scorpia

he world of adventures is a large one, spread out over many different computers. To make this book even more valuable, I asked Scorpia, the sysop of CompuServe's Game Special Interest Group, to write some reviews of Apple adventures for inclusion here. The result is this chapter. Scorpia's material not only covers an important adventure publisher that otherwise would have been neglected, but it gives you a chance to read a viewpoint on graphic adventures that is more sympathetic than my own! —GM

In the beginning it was nothing but words. Text flowed across the screen, evoking mental images of people, places, and events. For a nongraphic computer, like the TRS-80, that was fine. But for a computer with potential for high-resolution color displays, like the Apple, that wasn't quite enough. While arcade games had begun to take advantage of Apple's graphic abilities, the art of hi-res adventures languished. And then along came Online Systems.

Their first entry into the graphic-adventure field, *Mystery House,* was hardly calculated to raise eyebrows. It was simply done, with line drawings on a black background and no color, and it wasn't what we now think of as hi-res. That really came in with the second adventure, *The Wizard and the Princess,* and graphic-adventure games haven't been the same since.

The Wizard and the Princess set the pattern for most of the hi-res games that were to follow: full-screen color graphics, with four lines of text and player input at the bottom of the screen, along with a means of flipping back and forth between hi-res screen and all-text screen. The game itself had very nice, smoothly drawn graphics, with a clean color-fill. Even today, with some very sophisticated hi-res displays around, *Wizard's* graphics stand up well. That is even more remarkable when you consider that the game was first published back in 1981 (ancient times, in computer terms).

Sierra On-Line Adventures

Aside from graphics, Online (later renamed Sierra On-Line) also established a reputation for writing tough adventures. *Wizard* is not a game for novice players, nor are most of the others in Sierra On-Line's stable. For these adventures are more than pretty pictures. They present you with some tough puzzles as well, and the combination quickly brought On-Line to the forefront in the world of computer adventures.

One of the more interesting features of Sierra On-Line adventures is their linearity. In most (although not all) of the games, you move only forward, never back. The games tend to have small clusters of locations which, once visited, need never be visited again. The desert in *Wizard* is a case in point. Once you have been through all the desert areas, and there aren't too many, you don't ever have to go back there.

Another oddity of On-Line's adventures is the "limitless inventory." Where many games limit the number of objects you can hold at one time, On-Line's don't care in the least how many you have with you. You can literally pick up every item you come across, hauling them all around with you for however long you please. This may sound like a big advantage, but first you have to *get* the item—and that's not always easy! These games are good at concealing objects without seeming to do so, which is why the reviews that follow all tell you to look at the graphics carefully. Those pictures aren't there just for show.

Mystery House

Mystery House, by Ken and Roberta Williams, was On-Line's first adventure game. It puts you in a spooky-looking Victorian mansion, set in the middle of a dark forest. Somewhere in that house is hidden a fortune in jewels—and someone wants those jewels badly enough to kill for them.

You'll need to find a light source pretty quickly, since night is rapidly approaching. Then, as you explore inside, outside, upstairs and down, you will find clues to the identity of the killer, as well as to that person's secret hiding place. And you'd better find a weapon, too; when you finally track down the killer, only one of you will walk away from the encounter.

That's pretty tough stuff, but the adventure itself is somewhat easier than later Sierra On-Line games. There are a couple of hard puzzles and one that might be considered a little unfair, since it requires more of an intuitive leap than careful

reasoning. Also, players familiar with the dynamic *Deadline* from Infocom may be disappointed by the static nature of this game. No matter what you do, the same people will always turn up dead in the same places. There is no way to save them.

The graphics may come as a surprise to some people, too. Unlike later games, *Mystery House* is not hi-res. Instead, the pictures are all done as line drawings on a black background, with no color-fill. This occasionally produces some odd results (for example, allowing you to see through many of the solid objects in the game). However, the drawings themselves are quite good.

The save game feature is somewhat limited. Saving is done to the game disk itself, and you can save only one position at a time. Each new save wipes out the previous one. This can sometimes make you think twice when saving the game, since you have only the one save.

By current standards, *Mystery House* is a little crude. But overall it's a pretty good intermediate-level adventure.

The Wizard and the Princess

There you stand in the desert town of Serenia, with only a few meager possessions in hand. Looking out over the endless desert sands, you wonder if you can ever accomplish your mission: rescuing the princess from the clutches of the evil wizard. With a heavy sigh, you trudge northwards, only to come face-to-face with a nasty snake.

Thus begins one of Sierra On-Line's most famous games, *The Wizard and the Princess,* by Ken and Roberta Williams. One of the earliest of the hi-res graphic adventures, it has stood the test of time and is as playable today as it was when it first came out.

Your journey to the wizard's castle will be long and hard. Trackless desert, bottomless chasms, endless oceans, and towering mountains are only some of the obstacles you must overcome. Maddening gnomes, ferocious lions, and grumpy giants will all do their best to stop you cold. And, when you finally get through all this, there is still the magic castle, full of tricks and traps for the unwary.

Each part of the game builds on previous ones, so once you move ahead, there is no going back. It's important to search every area thoroughly to make sure you haven't overlooked anything, which is very easy to do in this adventure.

The location or importance of some items may not be immediately obvious; careful attention to the graphics is essential to complete the game successfully.

The graphics themselves are disk-driven. Each time you move to a new area, the program must read in the picture from the disk. The fill routines are a bit slow in comparison to those in more recent hi-res games, but they are pretty to watch. The graphics, while not spectacular, are quite good.

There is a save game feature, which allows you to make a "snapshot" of up to 15 different positions on a separate save disk. Frequent saving is highly recommended, since you never know (until it's too late) when you've done something that will prevent you from finishing the game.

This is a good, tough adventure for experienced players.

Cranston Manor

Cranston Manor, by Harold DeWitz and Ken Williams, is an old-fashioned treasure hunt through a haunted house. The object is to explore the estate of Old Man Cranston and to recover 20 treasures, which will help revitalize the town of Coarsegold, California. If you know anything about Sierra On-Line games, you know this isn't going to be as easy as it sounds.

Old Cranston was something of an evil-minded, old miser. The treasures that he left behind are guarded by an animated suit of armor in the house and by deadly tin soldiers who haunt the caverns beneath the mansion. The armor is pesky but not fierce; it simply follows you around the ground floor, preventing you from taking any goodies you may find. The soldiers, however, will shoot at you the first chance they get—and if they hit you, it's time to restore the game.

Although the manor itself has a fair number of rooms, its layout is straightforward and mapping is simple. The underground areas, however, are another matter. The many corridors and openings take some strange twists and turns, so you may not always end up where you want to. That can easily make a shambles of any map you try to draw. While there aren't any mazes per se, just trying to get the underground area figured out is frustrating enough.

You must learn your way around down there, for it is only in the depths of the caverns that you will uncover the secret of the armor and the soldiers. But even after you banish

those troublesome contraptions, your problems aren't over. Some of the treasures (such as the bottle in the cistern) are not easily accessible; others, like the emeralds, are not immediately evident. As with all Sierra On-Line games, it is vital to pay close attention to the graphics display.

Cranston Manor features typically good, if slow, graphics. The pictures are well drawn, nicely detailed, and the colors don't appear to run into each other. The usual save game feature is also available, allowing you to save up to 15 different positions to a separate disk.

This is a good, hard adventure game for experienced players.

Ulysses and the Golden Fleece

Someday, Sierra On-Line will explain why they named this game for Ulysses, since most people know that it was Jason who went after the Golden Fleece. Even so, anyone who is at all familiar with *The Odyssey,* or who knows something about the legends surrounding Ulysses, will be one step ahead of everyone else in this adventure.

The game starts with your character (Ulysses) standing penniless and empty-handed in a Greek town. From that humble beginning, you must buy supplies, obtain a ship, hire a crew, and set sail on your quest for the fabled Fleece.

Once underway, you will find several nasty puzzles and traps in your path. The first is a hurricane; getting past it is one of the more difficult problems in the game. The solution is particularly maddening, too, as it isn't something you would ordinarily think of—and there are absolutely no clues to help you figure it out. You must just make an intuitive leap to discover the answer.

Having done so, you proceed onwards to other obstacles. These include Neptune, God of the Sea; Pluto, God of the Underworld; a wall of flame; several chasms; the Sirens; the Cyclops; Harpies; and animated skeletons. There's even a dragon thrown in for good measure.

Your mind (and recollection of Greek myths) will get quite a workout. Some of the solutions are quite subtle. It is very easy to pass by something important and never even know that it's there. And passing something by could spell disaster, since you can't backtrack in this game.

Fortunately, you can save the game. Up to 15 different

positions can be saved to a separate diskette. Considering the difficulty of the game and the ease with which you can paint yourself into a corner, frequent saving is highly recommended.

The graphics, as you might expect, are of high quality. While simple in design, they are well drawn and use clean, bright colors.

Overall, the game is excellent. But there is one thing to be aware of. In the very beginning, when you go to the store, you will find several items for sale. You can purchase all but one of them. The only way to figure out which item *not* to buy is through trial and error. This can become extremely frustrating if you get very far into the game and then find that you can't go on because you don't have what you need. If that happens, there is nothing to do but start all over again from the beginning of the game. But such challenges help to make this tough adventure one of Sierra On-Line's best.

Time Zone

Time Zone, by Roberta Williams, may well be the most ambitious hi-res adventure game ever designed. It spans the course of history from the dawn of time to the far future, taking you across the continents of Earth and out into the depths of space. As a result, the program fills six double-sided diskettes and is advertised as requiring "a year to complete." That may or may not be true, but in any case, finishing this game is definitely an achievement.

The story line is simple: you have been chosen to save the Earth from destruction at the hands of an alien tyrant on Neburon, a planet located many light years from Earth. You are given a space/time machine that can transport you to different continents and time periods on Earth, as well as into space to the planet Neburon. The time machine is needed because you have to pick up a few useful objects first, and these objects are in different time periods on Earth. Thus, most of the game is spent traveling back and forth among time zones, solving puzzles to obtain the items you will need at the finish.

Make no mistake about it: the puzzles in this game are *tough*. Part of the difficulty comes from having all those time zones—figuring out which items to get from which time periods is not easy! The difficulty is compounded by the fact that if you try taking something backwards in time before it

could exist, that item will disappear forever. Fortunately, there is no problem with taking anything *forward* in time.

When you finally arrive on Neburon, things are no easier. You will have to make your way around a hostile planet, overcoming a number of nasty tricks and traps as you go. There is also a maze that must be negotiated before you can locate and destroy the tyrant's weapon. All in all, it's a rough trip.

Since it's likely to take some time to finish the game, Sierra On-Line recommends that you back up the diskettes. They provide a utility that will copy all but disk 1, side 1 (the main boot program). They also provide another utility that will initialize a blank diskette for saving the game. Each disk can hold up to 15 separate positions, and you can have as many different save disks as you like.

There are also some drawbacks to *Time Zone*. For one thing, it is a *very* disk-intensive game. Every graphic display is stored on disk, so there has to be an access every time you move to a new location...and there are, as you can guess, quite a few locations on 12 disk sides! Further, a lot of disk swapping is necessary as you move between the zones. Since the game supports only one drive, that can become tedious during long sessions of play.

In addition, while the graphics are quite good (and the fill routines are faster than in previous On-Line games), there is still waiting time while the new picture is loaded and drawn. All this makes the game a bit more time-consuming than some people might wish.

However, the most bothersome feature of this otherwise superbly designed game is its over-use of "empty" zones and red herrings. More than half the time periods are dead ends, whose sole purpose is to slow you down, frustrate you, and misdirect your efforts. Having red herrings is a time-honored convention in adventure games, but they should be carefully placed and used sparingly. Here, they have been used almost with abandon, adding extra frustration and confusion to an already-tough adventure. That's a shame, because the basic game is very well thought out and executed.

This is a good but hard adventure game, marred by a lot of filler. But if you want to spend the money and time, the puzzles will provide a real challenge for the experienced adventurer.

Sierra On-Line Adventures

Dark Crystal

This is an adaptation of the movie of the same title. However, while having seen the movie might help a little, it really isn't necessary to finishing the adventure.

The object is to guide Jen, the Gelfling, towards his goal of healing the magical crystal and thus ending the reign of the evil Skeksis. Jen must first make his way to Aughra (from whom he will obtain the missing piece of crystal) and then to the palace of the Skeksis (where the piece will be returned to its rightful place in the crytal).

Naturally, this will not be as easy as it sounds. Jen must make his way through forests, swamps, and deserts, avoiding traps, solving puzzles, and keeping out of the way of the dreaded Garthim. He must uncover the answer to Aughra's riddle, make friends with Kira, and learn to ride the Land-striders. In the palace of the Skeksis, he must tread warily and keep his ears open as he seeks a route to the crystal.

In spite of all that, however, the adventure is not too difficult, and is in the novice category. It's a good game for those who don't have much experience with adventuring, although some familiarity with adventures is helpful when trying to solve a couple of tough puzzles.

The graphics in the game are of uneven quality. Some seem to be lacking in detail, almost as if they are unfinished, while others are well drawn. There is a feeling that there might have been a rush to get the game out, and some of the graphics suffer because of it. This detracts a little from the enjoyment of the adventure, but it is by no means a major flaw.

The game offers the usual capability to save to a separate disk, with the standard (for Sierra On-Line) of 15 different positions per disk. Also typical with Sierra On-Line, the game supports only one drive, so disk swapping is necessary before and after saving a position.

Bottom line: This is a fairly good adventure in the novice-intermediate area.

More Adventures

More Adventures

Adventure

Adventure *is published by the Digital Equipment Corporation Users Society (DECUS), Microsoft, and other suppliers.*

This is the original, from which the genre got its name. You must find a way into a vast complex of caverns, pick up treasures, and get them back to the house at the surface to get full points.

You face an assortment of enemies along the way. Shortly after entering, you will come into the Hall of the Mountain King, which is guarded by a huge, green snake. Unless you discover the way to get past it, you will get to see very little of the cave. Help is available, though it comes in an unlikely form.

Having vanquished the snake, you will soon find several treasures. At one point a broad chasm blocks your path, but there is a way across it (if you're imaginative in using what you've found so far). Soon after this, a more persistent danger will appear: a number of threatening dwarves who follow you around throwing knives. And if they aren't enough trouble, a bearded pirate will leap out of the shadows at inconvenient moments, grabbing all your valuables and escaping before you can make a move. Don't worry too much, though; there are ways to defeat the dwarves and recover your treasure from the pirate.

Eventually you may get to the Troll Bridge (complete with a sign saying "Stop! Pay Troll!"), and then you will have to give a treasure to the troll before he'll let you cross the bridge. He can be outfoxed. But what will you do when you get to the barren room and find the bear? And how can you get an Oriental rug out from under a dragon? Well, if it were easy, it wouldn't be fun.

Various versions of *Adventure* are in circulation. The original version takes only two-word commands but has a fairly good vocabulary. Later versions include multiple direct objects and prepositional phrases. There is also a cut-back version available from Radio Shack on TRS-80 computers, misleadingly published under the title *Pyramid 2000.* In all versions, the descriptions are well written and picturesque. The sense they give of exploring a spectacular environment remains

Some of the puzzles are quite clever, while others don't really play fair. One of the best is the aforementioned encounter with the troll. It seems at first that he will charge you one treasure, if not two, but making the right moves lets you get everything back in the end. Less satisfying, on the other hand, is the encounter with a dragon. Success here depends on answering a seemingly rhetorical question correctly. The unfair aspect is that the program asks the same question at other points in the game but doesn't recognize an answer from the player.

Adventure has proven its durability and ranks high among its descendants.

Asylum

Asylum, *by David C. Willen, is based on the original* Asylum *adventure by Frank Corr, Jr. and William F. Denman, Jr. (Med Systems/Screenplay).*

You have been put away in the asylum. Can you get out? Perspective graphics, good command handling, and a large map combine to make this an intriguing game. The going isn't easy, and it may be a long time before you can even find the way out of your cell. You do have a hand grenade that someone slipped into the cell, but how can you use it in a small room without blowing yourself to bits?

Once you get out, you find yourself roaming through corridors and trying doors in the hope of escaping before the guards catch you. If you do meet a guard, you may still be able to outwit him. Patients might be able to help you if you think you can trust them.

Graphic adventures often sacrifice command handling for pictures, but *Asylum* is better with commands than many all-text adventures. Its vocabulary (which can be displayed on request) is over 200 words. You can enter commands as complex as GIVE COAT TO GUARD and OPEN DOOR WITH SILVER KEY.

There are some oddities, though. Objects that aren't being carried are always in "boxes." If you drop an object, a box magically appears to hold it. To see what is in the room, you have to OPEN BOX. You can also put an object under something else. For instance, you can PUT GRENADE UNDER BED. Once you do this, no amount of searching will reveal its

presence. Only if you remember where it is and type TAKE GRENADE UNDER BED can you get it back.

As you make your attempt to escape, a clock keeps ticking whether you move or not. You have only 12 hours of game time (about 6 hours of real time) to flee before the more competent day shift comes to lock you safely away. Fortunately, you don't have to do the 6 hours at one sitting; you can suspend or save the game at any time.

One last word of warning: Don't look up! You never know when they might decide to drop something on you.

Asylum II

Asylum II, *by William F. Denman, Jr., is published by Med Systems Software.*

Many creators of successful products have tried to follow up their success with a sequel, only to produce a pale shadow of the original. This is largely true of *Asylum II*. The idea, the graphic approach, and the command handling are all essentially unchanged from *Asylum*. Once again, you start in a cell and must escape. Why did Denman want to go over the same material again? Perhaps he was trying to create an easier version for players who were frustrated with the original. In many ways, this is easier. Getting out of the cell initially is not as difficult. There are no guards roaming the halls to put you back. A player can map out large areas of the asylum after getting past the first locked door.

However, this simplification results in long, arid stretches in which the player doesn't have much to do besides mapping the convoluted passages. Most of the challenge comes from accumulating the keys needed to open various doors. Some doors, unfortunately, will set off alarms when opened and end the game. The pattern behind these alarms seems random and varies each time the game is played; often a door that leads to a needed item will set off an alarm when opened. This can lead to a lot of frustration, even with frequent saving of the game.

Many players of *Asylum* have undoubtedly been unable to get far enough to discover the oddities of its map. *Asylum II* lets you discover early in the game how strange its layout is. There are at least two locations that quietly teleport you from one location to another, as well as a block of rooms in the form of a five-sided square! Interesting locations to visit include shock therapy, plastic surgery (performed without

anesthesia), and psychiatry (where you can literally, be bored to death if you hang around too long).

The documentation for the program cassette claims that one side of the tape holds a 16K version and the other a 32K version, the difference being that the 16K version has shorter messages and a smaller vocabulary. On my tape, though, the reverse side was marked "Duplicate," and I couldn't detect any differences in the programs on the two sides. The messages, unlike those in my IBM PC version of *Asylum*, are just phrases rather than full sentences, and the vocabulary is smaller.

Bedlam

Bedlam *is available from Radio Shack.*

Here is another adventure in which you have to escape from an insane asylum. No, not an asylum for the insane, but an insane asylum. Your goal is to get out in spite of the efforts of a needle-waving doctor, a sadistic therapist, and a talking guard dog (if you say HELLO to the dog, he answers). You may be able to get help from some of the other patients. Each time the program is started, these characters start out in different cells. Once you find them, they will often follow you around.

Unlike most adventures, the solution varies each time you run the game. This adds to *Bedlam*'s staying power, since you can play it several times before you have found all the possible solutions.

• Radio Shack's adventure writers seem to have a fascination with the word *PLUGH*, which originally appeared in *Adventure* itself. It shows up in several of their adventures for one purpose or another. In this one, it can get you out of some otherwise hopeless situations.

The command parser has the inconvenient feature, in common with several other Radio Shack adventures, that when you enter an invalid command, the cursor remains in the command line to let you edit it. It turns out that hitting the CLEAR key lets you make a fresh start, though this isn't mentioned anywhere in the instructions. The program's descriptions, entirely in upper case, are grammatical but a bit dull.

Blade of Blackpoole

Blade of Blackpoole, *by Tim Wilson, is published by Sirius Software.*

Your goal in this adventure is to find the magical sword Myraglym and bring it back. Aiding you in your quest are a command parser, which is better than many well-written text descriptions, and reasonably good high-resolution graphic illustrations of each scene. Still, it may be a long time before the score that measures your progress gets above zero. There are two obvious obstacles near the beginning, and one of them turns out to be completely impassable. Now there's nothing wrong with having dead ends in an adventure; but having so tempting a dead end so close to the start is bound to cause any player hours of frustration.

Many of the other puzzles are simply capricious. A normal-looking rope exhibits magical properties if thrown, but only in two locations. A pit is sometimes there and sometimes absent. A shrinking potion works on some objects but not on others. Another irritation is the patronizing tone of the messages that tell you a command doesn't work. Is it really necessary to tell a death-defying adventurer that he might get hurt if he climbs a tree?

A good number of features are included. You can store up to ten save files on disk. Commands such as TURN ON THE LAMP and MOVE THE ROCK WITH THE LEVER are supported. If further information is required, the program may ask "With what?" or "On what?" and accept a completion. You can get hints with the HELP command; considering how silly some of the puzzles are, this is something you shouldn't be ashamed to do.

Blade of Blackpoole makes a good first impression, but its puzzles are apt to drive even an experienced adventurer to frustration. Approach it with patience and willingness to try the unreasonable.

Cyborg

Cyborg, *by Michael Berlyn, is published by Sentient Software.*

Cyborg is a science-fiction concept meaning a being that is partly human and partly machine, and the concept is a natural for describing the view that an adventure player has of the

game world. You *and* the computer are the cyborg. Your real-life computer represents the machine part of your mind; it gives you all your data, and you can ask it to SCAN things or even to give an OPINION.

When the game starts, you're lost in a very strange forest. Your memory banks are damaged, and you haven't the faintest idea of how you got here. Scattered around the forest are portals that lead into rooms where there isn't even a building. You can pass through some of these portals, but getting through others takes a little work. Your first goal is just to get enough energy to survive for a while; then you can worry about what you're doing here and how you can get out.

Perhaps the space-suited lizard can help you—if you meet his price. Or maybe the mini-droid can do things you can't do yourself. The ultrafiche might have the information you need, but how do you read it?

The command handler is quite good. You can take and drop multiple objects with one command, and you can give indirect objects with prepositions (GIVE FOOD TO LIZARD, INSERT COIN IN SLOT). Certain common commands, such as LOOK, don't work; but this is because more cyborg-like commands, such as SCAN, have replaced them for the sake of atmosphere. As the instructions explain, you can type AREA SCAN to look around or BODY SCAN to check how healthy your human and cyborg halves are.

This game, by the author who has since done *Suspended* and *Infidel*, is one of the best-written adventures available.

Dungeon

Dungeon *is published by the Digital Equipment Corporation Users Society (DECUS).*

The creators of *Zork* claimed that their game couldn't possibly be done in FORTRAN (it was done in MDL, a LISP-like language). Robert Supnik of Digital Equipment Corporation took them up on that claim, and the result was *Dungeon*.

Dungeon contains all the locations, descriptions, and puzzles of the original *Zork*. It falls short only in its command parser, which cannot handle all the complex commands of *Zork*. This shortcoming does make some of the puzzles more difficult, since certain commands won't work.

There are several versions of *Dungeon* in circulation, varying in completeness. The most complete one includes the Puz-

zle Room, the Tomb of the Unknown Implementer, and an end game when you have gathered all the treasures into the trophy case.

Another incarnation of *Dungeon* is *House of Banshi*, which is available on the CompuServe Information Service. At $6.00 an hour, you could go broke trying to solve it, but it's a good opportunity for CompuServe subscribers to try a few moves and get the flavor of *Zork*.

Empire of the Over-Mind

Empire of the Over-Mind *is published by Microcomputer Games, Inc., a division of The Avalon Hill Game Company.*

Your quest is to destroy the evil Over-Mind, which is powerful in its own right and has various creatures guarding it, too. The action takes place on two planets; magical means let you get from one to the other.

Eventually you will meet the Over-Mind itself, and that encounter may be the toughest part of the whole quest. Unlike most adventures, this one does not use compass directions at all. Movement commands always specify a portal, such as a road, a cliff, or a door, that leads from one place to another. Typical movement commands are GO TO DOOR, CLIMB TREE, and FOLLOW ROAD.

The command parser is a little more sophisticated than the usual verb-object handler, in that it lets you use adjectives. Unfortunately, it is not very good at indicating what is wrong when you type in a command that it can't interpret. It also makes assumptions when it encounters an ambiguous command, so you may find yourself doing something you didn't intend to do.

Included in the package is a poem that gives the history behind the Over-Mind and contains a number of vital clues. Read it carefully, and reread it if you get stuck.

A subtlety of *Over-Mind*, not found in any other game that I know of, is the distinction between *carrying* an object and *holding* it. You can carry several objects but hold only one, and you can make effective use only of the object you are holding. This requires planning ahead, since you could be in trouble if you have to rummage through your knapsack while a monster is attacking you.

A game in progress can be saved on tape, but on TRS-80 computers this requires having the TRUG program

More Adventures

G.F.S. Sorceress

G.F.S. Sorceress, *by Michael G. Cullum, is published by Microcomputer Games, Inc., a division of The Avalon Hill Game Company.*

This science-fiction adventure casts you in the role of Lieutenant Joe Justin of the Galactic Federation Navy. As the game opens, you have just been falsely convicted of mutiny and sentenced to float in space forever in a life-sustaining spacesuit. But a friend has hidden a thruster in your suit, and with it you are able to find your way to another starship.

This starship, the G.F.S. *Sorceress,* has a single occupant, Captain Selena Sakarov. Why is she commanding a starship without a crew? How does she know so much about you? Why does she entrust the ship to you and then go into a hybernation chamber? Sadly, these are things you will not learn in this adventure. But the ship's computer contains programming to go to any of four worlds besides Earth. You have nothing better to do with the rest of your life; you may as well go to these worlds on the wild chance that something you find will help you prove your innocence.

The underlying program is very similar to *Empire of the Over-Mind.* It has the same strengths and weaknesses as *Over-Mind*'s command handler, except that it seems to run a little faster. Avalon Hill still hasn't learned that proofreading is necessary in adventure programs; spelling errors abound, although the use of the language is otherwise good. An unsuccessful "improvement" over *Over-Mind* is that the screen is cleared after every command you type.

The puzzles are fairly simple, and nothing in this adventure should cause an experienced player excessive trouble. Until you get close to the end, though, there is no sign that you're making any progress toward your goal.

What is especially noteworthy about this adventure is the supplementary material. One of the booklets included with it is entitled "Restless Universe" and tells Joe Justin's story up to the beginning of the adventure. The other booklet, the "Galactic Federation Naval Officer's Manual," contains descriptions of the planets you can visit, along with color illustrations. These illustrations are an intelligent alternative to screen graphics, since they look better and don't take up memory space. The "Officer's Manual" also includes several sections

that you are instructed not to read until the program tells you to read them. You should observe this request, since reading them would give away developments in the adventure.

