THE COMPUTER & VIDEO GAMES

BOOK OF ADVENTURE

by KEITH CAMPBELL

With listings for: SPECTRUM, COMMODORE 64 and BBC

Foreword by Scott Adams

THE COMPUTER & VIDEO GAMES BOOK OF ADVENTURE

by Keith Campbell

MELBOURNE HOUSE

Published in the United Kingdom by:
Melbourne House (Publishers) Ltd.,
Church Yard,
Tring, Hertfordshire HP23 5LU.
ISBN 0 86161 143 8

Published in Australia by:
Melbourne House (Australia) Pty. Ltd.,
Suite 4, 75 Palmerston Crescent,
South Melbourne, Victoria, 3205.

Published in the United States of America by:
Melbourne House Software Inc.,
347 Reedwood Drive,
Nashville TN 37217.

Printed in Hong Kong by Colorcraft Ltd.
9 8 7 6 5 4 3 2 1 0

ACKNOWLEDGEMENTS

My sincere thanks to:

Terry Pratt
Editor of Computer & Video Games, whose professional eye spent many hours checking the draft. His faith in this book kept me going — I hope it was justified!

Steve Willis
Colleague at Seeboard and co-translater of my TRS-80 Adventures into BBC Basic, who painstakingly checked the program and references to it.

Ruth, Bruce, Neil and Veronica
My wife and children, who claim to have solved more Adventures than I, and whose telephone calls and TV programs were wrecked many times by my noisy line printer!

Keith Campbell
June 1983

[illegible]
1991

CONTENTS

FOREWORD

PUDDING! Pudding? PUDDING!

In the musical Hello Dolly, Barnaby and Cornelius have decided to go to the Great City of New York and have the Adventure of their life. To be on the safe side they want to have a secret password, to use to tell the other that they are indeed in an Adventure! You guessed it! Their secret word is pudding.

I've always thought that computer Adventures are indeed very like a traditional pudding. From the outside both present a smooth even exterior with nary a look at what is really inside. A good pudding like a good Adventure requires a basic recipe and then a chef to put it all together. With the best recipe in the world and a poor chef a pudding can turn out as a disaster, but a great cook can turn the most basic recipe into a masterpiece.

You will find inside this book a wonderful recipe for baking your own Adventures. Only by trying will you find out what quality Adventure Chef you really are. But no matter if you are a Short Order or Cordon-Bleu you will find the baking process fascinating! And as they say: The proof is in the pudding . . .

December 1983
Scott Adams
Longwood Florida
President
Adventure International

CHAPTER 1
What is Adventure?

I am in a Ghost Town. I can see a barbershop and a jail.

"GO BARBERSHOP"

I find myself in a barbershop and can see a stetson hat.

"WEAR STETSON"

OK — odd, something seems strange.

Everywhere is deserted, yet somewhere in the distance a bell rings. A ghostly voice whispers

"VAIN"

I shudder and make for the saloon down the street. Deserted. No chance of a drink to steady my nerves — all I can see is a bell and a mirror.

"LOOK MIRROR"

OK — very pretty.

I get annoyed —

"BREAK MIRROR"

Crack! Flying glass slices me. I'm dead. My Adventure is over.

In the beginning there were stories. Some stories were of excitement and suspense, mystery and intrigue. Such stories were called "Adventure Yarns", and, as with all fiction, transported the attentive reader to a fantasy world created by the author. The reader often became so absorbed in the story that he found it quite impossible to put down the book. Being so caught up in the excitement of the plot, he had an overwhelming desire to see how the hero would fare next. He didn't want to break the spell of fantasy created by the story; putting the book down would be similar to being awoken before the end of a fascinating dream!

But how often might the reader put down the book and think "If only . . . " If only the hero had taken a different course of action from that described by the author, then the whole outcome would have been better.

Although the illusion of realism is so well created that the reader feels that he is in the shoes of the hero, however hard he might try to move those shoes along his own path, they will refuse to co-operate and will follow a set route. The story has a fixed narrative and conclusion. Be it happy or sad, good or evil, it is written, take it or leave it. How could things be any different?

Throughout the ages man has played games for relaxation, enjoyment and excitement. The outcome of a game is not predetermined, and in this way it differs from a story, although there are strong similarities between the two. Many games have an element of fantasy, in which each player takes the role of a character quite different from that which he plays in life.

There are many different fantasy games, but perhaps the best known is a game called Dungeons and Dragons. Here the players travel through a network of unknown "dungeons". A Dungeonmaster is the author of the game. He populates his dungeon with many different creatures, both benign and malevolent. These are very likely goblins, trolls and dragons. He will also include a number of cunning traps and a few subtle problems to be solved by the players. After many hours of work creating his dungeon, the Dungeonmaster will invite the players to enter the dungeon.

One by one, the players move into the imaginary dungeon, and the Dungeonmaster gradually reveals a table-top map of the dungeon, and on it he represents the population with miniature figures. By looking at his maps with room and monster information he will, strictly in accordance with his pre-ordained rules, tell the players the results of their actions. Each dungeon might be played in an infinite variety of ways, dependent upon the randomness of the dice, and the unpredictability of the players.

In such a game there is a plot, and the players interact with it. Provided all the players are gathered together with the Dungeonmaster the adventure of the exploration continues. When the game is adjourned a player may think "If only . . . ". But this time his thoughts relate to his own actions within the Dungeonmaster's plot, rather than to the author's decision as to the actions of the characters. Next time round the player can try something different to improve the outcome.

Thus stories and games were combined. And then, along came the computer . . .

Take an Adventure story with all the traps and pitfalls into which the hero might, and often does, fall. Some traps are fatal — in the story our hero never falls into them! Some traps may be avoided, but if not, the hero will have to overcome additional problems and difficulties to survive. Of course, in the story, he always does!

Now computerise the story, adding the interactive ingredient of the Dungeon.

In a story, at every major choice of action, what the characters actually do is determined by the author. The outcome of alternative possibilities is left unexplored except in the imagination of the reader.

The computer enables the author and "reader" to bridge the gap between story and limitless fantasy. The author writes the plot, but he now includes in it a number of alternatives at every stage, allowing the plot to branch in an ever-increasing number of directions. Thus the author writes a plot that allows the hero to make "wrong" moves — fatal or otherwise — and to continue the action to a conclusion, within the constraints imposed by his mistake. The "reader" is put in the position of the hero and must play his part, deciding on each action to take. He supplies commands on which the hero acts, and the computer supplies a description of the outcome. If all the commands are the "right" ones, then the story will end "happily ever after". If some wrong moves are made, those that are not fatal to the player will certainly lead to a less than satisfactory conclusion.

The reader can now be described as a "player", and the player will "win" if the outcome is successful. The story has now become a game.

And yet this description is not entirely accurate. Agreed, it can be argued that a computer Adventure is no great literary work, but there is a storyline, together with excitement and humour.

I have deliberately used the word "story" rather than "novel" in developing this discussion, as "story" implies something less verbose than "novel". However, Adventures played on large (mainframe) business computers can be relatively wordy, with replies extending over more than one page (or screenful). Commands can be fairly complex sentences, albeit with a restricted structure. Microcomputer Adventures, spawned from mainframe games, necessarily have less to say for themselves due to the limitations of available memory. That this could be achieved at all whilst retaining the compelling nature of Adventure stories is a tribute to the pioneer micro-adventure authors.

As micros became commonplace, and peripherals became cheaper, so it was possible for mainframe games to be adapted with little loss of text, using disk storage. Access to the disk is made frequently throughout the game, and commands such as "verbose", "brief", "super-brief" are used to give the player control over the volume of text he receives in replies.

Looking at the trends in the development of Adventure games, it is not unlikely that before long the computer Adventurer may well become a true "reader", playing the part of the hero and commanding actions that will lead to true novel-like narrative responses from the computer. Not only will novel-length text be presented to the reader, but the game will be fully illustrated in colour, accompanied by realistic sound effects and music. And if required, be output to a printer/plotter that would in fact create a "book". Already two-player games are beginning to appear, in which the players interact with each other as well as with the plot. How long will it be before many machines can be linked together to allow their owners to join in a simultaneous fully interactive Adventure quest?

Could an ordinary novel offer all this?

CHAPTER 2

Playing Adventure

Adventure is one of the most popular types of game played on computers. Adventure, certainly, is a game uniquely suited to the computer, combining the intricate plot of a story with the interactive character of Dungeons and Dragons. It is a game that generates enthusiasts, even addicts, those with a passion for the tantalising puzzle whose solution is staring them in the face, but is seemingly impossible!

Scott Adams, author of the famous "Adventureland" series which set the standard for micro-Adventures, scarcely exaggerates when he proclaims in the instructions introducing his games that the Adventure player is able to experience the thrill of adventure without ever leaving his armchair! A good Adventure can be totally absorbing; the danger of making a false move can become to seem a very real danger, and the excitement of discovering a hidden exit or new location can give a real sense of thrill.

So, for the uninitiated, what exactly is Adventure like to play?

In general the game consists of a logically connected network of "locations" which must be explored or traversed. The network can contain objects, some useful or valuable, others dangerous or red-herrings, and creatures, benign or otherwise. Some of the objects and creatures may be picked up and carried, and thence transported to be used or manipulated elsewhere in the network, often in obscure ways, in the progress towards the ultimate goal of the particular game.