Haunted House

Haunted House *is published by Radio Shack.*

The unique virtue (if virtue it is) of this adventure is that you can have lunch, solve it completely, and still be back to work within the hour. Running in a mere four kilobytes of memory (but broken into two tape loads), it lets you go into a haunted house and try to get back out alive. As is usual for tiny adventures, more of the challenge comes from figuring out the program than from the actual situation. A special source of frustration is that the program doesn't tell you whether your movement commands are successful or not; it simply repeats the description of the place you were in if you can't go in the direction you specify. Combined with the fact that adjacent rooms often have identical descriptions, this makes figuring out where you are rather difficult. Another irritation is that you can't restart the game when you lose, except by reloading it from tape.

The magic word *PLUGH* makes its appearance in this adventure as the word that gets you into trouble in the first place. So if you just don't say it, you're as well off as if you had gone through the whole adventure and solved it. But solving it is more fun.

An amusing feature of *Haunted House* is that you can kill a ghost. No, I don't know how that can be, but the program tells you afterward that the "body of a dead ghost" is in the room.

Whatever its weaknesses, this game shows an offbeat sense of humor. It's also one of the lowest-priced adventures available.

Lords of Karma

Lords of Karma *is published by Microcomputer Games, Inc., a division of The Avalon Hill Game Company.*

Your goal here is to gain karma points by doing appropriate deeds in a world of rather strangely mixed myths. You are reborn after every death, but if your karma points are negative at the time of your death, you must first suffer in Purgatory. Tolerance must not rank high in the eyes of the Karma Lords,

since they award you a large number of points for destroying an idol. The map is large but repetitious. Most of the locations are forests of various kinds with nothing exciting in them. There are some interesting puzzles, but they are too few and far between. Nor does the inattention to proofreading help.

Some story fragments are present. You can rescue a princess from a ruffian, after which she will DEMAND (emphasis hers) that you take her to her father and will keep following you until you do. There is also a wizard who commands you to bring him the staff of the evil shimmering wizard. Off in the woods, there is a not-so-jolly green giant. You might eventually wonder why you should bother with karma points for helping these unpleasant characters; in fact, the game works just as well if you play to get the lowest possible karma score instead of the highest. But if you think this way, you might be better off playing *Prisoner 2* (see below).

This adventure accepts simple two-word commands and has a reasonably large vocabulary. By taking up 48K bytes of memory, it manages to have a less constrained feel—but it seems more diluted than expanded.

Merlin's Treasure

Merlins's Treasure, *by Michael D. Wile, is published by Adventure International. It is packaged on disk as the flip side of* The Sledge of Rahmul.

If you've ever wanted to do in a wizard and then steal all his possessions, this adventure may be for you. Each orientation in each room has its own graphic presentation (within the TRS-80's rather meager capabilities), and a box on the screen displays the last five commands entered. Movable objects in the room do not appear in the display.

The adventure does have some challenging puzzles. Unfortunately, the command language is extremely limited, and many commands simply have no effect. You can beat the wizard up and even stab him with a dagger; he'll just ignore you and keep contemplating the ineffable until you either try to take something (he does defend his property) or find the way to kill him.

One element of storytelling that should be present in any adventure is motivation. That is, if the character must take some action in order to succeed, he should first have access to a clue that suggests that action. For instance, if going into a

certain passage is dangerous and requires special precautions, the character should be able to get prior warning of the danger. Without this warning, you would have the character take precautions only because you knew from previous play that doing otherwise would be fatal. From your character's standpoint, this means taking actions for no reason at all.

Merlin's Treasure violates this rule in a particularly blatant way. One room is protected by a beam of light which triggers a trap if you walk through it. There is no indication that you are coming up to this beam until it's too late. The net effect is that your character will crawl through the room only because of knowledge gained in a previous incarnation.

A hint sheet is included; the hints are scrambled to prevent unintentional peeking.

Prisoner 2

Prisoner 2, *by David Mullich, is published by Interactive Fantasies, a division of Edu-Ware, Inc.*

Most graphic adventures use the graphics just to add pictures; but *Prisoner 2* integrates text and graphics in an innovative and effective way. The adventure is based on the television series *The Prisoner,* in which Patrick McGoohan played a captive secret agent struggling to keep the independence of his own mind despite the insidious methods of "The Village." Here, an animated sequence, similar in spirit to the sequence that opened each episode of *The Prisoner,* graphically transports you to "The Island," where your captors attempt to get a piece of secret information from you. This information has been compressed into a three-digit number, so your first priority is never to get careless and type in that number for any reason.

On arriving at the Island, you must find your way through a maze. You move with one-letter commands; just to break your morale down a little more, the commands that are echoed on the screen aren't the same as those you typed in. For the sake of your sanity, though, you really do move according to the commands you type.

Once you've gotten out (and there's an exceedingly devious way to make it easier), you emerge among a group of buildings where you can go through a battery of tests. These tests establish how well you are "adapting" and also include an occasional sneaky attempt to get you to reveal the secret

number. Remember, no matter *what* the program does, *don't let them have that number!*

The more individualistic you are, the more points you get. But if your actions get out of bounds, you will incur the wrath of Pax, a yellow sphere with an eye and a mouth in profile (where have I seen that before?). And if you do disclose the secret number, the program rubs it in quite thoroughly in a final graphic sequence that welcomes you to "the flock." *Prisoner 2* is the best example I have seen of social satire in a computer game.

Pyramid

Pyramid *is published by Aardvark Systems Ltd.*

This adventure is very much in the Scott Adams mold. It has two-word commands, terse responses, clever puzzles, amusing solutions, and bad spelling. Like the Adams adventures, it is presented in the first person ("I am in...."). The premise of *Pyramid* is simple enough: you have set out to explore a pyramid and get treasures. Your first problem is that there is only one obvious entrance to the pyramid, and that one is locked. But you can get in, and after solving several puzzles, you will find yourself wandering through the twisting corridors of the pyramid.

Some of these corridors contain inscriptions that don't look at all like hieroglyphics; oddly enough, they give the impression of being English cryptograms! The code isn't hard to break, and the messages give vital clues.

One passage is guarded by a mummy. If you try to hit or push it, it becomes a "very mad mummy." Try again and it becomes a "very mad very mad mummy," and so on. Fortunately, it never gets quite mad enough to do anything but block the passage.

You are limited in the number of items you can carry, but you have a knapsack that lets you carry more. To store an item, you type PUT (item). The program asks "Where?" and you answer IN KNAPSACK (or wherever else it might be appropriate to put an item).

While this is a solid adventure in many ways, its interface to the player is unforgivably poor. Words are truncated to two letters, causing a great deal of confusion; for instance, if you type FLOOR, the program will take the word as FLASH-LIGHT. Many actions, both successful and unsuccessful, get

no response from the program; you have to LOOK to see if anything has happened. (Unlike the Scott Adams adventures, *Pyramid* does not reserve part of the screen for a constantly updated description of your surroundings.)

Only 17 verbs are recognized. You can't ENTER or CLIMB something; you can only GO it. LOOKing things works, but EXAMINEing them or SEARCHing them gets the response, "I don't know how." You can't make northward progress by typing N or NORTH or even GO N; you must type GO NORTH.

Aardvark claims that this is one of its toughest adventures, with the average time to solve it being 50 to 70 hours. While it isn't exactly easy—and while I'm not entitled to say for certain without having solved it completely—this claim seems exaggerated. The problems are tricky, and the limited command handling doesn't help, but intermediate players shouldn't be scared away from *Pyramid.*

Raaka-Tu

Raaka-Tu, *by Robert Arnstein, is published by Radio Shack.*
You have located a temple on the fringe of the Himalayas, patrolled by implacable guards. Being a dedicated (and slightly greedy) anthropologist, you are determined to get into the temple in spite of the risks. Once you have made it in, you must make it past various guardians of the temple in order to pick up treasures and then leave. To get full credit for your treasures, you have to bring them back to your starting point.

The command parser allows adjectives and prepositions, but the program doesn't make much use of this capability. When you want to FEED something, the program insists that you tell it with what (with the food, obviously!), and you must GO INTO some places when ENTER would have served just as well. For the most part, the program could have used a simple two-word parser and made more room for commands and messages.

The responses range from the humorous to the irritating. If you try to rub the lamp, it goes out, and the program explains that "you must have rubbed it the wrong way!" Less amusing is the response, "Why don't you leave the poor ... alone," which the program frequently gives when it can't handle your command. No adventure can cover every possible

action, but it doesn't have to try making the player feel guilty about its inadequacies.

The major irritation of the game is that when you find the exit, there's a fifty-fifty chance that the guards will see you and kill you on your way out. This is very frustrating when you've picked up all the treasures and are leaving. If there is a way to avoid this hazard, I don't know it. And even when you do get the maximum score, there are no fireworks or congratulations; the program just routinely advises you, when you type SCORE, that your score is 50 out of a possible 50.

The documentation includes a package of hints marked "for hopeless situations only," which are geared toward the beginner. Much of the game is simple, even though the highest possible score is elusive. It's important to SEARCH every room in order to find everything.

The Sledge of Rahmul

The Sledge of Rahmul, *by Roger Jonathan Schrag, is published by Adventure International. It is packaged with* Merlin's Treasure.

This game is the flip side of *Merlin's Treasure*, and it is the better of the two. Unlike most adventures, this one checks each entire word of each command rather than truncating words to some number of letters. Simple graphics are included to give a perspective view of the way you are facing; you have to LOOK in all four directions to make sure you know about everything in a room. In this adventure, you start out in serious trouble; the lights go out on you, and you have no apparent source of illumination!

Once you overcome that little difficulty, you find yourself wandering through a maze, hindered by such foes as a nasty rat, an ugly bird, and a prison guard.

The command handler goes beyond the usual verb-object parsing to allow commands of as many as four words. Unfortunately, if you type more words than the program regards as necessary, it simply ignores the rest. This can be misleading at times. Also, there are some things that the program won't let you do, but without giving any explanation. For instance, your shears are sharp enough to cut a dead bush, but inexplicably won't cut a green twig.

While this program is a good one in many ways, it pro-

command-handling possibilities for graphics—especially on a computer that has such a limited graphics capability in the first place.

An interesting technical feature is Russ Wetmore's Duo-Loader, which allows the same disk to run on either a TRS-80 Model I, a Model III, or a Model IV.

Spook House

Spook House, *by Roger Jonathan Schrag, is published by Adventure International. It is packaged with* Toxic Dumpsite.

This is my favorite of Schrag's adventures using the *Sledge of Rahmul* system. Its most unusual feature is the realtime limit; if you fail to get out of the house within half an hour, a bomb goes off and kills your character. Perspective line drawings are used to indicate your point of view as you look in one of the four cardinal directions.

Semianimated graphics are used to depict a strobe-lit room and a spinning room, and the graphics in general add a bit of fun. One corridor, for instance, doesn't lead where it seems that it should; looking in various directions, you see signs reading "Lost?" and "Confused?" Elsewhere, when you land in some water, the program tries its best to persuade you that you're sinking, but the pictures indicate that you're not exactly in "deep" trouble.

Examining some objects will provide strange, cryptic messages. These may be hints of some sort; more likely they're bugs in the program. Sliding down the fireman's pole is fun and sometimes necessary, but the program keeps track of how many times you do it, and it makes you pay the price for over-exertion. Having good peripheral vision is important; check the pictures for directions in which you should look.

One of the toughest traps in the adventure is the "continuous corridor." You can go east or west as far as you like, but you'll never get to the exit. If you do find a way out, please let me know; I'm stuck there, and the bomb is about to....

Stone of Sisyphus

Stone of Sisyphus, *by Dixon Scott, is published by Adventure International.*

This adventure, part of the Maces and Magic series, incor-

first time you play, you are given a character with randomly generated strength, dexterity, charisma, and so on. You must outfit this character by spending your initial allotment of gold on weapons and armor. What you can bring into the dungeon depends not only on what you can afford, but on how much you can carry. In addition, your dexterity must be good enough to handle the weapon you choose. If your character survives the expedition, his record is saved on disk for reuse in a later game.

Having outfitted your character, you proceed toward a building with two entrances. All scenes are illustrated with graphics; the quality, naturally, varies greatly depending on which computer the program is running on. You indicate your choice of action with single-keystroke commands. At each location, you are usually presented with a choice of several actions designated by numbers. For instance, when you come to the Troll Bridge, you can enter 1 to pay the troll, 2 to fight him, or 3 to turn back.

In addition, several single-letter commands always work. G lets you grab an item. L lets you leave it behind, I lists your inventory, and so on. Especially important is the P command, which lets you pick an item that you're carrying and do something with it. Only with the P command can you exercise your imagination and enter commands of your choice. Usually you will enter just a verb, with the thing you picked being understood as the object; sometimes a verb and object will work. For instance, after picking a pile of timbers, you might give the command, BUILD HOUSE.

The puzzles often show a strange sense of humor, and some of them aren't too difficult. One room has a tiny red creature that suicidally attacks you and then remains on the screen as a squashed blob afterwards. Nearby is a manticore who covers the walls of his room with pictures of monster stars. The text descriptions that go with these are quite a bit of fun.

With all this going for it, it's a great shame that the way the game runs virtually ruins it. The program runs under the BASIC interpreter, which doesn't make it speedy, and it constantly accesses the disk. It often has to go to the disk several times even to put up a single screen display. Some commands must be terminated with the Return key, while others are just single keystrokes. At other times, you must hit a key just to

redisplay the message on the screen so that you can enter a command. The response is so slow that you can't be sure whether you've completed the command or not.

Sisyphus was a character in Greek mythology who was condemned to roll a boulder up a hill in Hades, only to see it roll back down and have to repeat the process eternally. How that myth fits into the adventure isn't apparent from the parts of the dungeon that I've reached; but the way the program runs gives something of the sense of frustration that Sisyphus must have felt. *Stone of Sisyphus* is a case of a good idea that wasn't carried through with good programming; if Chameleon Software, the group that developed the program, would only redo it in a more efficient language and improve its disk management, its next edition could yet be a success.

Toxic Dumpsite

Toxic Dumpsite, *by Roger Jonathan Schrag, is published by Adventure International. It is packaged with* Spook House. This adventure is packaged on a two-sided floppy with *Spook House*. The command handler is similar to the one in *The Sledge of Rahmul*. In addition to having simple graphics and multiword commands, this adventure (like *Spook House*) is a race against real time. An underground toxic waste system has failed, and you have been sent in to pull the master shutoff lever before the dump blows up. You have about 20 minutes to do this, regardless of how many moves you make.

The first part of the adventure takes place on the office level of the dumpsite. You've been sent in with no training or preparation, so you have to find the things you need just to get through the front door. Looking in, under, and behind things is important. Your next major job is to descend in the elevator to the underground level. This puzzle is difficult but not especially clever; it involves doing something quite far away, which you would have no reason to associate with the elevator.

Down below, you have to reckon with a lack of ventilation; and you'd better have brought a light source. There is lots of button-pushing to be done, and you must always specify the color of the button. Why this should be necessary, I don't know; I haven't found any situation in which buttons of two different colors can be in the same room.

A hint sheet is included for both this adventure and *Spook*

House. This adventure lacks the humorous touches that make its flip-side companion fun, but it plays fair (with the one exception noted earlier) and rewards the systematic player.

Xenos

Xenos *is published by Radio Shack.*

The programming underlying this adventure is very similar to *Bedlam* and *Raaka-Tu,* but the elbow room, which the disk format provides, makes it significantly better than the others. You have been assigned to investigate a reported UFO landing and rushed to a recently evacuated town near the landing site. You will probably want to spend a while investigating the deserted buildings and looking for any equipment that might help you. Like most adventure characters, you have been sent out without decent preparations, and it's a poor idea to go out into the desert without food and water!

When you do find food, things become easier. Even though what you find is described as a "small amount," there's always more left after you eat it—and it somehow fills your need for water as well. Who says you can't have your lunch and eat it, too?

Properly equipped, you can venture out into the desert. But watch out for alien creatures, an unfriendly prospector, and a magnetic anomaly that can have you going in circles if you trust your compass.

With perseverance, you can eventually reach the alien craft. To get in, though, you have to get past a sizable obstacle. The program describes the rooms on the ship in the aliens' language; this adds quite a bit of color, as well as reflecting the fact that you don't know what anything in this ship is for. Once you decipher these mysteries, you can find out whether the aliens are simply trying to phone home or are up to something more sinister, and you can take appropriate action. Be careful poking around the ship—pushing the wrong button could destroy the world!

The program is overlaid from the disk, so that more descriptions can be presented than will fit in memory at one time. This allows for fairly detailed descriptions. However, the overlay structure isn't very clever. If you go back and forth between two adjacent locations that are in different overlays, the program will access the disk every time you go from one to the other.

As in other adventures of the *Bedlam* family, the command parser can handle adjectives and prepositional phrases. *Xenos* makes much more use of this capability than the others do. There is, for instance, a desk with three drawers and an assortment of keys to try on them; so you might try to OPEN MIDDLE DRAWER WITH BRASS KEY. The only problem is that the program sometimes insists on too much information. If you try to SHOOT a creature, it wants to know what you're shooting it with, even though there's only one possibility.

There are a number of logical red herrings. Don't assume that every difficulty represents a puzzle to be solved; if you try too hard, you may use up resources that are necessary for the real work.

Xenos is easily the best text adventure (not counting *Zork*) that Radio Shack has offered.

Action Adventures

Action Adventures

close relative to the text adventure is another kind of computer game, frequently called an "action adventure." This type of game lets the player control the actions of a character who is exploring some area for the sake of gold or glory, just as a text adventure does. The difference is that it doesn't use English-language commands and text descriptions, but instead displays a map of the area the character is moving through and accepts single-keystroke commands.

While action adventures are a different kind of game from text adventures, each borrows characteristics of the other. *Prisoner 2* uses single-key commands and map displays, for instance, and *Stone of Sisyphus* incorporates many of the features of action adventures, allowing both single-keystroke and word commands. Future adventure games may well show even more of a blend of the two types.

The basic similarity between the two types of adventures is that in each case, the player controls the actions of a character in a hostile environment. The goal in either type of game can be either to pick up treasures or to fulfill a quest. In each type of game, the player has a variety of actions at his disposal.

However, there are many differences between the two types of adventures. Here are some of the major distinguishing characteristics of action adventures, starting with the most basic.

Single-key commands and joystick control. The command structure of action adventures is simplified so that the player can concentrate on other things. Typically, the player can move the character with the joystick or the cursor keys. Other commands have well-defined functions; O might open a box or a door, S might cause something to be searched, and so on.

Display of the character's position on a map. Usually, this is an overhead view of the map, showing only what has been explored so far. In some cases, the display may show a perspective view from where the character is standing; this technique is also used in some graphic text adventures, such as *Asylum*.

Realtime action. The *action* in an action adventure comes

player doesn't do anything for a while, a monster may wander into the room and attack his character. In combat, the player must keep active and be ready to decide when to run away.

Continuous movement. The character is not located simply "in a room," but is always at a particular point in a room. Treasures, monsters, and the like also have specific locations, and the character must move toward them to carry them away or fight them.

Elaborate combat sequences. In text adventures, the player usually fights a monster by typing FIGHT MONSTER and lets the results fall as they may. In action adventures, however, as a general rule, there is much more detail. To strike a blow requires hitting a key. As the character receives blows, he weakens; if he gets into trouble, he can try to move away and elude the pursuing enemy. Often the character can either thrust or parry. He may also be able to shoot arrows or laser beams instead of moving into hand-to-hand combat.

Character generation and experience. In text adventures, the character's abilities are generally fixed. He is strong or agile enough to do certain things but not others. The reason for this is that text adventures are mostly concerned with problem solving; if a powerful character could just bludgeon his way through obstacles, much of the point would be lost. However, action adventures are based on different expectations. The character will fight and run, sometimes winning and sometimes dying. Skill and luck are necessary, as well as just knowledge of what to do. The character who comes out of the dungeon alive and wealthy is distinct from his predecessors who died in the attempt. Moreover, in the process of battling his enemies, he may gain experience points. The more of these he gets, the more effectively he will be able to cast spells and fight.

Variety in repeated playing. A text adventure that you've solved is like a novel that you've read—you might want to go back over it to relish details that you had missed, but it no longer offers a challenge as a game. An action adventure, though, makes demands on your skill every time you play it. You may have beaten the wyvern last time, but that doesn't guarantee you can do it again.

Simple plots and little characterization. This is the price that action adventures pay for their other features. The charac-

mined ways, most of them violent. Other beings appear only as enemies to fight or to perform a specific function before they disappear. Lacking dialogue, action adventures cannot have the subtlety of text adventures. This isn't to say, however, that they can't pull some utter surprises on the player.

One of the oldest and best-known action adventures is *Temple of Apshai* from Epyx, which is in many ways quite similar to Dungeons and Dragons.

The first part of the game generates the character. The player can either specify his own character's attributes or let the program randomly generate a set of attributes for him. These include intelligence, intuition, ego, strength, constitution, and dexterity. The range for these attributes is the same as in D&D, so it is not difficult to transplant a D&D character into *Apshai*. Intuition replaces the D&D characteristic of wisdom, and ego replaces charisma.

The character must purchase supplies from the Innkeeper, using a limited amount of silver. The Innkeeper has armor, weapons, and healing salves. The player can bargain with him over each item until a price is agreed upon.

Once the character is fully equipped, he can venture into the ruined Temple of Apshai to go treasure hunting. There are four levels to the dungeon; the higher numbered levels (which are deeper underground) have greater treasures and correspondingly more powerful monsters.

As the character moves through the dungeon, the area he occupies is mapped on the screen. Only a small part of the dungeon fits on the screen at one time, so the program frequently redraws the map as the character moves to new rooms. This mapping is extremely slow, as the program is written largely in interpreted BASIC.

The booklet that comes with the program contains much of the information about the dungeon. When the adventurer enters a room, the computer's screen displays the number of that room. The player can then look up the number in the booklet and find out what he is seeing. These descriptions often provide important information; but if you're going to try to read them while an ant man is charging toward you, you'd better be a fast reader. The booklet also contains explanations of the various treasures and traps found around the ruined temple.

Action Adventures

There are a large number of possible actions. Single keystrokes let the character open doors, search for secret doors and traps, listen for sounds in the next room, apply healing salves, speak with monsters to avoid combat, or fight. In combat, the character has the choice of attacking, thrusting, parrying, and firing normal or (if he has them) magic arrows.

The adventurer can move rapidly or slowly. Rapid movement is sometimes essential for escaping monsters, but it builds up fatigue. In addition, the heavier the adventurer's armor, the more he will be tired by rapid movement. Fatigue impairs his ability to fight; complete fatigue makes him unable to move or fight until he has rested. If a monster wanders by while the adventurer is in this state, it could mean big trouble!

As in Dungeons and Dragons, a dead adventurer can often be resurrected. Since the *Apshai* adventurer goes in alone, however, his life can be restored only if another adventurer happens by and brings his body out. Some rescuers will do this deed simply out of generosity; others will grant themselves a large reward out of what the adventurer was carrying.

Temple of Apshai has been so successful that Epyx has published modules for additional portions of the temple as well as for applying the game system to other situations.

Other game publishers have carried the concepts of *Apshai* to even greater lengths. The Ultima series, from Original Systems, Inc., lets the player incorporate an entire party of characters instead of just one. The characters can be of a variety of species (human, elf, dwarf, etc.) and professions (thief, cleric, fighter, etc.). Magic spells are included in the game, and partial maps are provided to aid the party in its quest. *Ultima III* provides for a limited opportunity to enter words as well as single keystrokes as commands, thus crossing partially into the realm of text adventures.

While the Ultima adventures make the maximum use of a computer's graphics and sound capabilities, it is not true that all action adventures need a computer with high-resolution graphics. *Rogue*, a program widely circulated on Unix systems, presents the essentials of an action adventure using computer terminals with no graphics capability at all. Simple characters are used to draw the maps of the 20 levels of the dungeon. The adventurer is represented by the *at* sign (@);

monsters are represented by a single letter (for instance, Z for zombie).

The rogue moves about the dungeon under keyboard control; he attacks a monster by attempting to move over it. As he defeats monsters, he gains experience points and becomes more powerful. Once he has reached a high enough experience level, he can venture into the lower levels of the dungeon, where more powerful monsters await him. If he defeats all the monsters, there is a treasure awaiting him on the twentieth level; but then he has to bring it all the way up again. The state of the game can be saved to a file at any time.

In exploring the dungeon, the adventurer will find magical potions, weapons, scrolls, and armor to help him. Some items are cursed, though, and rust monsters can reduce even the best armor to a corroded mess. Other monsters to watch out for are giant ants, which sap the adventurer's strength with their stings, and umber hulks, which cause him to blunder around aimlessly.

Rogue serves as the basis of *Sword of Fargoal* (from Epyx) which offers essentially the same ideas in an elaborate graphic version. *Fargoal,* unfortunately, does not have the game-saving feature of *Rogue,* and it takes many hours to play.

Some adventures vary the perspective from that shown in the overhead map. *Aztec,* from Datamost, shows a cross section of part of a pyramid that holds a golden idol. The adventurer is shown in full-body profile as he walks down the hall or up or down the stairs. When he misses a step (which he does surprisingly often), he lands flat on his back with stars circling over his head until he recovers. His enemies, which include spiders, pumas, giant lizards, carnivorous plants, and Aztec guards, are strikingly animated. When the adventurer moves from one screen to the next, the new cross section appears almost instantaneously.

The adventurer can fight using either a pistol or a machete. In the hands of a skilled player, he can be made to move, thrust, and lunge in a very realistic-looking manner.

Aztec does have its quirks. Treasure chests are often seen floating in midair with no floor beneath them. Stairways may lead nowhere. Occasionally the adventurer can manage to slip through a wall. In a few cases, he will find himself standing in midair, only to fall the moment he takes a step.

Action Adventures

While most action adventures are set in mythical or ancient realms, *Castle Wolfenstein* offers a more modern setting. In this one, the player takes on the role of an American soldier trying to escape from a Nazi prison. Firing bullets and throwing grenades at guards provides most of the action, but occasionally the fugitive must open a chest to get more ammunition. This is a very time-consuming process, and he has to hope no guards show up before he has it open.

In the world of noncomputer role-playing games, there are two kinds of players. One kind wants, above all, a game that is full of action and battles. What he enjoys most is the challenge of being a master tactician or the intricate mechanics of game combat or perhaps the pleasure of chopping an imaginary enemy's head off. The other kind prefers deep puzzles in which the characters must go searching and spying—or just ponder a situation until they can devise a way to approach it.

In computer games, action adventures appeal to the role-playing game player whose first love is battles; text adventures appeal to the problem solver. The two areas aren't mutually exclusive; *Apshai* and the Ultima series offer challenges to the problem solver, and many text adventures include battles with monsters or villains. Nor are the two kinds of players mutually exclusive; most people who enjoy one type of adventure will enjoy at least an occasional fling with the other. But text adventures and action adventures do generally require different frames of mind if the quest is to be successful.

A Field Guide for Frustrated Adventurers

A Field Guide for Frustrated Adventurers

very sort of game has its challenges, but text adventures can be among the most frustrating. Overcoming a game's difficulties isn't just a matter of practicing until you get good enough; it's more a matter of thinking about the situation until the right idea finally hits you. That gives a feeling of discovery and achievement that is uniquely close to the satisfaction of solving a real-life problem.