The objective of an Adventure game may be to collect and store treasures or to carry out a particular task such as preventing a bomb from detonating and thus saving the world!

The game is actually played by typing commands to the computer in plain English usually in the form of simple two-word sentences consisting of a verb followed by a noun. The player decides on a suitable command by considering the information displayed successively before him on the screen. This will usually describe where the player is, certain specific things that are visible, and perhaps a few clues as to how to move on to another location. There will also be displayed part of the continuing conversation between player and computer, that is, the player's commands and the computer's responses to them. If the player uses a word that is not included in the vocabulary of the game, then the computer will reply with a suitable comment. By a careful choice of vocabulary, and by inserting the player's "unknown" command words into the response, the author is able to create the uncanny illusion that the player is freely conversing with the computer.

For example, if the player types "TAKE LAMP", and "lamp" is not a word in the vocabulary of the game, then the response might be "I do not know what a lamp is". If the word "lamp" is known, but the lamp is not in the player's current location, then the computer might reply, "I do not see a lamp here". If the lamp is present, however, the response may be simply, "OK", and the word "lamp" will disappear from the display. It is now no longer in the location, it is in the player's hand! Thus, if the player now commands "INVENTORY", the reply will be "I am carrying the following: Lamp."

To move from one location to another the player must use a suitable "moving" verb followed by a direction or exit. For example, by commanding "GO NORTH". Then this, if a legal exit, will move the player to the location logically north of the current one in the network. Details of the new location, together with any objects present there, will now replace the old location details on the screen.

In some games, graphics, sometimes in colour, are provided in the game. There are Adventures that talk, and some with sound effects and music to accompany the responses on the screen. These features may enhance a game, but words are the bread and butter of Adventure. Without words a graphics oriented game becomes a cross between a maze and an Arcade game, and can no longer be truly described as an Adventure.

The player, unless he cheats by LISTing the program, has no way of knowing how many locations or objects are in the game, or what they might be, until he comes across them. The locations might be so intricately connected that unless he has carefully drawn up a map from the outset, he will very easily lose his way and be unable to effectively exploit the clues and opportunities presented to him as the game progresses. A map will certainly aid him in travelling back and forth (as surely he will need to) to complete the puzzle.

When a game seems to have ground to a complete halt, and the player is desperate for inspiration, he will find that cheating is not easy whether the game is written in Basic, the English-like language provided on most micro-computers, or is in machine language.

A machine language program consists of a series of code numbers recognised and acted upon by the particular microprocessor used in the computer. To "read" such a program involves returning these codes to their original Assembly language listing, and this requires a piece of software called a "disassembler". However, an Assembly language listing itself consists of a series of mnemonics or "aides-memoire" designed to make it easier for mere humans to recognise and understand the machine codes. How helpful would lines like:

```
LD HL,3C00H
```

be to the frustrated player trying to read the logic of an Adventure game?

The reasons for using Assembly language in preference to Basic are twofold. Firstly, it is not only more compact, but can occupy part of memory normally set aside for Basic housekeeping tasks, thus allowing a bigger program to be loaded into a given machine. Secondly, such a program will execute faster than if Basic had been used. This is because machine code is a series of direct instructions to the machine, whereas Basic must be first converted into the codes by the interpreter before being executed. However, Assembly language has the disadvantage of being difficult to learn, and extremely time-consuming to write in comparison with English-like Basic. It is also less compatible between machines. The language will differ depending upon which microprocessor is driving the computer. Furthermore, the differing memory maps (the way computers have their different sections of memory laid out) of computers using the same processor will make translation difficult, especially if maximum use is made of ROM calls to make use of sections of code or routines already resident in the machine.

Cheating Adventures in Basic is a lot easier, so quite likely the author will have made it difficult to list, print or copy. However, although a Basic listing will usually make plain what the objects and locations are, and give away the responses, it may still be quicker to solve the game by playing it than by trying to unravel the logic of the program!

For the really desperate player of a machine language Adventure who wants to know what vocabulary is used, and the objects or locations still to be discovered, here is a way to display these secrets, provided that the English text has not been encoded by the author. Get out of your game by typing "QUIT", pressing BREAK, or RESET. Do not turn the machine off. From the command mode type:

```
FOR I = ***** TO %%%%% : PRINT CHR$(PEEK(I));: NEXT I
```

In place of the asterisks type the decimal number denoting the start of user memory (RAM), and in place of the percent signs, the end of user memory. Now press ENTER and interspersed with dark patches and graphics characters, all the text used in the program will scroll up the screen. This won't tell you how to actually complete the game, but it will give you a few jolly good clues to go to work on!

CHAPTER 3
Memorable Adventures

Adventure was around before micros. The game recognised as having started it all, the "big bang" of Adventures, was written by Crowther and Woods in Fortran, a scientific and engineering language deriving its name from FORmula TRANslation. It was played on large business and university "mainframe" computers in the USA. Variously known as "The Original Adventure" or "Colossal Caves", it had college students and bank clerks alike writing letters to journals requesting help from other readers on how to solve some of its more tricky problems. It was played in lunch hours and spare time over periods of many months. Adaptations of it are now available for a number of popular micros.

When micros started appearing on the US scene in large numbers, Adventures appeared written in Basic. Greg Hassett, (still only in his teens), and Lance Micklus were authors whose names became commonplace. The majority of these games were written for the TRS-80, which was then the most popular personal micro in the States.

Among these early authors was an advanced computer programmer at the Florida Institute of Technology named Scott Adams, who having come across the Colossal Caves became excited with the Crowther and Woods game because it combined the challenge and logic of a good game with the imagination of a story. He had just bought a TRS-80 and, wanting to explore the possibilities of Adventure on a microcomputer, wrote his first Adventure. This was in 1978, and the Adventure was Adventureland. It was written in Basic and featured a split screen showing locations objects and exits at the top, with the player's instructions and computer replies scrolling independently underneath. On the advice of Lance Micklus he translated Adventureland into machine language, having been persuaded that people were concerned about speed. He realized afterwards what a big difference it made. After that, he wrote his own Adventure language, formed his own company, Adventure International in 1979, and followed with 11 more Adventures.

Scott's Adventures are undoubtedly the most popular games on the market today, and the fact that in addition to the TRS-80 they have been made available for the Apple, Sorceror, Atari, Texas and VIC-20 computers testifies to this. More letters for "HELP", the Adventurer's eternal cry, arrive at the offices of Computer & Video Games magazine concerning Scott's adventures, than for any other game or series. This is not to say that they are necessarily more difficult than the rest, but they imbue the player with a sense of compulsion, their problems are so intriguing that the player rarely, if ever tires of those that he has not managed to solve!

Another series of machine language Adventures that is very popular are the "Mysterious Adventures" written by an English author, Brian Howarth, and introduced by Molimerx Ltd. Again, originally for the TRS-80, these games, also published in the US, are now available for BBC micro and VIC-20. There are currently 10 games in the series, with titles ranging from "The Golden Baton" to "Ten little Indians".

The arrival of the Spectrum has brought with it a new wave of Adventure games. A series of four games from Artic Computing vary from poor to interesting — none of them reaching anything like the standards of Scott Adams. From Melbourne House, "The Hobbit", another Spectrum Adventure is quite novel, following very closely the story in the book, and featuring colour illustrations. The pictures are good, but does not the main attraction of Adventure games lie in the use of the imagination? The Spectrum has brought in its wake a series of "prize" Adventure games, notably Pimania, later made available for the Dragon and BBC micros.

The BBC micro, a little short of Adventures yet, has its own "standard" Adventure, "Philosopher's Quest" from Acornsoft, as well as the Brian Howarth series. Another series of three Adventures, recently converted from their original TRS-80 version, is the "Fairytale" series by yours truly. Written along lines very similar to those described in this book, the BBC versions are enhanced from the TRS-80 versions by having sound and colour added, also, Molimerx Ltd which now distributes these games, is exporting them to the U.S.A. where they have been converted for the IBM PC.

A really mind-bending series of Adventures by Frank Corr, and available for the TRS-80 and Apple computers, is "Death Maze 5000" and Asylum I and II. These combine the text entry typical of Adventures with a fast-reacting graphical maze. They are particularly difficult, and require careful mapping and note-making on the part of the player.

Graphics also feature in another series, Dunjonquest, whose titles include The Temples of Aphsai and Hellfire Warrior. A departure from the usual type of Adventure, these games follow more closely the Dungeons and Dragon theme, and are variously available on disk or cassette for

Apple, Atari and TRS-80 computers.

Avalon Hill, despite specialising more in war-gaming, produce a couple of nice Adventures in "Empire of the Overmind" and "Lords of Karma", for the Atari, Apple, Pet and TRS-80 micros. These are available on disk and cassette, and all versions require more than the standard memory configuration for the respective machines.

Many other Adventures are to be found in software shops and magazine pages. Tandy produce their own series which includes Pyramid and Raaka-Tu. Big disk adventures are available in the form of Zork for TRS-80 and Atari, running into some 96k of disk space. Zork II and III is available for the Atari, and Xenos for the TRS-80. More Adventures appear every month. Below are some short reviews of just a few of the many games currently available.

Adventureland by Scott Adams

My very first taste of Adventure! Set in a swampy forest near a sunny meadow, the scene soon moves underground to royal chambers and a chasm. Can you think of uses for an empty wine bladder and evil smelling mud? The large dragon sleeping peacefully in the meadow begs a good hard kick — is he really as harmless as it seems? Is it possible to get past the thin bear and still collect the 13 treasures to complete the game? Some devious thinking is needed to solve this — but it has been enjoyed by many beginners and experienced players alike.

Pirate Adventure by Scott Adams

A strong theme runs through this game where you will come across bottles of rum, old chests, an anchor, and an extremely greedy and loquacious parrot! The story begins in the player's London flat, and after some chilling discoveries, moves to Pirate's Island. Easier than Adventureland, this one has a keener sense of humour, and reaches a climax with a gigantic hoax! If you can't take a joke, be prepared to put your fist through your screen when you get there!

Mission Impossible by Scott Adams

This one is different, set in an automated power plant where a saboteur is loose with a bomb. A tape recorded message tells you, Mr Phelps, that your mission, should you decide to accept it, is to prevent the bomb from detonating. Rather a drab game as far as descriptive content is concerned, and quite a difficult one. Breakthroughs seem to come rarely after long periods of frustration, and it is a game that lends itself to "stop-start" playing! What upset me — when I finally succeeded, I hardly even got a pat on the back!

Deathmaze 5000 by Frank Corr

Deathmaze is displayed graphically as a perspective maze, movement through which is by depressing the arrow keys, and text commands and replies are also featured. Said to have 500 cells, but I can't verify this! Boxes found in the maze contain a variety of things — a precision crafted frisbee and mutilated sneakers amongst others! There is also THAT ROOM that seems impossible to exit. A calculator, 317, 317.2, TURN, TURN, TURN . . . A difficult game that could well lead you to:

Asylum by Frank Corr

Bigger, with better graphics, and even harder than deathmaze, Asylum has been known to send people to the verge of lunacy. Armed with a hand grenade you must escape the asylum, defeating some very alert guards, and escaping from straightjackets. Very difficult, I have heard of only a few people claiming to have completed this game.

Pyramid of Doom by Scott Adams

Difficult in parts — notably at the beginning and the end, but easy enough in the middle to give the novice some encouragement. Nervous tension is created by the appearance of a small nomad, who proceeds to follow the Adventurer around. There is humour in the throne room, and a counterfeit object. If you can successfully avoid the rats and the mummy, you're well on the way to success.

Hellfire Warrior

This is a game more along the lines of Dungeons and Dragons, and one of the "Dunjonquest" series. After bartering with an Innkeeper to collect weaponry and armour, you proceed to the Dungeon, where exploration is carried out graphically. The monsters therein move of their own accord — so the player must be quick with the command codes to avoid injury or death. Good of its kind.

Abersoft Adventure

For ZX81 (16k) and Spectrum fans, this game is very similar to Tandy's Pyramid, once you get inside. Both are similar to the Colossal Cave in an abridged form.

Savage Island by Scott Adams

A two-part Adventure, the first is considered extremely difficult, and even hardy Adventurers try not to mention part 2! Starting off on a deserted beach, the chilling discovery of a large stone head in your likeness, lends to a feeling of unease. Soon forgotten in the exploration of a volcano, lake and caves, pursued by an over-friendly bear! Beware the hurricane whilst you struggle over the first hurdle — the bear! Part 1 is a good game, which may take quite a few months to complete. And as for Part 2 — it's breathtaking!

Atom Adventure

A trip down the road from the station and through the forest will lead to the cavern — somewhat difficult to get in. You will be told it is too obvious to mention if you ask for help! Eventually of course, you will get in, and experience the magnificent chambers and rock pools, and encounter a green frog. Then you can set about rescuing the Princess.

Ghost Town by Scott Adams

As featured in the opening sentences of this book, Ghost Town has all the feel of a deserted Western town in this game, complete with Saloon, Barbershop and Jail. Who is ringing the bell, can you ride a horse, and how do you set about breaking **in** to a jail, are some of the problems you will come up against. One of Scott's best — I raise my stetson to it!

Time Machine by Brian Howarth

As a journalist seeking to interview Dr Potter, a famous scientist, you find he has mysteriously disappeared, leaving only his slightly faulty time machine behind. Your quest will take you to such diverse places as Troy, a Black Monolith, and the Marie Celeste.

The Curse of Crowley Manor by Jyym Pearson

A detective story with a difference, you are part compelled to follow the narrative, until you come across a body. The secret of Crowley Manor will eventually be revealed if you can break through the inconsistent commands required.

Lords of Karma

Opening in the city of Golconda, your journey takes you through forests where sunlight filters softly through the leaves of Aspen, Oak and Redwood. Pray in the Chapel of Prayer if you must, and beware the jolly green giant. The object is to collect Karma Points by doing kind and brave deeds, and fulfilling specific quests. A big game requiring 32k on an Appel or Pet, 40k on an Atari, and 48k on a TRS-80.

Strange Odyssey by Scott Adams

From a broken-down spaceship, the Adventurer in an alien environment must collect treausures of alien civilizations — if he can successfully manipulate the controls of a strange travelling device! Can you mend your crippled ship, climb slime trees, and tame a dia-ice hound?

Love (for ZX81 with 16k)

Described as an adventure for women, its unusual objective is the successful seduction of Tom, a character among the house guests at a party. A range of glamorous dresses, make-up and washing facilities are all available for the would-be seductress or transvestite.

Philosopher's Quest

A game set in a labyrinth, with many logical puzzles to solve. Meet the automatic solicitor (if you can find him!) and avoid the albatross. A difficult but interesting text Adventure for the BBC. It's weakness is its slow (for the Beeb) response time of 5 seconds.

Espionage Island (Spectrum)

You are the pilot of an aircraft, which must be ditched after it is shot down, then find the secret of the island on which you land. What are the mysteries of the heavily guarded camp and can you penetrate it? Will you be able to return to your aircraft carrier base?

Ship of Doom (Spectrum)

Your space ship has been drawn into an alien cruiser by gravitron beam, and you are about to be turned into an android. The main control button on the ship's computer must be pressed to free your ship — but although easily found, cannot easily be touched!

Pimania

More a series of unconnected puzzles, perhaps Adventure is not the correct description of this game. Catchy tunes and jokes break the monotony of trying to solve the difficult puzzles and win the ornament prize offered by the publishers, Automata.

The Hobbit

An adventure with a difference, this one has pictures with every location, and follows closely the theme of the book. The creatures of Wilderland go about their business whilst you are thinking, so that different outcomes may be reached each time you play the game.

Zork

A game requiring at least one disc drive, this is a really big game, with screenfuls of text, most of it tongue-in-cheek. The descriptions are extremely detailed, giving, for instance, whole pages of a guidebook to a dam, and the wordy satirical advertisement on the back of a book of matches. The network of locations is seemingly endless. Almost worth buying a disc drive to play!

Empire of the Overmind

Overmind is a computer on another world controlling Earth's population, which it has enslaved. It must be found and disabled to free the world. A Sci-Fi background, but not presented in the space-ship form — other forces are at work to help (or hinder) you in reaching your objective!

Mystery Fun House by Scott Adams

Yet another of Scott's, and another of his better ones! Successfully recreating the atmosphere of a fun-house on a computer, this is a quest to find secret plans. Not an easy one, but so appealing and intriguing, that it is always fun to come back to in search of a solution.

Xanadu

A large underground network for the player to explore, this game has a novel two-player option, allowing players to either compete or form a common alliance against the monsters. In this mode, of course, both players cannot be holding the same object as the other . . . including treasures! And fighting has been known to break out between the adventurers playing on a BBC micro.

Xenos

Another disc-based adventure, this is somewhat reminiscent of Ghost Town. An air of foreboding hangs over you as you explore a small desert town that has been hurriedly abandoned by its population. Can you drive a rusty jeep? Can you survive a meeting with a rattlesnake? And why have the people fled? Your score here is not in treasures, but percentage of mission accomplished.

Rescue at Rigel

Memorable enough to feature in "The Worst of Adventures" book. Play Hellfire Warrior and you'll find the same but less in this one. Amazingly thought worth producing for a range of machines.

Dungeon Adventure

One of the best Adventures around for the Spectrum. Over 200 locations, including one on a mudbank by a large packing case, from where you start. The many problems and traps include a dice game with the Rakshasa, a strange cubic machine, and a siren! Verbose text descriptions and a fast response make this game both interesting and enjoyable to play.

Temple of Bast

A TRS-80 Adventure in machine code, taking the player (if he is able to go) from his London house to Egypt. There's some tasks for the D-I-Y enthusiast to complete before the journey, and a very strange caller at the front door . . . !

Forbidden City

A non-game for the Dragon and more recently, the Texas. This one could be unique in its poor descriptions, its inability to understand most of its own vocabulary, and its totally boring plot. One to be avoided at all costs.

Bedlam

A text Adventure set in an Asylum, this is from Tandy for Tandy. Meet Napoleon and picasso, a mad doctor and a vicious dog. What there is is good — but there is very little of it. The way out changes every time — so they say. It didn't change once for me.

Ring of Light

Not quite an orthodox Adventure, but involving movement graphically around a map. The explorer can enter townships, and buy food and other necessities, doing battle with bandits and others. More strategy than problem solving in this game for the Dragon.