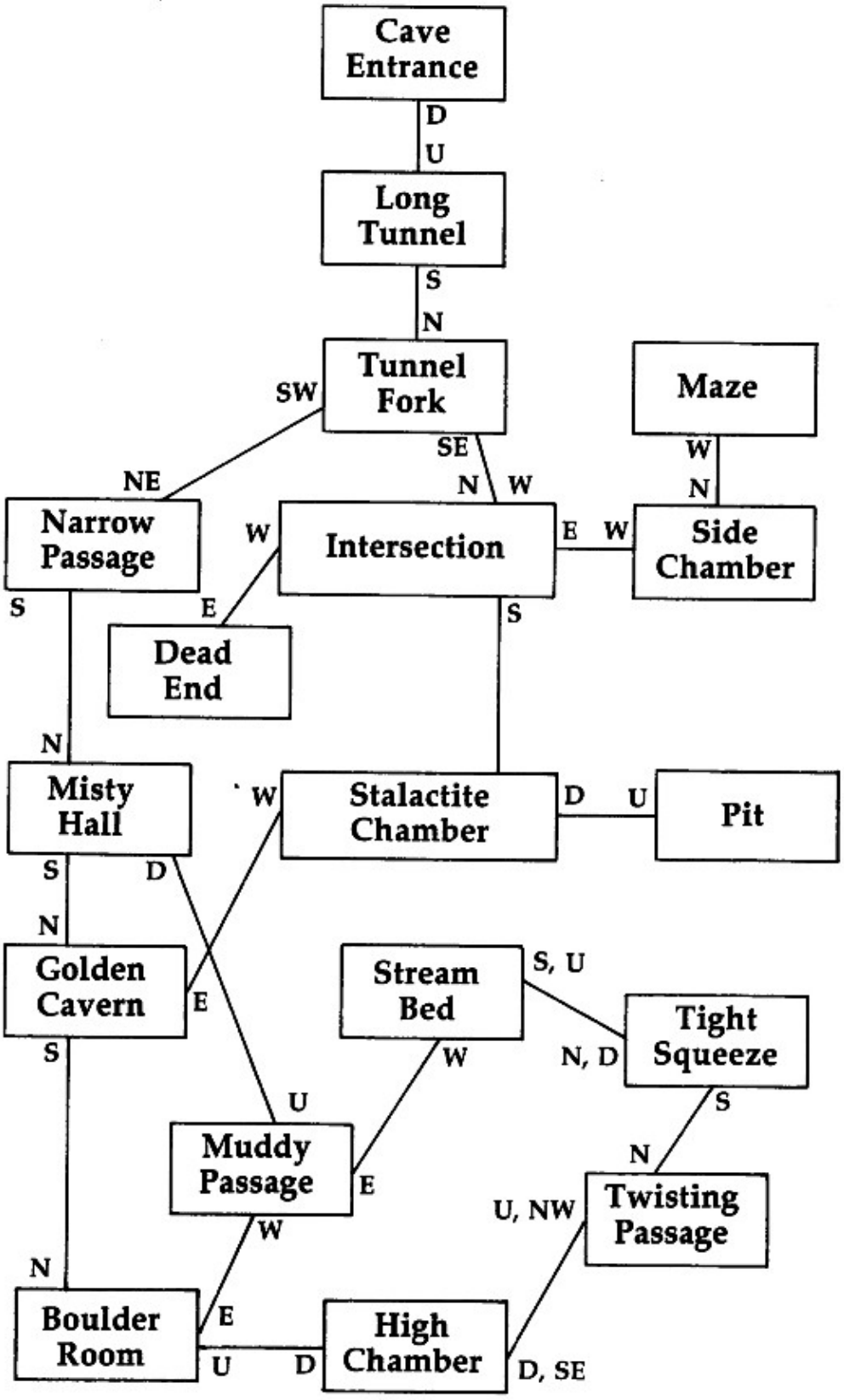
To achieve those successes, you have to learn method and strategy. To a large extent, expertise at adventure gaming comes only with experience. But if you take the right approach, that experience can be much more profitable.

Take these guidelines as a starting point, not as rigid rules. Your goal should be to develop a way of thinking that lets you solve problems easily, not to learn a set procedure. But if you come across an especially tough problem, the best approach may be to go back to the guidelines given here (and any that you may have added yourself) and to consider each as a possible approach to a solution.

Draw a map. Some people consider it a point of honor to go through an adventure without writing anything down, but unless you have a photographic memory or a masochistic streak, it makes more sense to map out the *rooms* of the adventure.

In most adventures, the rooms are found on multiple levels and are connected by passages that twist in unexpected ways. So it's usually impractical to make a geographically realistic map during the course of play. All that you really need is a *topologically* correct map—that is, one that indicates the correct connections between rooms. The simplest method is to represent each room with a box and to indicate the passages with connecting lines labeled with the appropriate direction or movement command. Label each box with a short name that matches the room's description. Figure 8-1 shows part of a map from a typical adventure.

Figure 8-1. A Typical Adventure Map



After you've wandered through the labyrinth for a while, you'll probably find your map getting messy, with lines going over long distances as you discover unexpected connections between rooms. You may want to stop at that point and make a clean copy of the map. If you're geographically inclined, you may want to lay it out so that the rooms are positioned correctly relative to each other. Don't count on the author to be completely consistent about geography, though.

Map mazes by dropping objects. A feature found in many adventures is a "maze of twisty passages" in which every room has the same description and the connections follow no logical pattern at all. Going east and going east again may take you back to the room you started from; going south might not take you back to the room you left by going north. If you just blunder around, you can be lost forever.

The way to handle a maze is to drop objects in the rooms, making them distinguishable. This requires some cleverness, since there are usually more rooms in the maze than objects in your inventory. (And please, don't drop your lamp!) Get some little scraps of paper and mark each one with the name of one of the objects you can drop. When you enter a new room in the maze, make an entry for it on the map and put the corresponding scrap on that map entry.

Try moving in every possible direction from each room. Mark the connections between rooms in the usual way; also mark any directions for which you get a "you can't go that way" response. Once you've run out of objects to drop, keep trying to go back to the rooms that have objects in them and learn where other directions from those rooms lead.

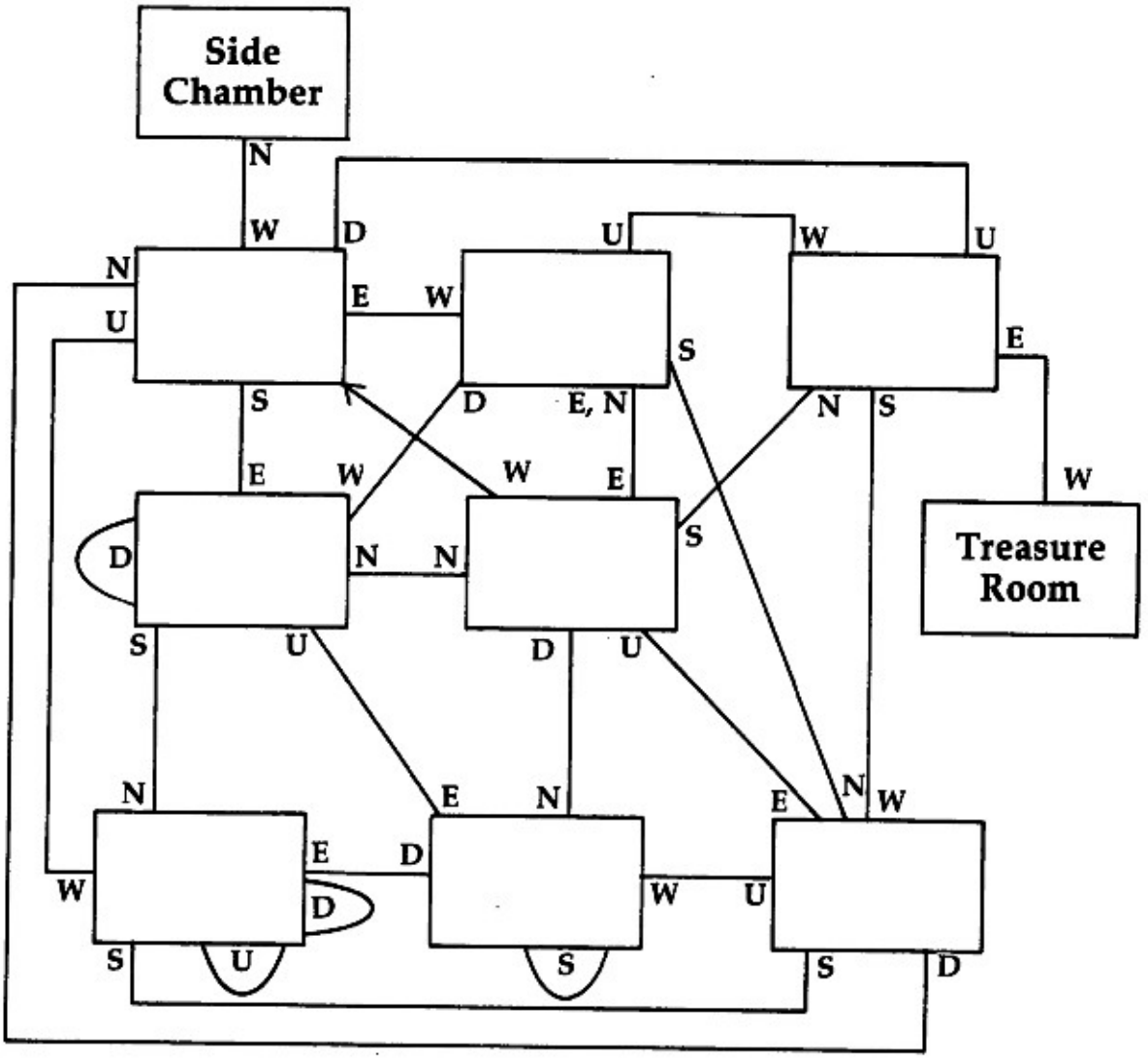
Once you've got these rooms mostly mapped out, you can pick up some of the objects and move them to other rooms. The map you've already made will usually be able to guide you around the rooms you had previously mapped, and you'll be able to add the new rooms.

After a while you should go over your map and see if any pairs of rooms have suspiciously similar connections. Since you can't drop objects everywhere at once, it may turn out that what you thought were two different rooms were really the same room. Comparing the dead-end directions from different rooms is also very helpful; if, in each of two rooms, you can't go north, south, or northeast, it's quite possible that they're really the same room.

Some adventures add complications to their mazes. In *Zork I*, for instance, the Thief will pick up objects you've dropped and later drop the worthless ones somewhere else. But the same technique still applies. The only difference is that you have to be willing to throw information out if it looks misleading.

Figure 8-2 shows a full map of a hypothetical maze. Six directions are allowed: north, south, east, west, up, and down. As is typical for mazes, the compass direction is no help for relating room positions, and some passages double back to the room they came from. This maze is entered by going north from the Side Chamber.

Figure 8-2. A Map of a Hypothetical Maze



Let's assume you have four objects that you can afford to drop: an apple, a banana, a cherry, and a doughnut. Figure 8-3 shows the first steps to take in mapping the maze. Each time you enter a new room of the maze, drop an object there and mark it in on your map.

When you run out of objects to drop, try to go back to the rooms where you dropped objects and concentrate on filling in the connections for these rooms (Figure 8-4). When you do this, you'll probably map more rooms than there actually are, since you won't know that the room entered by going east from "Apple" is the same as the room reached by going west from "Banana." Keep an eye out for such duplications, and be ready to correct your map when you find one. A good eraser is important when mapping mazes.

After you've gotten a fair number of connections mapped, pick up an object from a room for which you've discovered most of the connections and move it to a room without an object. In Figure 8-5, you've moved the banana to a different room. This will help you to establish the connections from that room; it will also quickly show if you've put that room down more than once on your map.

By repeating this process several times, you can get the maze fully mapped. While it isn't necessary to get every last connection mapped, you should at least make sure you haven't missed any exits into interesting rooms.

Figure 8-3. Maze Mapping, Step One

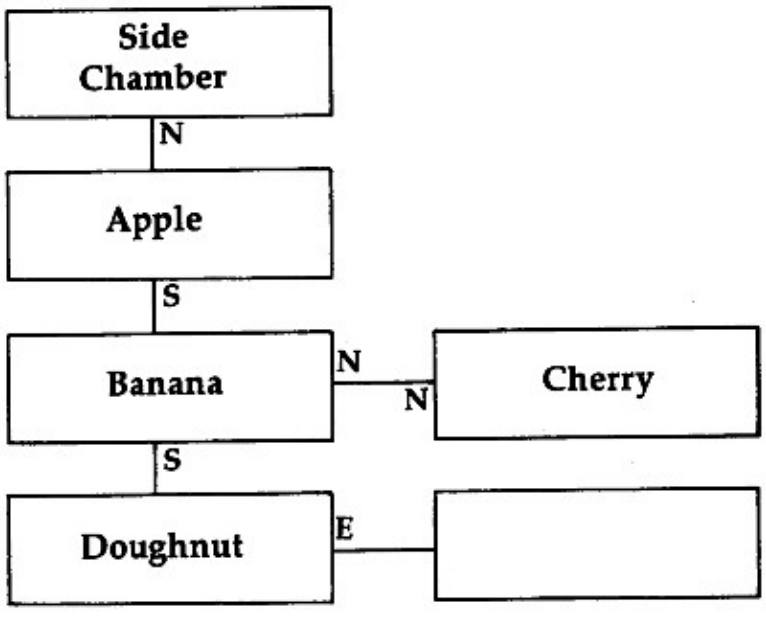


Figure 8-4. Maze Mapping, Step Two

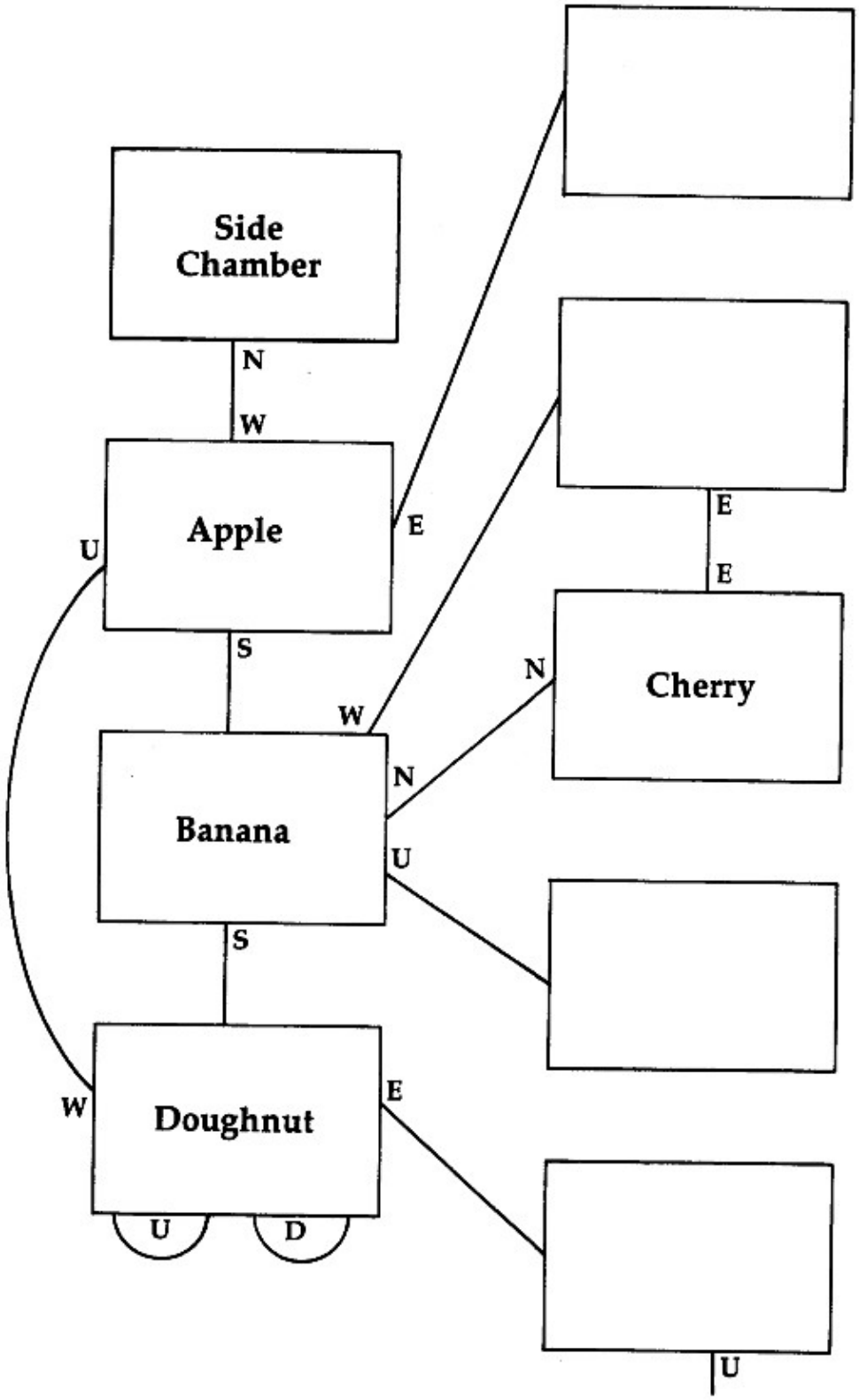
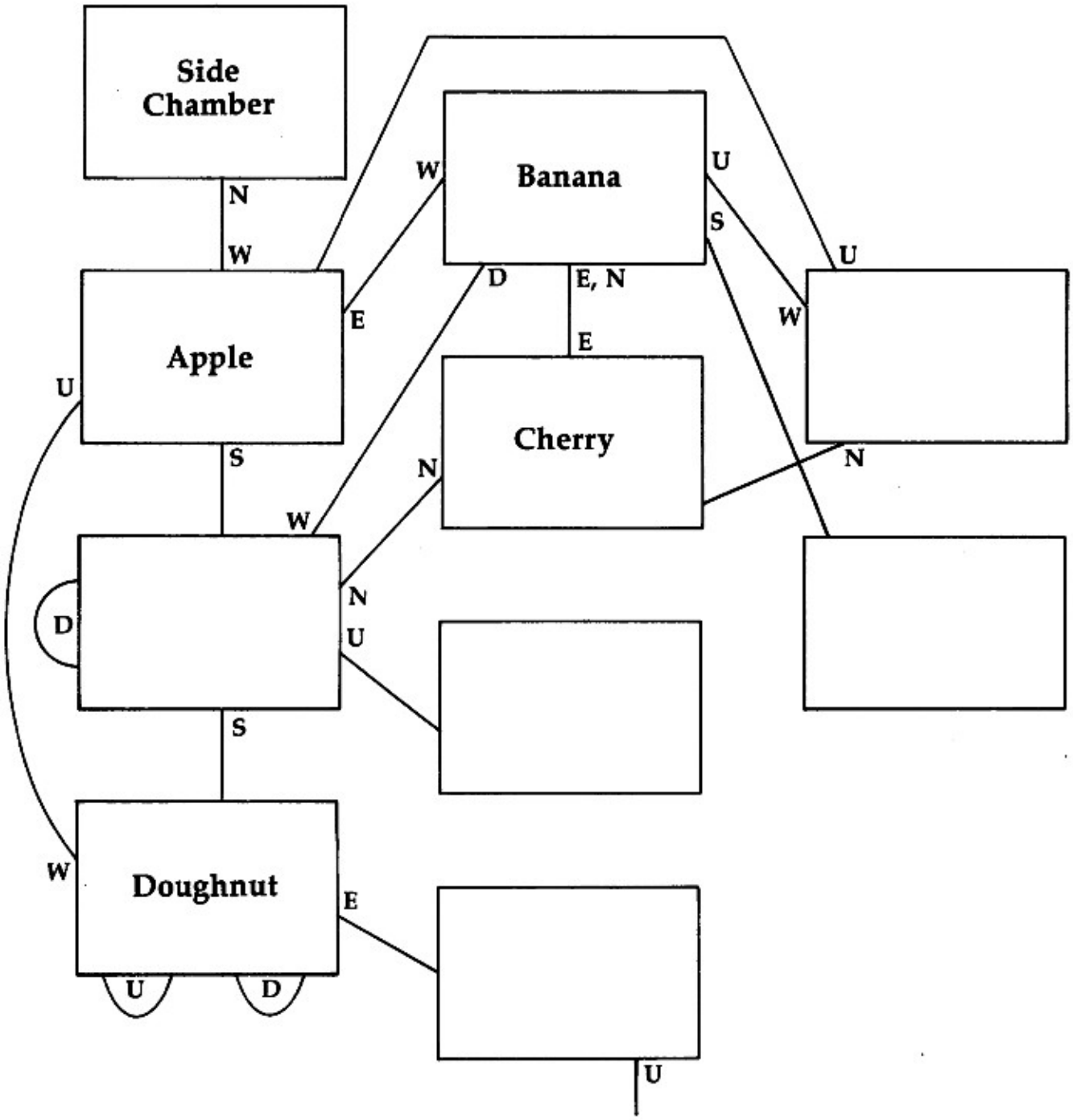


Figure 8-5. Maze Mapping, Step Three



Go everywhere. Missing a room that you need is one of the major causes of frustration for adventurers. If you think you've gone everywhere, double-check all rooms for exits that you might not have tried. Room descriptions are often devious and don't tell you outright where all the exits are. Watch for any peculiarities in the description that might indicate a hidden path.

Don't forget to try the diagonal directions if the adventure includes them. If a room is described as having "exits in all directions," try going northeast, northwest, southeast, and southwest. Don't forget, too, that you may be able to walk through certain walls; the worst you can get out of trying something like this is a bruised nose!

For example:

```
>EXAMINE WALLS
    The southern wall seems to shimmer.
>SOUTH
    There is a wall there.
>WALK INTO WALL
    The wall turns out to be insubstantial, and you easily pass
    through it.
```

Examine everything. The EXAMINE command is one of the most important commands in many adventures. Sometimes this command is given as SEARCH or (in simple adventures) as LOOK. If this command is available, examine *every* object you come across. Most of the time you'll get a noncommittal message, such as "I see nothing special about it." But in a few cases, you'll get information that is vital to solving the adventure.

The information you get may seem obvious. But if you get anything distinctive in response to EXAMINE, there's a good chance you should pay attention to it. Suppose you examine a statue and get back the response, "It's covered with half an inch of dust." No big deal? Perhaps, but it may turn out that when you DUST the statue or do something similar, an engraved message will become visible, or some other exciting event will occur.

Watch out for objects that seem to be there just for atmosphere. They may be important to you. Suppose the room is described as being "full of debris"; EXAMINE the debris just in case there's something important in it. Maybe there's a wandering junk collector around who'd like something from

the room, or maybe something valuable is buried amidst the junk. Other verbs that may give you information about an object include READ, TOUCH, and FEEL. Try them whenever they seem applicable. It may save your life.

For example:

>EXAMINE BOOK
The title of the book is "Great Magic Spells."
>READ BOOK
The book seems to be a cookbook for preparing magic spells, but it's too technical for you.

These responses might lead you to guess that the magic book would interest a wizard (and that you might be able to trade it for assistance later on). Maybe, with some help, you could even use the spells yourself.

Think of possible uses for objects. Any object that you can refer to by name stands a good chance of being useful, even if it isn't valuable (worth points) in itself. It might be a tool for doing a necessary job. It might be a source of light if you can figure out how to turn it on or ignite it. It might have something important hidden inside it.

If something *is* valuable, don't assume it's only good for points. It may turn out to be useful in itself, or you may have to trade it or even let it be stolen in order to achieve some goal. Be especially wary if an object seems valuable but doesn't add any points to your score.

For example:

>EXAMINE ROOM
There is a door at the end of the room, next to a shelf.
>EXAMINE SHELF
There is a diamond on the shelf.
>EXAMINE DIAMOND
The diamond is expertly cut, with a prominent knife edge.
>EXAMINE DOOR
The door is held shut by a steel plate welded to it.
>CUT PLATE WITH DIAMOND
With great effort, you manage to cut the plate in half.

Watch for interactions between objects. Pairs of objects often go together or conflict in interesting ways. A certain key may be needed to open a particular lock, or a special chip may fit into a socket. In cases like this, watch for a key that has the same color or material as the lock or for some other clue indicating that the two are a pair.

Pairs of objects can also conflict subtly or fatally. A bull might react nastily if you carry a red cape into his presence. Similarly, picking up a magnet and a reel of computer tape at the same time could be a bad idea.

Figure out what you need to solve a problem. Figuring out what a tool is good for is a useful approach. But equally helpful is working from the other end—looking at a situation and trying to decide what tool would be useful in it. *Tools* can cover a lot of ground; the object you need might be a weapon to fight an enemy, a gift to gain someone's cooperation, a distraction to throw in a creature's way, a vehicle to help you get to your destination, or an actual tool to perform some mechanical operation.

The description of a situation will often give a hint about what you need. If it doesn't, then trying to solve the problem with the wrong tool (or none at all) may elicit a message that will suggest the kind of tool you need.

>EXAMINE DOLL
The doll looks like a wind-up toy. It has a keyhole, but no key.
>EXAMINE KEYHOLE
The keyhole in the doll is about an eighth of an inch in diameter and one inch deep.

What will happen when you wind up the doll? You do want to find out, don't you? It may turn out that the object you need isn't actually a key, but something else that will fit the keyhole. If you come across a nail, a pen, or anything else that might do the job, give it a try. Trying the wrong object might give a further indication of what you need.

>WIND DOLL WITH PEN
The pen is too wide to fit in the keyhole.
>WIND DOLL WITH NAIL
The nail works! The doll walks across the room, indicating a spot on the wall.

Read descriptions carefully. Information is often intentionally buried in a long description. Read all the information that is given and pay special attention to anything that seems odd. Trying to visualize the situation is often a great help. If graphics are included, examine the pictures for items that the text might not have mentioned.

>EXAMINE ROOM
You are in an office of the Pottsylvanian Embassy. As you

would expect of Pottsylvanian diplomats, the place is a general mess. Several chairs are scattered about, and a large desk is located by the east wall. This desk has an abundance of junk on it, including an unwashed coffee cup, an ink-stained blotter, and several loose papers. The floor is scratched and marred and has obviously not been swept in several weeks .

>READ PAPERS

The papers are mostly unreadable from coffee stains, but on one you can discern the message, "Boris will leave the keys in Box 552."

In this case, the detail to watch for was the paper on the desk, and all the clutter was just a distraction.

Make use of negative responses. Many times, an adventure won't let you do something just because the author didn't consider it or didn't have room in the program to handle it. This can lead to immense frustration when you try something reasonable and the program keeps declaring, "You can't do that." Still, finding out that you can't do something can be a source of positive information. If you get a special negative response, rather than a standard one, such as "Don't be silly," then you should pay closer attention to it. The message may call attention to something else that will work.

>EXAMINE GARDEN

You are in a garden behind the house.
There is a delicate flower growing here.

>PICK FLOWER

The flower resists your attempt to pull it out of the ground.