Franklin's Tomb

You are Dan Diamond, a private detective, although this does not seem to have much relevance to the scenario of the game. The tomb holds a series of puzzles, requiring the collection of certain items to use the final exit. This exit leads to the second game in the trilogy, "Lost in Space". A pleasing Adventure available for Dragon, BBC and Oric.

Circus

One of the Mysterious Adventures available for an ever-growing range of micros, Circus leaves you stranded without petrol. You walk across the fields to find some, and stumble upon a strangely deserted Circus site — or is it . . . ? Creepy!

Golden Apple

The fifth game in Arctic's series, requiring 13 treasures plus one apple to be collected. A miscellaneous-problem type of Adventure that will no doubt be popular amongst Spectrum owners, although sometimes the logic is a bit difficult to swallow.

Groucho

The sequel to pimania, groucho offers a prize. Groucho also offers fat cigars as currency, which the player uses to buy clues to guess film stars, to be given clues to guess the 'big' one. Interspersed with 'knock knock' and 'I say' jokes, and set against a background of superb graphics, Groucho is a tribute to Hollywood. Groucho is painfully slow, but extremely addictive.

Pettigrew's Diary

One of the most unusual Adventures arund, Pettigrew's Diary consists of three games (or chapters) in one. The second of the two, a full-length Adventure in its own right, has a most original display, and wicked sense of humour. In this chapter, 'London Frolics' you can travel on the tube, work, gamble, dodge traffic, and many other things. A really worthwhile Dragon package.

Valhalla

Valhalla requires more beating about with sword and axe than puzzle solving, but in the best possible way. Set against fast-displaying graphics, all the characters march around the screen doing their own thing. Tell your little figure to TAKE SWORD, and he will walk up to it and actually take it in his hand. And a fight is a real one — you can watch them actually beating the living daylights out of each other! Pity about the bug.

These games represent just a small sample of the many games available for current micros.

CHAPTER 4

Essentials of an Adventure Program

An Adventure game is, as we have seen, rather like an exciting book. Perhaps there will not be much opportunity when writing it to make the text a great work of literature, but the key to an absorbing book is its plot, and so it is with Adventures.

Ideally, the plot of the game should have a definite theme, or the whole thing will appear to be a hotch-potch of fragmented puzzles that don't link together very well.

There are many ideas that can be taken as the basis for a suitable plot. A popular theme is science fiction — a subject closely related to computers themselves! Here there is plenty of scope for the Adventure author to invent strange planets, alien beings, and new modes of travel through time and space! A science fiction game does not perhaps lend itself quite so easily to the treasure collecting type of Adventure, although treasures can be found in any setting given a little ingenuity.

Spy stories make good adventures — keeping the player on the alert for tell-tale signs that he is about to get caught. These can equally well be set in war-time Germany, or present day iron curtain countries.

If you are not too adept at creating a whole plot from scratch, then have a go at converting a book that you found exciting and enjoyable, preferably a well-known one, into an Adventure game. Here you have the scope to change the parts of the story you found disappointing into a plot of your own liking — and keeping the players from becoming too complacent about the game if they've read the book!

The scope for Adventure plots is almost limitless, and on a more serious note, the game/program method lends itself well to educational purposes. Perhaps with research you could recreate a particular period of history, teach someone how to service a car (Tell me what to do —

Remove oil filter!), or if you are a linguist — write a simple adventure in a foreign language!

An Adventure essentially takes place somewhere, and therefore the environment, a series of discrete and interconnecting locations, must be carefully drawn up. This means that not only must the geographical network in which the action takes place be mapped out, but each location must be described, and the exits from each together with their respective destinations must be defined.

Similarly, without treasures or certain tools and objects available to help the player manipulate his fantasy world, the Adventure will fall flat. Therefore planning must include not only what those objects and treasures will be, but in what locations they will be found, and whether they will be available to the player unconditionally. Will any be hidden and revealed only after a search? Will some be constructed with or from other objects?

Although an Adventure game consists entirely of text, this doesn't mean numeric values will not be used. The central feature of any Adventure program is its ability to interpret the words used by the player, and the end result of this interpretation is to arrive at a numeric value for each word. Thus we will develop a routine that will turn those words into numbers. For simplicity we will be limiting ourselves in this book to replies consisting of no more than two words, — the simplest sentence that can be phrased — a verb followed by a noun.

Having established that the verb is a valid one for the game the program will establish a numeric code for it, and after doing the same for the noun, will use the verb number to find the appropriate verb routine. This part of the program will check to see if the numeric value derived for the noun is compatible with that for the verb as far as the plot is concerned. If it is, the program will then check to see if the conditions for applying the verb to that noun are satisifed. Has the player the right equipment with him? Is the act being performed in a suitable location? Have other conditions been fulfilled?

Thus the course of the game will turn, for better or for worse as far as the player is concerned. He may be helped with a clue, or rewarded with a treasure. Alternatively, he may be killed!

The computer having evaluated the command given by the player and privately arrived at its own conclusion as to the present state of play, must now be persuaded to part with its knowledge by way of presenting it on the screen for the benefit of the player. So now numbers will be turned back into words and the words will be displayed. The way in which this is done may make or mar the whole game — to grab the player's attention we must ensure that information is displayed quickly and attractively, and in a form that is easy on the eye and readily absorbed.

Screen presentation, is, in fact, our end product, and requires no less thought in programming than the logic of the game.

So with these thoughts in mind, we can start. I hope to be able to talk you through the elemental parts of Adventure programming, so that, given a good plot, your Adventure will rank among the best of them. I would stress that the techniques I am about to describe were those that struck me personally as the most obvious way to go about writing an Adventure program when I first went about the task. In computing there are many different ways to achieve the same result, more or less, and my way is probably no better, no worse, than most. To me, it is a logical method, and it works!

CHAPTER 5
The Plot and Locations

An essential stage in writing an Adventure program is to set out the plot. This must be fully scripted before the programming can begin. Although minor deviations from the plot may become necessary due to programming constraints in the course of writing the actual code, a major change after a considerable amount of program has been written may result in so many program alterations as to make it more desirable to start again from scratch.

Before we can even consider approaching the computer we must establish the details of the plot. This will involve sitting down and putting pen to paper, drawing out ideas in the form of a map of the network, making notes and comments. When the plot is firm, the information on the map should be written out in tabular form, ready for typing into the computer.

If writing your own game, then your first task is to devise the plot and draw up a map of the network, showing details of each location. The locations should be numbered, and the finished map should end up looking something like Appendix 7, a map showing the network of a short Adventure I have written especially to demonstrate how the program is built up. I will be referring to this game throughout the book, and you will find a complete listing for it in Appendix 1. In order to follow through the 'tutorial' section that follows, I suggest that you do not enter the listing as a whole into your computer, but enter it section by section as the program is explained. These explanations follow the order in which it is most convenient to build up the program, and this is not in numerical order of line numbers!

The game we are about to write will break the first rule I mentioned as it does not have a particularly compelling theme; I have tried to include in it a variety of situations in the plot that call for a number of different techniques in the program, whilst limiting it in size, to enable all the steps in its creation to be fully explained. I would add at this point, that although it is a 'going concern', the game is not complete in the sense that the final 'polishing' up has not been done. This is deliberate, so that later we can discuss how to discover and overcome any problems and weaknesses.

Let us start from the point where the plot is decided, and the map is drawn. This is shown in Appendix 7.

We will shortly be looking at the map of the network in detail, but first we will set up a table to lay out the actual wording we will use to describe the locations, and any other information that will be needed relating to them. So draw out a blank table of 4 columns. The first column is for the element number of the arrays. Column 2 contains the text description of the location, and we will call this array L$. In column 3 will be placed another array, E$, each entry containing a number of code-letters, denoting exits from the location appearing alongside it in the same row. Before we complete this column we must make an entry in another table, Table 2. This table will have one column for the direction words, and another for the code letter of each direction. First of all we can enter in column 1 of this table the obvious directions, north, south, east, west, up, down, out. Alongside these, in column 2, enter the initial letter of each word — N,S,E,W etc. Eventually we will have to add some implied directions. To establish these, we will have to examine the plot, and make the entries to the table as we encounter them.

Looking again at Table 1, each exit will, of course, lead to a destination, so in column 4 will be placed a series of numbers, each being the location number that will be arrived at by taking the exit whose code letter is in a position corresponding to it in the same row of E$. Examples of completed Tables 1 and 2 are in Appendix 11 and 12 respectively.

Although the table entry will consist of numbers, the array itself will be a string array D$, since we will be using string handling techniques to isolate an individual location. Since we will need to derive numeric values from this array, and these values may contain 1 or 2 digits, for single digit destinations, we will follow that digit by an asterisk. Thus, elements of array D$ will be double the length of those in E$, and there will be a simple arithmetic relationship between the relative positions of the exit codes, and the destinations to which they lead.

We are at last ready to examine the plot in detail.

Location 0

The action starts in location 0, a small room without any apparent exits. Since it is a short game we will make it fairly difficult for the player to escape. Wallpaper will be visible on the walls but the player will not be able to take it — yet. The clue to escaping the room will be given if the player commands "LOOK AROUND", when he will be rewarded with a suitable phrase informing him that the wallpaper is beginning to peel.

This should be enough to get the player trying to peel or pull the paper, in which case it will fall off the wall and become a TAKEable object. In so doing, it will also reveal a door, which will be our exit.

We can now begin to fill in the tables we have drawn up. In table 1 we can fill column 1 with a 0; this is element zero in the arrays L$, E$, D$. It will describe location 0. In column 2, under L$ we can write "in a small room". Initially there will be no exits from this location, so column 3 (E$) will be left blank. However, eventually there will be an exit through the door, so in Table 2 make another entry; DOOR in column 1 and its code, say A, since D has already been used for DOWN, in column 2. This exit will lead to location 1, so while we're at it, we might as well complete column 4 (D$) in Table 1, and fill it with '1*'.

Having described and demonstrated the principle by which the exit codes are arrived at, I have provided a complete list of those we will need in Appendix 11. Refer to this to see how the E$ strings are arrived at as we continue to examine the details of each location.

Location 1

The door from location 0 will lead us into a hallway, which will have exits 'south' leading back into the small room, 'west' going into a kitchen, and 'out' leading via the front door to the outside. So enter a '1' in column 1, the words "in a dimly lit hallway" in column 2, the exit codes "SWO" in column 3, and the destinations in column 4. The destination reached by moving south will be 0, so write '0*'. West will lead to location 2 so add '2*', and Out will lead to location 3 so add '3*'. Column 4 will now read "0*2*3*".

Location 2

This is a kitchen off the hallway. It's only purpose is to house a packed lunch, an object which is really a red-herring. The exit is east, so fill in the next row with location number 2, "in the kitchen of a cottage", E$ will be 'E', leading to destination D$ '1*'.

Location 3

The player is now outside the cottage in a forest, so fill in the next row for location 3 with "outside a forest cottage". The exits are north going deeper into the forest at location 9, east to location 4, and cottage, back to location 1. So in Table 2 enter COTTAGE and C in the two columns, whilst the entry for E$ in Table 1 will be NEC, and D$ will be 9*4*1*.

Location 4

This location is "by the moat of a castle". As it stands there is no immediate way of crossing this moat, and the player can only go back the way he came. So there is one exit for E$, 'W', and the corresponding entry for D$ is '3*'.

Location 5

This location is one the player is striving to get to, but will find difficult to reach. Describe it as "in a crumbling castle". The means by which the player reaches this location will be explained in due course, suffice to say for the moment that he will not be able to get back easily! There is therefore one exit, up via the stairs. We will allow two exit codes for this, UF, allowing the player to use the words 'up' or 'stairs'. Both will lead to the same destination, so D$ will be '6*6*'.

Location 6

Here, enter "in a tower room" in column 2 for L$, and the two exits, down the stairs or up the ladder will have four entries, DFUG, down, stairs, up, ladder. These will lead to two destinations, and D$ will be '5*5*7*7*'. The tower room holds a secret. On looking around, the player will see a secret cupboard, which when opened will reveal the one treasure in the game, a priceless crown. So the player will at last be able to take the treasure, but must now return it to the room where he first started. How will he get back?

Location 7

This is "on a parapet at tower top". (This not very clever abbreviated description is used to accomodate the smaller screens). The only exit initially is down, so enter 'D' for exits. There will eventually be another exit to location 8. Location 8 doesn't really exist as we shall see. However, enter '6*8*' for D$. If the player becomes desperate in trying to solve how to move from this location, should he decide to jump he wil meet a sticky end in the moat. However, the way out from here will be to 'TIE ROPE'. This will let him 'go rope'. This assumes, of course, that the player has been prudent enough to bring the rope along with him.

Location 8

Enter the description for this location as "hanging on a rope above moat". The only obvious way out here is 'UP', so E$ will be 'U', and D$, '7*'. The player will, however, be able to jump, in which case he will end up dead in the moat again. Unless, that is, he decides to "swing" first, and a suitable message will hint that it's safe to jump. He will then land on firm ground in location 4.

Location 9

This is in the depths of the forest, so L$ will be "in the forest". Let us fool the player into thinking he is moving in the forest, and give him 3 exits that appear to move him, but really leave him in the same location. He will, at first, think he has got lost. This is easily done by writing in 'NESW' for exits, and '9*9*3*9*' for destinations. This way, any direction taken except south, which leads back to outside the cottage, will take the player to the same place. In the forest will be a tree, which can't be taken unless

it is chopped down with the axe. Even then it can't be taken — it would be absurd to allow our player to stagger around carrying a tree trunk. We will arrange for the tree to follow the player to the next location, providing that each time, before he moves, he types "PULL TREE". If and when he reaches location 4 with the tree, if he types "PULL TREE" once more, the tree will slide in position over the moat and form a bridge, opening up an exit to the castle. When he crosses the tree, we will arrange for the tree to slip into the moat, effectively isolating him in the castle until he has caught on about the rope. We won't be so unkind, though, as to prevent him from going back into the castle, we will replant the tree back in its forest location, and allow the procedure to be repeated.

Location 10

Finally, we must have a location for "dead", and that is exactly how this location can be described. Being nasty, we will allow no escape from dead, but let the player suffer a bit before he discovers this. Therefore we will lock him in by allowing four exits 'NESW' and making all the destinations the same '10101010'. Note that the asterisks have now disappeared as we need both digit positions to contain the destination number, which in all directions is 10. This avoids the player being told he cannot go north, south, east or west, but at the same time it doesn't move him anywhere.

At last we can turn the computer on! We are not going to actually write the program, but can at least get some of the groundwork done.

Using two lines, 50000 and 50010, we can type in the contents of Table 1, in the form of DATA statements. Ignoring column 1, the array element number, type in the description, the entry in the exit column, and entry in the destination column for each location in turn in the table. Each piece of data should be separated by a comma, and the DATA statements, when complete, should look similar to those in the program listing. For good measure add line:

```
49999 REM **** DATA ****
```

Before you do all this, a word of warning about screen width. I have deliberately abbreviated the text so that it will fit in one line across all screens of 32 columns width and greater. If your screen width is less than 32 columns then you will have to abbreviate it even further, or convert it along the lines described in chapter 8. On the other hand, if you have plenty of width, then pad the text out with more details to add more interest and character to the game.

Finally note that some computers require string data to be enclosed in quotes.

CHAPTER 6

Objects, fixed and moveable

In a similar way to our treatment of locations, we will now draw up a table of objects, Table 3. An example of the completed table is found in Appendix 13. This will again be a table of four columns, where the first column will contain the array element numbers. Column 2 will have the text description of the object, and be array O$. Next, we will have a column with an array telling us in which location that object is present. Since the locations are numbered by virtue of their positions in the location array L$, this array will only contain whole numbers, and can therefore be numeric with integer precision. As it describes the Position of an object, we will call it P%. If your computer cannot specificially define a variable as integer, don't worry, just omit the '%' sign.

Finally, we will require another array which can also be integer numeric. We will call this C%, and it will be used as a series of 'flags' to signify different conditions applying to the objects from time to time. These flags will include information about whether or not their respective objects can be TAKEn. If an object is TAKEable we will assign to it a positive value in array C%, if not, negative. We will go further than that, and say if it is permanently unTAKEable, the value in C% will be −2 or less, but if we want a message like "I can't take it yet", implying that there is work to be done before it can be taken, we will assign its value in C% to be −1. For consistency then, if it can be taken unconditionally, it will have a value of +2 or greater in C%. The values in the flags C% will be further manipulated as we progress through the program, but we will discuss this in a later chapter.

Object 0

First there is the axe, so in table 2, column 1, enter 0, and in column 2 enter "axe". Where shall we hide this? What better than the obvious, in the player's own hands. Who knows, it may be ages before he thinks of looking there! We will assign a non-existent location for the player's inventory, so that it doesn't get displayed unless requested. So in column P% enter '55', which will be the number we use for held objects. We will allow him to drop and take the axe at will, so for C%, enter '2' in column 4.

Object 1

Next we have "wallpaper". This is found in location 0, so enter this in column 3 under P%. It can't be taken until it is peeled off, so we will assign it a value of −1 in column 4 under C%.

Object 2

The red-herring "packed lunch" can be listed next with a location P% of 2, and a "TAKEability factor", C%, 2.

Object 3

The tree is in location 9, so write 'tree', '9', and '−1' for L$, P%, and C% respectively.

Object 4

Next, a non-object, the entrance to the castle, seen from the inside, but unattainable in practice. As it must be displayed to frustrate the player, we will display it as an object. So write, 'Entrance', '5', '−2'.

Object 5

The rope comes next, in location 0, with TAKEability factor of 2.

Object 6

Now for the treasure. Priceless Crown will be the description, and while we're about it, let's write an asterisk each side of it to mark it as a treasure. Since it is hidden in the cupboard initially, and must not be displayed, we will put it in another non-existent location 99. This location number we will use for objects yet to appear. TAKEability can be 2, since we will be writing the code for taking objects so as to require that it is in the same location as the player. It will thus not be capable of being taken until it is revealed.

Object 7

This is a "rolled umbrella". It is situated in the hallway, location 1, and can be taken, C% is 2. We will use this to create another problem for the player. Outside, the player will find it is raining. If he continues outside, a message will warn him that he might catch a cold, and if this is ignored, he will die. To overcome this, he will have to open the umbrella.