Maybe the author of the adventure just didn't want to be bothered with moving the flower around. But the fact that the message is so specific suggests that you should look further. Maybe you have to cut the flower, in which case you might need a special tool for the job. Or maybe the flower would be a good thing to hang on to in a hurricane. It's an anomaly, so keep it in mind.

Make use of the SAVE feature, if present. The better adventure programs provide a command to let you save the state of the game on disk or tape. This has the obvious use of letting you leave the computer and continue later. In addition, it can provide you with a safety belt before undertaking a risky action. If the action you try blows your character to smithereens, it's much more satisfying just to be able to restore the game where you were than it is to have to go

through all the actions again to get to the same point.

Using the SAVE command is especially important if you have to take an action that you know may not work. In Scott Adams's *The Count,* for instance, you have to take a certain risky action several times during the adventure. Unless you save the game before each attempt, you may go crazy trying to get all the way through without breaking your neck.

Using SAVE with tape-based adventures is tedious, especially since you should really erase the tape before reusing it. Still, it's better to spend five minutes saving the game than to have to spend half an hour repeating all your moves.

It's also a good idea to save the game periodically just on general principles. In the course of the game, it's possible that your character will get killed unexpectedly, or that you'll take an action that irreversibly blocks your further progress. When that happens, you'll probably want to go back to a recent point rather than starting all over again.

Don't let SAVE become a crutch. It can be tempting to save the game and then repeatedly try an action with a very low probability of success until it works. But if you have to do that, you almost certainly haven't discovered the right way to solve the problem. SAVE allows you to return to life and correct your errors. But don't abuse the privilege.

```
>EXAMINE BOTTLE
     The bottle contains a clear fluid and is labeled with the words
     DRINK ME.
>SAVE
[Whir, whir, whir....] File saved.
DRINK FLUID
     You drink the fluid and die in horrible agony.
```

Since you had the presence of mind to save the game, you can restart the adventure at the point where you were examining the bottle. Your character, motivated by an acute sense of danger, will then realize that he probably shouldn't drink what's in the bottle.

If you're stumped on one problem, switch to another. Switching to a different task for a while is good problem-solving psychology. Besides, you may need to do something else before it's possible to solve the problem. In most adventures, there is a more or less fixed order in which you have to solve the problems. Getting the solution in one situation may provide you with the means to handle the next one.

Answer rhetorical questions. A popular (if somewhat dirty) trick in the original *Adventure* consists of responding to a command with a seemingly sarcastic question—which, if answered in the affirmative, lets you do something quite unexpected! Several other adventures have copied this trick. So if the program asks you whether you really intend to do something, try answering YES. It may work. (But be warned: In one expanded version of *Adventure*, the program asks the same rhetorical question at another point in the game—with disastrous consequences if you answer YES. But those are the risks an adventurer takes.)

There is a six-foot chasm to the north. Flames shoot up from below.
>JUMP OVER CHASM
Are you really going to jump over a flaming chasm?
>YES
You made it! You are now on the north side of a flaming chasm.

Experiment. Don't be afraid to try something dangerous. If the game has a SAVE feature, then save the game so that you can come back and take a different approach if you don't make it. An unknown liquid in a bottle might be poisonous, but it might also give you a magical power that you need to finish your quest. You'll never know unless you try.

Even trying something that seems ridiculous might have consequences. In two of the problems in *Zork I*, I found myself reduced to desperation, so I just started trying every word I could think of that had any bearing on the problem. In both cases, I was pleasantly surprised when I finally got the semi-magical word that solved the problem.

Just trying to do different things to objects will give you an idea of how much the program will let you do. If it allows lots of options, you may need to know about all of them.

The eastern wall of the room has a large red heart painted on it.
>KICK HEART
Why would anyone kick a heart?
>WHY NOT?
I don't understand that.
>PUSH HEART
Pushing the heart doesn't do any good.
>PUSH WALL
Pushing the wall doesn't do any good.

>BREAK HEART
Heartbroken, the wall collapses into a shower of tears.
>GROAN
I don't understand that.

Make full use of the command handler's capabilities. Whenever you're playing an adventure that allows more than two-word commands, try different combinations of words to see just how much it will let you do. The instructions with the adventure will often give examples of the kinds of commands you can give. Try giving nonsensical commands to see how the program reacts.

You are at the top of a mountain. A dwarf is sitting on a wooden chest.
>INVENTORY
You are carrying:
A lamp
A sack of coins
A lump of coal
>HELLO, DWARF
The dwarf ignores your greeting.
>GIVE COINS TO DWARF
The dwarf doesn't want the sack of coins.
>TAKE COINS FROM SACK
You can't take a sack of coins from a sack of coins.
>FEED DWARF
What do you want to feed the dwarf to?
>LUMP OF COAL
You can't feed a dwarf to a lump of coal.

You haven't made much progress with the dwarf, but you've learned that the program accepts indirect objects, that it lets you complete commands, and that it treats the sack of coins as a single object rather than a container with an object inside. Getting familiar with what the program can do will make it much easier to give the right commands later on.

Don't overestimate the program's abilities. The other side of knowing what the program can do is avoiding the mistake of thinking the program is "smarter" than it is. Some adventures will ignore extra words at the end of a command. If you think the extra words are doing something, you may get a false impression that will leave you confused for a long time.

You are in a room with doors on the north and south.
>LISTEN AT DOOR
You hear nothing

A naive player might decide from this that listening at doors can provide a warning of danger in some cases. If you follow this theory, though, you may just be wasting a lot of effort, since the program might have taken just the word LISTEN as a command and ignored everything that followed it. Try this to see if it's really paying attention to the whole command:

>LISTEN ZXCV
You hear nothing.

Now you know that for the LISTEN command, the program may ignore extra words. It's still possible that the program accepts the words AT DOOR but ignores nonsense; more trial and error will be necessary to get a good idea of how it handles your input. Check out other commands as well to see if their objects make any difference.

If a command doesn't work, try rephrasing it. This is especially important with adventures that handle complicated commands, since they allow more possibilities. If a command doesn't get a useful response, you may still be doing the right thing; it's just a matter of stating what you want to do on the program's terms.

In the northern wall of this cavern is a narrow hole, through which you can see a small sack tied with a cord.
>TAKE SACK
The hole is too narrow.
>INVENTORY
You are carrying:
A dagger
A lamp
>TAKE SACK WITH DAGGER
That doesn't work.
>HOOK SACK WITH DAGGER
You can't hook a sack.
>HOOK CORD WITH DAGGER
You catch the cord on the dagger and carefully lift the sack out.

Think of literary allusions. Many situations in adventures are drawn from mythology or from famous stories. If you notice a situation that's similar to a story you know, think of how something that happened in the story might apply. It may even turn out that a key word from the story is all that you need to make something happen.

This room contains a bust of Pallas on which a raven is sitting.
>TAKE BUST
The raven snaps at you with its beak, and you pull your hand back.
>NEVERMORE
On hearing that ill-fated word, the raven takes to the air and leaves the room.

Get help. Although it's most satisfying to completely solve an adventure without anyone's help, there may come a time when you find yourself stuck. If this happens, you may want to get a hint from a friend who's played the adventure so that you can get through the difficulty and be able to do the rest of the adventure.

If you can get your friend to give you a hint rather than telling you outright what to do, then you can still have at least part of the pleasure of solving the problem. There is an art to devising subtle hints. Here are a few examples, based on earlier situations in this chapter:

Q. I'm trapped in the shimmering room! Is there any way out?
A. Certainly. Just leave!

Q. Is there any way to get through the door with the steel plate?
A. I could come back with a cutting answer, but what you need is hard to get.

Q. I'm getting nowhere rifling the Pottsylvanian Embassy. Is there any way to get into the records room?
A. Don't believe everything that you read, but do read everything that you find.

Other than people you know personally, one of the best sources of help in adventure games can be found on the CompuServe Information Service. One of the many bulletin boards within the system belongs to the Game SIG (Special Interest Group), which has a large population of adventurers. Dropping off a message asking for *specific* help in an adventure will usually get you one or more useful responses, often in the form of cryptic hints like those given here.

In addition, the Game SIG maintains a data base of reviews, hints, and even complete walk-throughs of a number of adventures. The hints (which are often *very* cryptic) are good for getting a gentle push in the right direction; the walk-throughs are a last resort when you've run out of patience.

Finally, the adventure publishers themselves often make hints available. Sometimes these are packaged with the game; in other cases, you must buy them separately. Infocom's "InvisiClues" are especially elaborate hint packages, delivered with a special marker required to make the clues visible.

But remember that you need never despair. Try all the methods listed here, and add a few of your own to the list. If all of them fail, then set the game aside and try later—or ask for help without feeling ashamed, as long as you've given it your best effort. Remember, very few adventures are intended to be solved in a night.

How They Work

How They Work

This chapter discusses the way adventure programs actually do their work. The discussion here will cover practically everything that could go into a full-blown professional adventure. But don't be intimidated by it. Writing something in BASIC that handles a reasonable number of commands and provides treasure-hunting fun isn't nearly as onerous as writing the equivalent of *Zork*.

But the same basic ideas are involved. If you go through this chapter to get an idea of what's involved in what the pros write, you'll get a feel for how to scale the concepts down to a programming project of a few weekends. Chapter 10 deals specifically with that sort of project, but getting a broader picture first is always helpful.

All adventure programs, simple or complex, have certain basic features. A program need not be formally broken down according to this list of features, nor do the names that I am using here represent an industry standard. This breakdown is simply intended to show what functions the program must accomplish. These features are as follows:

- The *command parser* interprets the player's commands.
- *Output routines* tell the player what is going on.
- *Action routines* cause things to happen in the game world as a direct result of the player's commands.
- *Automatic routines* cause things to happen because of the passage of time or because of some event independent of the player's actions.

These pieces of the program depend on an assortment of data structures. The major ones are listed below:

- *Vocabulary lists* give the words in the program's vocabulary, grouped according to parts of speech.
- The *internal map* itemizes locations that your character can reach and the paths that allow access from one location to another.
- *Object descriptors* itemize various tools, treasures, and other objects available to you. The descriptors indicate where the objects are on the internal map, and they specify any distinc-

- The *character descriptor* holds information about your character (such as location, inventory, and state of health).

The Parser

Any program that accepts free-format input has to have a command parser. To *parse* a command means to break it down into its word components and identify their syntactic relationship.

Before going into the details of the parser, it's helpful to review a little English grammar. A sentence has two parts: a subject and a predicate. In a simple sentence like "I go," *I* is the subject and *go* is the predicate. In an imperative sentence, which expresses a command, the subject is omitted. The sentence "Go!" is made up only of a predicate.

Not surprisingly, commands in an adventure program are imperative sentences. This allows us to greatly simplify grammatical structure and means that the program has to analyze only the predicate.

For the purposes of a game, the predicate can be considered to have three parts: the verb, the direct object, and the indirect object (or prepositional phrase). Take the sentence, "Give me your gun." *Give* is the verb; *gun* is the direct object (the thing being given), and *me* is the indirect object (the recipient of the action).

You can also express the indirect object as a prepositional phrase; the preceding example could have been expressed as "Give your gun to me." The preposition *to* relates *me* to *give.*

An object (direct or indirect) or a prepositional phrase consists of a noun, possibly modified by one or more adjectives. An adjective describes the thing named by the noun, and—what is important for adventures—it can specify which object you are talking about. If you say, "Give me the smoking gun," then you are referring to one gun in particular (and not the cold gun that might also be nearby).

There can be more than one object. Multiple objects can be listed explicitly, one at a time; for instance, "Take the bell, the book, and the candle." The word *all* or *every* can imply multiple objects, as in "Take all the gems" or "Take everything."

Just as a noun can be modified by adjectives, a verb can be modified by adverbs. If you're afraid of being overheard,

you might say (or whisper), "Give me the gun quietly." In that case, *quietly* is the adverb modifying *give.*

Beyond this basic structure, the only other grammatical feature that an adventurer might encounter is a noun of address. This is just a noun added at the front to indicate whom you're talking to; for instance, "Robot, fix the nuclear reactor."

This is enough grammar to get us where we want to go. You won't be needing strange tenses, and you can ignore such niceties as subordinate clauses. You don't have to care whether there's a way to cleanly split an infinitive, or whether it's bad to end a sentence with a preposition. If a parser can handle the grammar described so far, it can do a very impressive job.

A parser operating on the command BURN RED BOOK, for example, would do the work necessary to determine that BURN is a verb, BOOK is the object of the verb, and RED is an adjective modifying BOOK. It is also responsible for recognizing meaningless input so that the program can complain to the player.

The parser takes the words of the command and converts them to units of internal storage called *tokens.* The tokens are passed to the action dispatcher, which calls appropriate action routines, depending on what tokens it sees. The parser might produce fewer tokens than the number of words it sees, since it can throw words out or combine two words (for instance, PICK UP) into one token.

The command parser is one of the most important parts of an adventure program, since you can get frustrated very quickly if you can't tell the program what you want to do. The power of a command parser depends upon the size of its vocabulary, the number of letters it uses per word, and the complexity of the syntax it can handle.

The simplest sort of command parser has a list of verbs and a list of objects in its vocabulary. It simply checks to see if the first word is a verb and the second word is an object, converts them to token form, and then calls the appropriate action routine. In fact, some programs dispense with syntax checking altogether; in *Pyramid 2000,* for example, the verb and the object can occur anywhere in the command, together with any number of noise words. Thus, you can say TAKE BOX or TAKE THE BOX or BOX THE TAKE with the same results.

Programs recognize a certain number of letters per word

and usually just disregard the rest. Thus, if a particular program recognizes five-letter words, then INVENTORY, INVEN, and INVENTION will all be equivalent. Truncating the words used keeps the size of the program down, since a smaller number of bytes is necessary for every word in the vocabulary. But if words are too short—and those with as few as three letters are not uncommon—the parser can misinterpret them, often with amusing or frustrating results. For instance, you could drown if the program thought you were trying to SWing an object when you were really trying to SWim.

Beyond the simple verb-object parser, it's possible to add complexity a little at a time. A simple enhancement, found in recent versions of *Adventure*, is using multiple objects with certain verbs. This is particularly useful with the TAKE and DROP verbs, since you will often want to pick up or drop several items at a time. This is just a matter of the parser recognizing the conjunction *and* and expecting the word following it to be another object, then constructing a list of tokens to match the objects.

Other additions in recent versions of *Adventure* are the uses of prepositional phrases and indirect objects. These allow you to specify the object you want to use for an action (FIGHT DEMON WITH PITCHFORK) or what your action will affect (GIVE HAY TO HORSE).

The use of indirect objects and prepositional phrases begins to involve some sophisticated command handling, since prepositions are among the trickiest elements of the language. Consider the commands FEED SEED TO BIRD, FEED BIRD WITH SEED, and FEED BIRD SEED. In the first two cases, the parser has to recognize that the first noun is the direct object and that the second one (the one after the preposition) is the indirect object. It also has to record the preposition; otherwise the action routines won't be able to tell whether you want the bird to eat the seed or the seed to devour the bird.

For the third case, the parser has to use the rule that when a verb is followed by two nouns, the first one is the indirect object and the second one is the direct object. It may also create a token for WITH as the implied preposition.

Even trickier are the words that go with the verb to change its meaning. In these cases, it makes sense to treat the verb and the extra word as a single verb: for instance, KNOCK

preposition and give the action dispatcher a single verb to work with. WAKE UP becomes the same as WAKE; PUT AWAY can be translated to the equivalent verb STORE. The parser has to be careful of word order, since PUT FUR COAT ON and PUT ON FUR COAT both mean WEAR FUR COAT, but PUT FUR ON COAT means something else entirely (although I'm not sure what).

Adjectives are a little more straightforward, since they normally precede the noun they modify. They can be strung together (MOVE BIG UGLY RED BOX). But adjectives are useful only if nouns by themselves are ambiguous. If there is only one box in the game, adjectives are unnecessary. But if there are two boxes with different attributes, then the program must determine whether BOX alone is sufficient. There might not be any need for adjectives, since only one of the boxes might be at your current location; but if both boxes are present, then an adjective is required. The command parser can't determine by itself whether the adjective is required; it's the job of the action routines to determine if the information was specific enough.

Adverbs can be positioned at various places in a sentence. If the adverb can be identified, it isn't too much of a problem since it can be assumed to modify the verb. Adverbs modifying adjectives (MODERATELY HEAVY) would add a level of subtlety that isn't likely to be needed. But some common adverbs look just like prepositions. This leads to such problems as distinguishing between PUT SUIT ON and PUT SUIT ON TABLE.

In general, the more parts of speech that the parser recognizes, the more vulnerable it is to ambiguities. *Zork III*, for instance, includes a place mat among its objects. If you say PLACE MAT ON TABLE, the program will complain that the command doesn't contain a verb, since it regards PLACE as an adjective.

An example might help to bring all these points together. Consider a relatively complicated command, such as GIVE THE LITTLE PIG TO THE BIG BAD WOLF QUICKLY. Let's see how a parser might handle this command.

Assume that the parser doesn't allow anything to precede the verb. This means that the first thing in the line has to be the verb, or the command is invalid. The first step is easy,

Next, THE is just a noise word that the parser lets you throw in to make your command look more like real English. It just skips over it.

LITTLE is an adjective, so it has to modify a subsequent noun. When PIG is found, the parser takes it as an object of GIVE, with LITTLE modifying PIG. At this point, the parser can't tell whether PIG is the direct or indirect object of GIVE; if the sentence were GIVE THE LITTLE PIG SOME FOOD, PIG would be the indirect object.

TO is a preposition, so what follows has to be the indirect object or a prepositional phrase. Therefore, PIG can now be identified as the direct object.

Throwing out another THE, the parser then finds the adjectives BIG and BAD, followed by the noun WOLF. So WOLF is the indirect object of GIVE, with BIG and BAD being its modifying adjectives, and TO being the preposition that relates the indirect object to the verb.

Finally, QUICKLY is an adverb, and as such all it can do is modify the verb. So the parser will create tokens for the following elements:

- The verb (GIVE), modified by an adverb (QUICKLY).
- The object (PIG), modified by one adjective (LITTLE).
- The indirect object (WOLF), related to the verb by a preposition (TO) and modified by two adjectives (BIG and BAD).

A common way to express the grammar of programming languages is using the Backus Normal Form (BNF). Figure 9-1 shows how the grammar of an adventure could be expressed in BNF. For those not familiar with the notation, a simple example should help to give the idea. Take the definition:

basic-command ::= predicate object [adverb]

This line defines what a basic-command is. It tells us that a *command* consists of a *predicate* followed by an *object*, optionally followed by an *adverb*. The brackets indicate that an item is optional. An item in braces, such as {adjective}, can be repeated any number of times or not used at all. A vertical bar (|) is used to separate multiple alternative definitions of a term.

A strict BNF definition goes all the way down to defining names in terms of their individual letters. The one used here doesn't involve that much painstaking detail; completely defining the language would be contrary to the spirit of

Figure 9-1. Expressing Adventure Game Grammar in BNF Form

```
command ::= basic-command [adverb]
basic-command ::= predicate object |
                  predicate indirect-object object |
                  predicate object preposition indirect-object
predicate ::= verb | verb short-adverb
indirect-object ::= object
object ::= {adjective} noun | object AND {adjective} noun
```

should be self-evident; for instance, the term *short-adverb* indicates such words as UP, DOWN, AWAY, and so on, which are used to change the meaning of a verb.

By simplifying or omitting definitions, you can create a system of grammar for simpler adventure programs. The simplest grammar, and the most common, needs just one definition:

command ::= verb noun

This grammatical framework also makes further enhancements easy to manage. For instance, allowing multiple commands separated by the word THEN can be expressed by adding the definition:

compound-command::= command {THEN command}

Putting a message on a terminal or screen is basically a simple matter, but it is complicated by a couple of things. First, it would be nice to keep players from cheating by dumping the program to a printer and looking for any obvious text strings. Second, computer storage has to be used economically, so it would be helpful to find a storage method that does better than using one byte per character.

In order to prevent cheating, the program may encrypt the output messages. There are many ways known to do this that require little overhead. It may be possible to break the encryption scheme, but if you're going to go to that much work, you might as well solve the adventure legitimately.

Reducing storage requirements is a more complicated matter. One way to do this is to pack three characters instead of two into two 8-bit bytes. The drawback to this method is that it allows only 40 different character codes to be stored (compared with 256, if 8 bits per character are used). This doesn't

Another way to compact text is to take advantage of the fact that only 128 of the 256 possible character codes in a byte are defined in the ASCII character set. The remaining 128 codes can be used to represent common words or groups of letters. These groups must be stored somewhere in the program so that it can translate the codes, but they need to be stored only once rather than every time a message needs them. This method can, of course, be combined with encryption.

Timing can also be a factor to consider in output routines. Some adventures—for instance, Adventure International's *Strange Odyssey*—make good use of programmed delays to add suspense to an event or to call your attention to the fact that something has slowed down.

Describing a World

Before seeing how an adventure program makes its simulated world respond to your commands, it's worth taking a look at the data structures that describe that world. The main parts of the description are the internal map, object descriptors, and character descriptor.

The main purposes of the internal map are to put an identifying tag on each location and to keep track of the connections among them. These locations are generally called *rooms* even though they may not be constructed or even enclosed. An obvious way to store the connections is to have an X by Y array of room numbers, with X being the number of rooms and Y being the number of different motion commands (e.g., NORTH, UP, etc.). Then, CONN(20,3) would contain the number of the room reached by going in direction 3 from room 20.

For every room there is a text description and possibly a graphic image to identify the room for you. Often there are two descriptions: a long one that is given when you first enter a room or when you type LOOK, and a short one when you return to a room you've already visited.

Each room may have some internal descriptive information associated with it. Many adventures characterize rooms by whether or not they are illuminated; the ones that aren't won't be described to you (and may even be fatal!) unless you bring your own light source. The room may also be under water (in which case you'd better not stay too long) or accessible

whether to give a long or a short description, it has to remember if each room has been visited yet or not.

Specific exits from a room may also have their own characteristics. A passage may be too narrow to let you carry a bulky object through. You may not be able to leave unless a door has been unlocked and opened. Using the exit may trigger a trap unless you've taken precautions.

The object descriptors indicate where an object is and what its characteristics are. The most important characteristic of an object is whether it is fixed or movable. If the object is fixed, no power on earth can change its location; this fact can be convenient for letting the program do certain things with it. Saying that an object is movable, though, doesn't necessarily mean that you can just pick it up and carry it off. The *way* to move something may not be at all obvious.

Doors, gates, and the like are an interesting class of fixed objects, since they are located between two locations rather than in one of them. If you open, close, or lock a door, the change in its state must be reported in the descriptions of both rooms.

Objects may act as containers that hold other objects or as surfaces that support them. This relationship can be carried out on multiple levels; a table might support a box, which contains a bag, which contains a book and a bookmark. Realistically, a container has a limited capacity, so the program must either consider the sizes of objects or else have a list of specific objects that can go in a container. The number of objects that a container can hold is also limited. Finally, two objects can't contain or support each other.

Some objects present special problems because they aren't in just one place at a time. Water is the most common example. A cave complex might contain a pool, a lake, and a stream, all of which contain water. But if you type TAKE WATER in one of these places, it won't work for the program to check to see if you're in a room that contains the object WATER. Instead, it has to determine if the room contains a water source and, if it does, create the specific object WATER (presumably in a container that you have handy). If you later type POUR WATER, you usually can't get it back into the bottle—so the specific object WATER has to be destroyed until the next time you get it from a water source.

adventure. For instance, color, size, and so on can be used to distinguish otherwise similar objects. These characteristics usually have no other effect on the game, unless you're wearing a red cape and enter a room housing a bull. The characteristics of an object may also specify what you can do with the object (whether you can eat it, drink it, open it, and so on). The action routines make use of these characteristics.

The remaining data structure to be considered is the character descriptor. Two items are basic enough to be included in virtually every adventure: where your character is and what he or she is carrying. Another common factor is the state of the character's health, which determines how much more hardship he can take before dying, and which may also affect his current ability to take various actions.

There may be distinctions that specify how the character is carrying an object. He may be wearing it (for instance, a helmet), in which case it might not count against the number of objects he can carry. In *Empire of the Over-Mind*, the character can *hold* only one object out of several that he is carrying. The only one he can actively use is the object he is holding, and he might not have time to reach for another one in a critical situation.

If there is more than one character under your control (as in *Suspended*), or if there are multiple players, each controlling a character, there must be a character descriptor for each one. Interaction between characters can occur when they are in the same room.

Action Routines

Now we come to the heart of the matter: what happens in the imaginary world created by the program. The factors involved are what your character does and what independent agents and other forces in the game do. These correspond to the *action routines* and *automatic routines* in the program.

As you may remember, the parser takes the words that you type for your command and translates them into tokens. The action routines take these tokens and convert them into action in the simulated world of the adventure; they, in turn, invoke an output routine to tell you what has happened. For instance, if you type TAKE COIN—and nothing unusual is involved in this action—then the action routines will remove the

object COIN from the list of objects in the room and add it to your inventory. Other things may happen instead. The coin might not be in the room, in which case the action is impossible. You might be so heavily loaded that you can't carry another nickel, in which case it's again impossible (but with a different output message required to explain the reason). Or the coin may be linked by a magic spell to another object in your inventory, causing something entirely unexpected to happen.

Action routines draw no clear-cut distinction between valid and erroneous commands. Every command causes something to happen, even if it's just putting out a message. Some messages may be considered *error messages* (for example, "Don't be ridiculous!"), but even an error message might give useful information. If the action is impossible, the message might explain why; or in the case of an EXAMINE or READ command, getting a message might be the only reason you entered the command. The strongest distinction that can be made is that some commands actually make something happen, with permanent effect, while others just produce output.

The more complicated the commands are, the more complicated the action routines have to get. In the case of a simple verb-object adventure, command handling is fairly straightforward.

The action routines are entered through a top level routine that we can call the *action dispatcher*, which determines what particular action routine to invoke. Usually, this depends on the verb. But special circumstances may make the dispatcher override the normal action routine for a verb (in *Zork*, for instance, if the character is in a vehicle). This is, in effect, a second dispatcher that determines what commands are allowed in the special situation. Some commands that are normally allowed, such as JUMP, might be impossible or fatal in the boat. Eventually, though, the program will call some action routine for the verb, whether it's the usual one or something unique to the situation.

The action routine for a verb first has to check whether the parser gave it an object in the command. If there isn't one, but one is required by the verb, then the action routine has to issue an error report and takes no action. If the verb doesn't need an object (for instance, WEST, SCREAM), it can go directly to determining what the effect of the action will be.

How They Work

If there is an object, the action routine must determine if you can get at the object. An object that has a specific location must either be in the room with your character or in your inventory for you to be able to do anything with it. (For some verbs, this might not be strictly true. It can make perfect sense for you to EXAMINE or ASK ABOUT an object that you can see at a distance.) Other objects, such as air, are everywhere and always available. So are directions, which are treated as objects in giving commands (GO SOUTH).

The next step, once the object is determined to be legitimate, is to see how the verb applies to the object. This is often just a matter of checking an appropriate entry in the object descriptors. If the verb is EAT, the action routine must check to determine if the object is designated as edible. For TAKE, it must check to see if the object is movable. If the action is inapplicable, the result will be just a message.

The possibility of applying the verb to the object may depend on other circumstances, such as inventory and location. Taking water requires a bottle; attacking an enemy is possible only if you're carrying a weapon. The room itself may affect the outcome of the action; waving a rod might have no effect in one place but could release a magic spell somewhere else.

If the parser allows commands more complicated than just a verb and an object, the action routines have to be able to handle those complications. If adjectives are allowed, the routine must determine if the command uniquely describes an object. If a room contains a short rope and a long rope, and the player types TAKE ROPE, the routine has to ask for more information.

Some programs take one of two short cuts with adjectives. For instance, they may ignore local information in deciding whether an adjective is needed. In *Bedlam,* there are a green key and a red key. Even if only one of the keys is in sight, you have to tell it which key you mean. On the other hand, there are programs like *Empire of the Over-Mind* that will make an arbitrary choice if you don't give enough adjectives to indicate which object you mean. Asking for too much information is the safer route, even if it's irritating; at least the program won't do something you didn't want it to do.

Infocom's adventures let you enter additional information in ambiguous situations, rather than making you enter the whole command over again. The dialogue might look like this:

>TAKE ROPE
Which rope do you mean, the long rope or the short rope?
>LONG
Taken.

When multiple objects occur, the action must be repeated once for each object specified. If the multiple object takes the form of ALL (something), then the program has to check each available item to see if it matches that "something."

If indirect objects and prepositional phrases are allowed, the action routines have to check for the validity of both objects. After that, the routine for the particular verb has to determine if an indirect object or prepositional phrase is allowed, required, or forbidden.

A prepositional phrase is like an adjective in that it narrows down a possible range of choices. The difference is that it affects the action itself rather than the object. For example, if the command is FIGHT VAMPIRE, a prepositional phrase can specify what weapon you want to use. If you have only one weapon, no qualification is needed, so the routine should consider the two-word command sufficient. But if you're carrying both a sword and a stake, for instance, it needs a prepositional phrase to tell just what you want to do.

The preposition that introduces the object has to be appropriate to the action. FIGHT VAMPIRE TO SWORD makes no sense, and the routine for the verb FIGHT must respond accordingly. Usually, only a few prepositions make sense with any given verb.

If the parser follows the model described in this chapter, prepositions will always introduce an indirect object or phrase, never a direct object. The reason for this is that the parser takes any preposition before the direct object and makes it part of the verb. In the command SWING OVER PIT WITH ROPE, SWING OVER is the verb, and the parser gives it to the action routine as a single unit.

Adverbs are similar to prepositional phrases in that they change the meaning of verbs. If you type OPEN DOOR in a murder mystery, the result might be, "Mr. Jenkins is in the room." Typing OPEN DOOR QUIETLY might get you the more useful response, "Mr. Jenkins, who has not noticed your quiet approach, is writing in a pocket notebook."

Automatic Routines

Further variations in the adventure are provided by events that are essentially out of your control. A lamp may run out of power after a certain amount of use. A bomb that you're trying to defuse may explode at a certain time, taking you with it, if you haven't accomplished your task by then. Another character may be running around the dungeon trying to frustrate you. These events are handled by automatic routines, also known as *daemons* (a term that originated with the Multics operating system to describe programs that run independently of any user).

Daemons may be set in motion at the beginning of the adventure, or they may be triggered by some action the character performs (such as entering a certain room). They run each time the player makes a move, although they may not have any visible effect for some time. A simple daemon might just count moves that the player makes and cause something to happen—perhaps making the sun set—after a certain number of moves. A complicated one might change something after each move.

One of the most complicated types of daemons is one that manipulates an adversary character (maybe even a demon). This almost requires keeping a complete second character descriptor. The daemon has to keep track of where the adversary is and determine how he moves each turn. If the adversary is involved in combat, it must keep track of his state of health. If he picks up objects from rooms or steals them from you, it has to keep track of his inventory. If he enters your presence, it has to tell you what he's doing. It may even let you overhear him if he's in a nearby room.

Combat, Wounds, and Recovery

In the borderland between the automatic routines and the action routines, your character will come into conflict with his adversaries. When this happens, the program must determine the outcome of the combat. Who will win—you or the monster? (Following the terminology established by Dungeons and Dragons, I'll use the word *monster* here to mean any creature, human or otherwise, that opposes your character.)

Combat in adventures can be utterly simple or extremely complicated. The simplest kind of combat is the kind in which

you only need to give the right command to rout your opponent. Figuring out what the command *is* may be devilishly difficult, but once you give the command, the program simply eliminates the monster and congratulates you on your cleverness. For instance, if you meet the Wicked Witch of the West, it may be necessary only to THROW WATER (provided you have some water) to wipe her out.

Unfortunately, the monster may be able to dispose of you just as easily. You have to have some way to defeat him, of course, or there's no point to the game. But it may happen that if you don't defeat him immediately, or if you meet him in a certain place, then he will determinedly rob, kill, or otherwise inconvenience you.

One step toward greater complexity is to add a random factor. The correct command may be simply to FIGHT the monster, but it may work only 50 percent of the time (and the monster may fight back with a 30 percent chance of success). This can allow a battle to go on for several rounds before only one survivor remains.

Only a simple-minded view of combat would leave the loser defeated and the winner unscratched. In any real fight, even the winner should come out battered and bruised. One way to simulate this is to assign each character a certain number of "hit points" and to let each successful attack take one or more hit points away. Then, the loser is the one whose hit points fall to zero first.

Another way to make combat more realistic is to provide an assortment of weapons, some of which are more effective than others. An ax might have a low probability of hitting a target but do a lot of damage when it does; a sword might be more likely to hit but do less damage. The effectiveness of the weapon might even vary with the monster it's being used against.

When you use a weapon in combat, you may run a small risk of breaking or dropping it. In that case you must pick it up, switch to another weapon (if you have one), or get out in a hurry.

If you survive the battle, you may find yourself tired and bleeding—in other words, down to a very few hit points. A command such as DIAGNOSE or HEALTH will tell you just how badly off you are. Then you have to recover from your wounds. Recovery may simply be a matter of time, in which

case you just have to avoid fighting long enough to heal. On the other hand, you may need to use a healing potion, invoke a healing spell, or even journey to some place of healing—but taking the necessary actions may require further travel and involve the risk of getting beaten up again.

But what happens to your foe while you're off licking your wounds? The most realistic answer would be that your adversary should heal in a manner similar to your own. Some adventures provide that whenever you encounter a monster, he is presumed in perfect health. You can wear him down as long as you continue to fight—but if you leave and come back, he will be fully recovered.

Language Considerations

Obviously, writing an adventure program can be a complicated task if the program is going to do much. This leads to an important question: What is a reasonable language for writing adventures?

The most popular language today, without question, is BASIC. But BASIC has restrictions. The line numbers make it difficult to expand a program in the middle or to treat one piece separately from the rest. The organization of BASIC programs into separate statements keeps programs from having any readable structure; you can't, for example, have an IF-THEN construction followed by more than a line's worth of statements to be executed if the condition is true. And BASIC has no way of describing groups of data other than with arrays.

What are some of the alternatives? Broadly, there are two classes: *assembly language* and *higher-level languages*.

In assembly language, the *statements* correspond to individual machine instructions. Each instruction has a mnemonic that identifies its function. For example, the instruction that adds two numbers might have the mnemonic ADD. Assembly language lets the programmer use the full power of the computer. Programs written in assembly language will generally take up less memory and run faster than any other programs.

But writing programs one machine instruction at a time is tedious, and it leaves lots of opportunities for programmers to introduce bugs into their code accidentally. Also, an assembly program can't be moved to a computer that uses a different processor, but must be completely rewritten in the assembly

language of the new machine. For these reasons, an assortment of higher-level languages have been devised. Some, like BASIC, use an *interpreter* to run programs. An interpreter is a program that takes each line of the user's language and executes it. Others use a *compiler* to translate the user's program into machine language. Interpreted programs save the programmer a step, since they don't have to be translated; on the other hand, they run more slowly. Compiled programs can be nearly as compact and as fast as assembly language programs if the compiler is a good one. Any language can be either interpreted or compiled, but most are designed for one use or the other.

Popular compiler languages include Pascal, FORTRAN, and C; newcomers that may soon gain equal popularity are Ada and Modula-2. FORTRAN is the oldest living higher-level language and is used mostly for scientific and mathematical programming. It has little to recommend itself for text-intensive programs like adventures. Still, the original *Adventure* was written in FORTRAN, and so was the translation of *Zork* called *Dungeon*.

The other languages mentioned above all have certain features in common. They allow programs to be organized into subroutines and blocks, which lets the flow of a program correspond to its readable structure. They provide facilities for organizing data in logically related groups (called records in Pascal and structures in C), even if the data are of different types. These capabilities make the job of writing a complex program much easier, since they allow breaking the program and data into manageable pieces.

A more esoteric language, but one worth considering, is LISP. LISP programs don't look at all like programs in most other languages; to the uninitiated (and often to the initiated as well) they look like masses of parentheses. But LISP has some very distinctive features. Its basic data structure is the list, which is simply a series of data items of any kind. A list can consist of lists, and programs themselves are lists. Items can be added to a list at any time; this is convenient for describing variable sets of objects, such as the items in a room. Another useful feature of LISP is the *property* list, which lets such properties as the adventurer's inventory and state of health be associated with an item. LISP is usually an interpreted language, but compilers have been written for it.

The original *Zork* was written in MDL, a greatly extended version of LISP. The resulting product is striking evidence of how much the right language can help the programmer.

Trends

With the range of options being as wide as it is, you can easily see why some adventures are much more sophisticated than others in their ability to handle your commands and make things happen around your character. An adventure that runs in a small amount of memory just won't have room for that much programming. But even if a lot of memory is available, there's a trade-off between using that memory for the program and using it for data structures.

For several reasons, we can expect to see future adventures grow in sophistication. The amount of memory in the typical home computer is steadily growing. The 16K "standard" main memory is giving way to 64K or even 128K—and by the time you read this, even more memory may be the norm. Increasingly powerful processors are also moving into home computers, and compilers for programming languages such as C and Modula-2 are becoming available as well. Better processors and languages will ease the programmer's job in creating fancier parsers and action routines. Floppy disk drives are getting cheaper, making it more feasible to get away from tape-loaded programs that have to fit everything into memory at once.

Finally, buyers' expectations will rise. When players see what *Cyborg* or *Zork* can do, they aren't likely to remain satisfied with adventures that limit them to two-word sentences. The more complex an adventure program gets in terms of what it can handle, the simpler the player's job will be. The ideal would be a program that could handle any grammatical command or question and come up with a reasonable response. If this sort of program existed, the player could deal with it as easily and flexibly as with a human gamesmaster in a Dungeons and Dragons game.

Getting to that point within the next century might be too much to hope for. But you can be sure that the state of the art will keep advancing.

Doing Your Own

Doing Your Own

f you enjoy playing adventure games and have written a few computer programs yourself, you might want to try writing one of your own. It isn't that hard, particularly if you limit your initial goals in a reasonable way.

Complex command handling is useful, and you should expect it from the professionals, but you can go a long way with the classic two-word format. Having a dozen events happening behind the scenes makes the adventure seem more like real life, but not every scenario needs such activity. What is important is to have a series of challenges and events that will keep anyone who plays your adventure working at it until it's solved.

The key to a good adventure is its puzzles. Puzzles that are too easy, too hard, or outright arbitrary will leave the player bored or frustrated. On the other hand, puzzles that the player can solve after a reasonable amount of cogitation will give him or her the satisfaction of beating a worthwhile opponent.

The puzzles should be somehow interconnected. Otherwise, you don't have a single big game but a lot of little games. The whole adventure should have some kind of theme; it could be a search of an island, a quest for lost treasure, a murder mystery, or anything that relates all the puzzles to a common idea. Some of the puzzles should provide tools for solving other puzzles. In many of the best adventures, everything leads to a single goal, and every puzzle solved (except for a few possible false leads) is a step toward the ultimate solution.

The player should have a sense of progress. If the adventure is a treasure hunt in which each treasure counts for points, this comes naturally. But if the object of the adventure is a single goal, the player needs some sense that he's getting there. He should be able to see that he's narrowing down the possibilities, getting close to the place he's trying to reach, or otherwise making progress.

The command handling doesn't have to be spectacular, but it shouldn't make the players want to pull out their hair (or yours). An adventure that handles a good number of verbs

is much easier to get along with than one that doesn't. Provide a few synonyms for common actions; some players might find GET more natural than TAKE, or they might think of MOVE rather than PUSH. Throw in a few verbs that do nothing but elicit a standard message (for instance, Eeeeeeek! as a response to SCREAM). It doesn't take much programming effort, and it gives the player a better chance of seeing a positive response to his commands.

Adventure Programming

This chapter is aimed at people with some experience in computer programming. It assumes that you are familiar with BASIC, that you have written a few programs, and that you are familiar with concepts like strings, bits, and arrays.

Writing an adventure program requires the same kind of skills and techniques as doing any other sort of computer programming. While this isn't a book on the art of computer programming, one point does need to be stressed: *design first and code afterwards.* It's been said many times, but plunging right into the code without a plan will make the job much harder than it has to be. The limitless possibilities of an adventure make this rule especially important.

Designing an adventure really begins with deciding what you want the game to do. How many rooms will it have? What objects will be in them? What verbs will the game handle? You may change your mind about some of these parameters once you see how easy or hard they are to code, but you should start off with an initial version of how you want the adventure to play.

Consider what special features you want. Are there other characters in the adventure? If so, what will they do? Are there different objects with the same name (such as buttons to push in different rooms)? Does anything depend on how many turns have passed?

Next comes the design of the program itself. The classic tool for program design is the flow chart, but my own preference is to design by writing *pseudocode.* Pseudocode is a program description that uses the structures of a programming language like Pascal or C, but fills out the structures with general statements of intent rather than program statements. A piece of pseudocode for handling a room description might resemble that shown in Figure 10-1.

Figure 10-1. Pseudocode for a Room Description

```
if torch is lit or room is illuminated
    {print room_description_array(room);
       for each object
          if item_location_array(object) = room
             print object description;
    }
else print "It's too dark to see."
```

This sample illustrates a few points about the pseudocode style that will be used here. Long, explanatory variable names are used. Braces are used to indicate blocks of code that are to be treated as a unit. Indentation is used as a further guide to how blocks of code are organized.

The program should be broken down into various modules, with each one designed separately. This is especially true if you're writing in BASIC, which has no built-in modular structure. As a matter of mental economy, each piece of the program should be small enough for you to understand as a unit.

You should decide early in your design what variables and data structures you need, since these are what hold all the modules together. Group the variables and structures according to the major functions that they serve, such as map, object descriptors, and so on.

Throughout this chapter, I'll be giving specific advice on how to use BASIC for adventure writing. The reason for this is simply that BASIC is the language most often available to microcomputer users and the one that most users know. There are definite advantages to writing adventures in a more structured language, such as C, Pascal, or Modula-2, but if you're using one of those languages, then you probably have enough programming experience to translate the concepts from BASIC.

Now let's take a look at the pieces of an adventure and see how to design each one to create a simple but exciting adventure.

The Main Loop

The program will consist of two parts: initialization code and the main loop. The initialization section dimensions arrays, reads data, and gives initial values to variables. Once that is done, the main loop runs until the adventure ends one way or another.

The major functions of the main loop are to present the room descriptions, accept and parse a command, execute the appropriate action routines for the command, and execute any automatic routines that are required. The overall flow of the program is shown in pseudocode in Figure 10-2.

Figure 10-2. Overall Program Flow in Pseudocode

```
initialize;
while adventure not over
   {if room description required
      describe room;
   input and parse command;
   execute action routines for command;
   execute automatic routines;
   }
```

The Parser

If all that the program expects is a verb and an object, the command parser is pretty simple. It has to take a line of input and end up with two numbers, representing the verb and the object. These numbers, called the *verb token* and the *object token*, will be used for dispatching to action routines and indexing into arrays.

The first step is to get a line of input and extract two words from it. This job is easiest if your BASIC interpreter supports the LINE INPUT statement. LINE INPUT accepts anything the player types as a single string. INPUT might respond with unwanted messages, such as "Extra ignored," if the player types in certain characters. The parser should be flexible about spaces, letting the player type extra spaces between words or at the end of the command. The pseudocode is given in Figure 10-3.

The next step is to convert the two words into the verb and object tokens. To do this, the parser has to look up each word in an array of recognized words. Let's call these arrays the *verb array* and *object array*. To save space, you may want to truncate all commands to a fixed number of characters. Truncation also saves the player some typing and provides a few synonyms for free; for instance, if all commands are truncated to five letters, then either STREAM or STREAMBED will be recognized as the same word. In this case, the words in the

Figure 10-3. Parser Pseudocode

```
first_word = "";
second_word = "";
flag = 1;
while characters remain to be processed
    {get next character;
    if character = space
        {if first_word = ""
            ignore character;
        else if second_word = ""
            {flag = 2;
            second_word = character;
            }
        else exit;
        }
    else if flag = 1
        append character to first_word;
    else append character to second_word;
```

array must be stored in truncated form, and the parser must chop off the words in the command to the same length.

After truncating the words of the command, the parser is ready to look them up. The parser starts by going through the verb array and comparing each string in turn to the first word. If a match is found, this yields an index. This index is then used as the index into another array, called the *verb token array*, to get the number which is the verb token.

Suppose, for example that the verb is LOOK. The parser goes through the verb array until it finds the string LOOK. If it finds it as the fifth element in the verb array, then the fifth element of the verb token array is the verb token.

Several different elements of the verb token array could have the same value; this corresponds to having different verbs that are treated identically. For instance, GET and TAKE might lead to the same verb token. Suppose these are the sixth and tenth elements of the verb array. Then the sixth and tenth elements of the verb token array would have the same value. This permits the program to treat both verbs identically.

The same thing happens with the second word. The program looks it up in the object array. This yields an index, which it uses as an index into the *object token array* to obtain the object token.

It's possible, of course, that the command has only one word in it. In this case, the object token should be set to a *null* value, such as 0 or −1, to indicate that there is no object.

If you're willing to do without synonyms, you can save a step and use the index into the verb array as the token, instead of using that index to look up the token in another array. The "Tower of Mystery" program presented in this book uses that shortcut with objects.

If either of the words isn't matched in the corresponding array, then the command is invalid. At this point, the parser should issue a message indicating which word it couldn't recognize and then ask for another command.

Unrecognized objects might be acceptable, though, for certain verbs. The verb SAY (or TYPE in the Tower program) could legitimately be followed by anything. In this case, the parser must set the object token to a value that stands for general nonsense. The pseudocode for getting from the two words to the two tokens is shown in Figure 10-4.

Figure 10-4. Pseudocode for Getting Two Tokens from Two Words

```
truncate first_word;
truncate second_word;
for i = 1 to length of verb_array
    {if verb_array(i) = first_word
        verb_token = verb_token_array(i);
    }
if no match found
    report error in first_word;
if second_word = ""
    object_token = 0;
else {for i = 1 to length of object_array
        {if object_array(i) = second_word
            object_token = object_token_array(i);
        }
    if no match found
        if verb accepts any object
            object_token = nonsense value;
        else report error in second_word;
    }
```

It may be tempting to try for more sophistication by allowing three-word commands with adjectives. Certainly, it would be simple enough to provide an adjective-token-look-up scheme along the lines of the treatment of the verb and object. The problem is that adjectives are useful only if the program has multiple objects that can be referred to with the same object name. Doing this would require a more sophisticated object-handling scheme than the one that's going to be presented in this chapter.

Describing a World

Before getting into what to do with the commands, we have to decide on some structures. These include the internal map, which describes the rooms, and the object descriptors, which describe the objects found there.

Rooms. The starting point is the collection of rooms. Each room needs a number to identify it. Once you've drawn up a map for your adventure, number the rooms from 1 to however many rooms you have. Any order will do, but it's convenient to use 1 for the room the character will start in. If some rooms are going to have special properties, it would be a good idea to give them low numbers for ease of reference.

To describe the rooms, the program needs an array of room descriptions. If you're willing to limit these descriptions to one line each, then an array of strings indexed by room numbers will do the job. The obvious name for this array is the *room description array.*

Another array is needed to describe the interconnections among the rooms. Let's call this array the *access array,* since it controls access to rooms. This is a two-dimensional array. The first dimension is the number of rooms; the second dimension is the number of different directions in which the character can go.

In order to look up a direction in the access array, directions have to be translated into numbers. Let's say that the directions north, south, east, west, up, and down are numbered respectively as 1, 2, 3, 4, 5, and 6. If the access array is named AC, then AC(6,4) would be the number of the room reached from room 6 by going west. If the character can't go west, then this element would be 0.

Notice that this scheme allows one-way passages if the

game requires them. If room 9 is west of room 10 and the character can go back and forth between them, then AC(9,3) would be 10 and AC(10,4) would be 9. If AC(10,4) is changed to 0, though, the passage allows movement only from west to east.

Usually you will want to keep track of such conditions as illumination that are true in some rooms but not in others. A useful structure for this purpose is a *room flag array*. A *flag* is simply a bit that indicates whether or not some condition is true. Most microcomputer systems can store 16 bits, and therefore 16 different flags, in an integer. Individual flags can be set, tested, and cleared in an integer by using the Boolean operators AND, OR, and NOT, provided these operators work on a bit-by-bit basis in your system.

Some BASIC interpreters don't support bit-by-bit Boolean operations. If yours doesn't, you'll have to use a separate array for each different flag. If you aren't sure, type PRINT 15 AND 40. If the result is 8, then you're all set. If it's 1 or −1, then you'll have to use one array per flag.

Suppose you need three flags to say, respectively, that the room has its own light, that magic works in it, and that water is present. These flags must all be powers of 2. Let's give them values 1, 2, and 4 respectively. You should give names to these flags, so that it's more obvious what references to them mean. Somewhere near the top of your program, you would have a line such as the following:

```
20 LT=1:MG=2:WT=4
```

If the room flag array is called RF, then you would test for the presence of water in room RM with the statement:

```
2660 IF RF(RM) AND WT THEN...
```

If the lights go on in the room, the appropriate statement would be:

```
3220 RF(RM) = RF(RM) OR LT
```

And if magic suddenly becomes useless, the statement is:

```
4330 RF(RM) = RF(RM) AND NOT MG
```

One of the flags might indicate whether the character has previously seen the room or not. This is useful if you have two room description arrays, one with the full description and one with just the name of the room. Checking the flag would let the program give a long description the first time the

character goes into the room and a short description after that.

In addition, you may need some special variables for situations that apply to just one room. These might include the number of times the character has entered a certain room, the presence of a hazard in another room, or whatever is needed for the situation. There isn't much to be said about these; just use them as necessary. In some cases, treating the condition as an "object" in the room might be the easiest way to handle it. For instance, you could make a room noisy by putting the object "noise" in it. This falls under the next topic, though, so let's move on.

Objects. Once the rooms are taken care of, the other major set of data structures is the one that deals with objects. The structures here will be arrays indexed by object tokens.

Objects include much more than simply things that can be touched and picked up. The kinds of objects even a simple adventure might deal with include the following:

- Objects that have a specific location and can be carried.
- Objects that have a specific location but are normally immovable.
- Objects that have a specific location and can't be carried, but may move and act on their own. These include nonplayer characters and monsters.
- Objects that specify the action of a certain verb. Any word following SAY or TYPE would fall into this category.
- Directions, such as NORTH and UP.
- Objects that can be ENTERed or CLIMBed to get to another room.

The most fundamental distinction is whether or not the object has a specific location. Objects with specific locations are usually objects in the ordinary sense of the word. A lamp, a boulder, a chair, or an ogre would fall into this category. Most verbs can be used with them; for example, the player might reasonably try to pick them up, examine them, break them, and so on. An object corresponding to a condition also falls into this class, even though physical actions don't have any meaning when used with it. Let's use the word *items* to refer to such objects.

Objects without specific locations don't correspond to physical entities, so they can be used only with special-purpose verbs. There's no need to consider what happens when

the player tries to eat, open, or kiss a SOUTH; a general message rejecting the very notion will do. Only GO, and perhaps LOOK, can mean anything with that object.

For this reason, most object descriptors can be limited to items. If object tokens are divided into two series, one for items and one for all the rest, it makes separation easier. You could do this by reserving the numbers 1 to 100 for items and 101 and above for other objects.

For items, two kinds of descriptions are needed. One is the word found in the object array; the other is a slightly longer description which is used when listing objects in a room or in the character's inventory. The longer descriptions are kept in an array, which can be called the *item description array*. For instance, if the name of the object is BAG, the item description array might have the text "Old leather bag."

Another structure gives the location of each item; let's call it the *item location array*. There are three possibilities for an item's location: (1) in a room, (2) in the character's inventory, or (3) nowhere at all. The third case comes about when an item has been destroyed or when it hasn't yet become accessible. A positive number in the item location array indicates that the item is in the room with that number. For other possibilities, use 0 for items in the character's inventory and −1 for items that aren't to be found.

This approach makes it simple to list items in a room, to list the character's inventory, and to determine whether the character is in the same room as an item. The first case just requires going through the item location array and noting the indices of all elements that match the current room. The second requires looking for elements whose value is zero. The third requires checking for values that equal *either* the current room or zero. After all, any item that the character is carrying is most likely in the room with him!

Another array that will usually be needed is one that indicates whether certain conditions are true for each particular item. Some items can be carried; others can't. Some items give off light; others don't. These characteristics can be handled by an *item flag array*, just as characteristics of rooms are handled by the room flag array. The techniques discussed for rooms work the same way here. Flags are tested with AND, set with OR, and cleared with AND NOT.

Other possibilities are optional. You might want to create

an array that gives the weight of each item. Doing this and setting a limit on the weight the character can carry would provide a more realistic inventory limit than just limiting the total number of items carried.

What about items that move on their own—people, monsters, and robots? Fortunately, they don't need much extra in the way of data structures; the complications arise mostly in writing the automatic routines. A few extra flags may be needed to indicate how they will react to the character—whether they're alive or dead, whether the character has tried to speak to them, and so on.

Summary. Selecting the right data structures is vital to laying out the design of an adventure. Once you have them, the rest will fall into place easily enough. To review, here are the major structures that a simple adventure will typically need:

- A *verb array,* consisting of all the words that could be the first word of a command.
- An *object array,* consisting of all the words that could be the second word of a command.
- A *verb token array,* which contains verb numbers (tokens) corresponding to verbs. This array has as many elements as the verb array.
- An *object token array,* which contains object numbers corresponding to object names. This array has as many elements as the object array.
- A *room description array,* which contains the text descriptions of the rooms. This array has one element for each room.
- An *access array,* specifying the paths between rooms. The two dimensions of this array are the number of rooms and the number of possible directions.
- A *room flag array,* whose elements are integers in which each bit corresponds to a condition that's true or false about a room.
- An *item description array,* which contains the text description of each item.
- An *item location array,* which tells where each item is.
- An *item flag array,* whose elements are integers in which each bit corresponds to a condition that's true or false about an item.

The Description Section

The major *output routine* in a simple adventure is the section of code that describes the room to the character. This code does two things: it prints the room description and it lists any visible items.