Object 8

A door will be the next object, again one that can never be taken, but must be displayed when the wallpaper is off. So type 'door', '99', and −2.

Object 9

Stairs similarly denote another exit, and will be in location 5, with C% of −2.

Object 10

Finally, another 'exit' object, 'iron ladder' used to go up from the tower room. Location 6, TAKEability −2.

Now you are ready to add the objects to the DATA statements already written. This will list an object followed by its entry in the P% column of Table 3, and then its C% column entry. Continue until the data has all been entered. (see page xx for table 3)

We have now covered the details of the plot, and listed all the locations and objects, together with the values of the variables which will be used to control them. You should now have three tables corresponding to tables 1, 2, and 3 shown. Keep them by you, together with the plan of the network, for reference whilst we write the program.

CHAPTER 7
Space, time and structure

Having written data statements for the arrays which will describe the locations and their relationship with each other, plus the arrays for the objects together with their dispositions around the network, we are surely ready to start writing the actual program!

Not quite! We must pause here for a moment to reflect upon the irrevocable course upon which we are to embark. Once we start writing the program in earnest, in no time it will be too late to turn back.

Consider the following questions, and imagine you are setting out, as hopefully you soon will, to write your own Adventure:

1. With the plot in mind is there any danger of running out of memory before it's complete?
2. What can be done to minimise the response time?

Having spent some weeks or months writing your great work, you will want to ensure that it can be played on as many machines as possible. So consider the most popular size of the micro on which you are writing the program and set that as your limit even if your own model has more memory than the popular version. That way, you will stand a better chance of selling it commercially should it turn out to be the most fascinating game since Ludo was invented. (And why should it not? You might well strike lucky and have a best seller on your hands!) Alternatively, if you cannot, or do not wish to market the game, do not preclude your friends who are not blessed with so much RAM as yourself from playing it. They at least will be more interested than most in the game, knowing the author! They will be keen to solve it quickly to tell you how easy it was!

Let us assume then, even though the program we are going to use here for demonstration purposes is small, that the memory limit might prove to be a problem. What steps can be taken to minimise this? The answer is very much dependent upon the machine you are using.

If you are able to specify a variable as integer, then typing in all those % signs will save more memory than leaving the computer to default to numbers of greater precision. All the numerics we shall use will be integers. Better still, if you have the facility of the DEFINT statement, use this at the beginning of the program, omit the % signs, and you will still have integers precision. Integers as well as occupying less memory, take less time for the computer to process, so we will have also taken one step towards speeding up the response time.

Similarly, if your micro has a DEFSTR statement, use this for all string arrays and variables, and you will save the memory overhead of the '$' signs — not to mention the effort required to repeatedly hit the shift key to type them in!

Having decided upon this course of action, how do we interpret the program lines in this book? All the variable names used are consistent with the use of DEFINT and DEFSTR statements; nevertheless, for clarity, the % and $ signs are shown. If your computer has no DEFINT, type in the %'s. If you have no integer precision then leave out the % signs and let your machine run as it will. In the same way, omit the $ signs if you use DEFSTR.

Another space saving device at our disposal is to type without spaces between the code. I have deliberately shown spaces in this book for clarity, but if you can — forget them. Squeeze everything up close together!

Computers with the optional use of LET need never have the word displayed on their screen. For 'portability', all the LETs have been shown. Don't use them if you don't have to.

Some Basics allow a statement like:

```
IF (condition) THEN 320
```

The GOTO between THEN and 320 is implied. Here, the GOTO is always present rather than implied, but omit this, too, if you are able. However, don't omit THENs. Although your program may work without THEN statements a renumber utility may have a nervous breakdown when used. The logic may go a trifle haywire as well. 'IFA=B320' may be construed as 'IF A=B3THEN GOTO 20' when you really meant to end up at line 320 if A=B!

Finally, you will save space by using fewer lines, even though the length of code may be the same. Each line carries a 'line overhead', usually about 4 or 5 bytes, depending upon the particular interpreter. If your Basic supports multi-statement lines use this facility as much as you can, and cram in every last bit of code possible.

We have already speeded things up a bit by using integers! In fact, most of the steps taken to preserve memory will also make a small contribution towards reducing the response time. Even spaces and REMarks take time to execute!

Many of the ways to elicit as quick a response as possible are embodied in the actual method I have used to structure the program. Obviously, Basic is not going to be anything like as fast as machine code. Or is it? Your interpreter and hardware will probably be the deciding factor here.

Nevertheless, there are a number of ways in which the programmer has some control over response time. The most frequently used variables should be the first to be assigned. Similarly, the words likely to be used most frequently by the player, should be placed as near as possible to the start of the string containing their keys.

In the first case, Basic will search the look-up table for the variables and find them in the order in which they have been assigned. In the second case, our string search subroutine will be satisfied earlier if the match is found. This does not alter the maximum and minimum search times, but speeds the game up overall, since the longer delays occur with words that are seldom used. An optical illusion, if you like!

Another time user is the 'special condition' line. You will meet this in a later chapter, but suffice to say here, that every such line must be executed every time the player hits enter. Therefore, the fewer the better. Here is the paradox. The 'special condition' lines are probably those that add the most dramatic interest to the game. The time delay due to them may detract from the pleasure of the player. In practice, you may find, depending upon your machine, that these must be limited to an absolute minimum. If this is the case, then you may have to alter your plot fairly substantially. For it is basically the plot that determines how many of these lines will be needed. If your response speed is good when all the necessary special lines are in, then you may even try adding a few more just for effect. But beware!

Once we get into the detail of programming we may not easily be able to see the wood for the trees, so let us first have a look at the overall structure of the program. We can then decide how we are going to assign the line numbers and where we will put remarks, so that whilst in progress, we can find our way around it.

There are a number of distinct tasks that the program has to carry out, and we will refer to the coding for each task as a 'block' of code.

Block 1.

Laying out the ground for the program to operate in:

CLEAR string space. (where required to allow manipulation of strings)

DEFine variable types. (string integer etc. where the particular micro has the facility.)

DIMension arrays. (telling the computer how many elements there will be in each array)

Block 2.

READ in DATA statements and/or directly assign variables.

Block 3.

This is the start of the main program loop, and communication with the player:

Check for special conditions.

Clear Screen.

PRINT screen display.

Reset input/output variables to null.

Await INPUT.

Block 4.

Interpret player's communication with the computer:

Decode verb and noun.

IF either are invalid singly or in combination set reply accordingly and return to block 3.

ELSE GOTO block 5.

Block 5.

Execute the plot:

This block comprises a number of routines, one for each valid verb. Each routine may alter game variables, and either sets a reply and returns to block 3, or if the reply is a common one, goes to block 6.

Block 6.

Sets standard replies.

Block 7.

DATA statements for locations and objects.

The deeper you get into writing the program the more difficult it will become to find/recognise the program lines and the purpose of each. This will be especially true if you have no printer and must catch the lines as they scroll up the screen. Another difficulty is that you may eventually need to renumber the lines to squeeze in a previously unforseen bit of logic. Then even the line numbers that you remembered will be lost!

Of course, a line printer will make life easier, but is not essential. We will work on the assumption that one is not available. Here are some guidelines for line numbering and REMarks that will make life easier.

The first rule is to number lines in increments of no less than 10. This will leave plenty of gaps for insertions, reducing the likelihood of having to renumber.

Line numbers must be planned in advance, and the whole program sprinkled liberally with REMarks. Having split the proposed program up into blocks of code, we can now lay down what line numbers will be used in each block, and this is shown in Appendix 3.

You can see that the bulk of the program is contained in Block 5 where the routines for each verb reside, whereas Block 1 is very short. Notice that I have shown each block having an upper line number limit ending with 98. This is because it is a good plan to place REMarks on the line immediately preceding the start of each block and/or routine. Thus REMs for Block 5 would appear on lines 999, 1999, 2999 etc. When the program has been completed you will want to delete these, both to avoid giving too many clues to would-be cheats, and to speed execution. When they're gone, the running of the program will not be affected if the REMark is on the line PRECEDING the one it refers to, and which is pointed to by a GOTO or GOSUB statement. As a bonus, deleting them itself will be easier, since they will be recognised as having numbers ending with a 9.

Refer to the full program listing for examples of the way I have suggested treating REM statements, and notice how they are highlighted by asterisks. Whether you check your program on screen or hardcopy listing, they will be much easier to find, and speed your job in correcting and debugging the program.

CHAPTER 8
Starting the program on any micro

Having written the DATA statements, we can start the program proper, but first a look at the kind of Basic we'll be using, and the points you may need to bear in mind when entering the program or one similar into your machine.

The demonstration game for this book was written on a TRS-80 Model III. Although written in Disk Basic for that machine, no statements specific to Disk Basic have been used, and so the program is effectively Level II Basic. A number of statements in the program are redundant as far as Level II is concerned, but have been included for maximum compatibility with other machines.

As listed, the program should run without any modification on a TRS-80 Model I, TRS-80 Model III, and Video Genie, with Level II or Model III Basic, including their Disk Basics.