In the interest of not being too verbose, the description section should be skipped if nothing has changed since the last command. However, the description should be printed if the character has moved or if the last command was LOOK. The action routines can notify the description section of these circumstances by setting the *description request flag.*

If the description request flag isn't set, the program simply skips the description section and waits for another command. If it is set, there may still be constraints on what the character can see. The most common is that there may not be any light in the room. Checking this may involve checking one of the room flags (as well as another variable to indicate whether the

Figure 10-5. Room Description Pseudocode

```
if description_request_flag <> 0
   {description_request_flag = 0;
   if (room is not lit and
       ((item_location_array(lamp) <> 0 and
         item_location_array(lamp) <> room) or
        lamp is not lit))
       {room_lit = FALSE;
       print "It's too dark to see."
       }
   else {print room_description_array(room);
       room_lit = TRUE;
       flag = 0;
       for i = 1 to number_of_items
           {if item_location_array(i) = room
               {if flag = 0
                   {print "Visible items are:"
                   flag = 1;
                   }
               print item_description_array(i);
               }
           }
       }
```

character is bearing his own light source). If these conditions aren't satisfied, the description section only has to print a message, such as IT'S TOO DARK TO SEE, and go on to the command input.

If all is well, the first thing to do is to print the appropriate string from the room description array. Then the program must check all items and print the names of the ones that are in the current room. The pseudocode is shown in Figure 10-5.

Automatic Routines

After describing the room, the program has to take care of any events that happen independently of the character. The position of these routines in the loop is important. By putting them after the room description but before the command input, the program insures that the player finds out what is in the room before he is told what is happening there. It's only fair that the player should know there's an ogre in the room before he's told that it is attacking him.

The automatic routines are the part of the program where you can really get imaginative. You could include moving creatures, lamps that burn out after a while, rooms that collapse three turns after the adventurer tampers with something, transitions from day to night, and much more.

The concept of time is important to many of these events. For this reason, the first job of the automatic routines should be to maintain a *move counter*. This counter is incremented after every move.

Not every command should be considered a move, however. If the parser can't recognize the words that were typed, the program shouldn't act as if time passed. The same is true of commands that the action routines can't make any sense of (such as EAT NORTH). In these cases, the program should go back to the command input rather than make things happen while the player is figuring out what commands work.

A creature wandering around the dungeon can make life interesting for the adventurer. Assuming there are no special restrictions on how the creature moves, this can be handled by generating a random number from 1 to the number of possible directions and having the creature attempt to move from its current room to the appropriate room in the access array. Another possibility is that the creature might follow the ad-

creature's element in the item location array to the adventurer's current room. Remember, a creature is just an animated item, so its location is found in the item location array.

Suitable information should be given to the player. The program should tell him about a change if it occurs in the room he is occupying. Some events may be noticed even from another room; for instance, the player may be told that, "You hear an explosion in the distance," when a bomb goes off elsewhere.

The pseudocode given in Figure 10-6 provides for an egg that hatches on the thirtieth move of the game.

Figure 10-6. Pseudocode for Hatching an Egg

```
move_counter = move_counter + 1;
if move_counter = 30
    {if item_location_array(egg) = 0
        {print "The egg you are carrying hatches.";
        item_location_array(roc) = room;
        }
    else {item_location_array(roc) =
            item_location_array(egg);
        if item_location_array(egg) = room
            print "The egg hatches into a baby roc.";
        }
    item_location_array(egg) = -1;
    }
```

Action Routines

Action routines allow the player's commands to cause something to happen. The action routines in a complex adventure can be thought of as a set of filters through which the commands pass. Each filter examines the command and may handle it in whole or part, depending on the particular features of the command. One filter might look at an object and decide whether it's accessible; another might look at the tool used and cause special events to occur. Yet another filter could make things happen according to the verb.

For simple adventures, the following three-step strategy can be used:

1. Check the object and make preliminary decisions about it.
2. Call an action routine specific to the verb.

Prefiltering. The reason for step 1 is to have a single body of code that handles issues common to all verbs. For instance, if an item isn't in the room, you probably can't do anything with it, and the command handling can stop right there. This step is called *prefiltering*.

Objects that aren't items usually make sense only when used with a limited number of verbs. The prefiltering process should check for these objects and reject the command if it uses objects with inappropriate verbs. It's difficult, for example, to do much with the objects NORTH, SOUTH, EAST, and WEST except GO there.

The action routines can vary quite a bit from one adventure to another, so the pseudocode examples in this section will be more in the way of examples than prescriptions. Typical pseudocode for prefiltering is shown in Figure 10-7.

Figure 10-7. Prefiltering Pseudocode

```
if object_token is a direction
    {if verb_token <> GO
        {print "That doesn't make sense.";
        return;
        }
    }
else if object_token is a magic word
    {if verb_token <> SAY
        {print "I don't understand that.";
        return;
        }
    }
else if item_location_array(object_token) <> 0 and
        item_location_array(object_token) <> room
            {print "It isn't here.";
            return;
            }
call action routine for verb_token;
```

Dispatching to action routines. If prefiltering reveals no problems, the next step is to call the action routine for the particular verb token. In BASIC, this is done with one or more ON–GOTO statements, using the verb token as the control variable.

Each action routine is different. Some verbs require an object, others can't use one, and for still others, it is optional. An action may have side effects on the room, the character, or items in the room.

include TAKE, DROP, INVENTORY, QUIT, LOOK, EXAMINE, and the directional commands.

TAKE. This verb, if successful, transfers an object from the room to the character's inventory. To be successful, the following conditions must be satisfied:

- The object must be an item.
- It must be in the room, rather than already in the character's inventory.
- It must be capable of being carried.
- The character must have some carrying capacity remaining.

We can assume that prefiltering has assured us that the object is an item and that it is either in the inventory or in the room.

Taking certain objects may have special results. Picking up a booby-trapped item might kill the character. Taking one item might reveal a previously concealed item. (This would be handled by changing the location of the concealed item, which was previously −1 [nowhere] to the number of the room.) Picking up the right item might open a secret passageway, thus changing the access array. Checking for special results should come last, after the routine has established that the character is capable of picking up the item.

Typical pseudocode for TAKE is shown in Figure 10-8.

Figure 10-8. Typical Pseudocode for TAKE

```
if item_location_array(object_token) = 0
    print "You're already carrying it!"
else if (item_flag_array(object_token) AND carry_flag) = 0
    print "You aren't able to do that."
else if items_carried = maximum
    print "You're carrying too much already."
else {item_location_array(object_token) = 0;
    items_carried = items_carried + 1;
    print "Taken.";
    handle side effects for object;
    }
```

DROP. This command transfers an object from the inventory to the room. It is simpler than TAKE, since an object that can be picked up can almost always be dropped. The major consideration is whether the item is in the inventory. As with TAKE (and just about any other command), there could be side effects. Typical pseudocode is shown in Figure 10-9.

Figure 10-9. Typical Pseudocode for DROP

```
if item_location_array(object_token) <> 0
    print "You're not carrying it!"
else {item_location_array(object_token) = room;
    print "Dropped.";
    items_carried = items_carried - 1;
    handle side effects for object;
    }
```

INVENTORY. This is a straightforward command, which simply lists all items whose location is 0 (that is, all items that the character is carrying). If the character is carrying nothing, the response should say so. Typical pseudocode is shown in Figure 10-10.

Figure 10-10. Typical Pseudocode for INVENTORY

```
if items_carried = 0
    print "You aren't carrying anything."
else {print "You are carrying:";
    for i = 1 to number_of_items
        if item_location_array(i) = 0
            print item_description_array(i);
    }
```

QUIT. This command is also straightforward. If applicable, QUIT should report the player's score. Asking the player for confirmation is generally a good idea. Figure 10-11 illustrates typical pseudocode.

Figure 10-11. Typical Pseudocode for QUIT

```
print current score;
print "Do you really want to quit now?"
input response;
if response = "Y" or "YES"
    exit program;
```

LOOK. This command does only one thing—it sets the description request flag, which causes the description of the room to be repeated before the program accepts the next command. The pseudocode is straightforward:

```
description_request_flag = TRUE;
```

EXAMINE. Examining certain objects will provide extra information about them. In some cases there could be other

effects; for instance, examining one object could make another one appear and examining a Gorgon's head could have petrifying consequences. Otherwise, the program should print a message like, "There's nothing special about it." Typical pseudocode is given in Figure 10-12.

Figure 10-12. Typical Pseudocode for EXAMINE

```
if object_token is not an item
    print "Huh?"
else if object_token = scroll
    print "Something is written on it."
else if object_token = box and item_location_array(key) = -1
    {print "There is a key in the box.";
    item_location_array(key) = room;
    }
else print "There's nothing special about it."
```

Directions. These are the commands NORTH, SOUTH, EAST, WEST, UP, DOWN, and possibly more. The verb GO followed by one of these words could serve the same purpose. In either case, all the directional commands can go into the same handler, after setting a variable to indicate which direction was chosen. For instance, the command NORTH would set the direction variable to 1, if the direction NORTH corresponds to an index of 1 in the access array.

If light is a factor, the direction commands must check for its presence. If it's dark, the adventurer might be reduced to bumping into walls until he can remedy the situation.

Figure 10-13. Typical Direction Pseudocode

```
if room_lit
    {if access_array (room,direction) = 0
        print "You can't go in that direction."
    else {room = access_array (room,direction);
        description_request_flag = TRUE;
        }
    }
else print "You stumble into a wall."
```

Assuming there's no hindrance to movement, the direction routine has to check the access array, indexing it with the current room and direction variables. If the corresponding element of the access array is 0, there is no way to go in that

direction and the character is stuck where he is. Otherwise, the value of that element is the number of the new room. The description request flag must also be set so that the description section will tell the player where he is. Pseudocode is given in Figure 10-13.

When You're Writing More Than One

The information in this chapter, plus a good amount of imagination, should be enough to let you write an adventure that will keep your friends entertained for hours. Once you've done this, don't be surprised when they come back for more.

Writing a second adventure will be less work than writing the first one. A large part of the code—the parser, the room description section, and many of the action routines—will hardly change at all. Just create a new map, new puzzles, and a few commands that are different from the last ones, and your old adventure becomes a new one.

After you've done a couple of adventures and started plans for several more, you might start wondering whether it would make sense to write an *adventure interpreter* rather than having a separate program for each adventure. An adventure interpreter is a program that takes adventure descriptions and runs them, just as a BASIC interpreter runs BASIC programs. Once you have an interpreter, you would write your adventures in an *adventure description language* rather than a general-purpose language like BASIC.

An adventure interpreter does make it easier to write new adventures. It also is likely to use memory more efficiently than an adventure written in BASIC. On the other hand, writing the interpreter isn't simple. An adventure definition language that offers enough flexibility to be worthwhile must include quite a few features. It has to be able to specify all the data structures: the access array, the item descriptions, the flag arrays, and so on. It has to let events be triggered by player action, by random events, and by timers based on either of these two.

An adventure interpreter is a project for the really ambitious. If you write your adventures in a modular fashion, you shouldn't find it difficult to churn out one adventure after another by modifying your initial program.

Tower of Mystery: A Simple Adventure Program

Tower of Mystery: A Simple Adventure Program

ower of Mystery" is a BASIC program designed to illustrate some of the concepts of adventure programming and to serve as a starting point for writing your own adventure programs. The object is to enter an old factory building, where the world's only remaining copy of *Adventure* is reputed to be stored, and to leave with a tape containing a copy of the program. The focus is largely on a computer, which the player must learn how to use in order to get the program onto the tape. The player can even ask the computer to RUN ADVENTURE. So not only is there a computer within the computer, there is an adventure within the adventure!

Tower of Mystery will run on virtually every popular home computer. Program 1 is in Microsoft BASIC; Program 2 gives modifications to the Microsoft version that will allow it to run on the Atari computers. Program 3 is in TI BASIC.

To play the game on your computer, follow the directions below:

Adam. Type in Program 1.
Apple. Type in Program 1.
Atari. Modify Program 1 by adding or changing the lines shown in Program 2; then type in the modified version. With cassette, at least 16K memory is required. With disk, you'll need at least 24K memory. *Note:* On the Atari, all commands must be completely spelled out (GO SOUTH, LOOK, INVENTORY, and so on).
Commodore PET. Type in Program 1.
Commodore VIC-20. Type in Program 1. At least 8K memory expansion is required.
Commodore 64. Type in Program 1.
Commodore 128. Type in Program 1 in 64 mode.
IBM PC. Type in Program 1.
IBM PCjr. Type in Program 1.
Radio Shack TRS-80 Model 1. Type in Program 1.

Tower of Mystery

Radio Shack TRS-80 Model 3. Type in Program 1.
Radio Shack TRS-80 Model 4. Type in Program 1.
Radio Shack TRS-80 Color Computer. Type in Program 1. The program requires almost 8K of memory and should run on a machine with 8K or more; however, since the disk drive uses 2K, you'll have to save the program to cassette if yours is an 8K machine. With Extended BASIC, type PCLEAR 1 before typing (or loading) the program. That clears additional RAM normally allocated as video RAM.
Texas Instruments TI-99/4A. Type in Program 3.

Like virtually every text adventure, Tower of Mystery does not conform to every guideline given in the previous chapter. As a result, it shows not only the basic principles but some of the areas in which personal variation is possible.

If you would like to play Tower before being told all its secrets, then stop here and type it in now. The rest of the chapter will explain the adventure in detail.

Variables and Arrays

The following variables and arrays are used in Tower:

Constants

NR Number of rooms
NT Number of "things" or items
NP Number of pseudo-objects (all other objects)
NO Number of objects of all kinds
NV Number of verbs

Flags

CA Computer (the one in the adventure) is active
CD Computer is dead
LI Character is logged into computer
CP COPY program is running on computer
MF Computer manual has been found
CF Coin has been found
WD Room description request flag
RT Rats are troublesome
BT Bats are troublesome
CT Time since computer has been started

General Variables

C$ Command input
C1$ First word of command

C1 Verb token
C2 Object token
RM Room currently occupied

Arrays

AC(NR,6) Access array
VB$(NV) Verb array
OB$(NO) Object array
RM$(NR) Room description array
TD$(NT) Item description array
VN(NV) Verb token array
TL(NT) Item location array
TF(NT) Item flag array (only flag is whether item can be carried)

Program Sections

The program breaks down into several sections. Lines 200–399 contain the room description code, while the command parser is located in lines 400–699. The action routines start at line 700 and continue up to the start of the completion messages at line 8000.

The small amount of code that passes for automatic routines is found in line 8200 to the end. Note that in this adventure, the automatic routines are located after the command input and action routines, rather than before them. This lets the automatic events occur immediately when the character steps into a new room, as is appropriate for these particular events.

At the highest level, the program runs in a loop that performs the following operations:

- Gives room description if the WD flag is set.
- Inputs a command and parses it into the verb token C1 and the object token C2. If there is no object, C2 is set to 0.
- Does some preliminary checks, then dispatches to the handler for C1.
- Performs the action specified in the handler.
- Performs any automatic actions required for the bats or the computer.

The parser takes the raw input (C$) and extracts two words (C1$ and C2$) from it. If the player types extra words, its action will be a bit erratic. It then translates these words into the verb token (C1) and the object token (C2). If only one word was typed, C2 is 0.

The program has a verb token array but no object token array. This allows it to recognize synonyms for verbs, such as GET and TAKE, but not for objects. For verbs, the index into the array VB$ is translated into a verb token number by the array VN. For objects, the index into OB$ is the object token number.

The access array controls passage from one room to another. The six possible directions are translated into numbers as follows:

1 North
2 South
3 East
4 West
5 Up
6 Down

To determine where the character goes when moving, the program considers the current room RM and the direction D. The room that will be reached is AC(RM,D). If movement in direction D is impossible, then AC(RM,D) is 0.

The location of an item is determined by the array TL. A value of 0 in the array indicates that the character is carrying it; a value of −1 means that it cannot currently be found anywhere. The item flag array TF indicates which items can be carried; a nonzero value permits carrying. Because of the small number of items in this adventure, there is no limit on how much the character can carry.

There are two kinds of objects in addition to items: directional words (NORTH, SOUTH, EAST, WEST, UP, and DOWN) and commands to be typed into the computer.

Some of the most interesting activity is organized around the computer. Once the player starts the computer, the counter CT starts ticking; when it has reached its limit, the computer will die (the flag CD is set) and the game is hopeless from that point on.

In the few turns that it runs, the TYPE command lets the player do a number of things with the computer, including logging on and off and running programs. In fact, TYPE will accept any object at all, including total nonsense. The flags LI, CP, and MT keep track of the state of the computer. LI is set when the character successfully logs on. CP is set when he runs the COPY program, so that the next TYPE command will be

interpreted as input to COPY. MT is set to indicate that the tape has been mounted on the computer, so that COPY will work.

A couple of objects have their descriptions changed during the course of the adventure, reflecting changes in their state. This happens to the rats when they are fed and the computer when it dies. Another way to do this would have been to remove the old object and insert a new object with a different token number. Doing so, however, would require a way to have the same word (RATS or COMPUTER) refer to different objects at different times.

Building the Tower

Devising ideas for an adventure game can be as difficult as writing the program. For what it may be worth to beginning adventure writers, here is a rough description of how the Tower of Mystery adventure took shape in my mind.

The starting point was the setting. The old factory of the adventure is, in fact, a real building in Manchester, NH, which now houses several high-tech companies. The computer room, vending machine, manufacturing area, and clock tower all followed from the decision to use this building. This suggested some obvious puzzles, such as how to get something from the vending machine and how to start the computer.

The next step was to decide on an overall goal for the adventurer. The one that I chose was to get a "treasure" in the form of a program from the computer. This gave an end point to which everything had to lead.

What obstacles could I put in the way of getting this program? Three finally made it into the adventure: getting media onto which to copy the program, turning on the computer, and logging onto the system. Each of these provided a chain of puzzles, tying in items which I had previously decided on.

The rats were introduced to make getting the media difficult; the vending machine provided the candy to feed the rats. The hidden coin made it possible to use the vending machine. Logging on required finding a computer manual, which was guarded by bats, and a noisy clock tower provided the way to scare the bats away. For starting up the computer, I decided not to introduce a chain of puzzles, but instead to make the player try different actions until he found the right one. The correct action—kicking the computer—seems almost natural after the usual start-up techniques fail to do the job.

Tower of Mystery

A complete map of Tower of Mystery, along with the commands you'll need to use in different places, is given in Figure 11-1, located after the programs. Again, if you want to try to complete the game without additional help, don't look at the map! But if you're interested in seeing how the programming makes the map come alive on your monitor, or if you want to create adventures of your own, the map will be helpful.

Program 1. Tower of Mystery, Microsoft BASIC Version

```
10 NR=14:NT=11:NP=11:NV=29:NO=NT+NP
20 DIM AC(NR,6),VB$(NV),OB$(NO),RM$(NR),TD$(NT),VN
   (NV),TL(NT),TF(NT)
99 REM *INITIALIZATION*
100 RM=1:CT=0:CF=0:RT=-1:CA=0:MF=0:CD=0:WD=-1:LI=0
    :BT=-1
120 FOR I=1 TO NR:READ RM$(I):NEXT I
130 FOR I=1 TO NV:READ VB$(I),VN(I):NEXT I
140 FOR I=1 TO NO:READ OB$(I):NEXT I
150 FOR I=1 TO NT:READ TD$(I),TL(I),TF(I):NEXT I
160 FOR I=1 TO NR:READ AC(I,1),AC(I,2),AC(I,3),AC(
    I,4),AC(I,5),AC(I,6):NEXT I
199 REM *MAIN LOOP: ROOM DESCRIPTION*
200 IF WD=0 THEN 400
210 WD=0:PRINT RM$(RM)
220 K=0
230 FOR I=1 TO NT
240 IF TL(I)<>RM THEN 270
250 IF K=0 THEN PRINT "YOU SEE:":K=1
260 PRINT TD$(I)
270 NEXT I
300 PRINT "EXITS ARE:"
310 FOR I=1 TO 6
320 IF AC(RM,I)<>0 THEN PRINT VB$(I);" ";
330 NEXT I
340 PRINT
399 REM *MAIN LOOP: COMMAND INPUT AND PARSER*
400 INPUT C$
410 L=LEN(C$):IF L=0 THEN 400
420 C1$="":C2$="":C2=0:X=0
430 FOR I=1 TO L
440 A$=MID$(C$,I,1):A=ASC(A$):IF A>=97 AND A<=122
    THEN A$=CHR$(A-32)
450 IF A$<>" " THEN 460
455 IF C2$<>"" THEN 500
457 X=1:GOTO 490
460 IF X=0 THEN C1$=C1$+A$:GOTO 490
```

```
470 C2$=C2$+A$
490 NEXT I
500 IF C1$="" THEN PRINT "EXCUSE ME?":GOTO 400
510 C1$=LEFT$(C1$,5):C2$=LEFT$(C2$,5)
520 FOR C1=1 TO NV
530 IF VB$(C1)=C1$ THEN C1=VN(C1):GOTO 600
540 NEXT C1
550 PRINT "I DON'T KNOW THE VERB ";C1$:GOTO 400
600 IF C2$="" THEN 700
610 FOR C2=1 TO NO
620 IF OB$(C2)=C2$ THEN 700
630 NEXT C2
635 IF C1=11 THEN C2=1:GOTO 700
640 PRINT "I DON'T KNOW THE OBJECT ";C2$:GOTO 400
699 REM *PRELIMINARY FILTERING OF COMMANDS*
700 IF C2>NT+5 AND C1<>7 AND C1<>11 THEN 640
710 IF C2>NT AND C2<NT+5 THEN IF C1<>11 THEN 8010
720 IF C2>NT THEN 750
725 TL=TL(C2)
730 IF C1<>11 AND TL<>RM AND TL<>0 THEN PRINT "IT
    ISN'T HERE.":GOTO 400
750 ON C1 GOTO 1000,1000,1000,1000,1000,1000,1100,
    1200,1300,1400
760 ON C1-10 GOTO 1500,1800,1900,2000,2100,2200,23
    00,2400,2500,9999
770 ON C1-20 GOTO 2600,2700,2800
999 REM *DIRECTIONS*
1000 IF AC(RM,C1)=0 THEN PRINT"YOU CAN'T GO THAT W
     AY.":GOTO 200
1010 IF RM=4 AND RT<>0 AND C1=3 THEN PRINT"THE RAT
     S LOOK TOO FIERCE.":GOTO 8200
1020 RM=AC(RM,C1):WD=-1:GOTO 8200
1099 REM *GO*
1100 IF C2=0 THEN 8050
1105 IF C2<=16 THEN 8040
1110 C1=C2-16:GOTO 1000
1199 REM *EAT*
1200 IF C2<>6 THEN 8010
1210 TL(C2)=-1:PRINT"IT TASTES STALE.":GOTO 8200
1299 REM *KICK*
1300 IF C2=0 THEN 8050
1305 IF C2<>7 OR CD<>0 THEN 8020
1310 IF CA<>0 THEN CT=9:GOTO 8200
1320 PRINT "THE COMPUTER STARTS UP!"
1330 PRINT "THE CONSOLE DISPLAYS: 'PLEASE LOG IN.'
     ":CA=-1:GOTO 8200
1399 REM *INSERT*
1400 IF C2=0 THEN 8050
1410 IF C2<>5 THEN 8010
```