The TRS-80 has a 64-column screen, but providing you change the number '61' in lines 310 and 4010 to a value equal to the column width of your screen minus one, then no modifications to the text or displayable variables will be needed for screens of width 32 columns or greater. If your screen is narrower than that, then you have a choice of possible modifications. You can suitably abbreviate the text; create additional arrays, (e.g. a second line for the location text could be an array L1$, which continues the overflow from array L$); or incorporate special control characters or 'line feed' characters to force the displayed text to move to the next line at appropriate positions in the text, to avoid text overflowing from the end of the screen to the beginning of the next line in mid-word. An alternative to using control characters is to pad the display variables and text with blanks to the end of the line and continue the text at a point where the next display line starts.

Below are the features used that may vary between machines, and you should check these against your manual to determine the conversions necessary. As most of them are used quite frequently, it might pay to make a note of the alternatives required on your machine for ready reference during the course of following the program explanation.

The Spectrum, however, is a micro with a Basic all of its own. An error message "Nonsense in Basic" may well occur for statements that work perfectly well in other Basics! What is meant, of course, is "Nonsense in Sinclair Basic", and because this is so different, I have devoted a special section at the end of this chapter to the Spectrum version of the demonstration game. If you own a Spectrum, then whilst you may find some of the general comments below are helpful, you should refer to the Spectrum section before attempting to write the program.

LINE NUMBERS

The main listing uses line numbers in the range 0-65536. This range is not enjoyed by all micros. For example, the BBC has a maximum line number of 32767, whilst the Spectrum only allows up to 4 digits, i.e. 9999. If your line number range is less than those used, I would suggest that in converting this game, rather than completely renumbering the program, you stick with the existing line numbers as far as possible, and renumber the higher ones to fit recognisably below the maximum. That way you will have fewer changes to make to other parts of the program, and the listing will be more easily compared with the original. The BBC listing demonstrates how I suggest tackling this problem.

IF/THEN/ELSE

The logic of the IF/THEN/ELSE construction used in the listing is worthy of study so that it is understood thoroughly to enable you to convert the program if the logic used by your Basic differs.

Where the condition following IF is true then subsequent statements on the same line will be executed until an ELSE is encountered. If the condition is false, then no more statements will be executed in that line until after the ELSE statement. Another IF statement may follow an ELSE or even an IF/THEN, and so on.

Care must be taken if your machine is a BBC micro, as funny things happen if you try to next IF/THEN/ELSEs. Refer to the BBC listing where these occur in the main listing, and you will see what must be avoided. If your machine hasn't an ELSE statement conversion will be a little more difficult. You will have to separate the ELSEs onto separate lines. For example, to convert line 3000:

```
3000  IF K2%>10 THEN GOTO 40070 ELSE IF P%(K2%)<>55
      THEN GOTO 40070 ELSE LET IN%=IN%-1 : IF PN%=7
      THEN LET P%(K2%)=88 : LET Q$(2)="FELL . . MOAT" :
      GOTO 100 ELSE LET P%(K2%)=PN% : GOTO 40020
```

Without the use of an ELSE statement, this line would translate into the following series of lines:

```
3000  IF K2%>10 THEN GOTO 40070
3001  IF P%(K2%)<>55 THEN GOTO 40070
3002  LET IN%=IN%-1 : IF PN%=7 THEN LET P%(K2%)=88 :
      LET Q$(2)="FELL. MOAT" : GOTO 100
3003  LET P%(K2%)=PN% : GOTO 40020
```

A situation to be aware of is where the statement before an ELSE does not pass control to a line with a GOTO. In such cases, you may have to repeat some of the conditions in the additional lines you write to maintain the same logic.

STRING HANDLING

Some micros use a variation to the statements used here to manipulate strings. An explanation of how the terms used in the demonstration game listing operate, should enable you to substitute the code applicable to your particular micro, if different.

LEFT$(A$,B)	returns a string containing the B leftmost characters of string A$. e.g. If A$="ADVENTURE" and B=3 then LEFT$(A$,B) returns ADV
RIGHT$(A$,B)	returns a string containing the B rightmost characters of string A$. e.g. In the above example RIGHT$(A$,B) returns URE
MID$(A$,B,C)	returns a string containing C characters starting from position B in string A$. e.g. Using the same example where C=4 MID$(A$,B,C) returns ENT.

The adding together of strings, or 'concatenation', is used frequently throughout the program. The concatenation operator is '+', and when this is applied to string variables and/or text, the characters are strung together.

For example, if
A$="ADV" and B$="ENT" then
A$ + B$ + "URE" = ADVENTURE.

ON variable GOTO

The ON-GOTO statement is not enjoyed by all micros.

```
ON X GOTO 100,200,300
```

will pass control to line 100 if X=1, line 200 if X=2, and line 300 if X=3 etc. Lack of ON-GOTO can be overcome by a series of IF statements:

```
IF X = 1 THEN GOTO 100
IF X = 2 THEN GOTO 200
IF X = 3 THEN GOTO 300
```

but this is not necessarily the most efficient way. If your micro has the facility of a 'computed' GOTO, (i.e. ON variable GOTO variable/computation), and the line numbers to be 'gone to' are, or can be arranged in an arithmetic sequence, then use this feature.

PREDEFINED VARIABLES

If you have the benefit of being able to pre-define variable types, then use this facility, and throughout the following program omit all '$' and '%' signs. The string variables all begin with the letters L, O, Q, E, D, V or W. All the others are integer variables which are defined by the '%' sign in the listing. Some machines do not provide a special integer precision, and therefore do not recognise the % sign. Just leave them out if that is the case.

If you can use the CLEAR and DEFSTR, DEFINT statements, then your first lines should look like this:

```
2 CLEAR 500
4 DEFSTR D,E,L,O,Q,V,W: DEFINT I-K,C,P,S
```

ASSIGNING VARIABLES

Some micros insist that a variable is assigned or initialised before it is otherwise referred to. For example, if the variable C has not been assigned an initial value, then the first reference to it cannot be in a condition statement such as IF C = 3 THEN . . .

This means that at the beginning of the program, all variables not given a value either by direct assignment or by READing DATA, must be set. You can safely do this by giving all such numeric variables a value of 0, and string variables a null (LET X$=" ") value.

One micro exhibiting this anti-social tendency is the BBC, which will give the notorious "No such variable" message if you fail to take the above precautions.

ARRAY DIMENSIONS

Most computers allow arrays of up to dimension 10 to be used without the need for a DIM statement. If this is true for your computer, then there is no need for the arrays to be explicitly dimensioned. However, for the sake of clarity, if the arrays are dimensioned at the outset, it will be easier later to recognise which array variables are used.

Note that not all computers have a zero element to an array, in which case you will have to dimension the arrays to 11. This will also mean that you must renumber the tables you have drawn up. The row numbers will now run from 1 to 11. All the numbers contained in the strings in the E$ and D$ columns in Table 1 should be increased by one, and the P% column in Table 3 will require all numbers from 0 to 10 to be increased by one. Then when you encounter a FOR-NEXT loop that scans one of these arrays, you will have to adjust the 'from' and 'to' values accordingly. See the special notes on the Spectrum for more details about this.

We can now dimension the arrays we will be using. The location-associated arrays all have the same number of elements as each other, by design, as do the object-associated arrays. Purely by coincidence, the number of objects is the same as the number of locations in the game we are about to write, and so all arrays will have the same dimensions. There are 11 elements in each, and not forgetting that the first element number of an array is 0, all arrays will be dimensioned to 10.

We can write the DIM statements:

```
10 DIM L$(10), E$(10), D$(10), O$(10), P%(10), C%(10), Q$(8),
V$(8)
```

and are then ready to read the data:

```
20 FOR I%=0 TO 10 : READ L$(I%),E$(I%),D$(I%) : NEXT I% :
FOR I%=0 TO 10 : READ O$(I%),P%(I%),C%(I%) : NEXT I%
```

If your micro is unable to READ DATA, and you therefore did not enter the DATA statements, you can now directly assign the variables instead. Use the line number range set aside for the DATA, and start thus:

```
50000 LET L$(0)="IN A SMALL ROOM" : LET E$(1)="" : . . etc.
```

When all the array variables have been assigned in this way, at the end add:

```
. . . . : RETURN
```

To assign the variables, line 20 will now read:

```
20 GOSUB 50000
```

There are a number of other variables that we will have to initialise at the start of the game. The plot calls for the game to commence with the player in location 0. The variable name we will give to hold the number of the player's current location could be LN% for L(ocatio)N but to keep in line with our convention that variable names beginning with L will be strings, let us instead use PN% for P(ositio)N. By typing RUN to start the game, all numeric variables will normally be set to 0, so strictly speaking there is no need to assign PN%. However, since this game will act as a model for others, and any location number may be chosen to commence, it will be better not to omit assigning PN%. Start line 50, then with

```
50 LET PN%=0
```

The inventory count IN% must also be initialised, and as we decided to place the axe in the player's hand without telling him, the number of items in his inventory at the start of the game will be 1. Add this fact to line 50, which will now read:

```
50 LET PN%=0 : LET IN%=1
```

Another value we might want to keep track of is a count of the number of moves the player has made. So now add:

```
.... : LET CT%=0
```

Somehow we will have to keep a list of the words that the computer will recognise in the game. In order to simplify word searching, and to reduce the amount of memory needed to store all the valid words, it is convenient to use only the first few letters of each word. In this case we will use the first three letters of each word; had we been using a very large number of words, then we might have had to extend this to four letters in order to ensure that each word was uniquely abbreviated.