Tower of Mystery

```
1430 TL(5)=-1:TL(6)=RM:PRINT "A CANDY BAR COMES OU
     T.":GOTO 8200
1499 REM *TYPE*
1500 IF C2=0 THEN 8050
1502 IF RM<>10 THEN 8030
1510 IF CA=0 THEN PRINT "THE COMPUTER ISN'T RUNNIN
     G.":GOTO 8200
1520 IF LI<>0 THEN 1600
1530 IF C2<>12 THEN PRINT "'INVALID LOGIN ID.'":GO
     TO 8200
1540 LI=-1:PRINT "'";C2$;" LOGGED IN.'":GOTO 8200
1600 IF CP<>0 THEN 1700
1605 IF C2<=12 THEN PRINT "'INVALID COMMAND.'":GOT
     O 8200
1610 IF C2=13 THEN PRINT "'COPY LOGOU ADVEN'":GOTO
      8200
1620 IF C2=14 THEN PRINT "'WELCOME TO ADVENTURE! W
     #ULD Y#$*'":CT=9:GOTO 8200
1630 IF C2=15 THEN PRINT "'MOUNT TAPE THEN TYPE FI
     LE NAME.'":CP=-1:GOTO 8200
1640 PRINT "'LOGGED OUT.'":LI=0:GOTO 8200
1700 CP=0:IF C2<=12 THEN PRINT "'NO SUCH FILE.'":G
     OTO 8200
1710 IF MT=0 THEN PRINT "'ERROR: TAPE NOT MOUNTED'
     ":GOTO 8200
1720 PRINT "THE TAPE SPINS...":FOR I=1 TO 500:NEXT
      I
1730 PRINT "'FILE COPIED.'"
1740 IF C2=14 THEN PRINT "CONGRATULATIONS, YOU'VE
     DONE IT!":END
1750 GOTO 8200
1799 REM *TAKE*
1800 IF C2=0 THEN 8050
1805 IF TF(C2)=0 THEN PRINT "THAT'S BEYOND YOUR AB
     ILITY.":GOTO 8200
1810 IF TL=0 THEN PRINT "YOU ALREADY HAVE IT!":GOT
     O 200
1820 IF C2=4 AND CF=0 THEN PRINT"THERE WAS A COIN
     UNDER IT.":TL(5)=RM:CF=-1
1825 IF C2=2 THEN MT=0
1830 TL(C2)=0:GOTO 8000
1899 REM *DROP*
1900 IF C2=0 THEN 8050
1905 IF TL<>0 THEN PRINT "YOU DON'T HAVE IT.":GOTO
      200
1910 TL(C2)=RM:IF RM<>4 OR C2<>6 THEN 8000
1920 PRINT "THE RATS DEVOUR THE CANDY AND GET SLEE
     PY."
1930 TD$(1)="SLEEPY RATS":TL(6)=-1:RT=0
```

```
1940 GOTO 8000
1999 REM *INVENTORY*
2000 K=0:PRINT "YOU ARE CARRYING:"
2010 FOR I=1 TO NT
2020 IF TL(I)=0 THEN PRINT TD$(I):K=1
2030 NEXT I
2040 IF K=0 THEN PRINT "NOTHING."
2050 GOTO 8200
2099 REM *MOUNT*
2100 IF C2=0 THEN 8050
2110 IF C2<>2 THEN 8010
2120 IF RM<>10 OR MT<>0 THEN 8030
2130 TL(2)=RM:MT=-1:GOTO 8000
2199 REM *READ*
2200 IF C2=0 THEN 8050
2210 IF C2=3 THEN PRINT "'INSERT COIN.'":GOTO 8200
2220 IF C2<>10 THEN PRINT "NOTHING IS WRITTEN ON I
     T.":GOTO 8200
2230 PRINT "'...USER ID IS ROAD...'"
2240 PRINT "'TYPE DIR FOR LIST OF COMMANDS...'"
2250 PRINT"THE REST IS ILLEGIBLE.":GOTO 8200
2299 REM *FIGHT*
2300 PRINT "THAT WON'T WORK.":GOTO 8200
2399 REM *START*
2400 IF C2=0 THEN 8050
2410 IF C2=7 THEN 8020
2420 GOTO 8010
2499 REM *OPEN*
2500 IF C2=0 THEN 8050
2510 IF C2<>9 THEN 8010
2520 IF MF<>0 THEN PRINT "IT ALREADY IS.":GOTO 820
     0
2530 PRINT "INSIDE IT IS A MANUAL.":TL(10)=RM:MF=-
     1:GOTO 8200
2599 REM *LOOK*
2600 IF C2<>0 THEN 8040
2610 WD=-1:GOTO 8200
2699 REM *WIND*
2700 IF C2<>11 THEN 8010
2710 IF BT=0 THEN PRINT "IT'S FULLY WOUND.":GOTO 8
     200
2720 PRINT "THE CLOCK CHIMES DEAFENINGLY AND SOMET
     HING FLIES PAST."
2730 BT=0:TL(8)=-1
2740 GOTO 8200
2799 REM *EXAMINE*
2800 IF C2=3 OR C2=10 THEN PRINT "SOMETHING IS WRI
     TTEN THERE.":GOTO 8200
2810 IF C2=9 AND MF=0 THEN PRINT "IT IS CLOSED.":G
```

```
2820 IF C2=4 THEN PRINT "IT LOOKS BEYOND REPAIR.":
     GOTO 8200
2830 IF C2=7 THEN PRINT"THIS IS AN ANCIENT MAINFRA
     ME WITH A CONSOLE.":GOTO 8200
2840 IFC2=11THENPRINT"THERE IS A LARGE HANDLE FOR
     WINDING THE CLOCK.":GOTO 8200
2845 IF C2=2 AND MT<>0 THEN PRINT "IT IS MOUNTED O
     N THE COMPUTER.":GOTO 8200
2850 PRINT "YOU SEE NOTHING SPECIAL.":GOTO 8200
7999 REM *COMPLETION MESSAGES*
8000 PRINT "OK.":GOTO 8200
8010 PRINT "THAT'S SILLY!":GOTO 200
8020 PRINT "NOTHING HAPPENS.":GOTO 8200
8030 PRINT "YOU CAN'T DO THAT NOW.":GOTO 8200
8040 PRINT "WHO'S YOUR ENGLISH TEACHER?":GOTO 200
8050 PRINT "PLEASE GIVE AN OBJECT.":GOTO 200
8199 REM *COMPLETION ROUTINES*
8200 IF CA=0 THEN 8300
8210 CT=CT+1:IF CT<10 THEN 8300
8220 IF RM=10 THEN PRINT "THE COMPUTER DIES WITH A
      LOUD POP."
8230 CD=-1:CA=0:TD$(7)="DEAD COMPUTER"
8300 IF RM=TL(8) THEN PRINT "A HORDE OF BATS CARRI
     ES YOU OUT.":RM=1:WD=-1
8310 GOTO 200
9000 DATA YOU ARE IN FRONT OF AN OLD FACTORY WITH
     A CLOCK TOWER.
9002 DATA YOU ARE AT THE BOTTOM OF A STAIRWELL.
9004 DATA YOU ARE AT THE TOP OF THE BASEMENT STEPS
     .
9006 DATA YOU ARE IN A DAMP CELLAR.
9008 DATA YOU ARE IN A STOREROOM.
9010 DATA YOU'RE IN THE CAFETERIA.
9012 DATA YOU'RE AT A LANDING ON THE STAIRS.
9014 DATA AROUND YOU IS A MANUFACTURING AREA.
9016 DATA YOU'RE AT A LANDING ON THE THIRD FLOOR.
9018 DATA YOU ARE IN THE COMPUTER ROOM.
9020 DATA YOU ARE INSIDE THE CLOCK TOWER.
9022 DATA YOU'RE AT THE TOP OF THE STAIRS.
9024 DATA YOU ARE IN A LONG CORRIDOR GOING EAST.
9026 DATA YOU'RE AT THE EAST END OF THE CORRIDOR.
9050 DATA N,1,S,2,E,3,W,4,U,5,D,6,GO,7,EAT,8,KICK,
     9,INSER,10,DEPOS,10,TYPE,11
9055 DATA TAKE,12,GET,12,DROP,13,THROW,13,INVEN,14
     ,I,14,MOUNT,15,READ,16
9060 DATA FIGHT,17,KILL,17,START,18,POWER,18,OPEN,
     19,QUIT,20,LOOK,21
9065 DATA WIND,22,EXAMI,23
9100 DATA RATS,TAPE,MACHI,TERMI,COIN,CANDY,COMPU,B
```

```
9105 DATA DIR,ADVEN,COPY,LOGOU,NORTH,SOUTH,EAST,WE
     ST,UP,DOWN
9150 DATA HUNGRY RATS,4,0,COMPUTER TAPE,5,1,VENDIN
     G MACHINE,6,0
9155 DATA BROKEN-DOWN TERMINAL,8,1,COIN,-1,1,CANDY
      BAR,-1,1,COMPUTER,10,0
9160 DATA BATS,13,0,DESK,14,0,COMPUTER MANUAL,-1,1
     ,ELABORATE CLOCKWORK,11,0
9200 DATA 2,0,0,0,0,0
9205 DATA 3,1,0,0,7,0
9210 DATA 0,2,0,0,0,4
9215 DATA 0,0,5,0,3,0
9220 DATA 0,0,0,4,0,0
9225 DATA 0,0,7,0,0,0
9230 DATA 0,0,8,6,9,2
9235 DATA 0,0,0,7,0,0
9240 DATA 0,0,10,0,12,7
9245 DATA 0,0,0,9,0,0
9250 DATA 0,12,0,0,0,0
9255 DATA 11,0,13,0,0,9
9260 DATA 0,0,14,12,0,0
9270 DATA 0,0,0,13,0,0
9999 END
```

carriable:
Computer Tape
Terminal
Coin
Candy
Manual

Program 2. Tower of Mystery, Modifications For Atari BASIC

Translation by Stephen Levy

```
2 GRAPHICS 0:SETCOLOR 1,0,12:SETCOLOR 2,0,4:
  SETCOLOR 4,0,4:POKE 752,1
5 POKE 82,0:PRINT "PLEASE WAIT WHILE I GET R
  EADY":PRINT :PRINT
15 DIM C1$(10),C2$(10),A$(41),B$(10),C$(15)
20 DIM AC(NR,6),VB$(NV*5),OB$(NO*5),RM$(NR*4
   1),TD$(NT*20),VN(NV),TL(NT),TF(NT)
30 RM$=" ":RM$(574)=RM$:RM$(2)=RM$:VB$=RM$:O
   B$=RM$:TD$=RM$
120 FOR I=1 TO NR:READ A$:RM$(I*41-40,I*41)=
    A$:NEXT I
130 FOR I=1 TO NV:READ A$,Q:VN(I)=Q:VB$(I*5-
    4,I*5)=A$:NEXT I
140 FOR I=1 TO NO:READ A$:OB$(I*5-4,I*5)=A$:
    NEXT I
150 FOR I=1 TO NT:READ A$,Q,Q1:TD$(I*20-19,I
    *20)=A$:TL(I)=Q:TF(I)=Q1:NEXT I
160 FOR I=1 TO NR:READ Q,Q1:AC(I,1)=Q:AC(I,2
    )=Q1
170 READ Q,Q1:AC(I,3)=Q:AC(I,4)=Q1:READ Q,Q1
    :AC(I,5)=Q:AC(I,6)=Q1:NEXT I
```

```
210 WD=0:PRINT RM$(RM*41-40,RM*41)
260 PRINT TD$(I*20-19,I*20)
320 IF AC(RM,I)<>0 THEN PRINT VB$(I*5-4,I*5)
    ;" ";
400 POKE 752,0:INPUT C$
405 IF C$="INVENTORY" THEN 2000
406 IF C$="LOOK" THEN 2610
440 A$=C$(I,I):A=ASC(A$):IF A>=97 AND A<=122
     THEN A$=CHR$(A-32)
460 IF X=0 THEN C1$(LEN(C1$)+1)=A$:GOTO 490
470 TRAP 490:C2$(LEN(C2$)+1,LEN(C2$)+1)=A$:T
    RAP 40000
510 Q=LEN(C1$):IF Q=5 THEN 515
512 FOR Q1=Q+1 TO 5:C1$(Q1,Q1)=" ":NEXT Q1:C
    1$=C1$(1,5)
515 Q=LEN(C2$):IF Q=5 THEN 520
517 FOR Q1=Q+1 TO 5:C2$(Q1,Q1)=" ":NEXT Q1:C
    2$=C2$(1,5)
530 IF VB$(C1*5-4,C1*5)=C1$ THEN C1=VN(C1):G
    OTO 600
620 IF OB$(C2*5-4,C2*5)=C2$ THEN 700
1930 TD$(1,19)="SLEEPY RATS":TL(6)=-1:RT=0
2020 IF TL(I)=0 THEN PRINT TD$(I*19-18,I*19)
     :K=1
8230 CD=-1:CA=0:TD$(115,133)="DEAD COMPUTER"
```

Program 3. Tower of Mystery, TI BASIC Version

Translation by Patrick Parrish

```
5 CALL CLEAR
10 NR=14
12 NT=11
14 NP=11
16 NV=29
18 NO=NT+NP
20 DIM AC(14,6),VB$(29),OB$(22),RM$(14),TD$(
   11),VN(29),TL(11),TF(11)
99 REM *INITIALIZATION*
100 RM=1
102 CT=0
104 CF=0
106 RT=-1
108 CA=0
110 MF=0
112 CD=0
114 WD=-1
116 LI=0
118 BT=-1
120 FOR I=1 TO NR
```

```
122 READ RM$(I)
124 NEXT I
130 FOR I=1 TO NV
132 READ VB$(I),VN(I)
134 NEXT I
140 FOR I=1 TO NO
142 READ OB$(I)
144 NEXT I
150 FOR I=1 TO NT
152 READ TD$(I),TL(I),TF(I)
154 NEXT I
160 FOR I=1 TO NR
162 READ AC(I,1),AC(I,2),AC(I,3),AC(I,4),AC(
    I,5),AC(I,6)
164 NEXT I
199 REM *MAIN LOOP: ROOM DESCRIPTION*
200 IF WD=0 THEN 400
210 WD=0
212 PRINT RM$(RM)
220 K=0
230 FOR I=1 TO NT
240 IF TL(I)<>RM THEN 270
250 IF K<>0 THEN 260
251 PRINT "YOU SEE:"
252 K=1
260 PRINT TD$(I)
270 NEXT I
300 PRINT "EXITS ARE:"
310 FOR I=1 TO 6
320 IF AC(RM,I)=0 THEN 330
322 PRINT VB$(I);" ";
330 NEXT I
340 PRINT
399 REM *MAIN LOOP: COMMAND INPUT AND PARSER
    *
400 INPUT C$
410 L=LEN(C$)
412 IF L=0 THEN 400
420 C1$=""
422 C2$=""
424 C2=0
426 X=0
430 FOR I=1 TO L
440 A$=SEG$(C$,I,1)
442 A=ASC(A$)
444 IF (A<97)+(A>122)THEN 450
446 A$=CHR$(A-32)
450 IF A$<>" " THEN 460
```

```
457 X=1
458 GOTO 490
460 IF X<>0 THEN 470
461 C1$=C1$&A$
462 GOTO 490
470 C2$=C2$&A$
490 NEXT I
500 IF C1$<>"" THEN 510
501 PRINT "EXCUSE ME?"
502 GOTO 400
510 C1$=SEG$(C1$,1,5)
512 C2$=SEG$(C2$,1,5)
520 FOR C1=1 TO NV
530 IF VB$(C1)<>C1$ THEN 540
531 C1=VN(C1)
532 GOTO 600
540 NEXT C1
550 PRINT "I DON'T KNOW THE VERB ";C1$
552 GOTO 400
600 IF C2$="" THEN 700
610 FOR C2=1 TO NO
620 IF OB$(C2)=C2$ THEN 700
630 NEXT C2
635 IF C1<>11 THEN 640
636 C2=1
637 GOTO 700
640 PRINT "I DON'T KNOW THE OBJECT ";C2$
642 GOTO 400
699 REM *PRELIMINARY FILTERING OF COMMANDS*
700 IF (C2>NT+5)*(C1<>7)*(C1<>11)THEN 640
710 IF (C2>NT)*(C2<NT+5)*(C1<>11)THEN 8010
720 IF C2>NT THEN 740
725 TLL=TL(C2)
730 IF (C1=11)+(TLL=RM)+(TLL=0)THEN 740
731 PRINT "IT ISN'T HERE."
732 GOTO 400
740 IF C1>10 THEN 760
750 ON C1 GOTO 1000,1000,1000,1000,1000,1000
    ,1100,1200,1300,1400
760 ON C1-10 GOTO 1500,1800,1900,2000,2100,2
    200,2300,2400,2500,9999,2600,2700,2800
999 REM *DIRECTIONS*
1000 IF AC(RM,C1)<>0 THEN 1010
1001 PRINT "YOU CAN'T GO THAT WAY."
1002 GOTO 200
1010 IF (RM<>4)+(RT=0)+(C1<>3)THEN 1020
1011 PRINT "THE RATS LOOK TOO FIERCE."
1012 GOTO 8200
1020 RM=AC(RM,C1)
```

```
1022 WD=-1
1024 GOTO 8200
1099 REM *GO*
1100 IF C2=0 THEN 8050
1105 IF C2<=16 THEN 8040
1110 C1=C2-16
1112 GOTO 1000
1199 REM *EAT*
1200 IF C2<>6 THEN 8010
1210 TL(C2)=-1
1212 PRINT "IT TASTES STALE."
1214 GOTO 8200
1299 REM *KICK*
1300 IF C2=0 THEN 8050
1305 IF (C2<>7)+(CD<>0)THEN 8020
1310 IF CA=0 THEN 1320
1311 CT=9
1312 GOTO 8200
1320 PRINT "THE COMPUTER STARTS UP!"
1330 PRINT "THE CONSOLE DISPLAYS : 'PLEASE L
     OG IN.'"
1332 CA=-1
1334 GOTO 8200
1399 REM *INSERT*
1400 IF C2=0 THEN 8050
1410 IF C2<>5 THEN 8010
1420 IF RM<>6 THEN 8030
1430 TL(5)=-1
1432 TL(6)=RM
1434 PRINT "A CANDY BAR COMES OUT."
1436 GOTO 8200
1499 REM *TYPE*
1500 IF C2=0 THEN 8050
1502 IF RM<>10 THEN 8030
1510 IF CA<>0 THEN 1520
1511 PRINT "THE COMPUTER ISN'T RUNNING."
1512 GOTO 8200
1520 IF LI<>0 THEN 1600
1530 IF C2=12 THEN 1540
1531 PRINT "'INVALID LOGIN ID.'"
1532 GOTO 8200
1540 LI=-1
1542 PRINT "'";C2$;" LOGGED IN.'"
1544 GOTO 8200
1600 IF CP<>0 THEN 1700
1605 IF C2>12 THEN 1610
1606 PRINT "'INVALID COMMAND.'"
1607 GOTO 8200
```

```
1611 PRINT "'COPY LOGOU ADVEN'"
1612 GOTO 8200
1620 IF C2<>14 THEN 1630
1621 PRINT "'WELCOME TO ADVENTURE! W#ULD Y#$
     $'"
1622 CT=9
1624 GOTO 8200
1630 IF C2<>15 THEN 1640
1631 PRINT "'MOUNT TAPE THEN TYPE FILE NAME.
     '"
1632 CP=-1
1634 GOTO 8200
1640 PRINT "'LOGGED OUT.'"
1642 LI=0
1644 GOTO 8200
1700 CP=0
1702 IF C2>12 THEN 1710
1703 PRINT "'NO SUCH FILE.'"
1704 GOTO 8200
1710 IF MT<>0 THEN 1720
1711 PRINT "'ERROR: TAPE NOT MOUNTED.'"
1712 GOTO 8200
1720 PRINT "THE TAPE SPINS..."
1722 FOR I=1 TO 500
1724 NEXT I
1730 PRINT "'FILE COPIED.'"
1740 IF C2<>14 THEN 8200
1741 PRINT "CONGRATULATIONS, YOU'VE DONE IT!
     "
1742 STOP
1799 REM *TAKE*
1800 IF C2=0 THEN 8050
1805 IF TF(C2)<>0 THEN 1810
1806 PRINT "THAT'S BEYOND YOUR ABILITY."
1807 GOTO 8200
1810 IF TLL<>0 THEN 1820
1812 PRINT "YOU ALREADY HAVE IT!"
1814 GOTO 200
1820 IF (C2<>4)+(CF<>0)THEN 1825
1821 PRINT "THERE WAS A COIN UNDER IT."
1822 TL(5)=RM
1823 CF=-1
1825 IF C2<>2 THEN 1830
1826 MT=0
1830 TL(C2)=0
1832 GOTO 8000
1899 REM *DROP*
1900 IF C2=0 THEN 8050
1905 IF TLL=0 THEN 1910
```

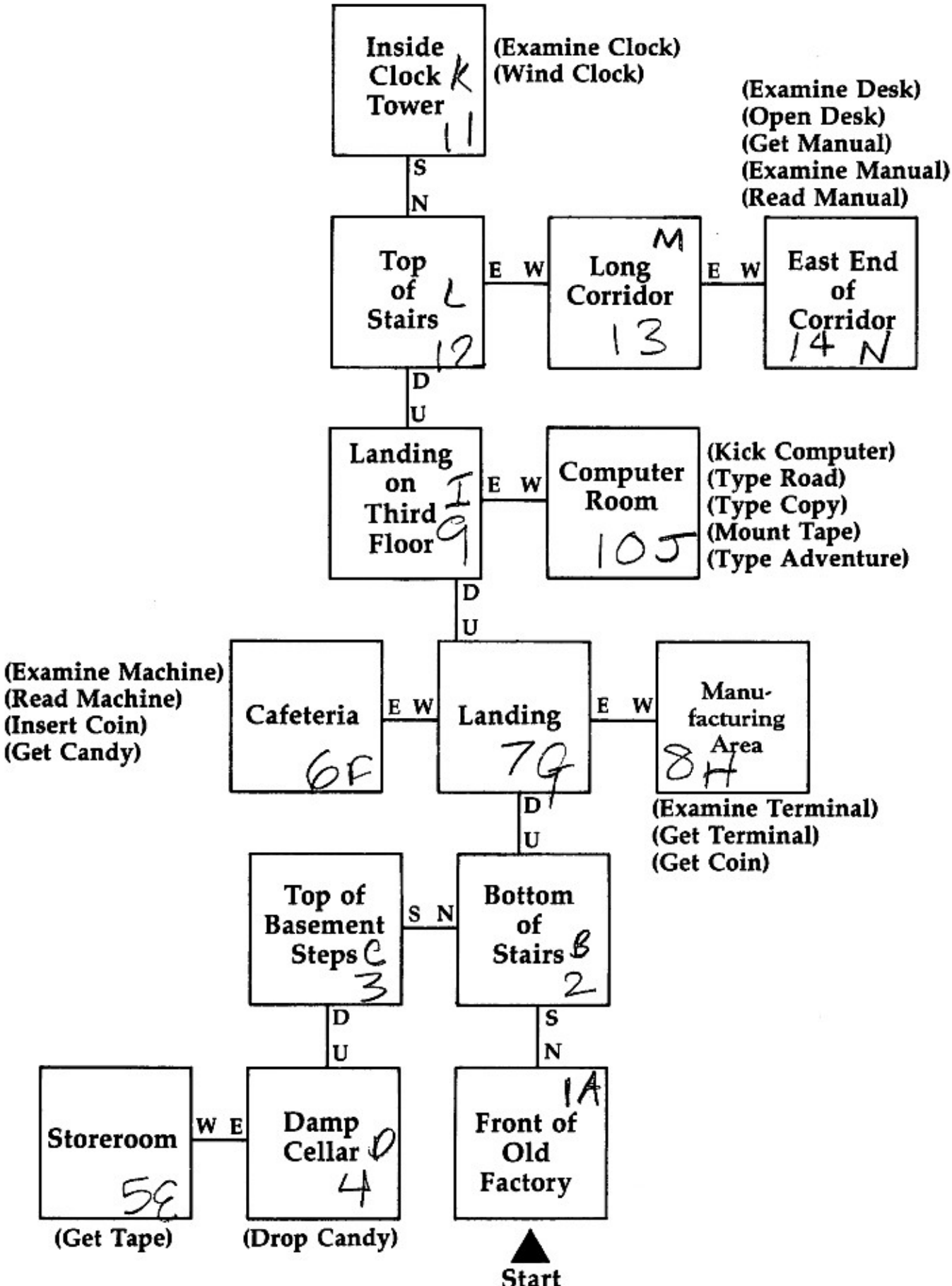
```
1906 PRINT "YOU DON'T HAVE IT."
1907 GOTO 200
1910 TL(C2)=RM
1912 IF (RM<>4)+(C2<>6)THEN 8000
1920 PRINT "THE RATS DEVOUR THE CANDY AND GE
     T SLEEPY."
1930 TD$(1)="SLEEPY RATS"
1932 TL(6)=-1
1934 RT=0
1940 GOTO 8000
1999 REM *INVENTORY*
2000 K=0
2002 PRINT "YOU ARE CARRYING:"
2010 FOR I=1 TO NT
2020 IF TL(I)<>0 THEN 2030
2022 PRINT TD$(I)
2024 K=1
2030 NEXT I
2040 IF K<>0 THEN 8200
2042 PRINT "NOTHING."
2050 GOTO 8200
2099 REM *MOUNT*
2100 IF C2=0 THEN 8050
2110 IF C2<>2 THEN 8010
2120 IF (RM<>10)+(MT<>0)THEN 8030
2130 TL(2)=RM
2132 MT=-1
2134 GOTO 8000
2199 REM *READ*
2200 IF C2=0 THEN 8050
2210 IF C2<>3 THEN 2220
2212 PRINT "'INSERT COIN.'"
2214 GOTO 8200
2220 IF C2=10 THEN 2230
2222 PRINT "NOTHING IS WRITTEN ON IT."
2224 GOTO 8200
2230 PRINT "'...USER ID IS ROAD...'"
2240 PRINT "'TYPE DIR FOR LIST OF COMMANDS..
     .'"
2250 PRINT "THE REST IS ILLEGIBLE."
2252 GOTO 8200
2299 REM *FIGHT*
2300 PRINT "THAT WON'T WORK."
2302 GOTO 8200
2399 REM *START*
2400 IF C2=0 THEN 8050
2410 IF C2=7 THEN 8020
2420 GOTO 8010
2499 REM *OPEN*
```

```
2500 IF C2=0 THEN 8050
2510 IF C2<>9 THEN 8010
2520 IF MF=0 THEN 2530
2522 PRINT "IT ALREADY IS."
2524 GOTO 8200
2530 PRINT "INSIDE IT IS A MANUAL."
2532 TL(10)=RM
2534 MF=-1
2536 GOTO 8200
2599 REM *LOOK*
2600 IF C2<>0 THEN 8040
2610 WD=-1
2612 GOTO 8200
2699 REM *WIND*
2700 IF C2<>11 THEN 8010
2710 IF BT<>0 THEN 2720
2712 PRINT "IT'S FULLY WOUND."
2714 GOTO 8200
2720 PRINT "THE CLOCK CHIMES DEAFENINGLY AND
      SOMETHING FLIES PAST."
2730 BT=0
2732 TL(8)=-1
2740 GOTO 8200
2799 REM *EXAMINE*
2800 IF (C2<>3)*(C2<>10)THEN 2810
2802 PRINT "SOMETHING IS WRITTEN THERE."
2804 GOTO 8200
2810 IF (C2<>9)+(MF<>0)THEN 2820
2812 PRINT "IT IS CLOSED."
2814 GOTO 8200
2820 IF C2<>4 THEN 2830
2822 PRINT "IT LOOKS BEYOND REPAIR."
2824 GOTO 8200
2830 IF C2<>7 THEN 2840
2832 PRINT "THIS IS AN ANCIENT MAINFRAME WIT
     H A CONSOLE."
2834 GOTO 8200
2840 IF C2<>11 THEN 2845
2842 PRINT "THERE IS A LARGE HANDLE FOR WIND
     ING THE CLOCK."
2844 GOTO 8200
2845 IF (C2<>2)+(MT=0)THEN 2850
2846 PRINT "IT IS MOUNTED ON THE COMPUTER."
2847 GOTO 8200
2850 PRINT "YOU SEE NOTHING SPECIAL."
2852 GOTO 8200
7999 REM *COMPLETION MESSAGES*
8000 PRINT "OK."
8002 GOTO 8200
```

```
8010 PRINT "THAT'S SILLY!"
8012 GOTO 200
8020 PRINT "NOTHING HAPPENS."
8022 GOTO 8200
8030 PRINT "YOU CAN'T DO THAT NOW."
8032 GOTO 8200
8040 PRINT "WHO'S YOUR ENGLISH TEACHER?"
8042 GOTO 200
8050 PRINT "PLEASE GIVE AN OBJECT."
8052 GOTO 200
8199 REM *COMPLETION ROUTINE*
8200 IF CA=0 THEN 8300
8210 CT=CT+1
8212 IF CT<10 THEN 8300
8220 IF RM<>10 THEN 8230
8222 PRINT "THE COMPUTER DIES WITH A LOUD PO
     P."
8230 CD=-1
8232 CA=0
8234 TD$(7)="DEAD COMPUTER"
8300 IF RM<>TL(8)THEN 200
8302 PRINT "A HORDE OF BATS CARRIES YOU OUT.
     "
8303 RM=1
8304 WD=-1
8310 GOTO 200
9000 DATA YOU ARE IN FRONT OF AN OLD FACTORY
      WITH A CLOCK TOWER.
9002 DATA YOU ARE AT THE BOTTOM OF A STAIRWE
     LL.
9004 DATA YOU ARE AT THE TOP OF THE BASEMENT
      STEPS.
9006 DATA YOU ARE IN A DAMP CELLAR.
9008 DATA YOU ARE IN A STOREROOM.
9010 DATA YOU'RE IN THE CAFETERIA.
9012 DATA YOU'RE AT A LANDING ON THE STAIRS.
9014 DATA AROUND YOU IS A MANUFACTURING AREA
     .
9016 DATA YOU'RE AT A LANDING ON THE THIRD F
     LOOR.
9018 DATA YOU ARE IN THE COMPUTER ROOM.
9020 DATA YOU ARE INSIDE THE CLOCK TOWER.
9022 DATA YOU'RE AT THE TOP OF THE STAIRS.
9024 DATA YOU ARE IN A LONG CORRIDOR GOING E
     AST.
9026 DATA YOU'RE AT THE EAST END OF THE CORR
     IDOR.
9050 DATA N,1,S,2,E,3,W,4,U,5,D,6,GO,7,EAT,8
     ,KICK,9,INSER,10,DEPOS,10,TYPE,11
```

```
9055 DATA TAKE,12,GET,12,DROP,13,THROW,13,IN
     VEN,14,I,14,MOUNT,15,READ,16
9060 DATA FIGHT,17,KILL,17,START,18,POWER,18
     ,OPEN,19,QUIT,20,LOOK,21
9065 DATA WIND,22,EXAMI,23
9100 DATA RATS,TAPE,MACHI,TERMI,COIN,CANDY,C
     OMPU,BATS,DESK,MANUA,CLOCK,ROAD
9105 DATA DIR,ADVEN,COPY,LOGOU,NORTH,SOUTH,E
     AST,WEST,UP,DOWN
9150 DATA HUNGRY RATS,4,0,COMPUTER TAPE,5,1,
     VENDING MACHINE,6,0
9155 DATA BROKEN-DOWN TERMINAL,8,1,COIN,-1,1
     ,CANDY BAR,-1,1,COMPUTER,10,0
9160 DATA BATS,13,0,DESK,14,0,COMPUTER MANUA
     L,-1,1,ELABORATE CLOCKWORK,11,0
9200 DATA 2,0,0,0,0,0
9205 DATA 3,1,0,0,7,0
9210 DATA 0,2,0,0,0,4
9215 DATA 0,0,5,0,3,0
9220 DATA 0,0,0,4,0,0
9225 DATA 0,0,7,0,0,0
9230 DATA 0,0,8,6,9,2
9235 DATA 0,0,0,7,0,0
9240 DATA 0,0,10,0,12,7
9245 DATA 0,0,0,9,0,0
9250 DATA 0,12,0,0,0,0
9255 DATA 11,0,13,0,0,9
9260 DATA 0,0,14,12,0,0
9270 DATA 0,0,0,13,0,0
9999 END
```

Figure 11-1. Map of Tower of Mystery



The Edge of the Future

The Edge of the Future

e are standing at the beginning of a new fusion of technology and literature. The best computer adventures that have appeared so far will undoubtedly appear crude and primitive in years to come, as the art of interactive computer storytelling advances. On the one hand, more powerful computers will become available for home use; on the other, authors will be able to build on their experience and create more sophisticated adventures.

Infocom's Stu Galley has aptly compared the present state of computer adventures to the early days of movies. In those days, the people who made the technology also made the movies. They would film anything that moved—trains, crowds of people, animals—and people would eagerly pay to see these shows just for the novelty. Later came the division of labor between technologists and dramatists, which resulted in the creation of movies that were worth seeing for their content.

Computer adventures—interactive fiction, if you will—are still in the process of moving toward that division of labor. Michael Berlyn, author of *Cyborg, Suspended,* and *Infidel,* is a pioneer in this area, being an adventure writer who is not also a programmer. Infocom's *Sea Quest,* still under development as this book is being written, involves a collaboration with a writer of young people's books to create a story reminiscent of the adventures of Tom Swift.

Well-known science-fiction writers have shown an interest in the adventure game format. Larry Niven and Jerry Pournelle made a start at devising an adventure based on their novel *Inferno,* although they have said little about it recently. Fred Saberhagen, author of the Berserker series of science-fiction stories and novels, is cofounder of a business called Berserker Works Limited, whose purpose is to convert well-known science-fiction and fantasy works into computer games. Again, as this is being written, nothing has actually reached the public.

Advancing from the present state to quality literature, though, will involve more than just better writing. Advances

in technology are also needed. The first movie-maker might have imagined filming a performance of *Hamlet*, but then he would have realized the difficulties. First, there would be no dialogue. Second, the presence and three-dimensionality of the stage performance would be lost. Overcoming the first problem required adding a new capability to film. Overcoming the second required more imaginative use of the existing technology through editing, changing camera angles, and so on. It required understanding how to use the medium effectively.

Similar advances will have to be made if computer adventures become serious plot vehicles. These advances will undoubtedly include many trimmings to enhance the impact of the events that are depicted. Graphics will advance to the point of presenting real atmosphere and information. Sound for both vocal and musical effects will undoubtedly follow. (Imagine really having a "hollow voice" emerge from the shadows of *Adventure*.)

The video disk may open the door to many of these effects. Video disks have already provided the basis for several games, and the potential of a computer-controlled laser disk unit is enormous. The games so far have provided simple multiple-choice alternatives (*Murder, Anyone?*) or joystick action (*Dragon's Lair*), neither of which leaves much scope for the player's imagination. But the interactive format of adventures could put these images to a more complex use, showing key scenes of the story in response to the player's commands.

Speech synthesis provides another path to enhanced special effects. Speech synthesis programs or peripherals are currently available for many computers, and with a little help on ambiguously pronounced words (like *bow* and *live),* such synthesis could add a great deal to adventure games.

Voice input is a little further off. Hardware is available today to let a computer recognize a few dozen words after it's been "trained" to recognize a particular individual's pronunciation of them. But recognition of hundreds of words, spoken by people with diverse voice qualities, intonations, and accents, will require some major advances.

But once voice recognition and speech synthesis are fully developed, the impact on adventure games could be tremendous. It will be possible for a player to have an actual dialogue with the computer instead of going through the keyboard and screen. The computer could assume the voices

of the various actors; it could be polite, threatening, and pleading in turn.

The Decision Tree

More than anything else, adventures need to grow in their ability to interact with the player. If the current trends hold, the most important features of computer adventures in coming decades will be great flexibility in the alternatives available to the player and great scope for the imagination.

Acquiring this flexibility and scope requires a huge decision tree. A decision tree is a set of alternatives that branch out again and again at successive junctures. A simple game like tic-tac-toe has a small decision tree: nine branches at the root, eight (for the eight remaining empty spots) coming from each of those nine branches, and so on. Chess has a larger decision tree, since dozens of different moves may be legal from a given position.

Decision trees can be massive things. Even the simple tree of tic-tac-toe has tens of thousands of possible paths; if you always played to fill the grid, instead of stopping when someone won, there would be 362,880 possible different games. The chess decision tree is so large that if the whole universe were a computer, with each atom storing a bit, it couldn't store the whole tree. Yet the decision tree for an adventure game is potentially even larger, since each possible command on each turn represents a new branch. The only trouble is that, because of computer limitations, most of these branches don't lead to anything interesting.

The adventure of the future will explore more of its potential tree. It will have more permissible actions, more crucial decisions, more things going on in the background, and more posssible endings than anything feasible today. Maybe this won't be true; maybe the market will decide that some other feature is more important. But you have to start with some assumptions, and this one is as reasonable as any. So let's see what path might lead to this kind of super-adventure.

Will visual effects dominate the game? It's unlikely. Recording every possible scene and action, either with actors or in animated form, would take far too much memory capacity. The computer might be able to generate scenes on the fly from stock picture elements, but it would take some really

good processing to avoid the stiff-bodied quality of Saturday morning cartoons.

Words are, and will always be, the best medium for conveying a multiplicity of alternatives. This doesn't necessarily mean text; perhaps your grandchildren will use a microphone to instruct their character to "open the box with the silver key." It does mean that words, in one form or another, will remain important both for controlling the game and for getting feedback from it.

Perhaps the main story line will be recorded visually, and the player will conduct a dialogue with a robot to advance the story. This robot would be argumentative, explaining to the player why certain actions don't accomplish anything useful, and would present the next scene only when the player gives a command that materially changes his surroundings.

For now, let's concentrate on this robot. Whatever audio-visual aids are included, the quality of the game is limited by its ability to interact with the player. So we can concentrate on the dialogue and ask what it might be like in "Zork XV." Possibly it would go something like this:

>GO NORTH

You are moving through a dense forest. Trees and shrubs of many kinds grow around you, with oaks and maples dominating. There is no clearly marked path here, so travel is slow.

[Pause]

You travel north for a couple of hundred yards and nothing much changes. There are sounds of small animals moving through the brush and of birds somewhere in the distance.

[Pause]

As you continue north, the forest begins to grow less dense. To the west the brush is more brightly lit.

[You could keep watching the descriptions roll by indefinitely until you reached some obstacle. But at this point you decide to change your course.]

>WEST

You go to the west about a hundred yards and come into a clearing some 40 yards wide. In the middle of the clearing is a small log house.

>WHAT IS THE HOUSE LIKE?

The house is built of logs, one story tall. It has several small windows of oiled paper. Smoke is rising from its chimney. It has an en-

[The program pauses at this point. If the player waited, he might get more details in the description of the house, or something might start happening. Rather than wait, he decides to...]

>KNOCK ON THE DOOR

The only sound in the area comes from the birds singing. A swallow flies over the house.

[Pause...but the player is patient.]

At last the door swings open, and you see a gray-haired man, old but not feeble, looking inquiringly at you. He is dressed in old, tattered garments. Beyond him, you can see a few pieces of simple wooden furniture in the cabin.

"Greetings," says the man uncertainly. "Who are you, and why are you passing through these woods?"

[Pause...and the player has let his mind wander.]

"Well, what do you want? Speak up!"

>I AM AN ADVENTURER, SEEKING THE LOST PALANTIR.

The man's eyebrows rise in amazement. "The Palantir? How did you come to know of that ancient relic, and why are you looking here? But I will ask for no long explanations. I need only one proof of your good intentions before I will give you the information you need. Leave your sword with me; you will have no further need of it until you return."

Places for Growth

As this example shows, what is important is the way the program can interact with the player's commands. Increasing the scope of future adventures will depend on making further improvements in that area. What sort of advances can we look for? Here are a few.

A more realistic presentation of the environment. A most peculiar aspect of today's adventure programs is that they invariably present a world divided up into rooms. That is, the character is always in one location, which he can leave or enter only by discrete paths. He goes north and ends up in the next room. Advancing just to the doorway is impossible, unless the doorway is marked as a room. In addition, each room contains a fairly small number of indivisible objects and nothing else.

In real life, you can move in any direction until you reach some barrier; you can also go halfway there or walk in a circle near the place you started. When you do this, you'll usually find large numbers of objects that might be useful to you,

depending on what you're doing. What's more, the objects that are present don't normally call themselves to your attention; instead, they might remain undiscovered until you do a little searching.

Making the environment more of a continuum would require quite a different approach from that of the current room-based type of adventures. Rooms are convenient for a computer representation because they allow binary decisions: either an object is where you are, or it isn't, with no ambiguity about what "where you are" means. Having things pinpointed by spatial coordinates would require the program to do more work in checking its data structure for objects that are near enough to affect the character. Devising data structures that make this operation reasonably efficient will be vital to improving the adventurer's environment.

Adding this kind of realism is an area in which Infocom is hard at work for its next generation of adventure programs.

An increased sense of continuous action. Action made of separate turns also limits the realism of an adventure. Major steps have already been taken in improving the time flow in *Deadline* and *Witness,* and less obviously in *Enchanter.* In each of these adventures, time passes in the game world according to how long it takes to perform an action. The time to solve the problem eventually runs out, or else night follows day, and the demands of hunger and fatigue affect what the character must do.

One way to get a more realistic time sense would be to have the clock continue to tick whether the player enters commands or not. This wouldn't be the best approach, however. It would put too great a sense of urgency on what should be a relaxed activity; besides, game time shouldn't pass through interesting and uninteresting events at the same rate.

Another approach, as illustrated in the earlier example, would be to let the character continue on a course of action for some time, giving him descriptions until something important happens or he stops the flow. *Deadline* does this with the WAIT command, which makes the program describe ongoing events and ask the player if he wants to keep waiting after each event. This principle could be expanded to let the player indicate a general course of action and stop it only when he saw a reason to do something else.

Better simulation of characters. Computer pioneer A. M. Turing said that people would agree that a computer could "think" if it could imitate a human being well enough to fool people. Certainly, realistically simulating human behavior is one of the toughest things to do on a computer. Covering the full range of human action is too much to ask of any program we're likely to see soon.

But it isn't beyond hope to expect future adventure programs to include characters that can answer questions on a limited range of subjects and can react to the player's character. It would be a major step forward to be able to ask characters questions like, "Why did you invite me?" or "Do you think Phong is lying?" Beyond that might come programs that let the player make statements (such as, "I am seeking the Palantir," or "You are in danger.") and then observe the reaction of the character he is addressing.

Characters in adventure games should also react to the player's character—not just to one statement or action at a time, but to what he shows himself to be. No adventure program to date has gone very far in making the player-character's actions have a long-term effect on how the other characters will respond to him. Suspects will answer the same question four times in a row. An abusive detective will not find people refusing to talk to him ten turns later, even if they walk out of the room when he insults them.

Adding this sort of feature, on a primitive level, should not be very difficult. The program could maintain an *attitude variable* for each character, which is affected one way or the other every time the player's character interacts with him. Once this variable fell below a certain level, the character would just become very uncooperative, answering all questions with, "Figure it out yourself!" or the like. Having the value rise high enough could cause the character to volunteer new information or offer help. This would force the player not only to solve problems but to develop his character consistently, too.

A wider range of alternatives. Early adventures had only one real ending. They could, of course, end unsuccessfully in many ways; but these conclusions were just a matter of running afoul of some hazard and getting killed. Unlike the true ending, they didn't represent a culmination of events.

The Edge of the Future

Deadline was the first adventure to provide a number of well-developed alternative endings. The detective could arrest the wrong suspect (or the right one without enough evidence), and he would get a message explaining why the suspect was not convicted at the trial. He could also exhaust the 12-hour limit without making an arrest, in which case he would be driven off into the sunset with nothing to show but a sense of dissatisfaction.

A further step would be to provide a variety of successful conclusions to an adventure, each reached by a different path, leaving to the player's judgment which of these endings is the most satisfactory. All of the conclusions would represent solutions to the main problem of the story, but in different ways and at different costs.

For instance, if the adventure gave the player the task of getting revenge against an enemy, one set of choices might lead not only to the death of the enemy but also of a friend. Another path might lead to merely humiliating the enemy while avoiding any tragic side effects. One ending might suggest the other, letting the player know that there are more alternatives, something like this:

The next day, Pierre brings you the news that Maraud has left France and abandoned his holdings. You feel a deep satisfaction at having driven the imposter away, although you wish he had not escaped with the ring. Perhaps you could have recovered it if you had moved more decisively, yet such rashness might have turned to ill. It's just as well, you decide, that people cannot go back and change their past actions.

While future adventures will undoubtedly allow more alternatives than present ones do, they will still run into the fact that the world includes more possibilities than any computer could ever represent. Currently, the restrictions are often very artificial. In *Adventure*, directions are distorted in the wilderness, so you can never get very far from your starting point. *Infidel* is set in the middle of an infinite desert. *G.F.S. Sorceress* has a spaceship with automatic controls that can take you only to certain planets.

Dungeons and spaceships are convenient for today's adventures, because they impose naturally restrictive environments. But if interactive stories are going to break the bounds of these settings, they will have to deal openly with the fact that the player might reasonably want to do things that the

program can't handle. A minimum would be to say simply, "I'm not programmed to handle that."

A further step would be to go into a different mode for handling activities outside the program's map. For instance, if the player tried to leave a house which is the scene of the main activity, the program could put up a menu of activities that are possible outside the house. These might include "Go shopping," "Go home to sleep," "Quit," and so forth. The range of alternatives in the menu might vary depending on the player's situation. Some of the choices might even be to travel to other areas where whole new maps are brought into play.

The more things that can happen in an adventure, the greater the burden is on the author. It isn't just memory limitations that keep every possible action from having a reasonable result; the author's ability to anticipate is also involved. For instance, the player might try to use something in a way the author never thought of but that might reasonably do the job. Nothing will ever replace thorough testing by imaginative players as a way to solve this problem. But if the author has programming aids that let him concentrate on these situations, then his job is much easier.

The key to letting the author be selective is the use of default actions. Even the simplest adventures use defaults, but only in a simple way. Obvious examples are having an object become part of your inventory when you pick it up or having a door open when you give that command. But much more than this is possible, at least in principle. The adventure program of the future will have to describe its world generically, with the writer intervening only when something distinctive is expected to happen. It will have to contain data about the usual characteristics of common objects, so that doing ordinary things with them will work.

The adventure writer of the future should be able, for instance, just to put a chair into a location and have the program recognize that it can be sat in, that only one person at a time can usually sit in it, and that it's movable but not something you can put in your pocket. Furnishing a room becomes a simple matter, and letting the player rearrange the room requires no extra effort. This leaves him or her free to concentrate on any unusual items in the room.

Handling items generically will, of course, have its limits.

Anything described this way will lack the color and individuality of objects that the author takes the trouble to describe in detail; this can serve as a clue to the player about which objects are useful. But having a few well-described objects and a lot of sketchily described ones would be a major advance from today's room, which has only a few usable objects in it. Providing an adventure with more ordinary items would help to round out the game's environment.

The Artist's Tools

The adventure writer of the future should not have to be a programmer at all—at least, no more than the user of a good data-base system has to be a programmer today. He will have to understand the concept of a decision tree and sketch out his story in appropriate terms. But from there, he will not have to touch any programming language. If he is a free lancer, he might first put his adventure on a home adventure-development system. This system would not enable him to get in all the details of a real, published adventure; but it would let him thoroughly plot the story and all the significant alternatives.

Next, this writer might present his adventure to a publishing house for consideration. The editorial staff would look at his home-developed version; if they liked it, they would buy it and proceed to give it full development on their professional system. The progression would be similar to that of turning a screen play into a movie. The actual programming would not be done for any particular adventure, but for the development kit that allowed the writer to create adventures.

This kit would undoubtedly consist of several different parts. One would let the writer create nonplayer characters. He would give the characters names, indicate where they start and how they move about, and define their roles. He would create every line of dialogue that the characters might utter and would specify the conditions that would prompt each line. These lines would include responses for general "don't know" or "don't care" situations. He would indicate how the characters interact with each other.

Another part would deal with the map of the adventure. Here, the author would define areas (not *rooms)* that the player's character can enter. He would include the description of as many different points of view within the area as he cared to. He might also include a number of throwaway lines, which

the final program would use randomly within an area to describe incidentals like birds singing or trucks going by. He would fully describe any events triggered by the character's entry into a room.

Inanimate objects would also be taken care of. The writer would give their weight and size and provide a brief description. When applicable, he would state what predefined class an object fell into. He would specify what actions would produce unusual results—and, of course, he would give an exact description of those results, both in text and in terms of its effect in the game world.

The development kit would include a *debug mode* in which the writer could play the adventure for a while, setting up situations to try out, then go back into one of the stages to change or add something. The final shape of the adventure would be reached only after many cycles of playing and reshaping. This should not require programming skills, but it would, inevitably, require the ability to think logically and imaginatively at the same time. But then so does any good writing.

All the things discussed so far are possible with today's computer technology and will be economically feasible in the next several years. But what lies beyond that? What might new scientific discoveries do for the descendants of *Adventure*? Here we must enter the world of science fiction.

Vernor Vinge's novella *True Names* (published as half of Dell's *Binary Star #5*) presents an intriguing view of this future. The main character is an author of computer "participation novels." But the really intriguing part of Vinge's speculation lies in the next step beyond those novels: the Portals. These are described as data bases that are accessed with electroencephalograph (EEG) terminals, which provide sensory cues that let the user experience a complete environment with which he or she interacts. The adventurer, with a certain amount of help from his imagination, actually experiences traveling on paths, facing monsters, and dealing with the personas of other participants. A programming language (called Psylisp) allows the computer to present characters that are difficult to distinguish from real people.

Dream Park, by Larry Niven and Steven Barnes, suggests that in the next century, computer programs, holography (projection of 3-D images), and live acting will be combined to

let players participate in adventures with an entire environment set up for them. The game as presented is closer to Dungeons and Dragons than to computer adventure. However, the use of computers to control realistic sound and visual effects suggests the extent to which they may someday put players "in the story" even without using props and actors.

All these speculations about the future will undoubtedly be proved wrong in many details. But, barring global catastrophe, one thing is certain; the computer games of the future are going to be so advanced that they will make today's best efforts look feeble by comparison.

Before the 1950s, the only computers were experimental models that cost fortunes and could be used only by specialists. Until the sixties, no one considered the idea of interacting with a computer for entertainment; computer time was just too expensive. By the end of the decade, though, students were using the machines to play a wide assortment of board games—and in the seventies things really got going. Between 1975 and 1980 came personal computers, *Adventure, Zork,* the first Scott Adams adventures, and Infocom's first personal computer adventures. No science-fiction writer would ever have dared to suppose that a completely new form of entertainment would leap so far forward in just five years, when the technology for it didn't even exist at the start of that period.

The eighties, so far, haven't yielded advances to match that sudden period of growth. Infocom has continued to refine its game system, but the rest of the industry hasn't produced much in the way of technical advances, aside from better use of graphics.

Another burst may be coming soon, however, as serious writers begin to take notice of the format. It may not be long before computer games cross over the threshold of literature—before the first computer adventure is written that stands up not only as a game, but as a story worth experiencing as a story. Such a work will really deserve to be called an interactive novel. It will be the first literary work in history to demand the reader's active involvement and to provide the main character with the ability to make genuine choices. In a novel or play, the *writer* makes all the choices for the characters. Experimental theatrical productions have tried to give actors choices and involve the audience; but when this is done on anything but a very limited basis, the result is chaos. Books

that instruct the reader to turn to one of several possible pages do provide a choice, but not active involvement. The reader is limited to the alternatives that are spelled out, and he has no chance to think of a course of action on his own.

The interactive novel will create a new kind of rapport between the author and the reader. It will let the reader say, in effect, "Yes, but what if the character did this?"—and see the author's answer. It will impose a new burden on the author in the process; he or she will have to spell out the consequences of not just one set of actions, but a multitude of choices that the main character might make.

When will this happen? That's a dangerous question. But there is one safe answer: sooner than you think.

Suggested Reading

Adams, Scott. "Pirate's Adventure." *Byte*, December 1980.

Addams, Shay. "The Wizards of Infocom." *Computer Games*, February 1984.

"Call Yourself Ishmael: Micros Get the Literary Itch." *Softline*, September-October 1983. (Includes short articles by Fred Saberhagen, Robert Lafore, Scott Prussing, Redmond Simonsen, Marc Blank, and Michael Berlyn on text adventures and interactive fiction.)

Dacosta, Frank. *Writing Basic Adventure Programs for the TRS-80.* Tab Books, Inc., 1982.

Holmes, J. Eric, M.D. *Fantasy Role Playing Games.* Hippocrene Books, 1981.

Lebling, P. David, Marc S. Blank, and Timothy A. Anderson. "Zork: A Computerized Fantasy Simulation Game." *Computer*, April 1979.

Lebling, P. David. "Zork and the Future of Computerized Fantasy Simulations." *Byte*, December 1980.

McClung, M.M. "Is It Fun? An Interview with Infocom's Michael Berlyn and Marc Blank." SoftSide, #44, September 1983.

Peterson, Dale. *Genesis II: Creation and Recreation with Computers.* Reston Publishing Company, 1983.

Strehlo, Kevin. "Getting into Adventure Games." *Personal Software*, February 1984.

Weizenbaum, Joseph. "Eliza—A Computer Program for the Study of Natural Language Communication Between Man and Machine." *Communications of the ACM*, Volume 9, No. 1, January 1966.

Winston, Patrick. *Artificial Intelligence.* Addison-Wesley, 1977.

Index

Notes

Notes

Notes

Notes

Notes

Notes

If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!** Magazine. Use this form to order your subscription to **COMPUTE!.**

For Fastest Service,
Call Our **Toll-Free** US Order Line

800-334-0868

In NC call 919-275-9809

COMPUTE!
P.O. Box 5406
Greensboro NC 27403

My Computer Is:
☐ Commodore 64 ☐ TI-99/4A ☐ Timex/Sinclair ☐ VIC-20 ☐ PET
☐ Radio Shack Color Computer ☐ Apple ☐ Atari ☐ Other _____
☐ Don't yet have one...

☐ $24 One Year US Subscription
☐ $45 Two Year US Subscription
☐ $65 Three Year US Subscription

Subscription rates outside the US:

☐ $30 Canada
☐ $42 Europe, Australia, New Zealand/Air Delivery
☐ $52 Middle East, North Africa, Central America/Air Mail
☐ $72 Elsewhere/Air Mail
☐ $30 International Surface Mail (lengthy, unreliable delivery)

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

☐ Payment Enclosed ☐ VISA
☐ MasterCard ☐ American Express
Acct. No. _____ Expires ___/___

COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**

800-334-0868
In NC call 919-275-9809

Quantity	Title	Price	Total
_____	The Beginner's Guide To Buying A Personal Computer	**$ 3.95***	_____
_____	COMPUTE!'s First Book of Atari	**$12.95†**	_____
_____	Inside Atari DOS	**$19.95†**	_____
_____	COMPUTE!'s First Book of PET/CBM	**$12.95†**	_____
_____	Programming the PET/CBM	**$24.95‡**	_____
_____	Every Kid's First Book of Robots and Computers	**$ 4.95***	_____
_____	COMPUTE!'s Second Book of Atari	**$12.95†**	_____
_____	COMPUTE!'s First Book of VIC	**$12.95‡**	_____
_____	COMPUTE!'s First Book of VIC Games	**$12.95†**	_____
_____	COMPUTE!'s First Book of Atari Graphics	**$12.95†**	_____
_____	Mapping the Atari	**$14.95†**	_____
_____	Home Energy Applications On Your Personal Computer	**$14.95†**	_____
_____	Machine Language for Beginners	**$12.95†**	_____

* Add $1 shipping and handling. Outside US add $5 air mail; $2 surface mail.
† Add $2 shipping and handling. Outside US add $5 air mail; $2 surface mail.
‡ Add $3 shipping and handling. Outside US add $10 air mail; $3 surface mail.

Please add shipping and handling for each book ordered. _____

Total enclosed or to be charged. _____

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

☐ Payment enclosed Please charge my: ☐ VISA ☐ MasterCard
☐ American Express Acc't. No. _____ Expires ___/___

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Allow 4-5 weeks for delivery.

Words Worth a Thousand Pictures

You are in a dimly lit canyon, surrounded by swirling mists and hemmed in by towering stone walls. There is a dark pit to the north, and beyond it the glitter of gold. Behind you, from the south, come the shouts of the approaching horde....

What do you picture when you read those lines? It's up to you to decide. You read the words and your mind supplies the images. That's part of the magic of fiction.

That's also the magic of text adventures.

Combining the descriptive power of prose, the speed of your computer, and your own imagination, text adventures can provide hours of entertainment. *COMPUTE!'s Guide to Adventure Games* shows you how those games are created and how to create your own.

What makes a good adventure? You'll learn what goes into a good adventure game, what should be left out. Then, since many beginning text adventurers play commercial adventures first, a number of the most popular are reviewed so that you'll be able to buy with confidence. You'll also find reviews of several graphic adventures, games that combine the mental imagery of a text adventure with the visual excitement of good graphics displays.

But there's nothing like the thrill of creating text adventures of your own. This book shows you how. There is even a complete adventure program—"Tower of Mystery"—that demonstrates how the programming techniques are applied. It's ready to type in and run on almost every home computer.

Like other COMPUTE! books, *COMPUTE!'s Guide to Adventure Games* is a perfect combination of clear writing and solid information. With it, and your imagination, you can use your computer to explore an infinite number of new worlds—worlds inside your computer and in your mind.

ISBN 0-942386-67-1

COMPUTE!'s Guide to Adventure Games