The valid words can conveniently be classified into two groups, first words and second words of the player's input. In general, the first group will consist of verbs, the second, nouns. Considering the verbs that will be required, there are some that come immediately to mind; TAKE, DROP, LOOK, to mention a few. Our 'Word — Verb' string can thus be written:

```
60 LET WV$="TAKDROLOO"
```

Never mind that we haven't determined the full list at this stage, we will find it quite convenient to add to the list as we go.

The noun recognition string WN$ (Word — Noun), should start off with the first three letters of all the objects in array O$ in the order in which they appear in the array. When I say 'objects' I mean the object noun rather than its adjective. For example, when referring to the Rolled Umbrella the player will most likely type "TAKE UMBRELLA" rather than "TAKE ROLLED", so in this case the recognition letters will be UMB. There are a few nouns that are not contained in the object list, the word TRUNK for example, plus ones that are really directions, direct ones like NORTH, and implied directions such as FOREST. The 'nouns' can thus be split into two sub-groups. Similarly with the 'non-direction' second-input words. LOOK AROUND is a useful response, we will need to recognise TRUNK when the tree has been chopped down, and CUPBOARD which is an object that is described in the location description when discovered.

Other words which are not objects but are used in the game spring to mind, such as CASTLE, DOWN (as in LOOK DOWN), and MOAT. There may be more which we will find are needed later, but let us start off with the ones that have sprung to mind:

```
80 LET WN$="AROTRUCUPCASDOWMOA"
```

To check the validity of exit words, we can set up a string holding the first three letters of the response words valid after the word GO. We will call this string WG$ (Word — Go), and it will contain the first 3 letters of each exit in column 1 of Table 2:

```
70 LET WG$="NORSOUEASWESUP DOW
OUTDOOSTALADENTCOTROPTRE"
```

Note that a blank has been inserted after UP to make up the recognition length of three.

Next we will set up string of the exit codes, WD$ (Word — Direction), consisting of the entries in column 2 of table 2:

```
80 LET WD$="NSEWUDOAFGBCHJ"
```

If you now compare the string WG$ and WD$, you will see that there is a direct relationship between the start position of any 3-letter recognition group in WG$ and the position its 1-letter code in WD$. We will be using this fact as the key to changing the player's location. This relationship is:

Code position in WD$ =
(((Substring start position in WG$) – 1) / 3) + 1

SPECIAL FEATURES OF THE SPECTRUM

LINE NUMBERING

The maximum line number available on the Spectrum is 9999. In the Spectrum listing in Appendix 4, the same line numbers (subject to conversion requirements) as in the basic listing have been used up to 540. From that point, the verb routines start at line 600, in increments of 100 for each routine, instead of 1000 as in the original.

VARIABLE NAMES

String variable names consist of only one character, and so those in the main listing using more than one character, must be changed. The new names used in the Spectrum listing are as follows:

Original	Spectrum
EX$	G$
OS$	S$
WV$	V$
WG$	F$
WD$	T$
WN$	Z$

A1$	H$
A2$	I$
A3$	J$
A4$	K$

This means that the previously mentioned convention relating to variable names and types, for compatibility with predefinng the types, is broken. This does not matter, since predefining types is not a facility available on the Spectrum.

ARRAY SUBSCRIPTS

Array subscripts start from 1 and not 0. Therefore, all arrays must be dimensioned to 1 greater than in the main listing. This means that a value of 1 must be added to variables used as pointers to elements of an array. In the Spectrum listing, for example, line 290 makes reference to p(i + 1) instead of p(i).

STRING ARRAYS

Spectrum string arrays have elements of fixed length. The length is tated as a second subscript — the array is effectively a two dimensional array.

In dimensioning the aray, then, the second subscript is the length of the string. In line 10 of the Spectrum listing then, o$ is dimensioned o$(11,32), giving it elements 1 to 11, all of length 32. When the string data, which must be enclosed in quotes, is read, trailing blanks will be added to strings whose text is less than 32, to make up the length.

Therefore the LEN of any string array element will always be the same, and LEN cannot effectively be used. Since we need to know the length of the actual text of each object, so that when printed, these trailing blanks don't leave long gaps between successive words, I have added an extra numeric array n(11), which contains the text lengths of the respective elements of o$. Where, in the program, the description of an object is changed, (e.g. 'Rolled umbrella' to 'Open umbrella'), you will notice that the value for the same element in array n is also changed, to reflect the length of the new text.

STRING HANDLING

Once again the Spectrum has its very own methods! There are no LEFT$, MID$, and RIGHT$ expressions on the Spectrum. Instead, string manipulation is done by 'string slicing'.

The part of an element of a string array that is required is given by (a TO b) where a is the start position of the sub-string, and b is the end position of the substring. This information is written alongside the variable name.

Therefore if X$ (3) = "ADVENTURE", then the three leftmost letters are returned to Y$ by:

```
    LET Y$ = X$(3) (1 TO 3)
```

The two rightmost letters are returned to Y$ by:

```
    LET Y$ = X$(3) (8 TO 9)
```

and the letters VENT returned to Y$ by:

```
    LET Y$ = X$(3) (3 TO 7)
```

Now had the array X$ been of length 15 instead of length 9, element 3 would have been "ADVENTURE   ". To get the last three letters of text, we would have used the same expression as above. However, to obtain these last three letters for any element, we need to know the text length. This is where array n comes in! To obtain the three rightmost letters in Y$ we can now say:

```
    LET Y$ = X$(3) (n(3)-2 TO n(3) )
```

More importantly, when displaying a list of objects along one line of print, we can obtain the whole word without trailing blanks by:

```
    LET Y$ = X$(3) (1 TO n(3) )
```

FOR/NEXT LOOPS

Note that the loop variable must be stated after NEXT, e.g. FOR I = 1 TO 11 : LET L$(I)=" " : NEXT I

VALUE OF STRINGS

The Spectrum's VAL function will cause execution of the program to stop, and display the message "Nonsense in Basic" if used for the value of a string containing a character following a digit. (The message should read "Nonsense in Sinclair Basic".) This presents a problem in the destination strings in array D$. All is not lost, as a blank, or space, is ignored, and will give the required result. So it is necessary to replace the asterisks in destination strings with blanks.

CHAPTER 9

Interpreting the Player's Input

INPUT A$. The player is asked for his next command. Difficult for him, obviously, because we haven't yet given him any display or output on which he may base a decision. However we must start the programming somewhere, and before it can display anything, the computer will need to 'understand' what it's been told. So first we will cover the section that follows INPUT. This section will be common to all games using the programming method, so when you have understood and entered it into the computer, it may be worth your while saving the lines described for later use.

Once we have determined how to simplify what the player is saying into numeric terms, the rest, hopefully, will fall into place relatively easily.

We have already established that we are going to use the first three letters only of each word of the player's input. This input may consist of either one or two words, the one-word response being reserved for a few special commands — QUIT SCORE HELP INVENTORY. First, then, we must split A$, the player's whole input, into two words. This can be achieved in the following way:

```
    410    LET  J%=0  :  FOR  I%=1  TO  LEN(A$)  :  IF
           MID$(A$,I%,1)=" "
           THEN LET J%=I%
    420    NEXT : IF J%=0 THEN GOTO 40110
```

This looks for the space between the two words, and if one is not found, the result will be a value in J% of zero, and control will pass to line 40110. The latter line can now be written:

```
    40110 LET Q$(2)="HUH?" : GOTO 100
```

The valid single word commands, which will be limited in number, can be picked out before this line, so that anything remaining must contain a space to be valid. Line 420 will give to J% a value equal to the position of the space contained in the input string A$. So now we can split A$ into two complete words using the value in J%:

```
420     NEXT : LET A1$=LEFT$(A$,J%-1) :
        LET A3$=RIGHT$(A$,LEN(A$)-J%)
```

We now have the two words, A1$ is the first word, A3$ is the second. To arrive at the first three letters of each is now easy using LEFT$. But where we place this can make a difference to the apparent response time. Also, we have still to cover the valid single word commands.

Going back a little to explain the use of Q$(2) in line 40110, we will use elements of the array Q$ to contain the reply from the computer to the player. The reason for doing this will be further explained in the section on screen presentation. Some of these replies will be individual to a particular command, but some will be replies that will cover a number of situations. These, I call "standard replies" and use the line range starting at 40000. Since we may well have further situations that call for the reply "HUH?", we will set it up as a standard reply. We can get hold of the first three letters of the first word without looking for a space. So let us go back a line or two:

```
400     LET A2$=LEFT$(A$,3)
```

will sort that one out. But supposing the player mistypes, or is just plain awkward, and only types in two letters? We will be staring an illegal function call error straight in the face, as the program breaks and we go into the READY mode. We must make this foolproof against idiots and fiends! So for good measure, let's rewrite line 400 to cater for just that eventuality:

```
400     IF  LEN(A$)<3  THEN  40000  ELSE  LET
        A2$=LEFT$(A$,3)
```

We have got rid of the problem to line 40000, which can be another standard reply:

```
40000  LET Q$(2)="IMPOSSIBLE!" : GOTO 100
```

Now we can sieve out our valid single-word responses. As these are fairly frequently used commands, we can afford to despatch them straight away to speed things up:

```
405 IF A2$="INV" THEN GOTO 4000 ELSE IF A2$="SCO" THEN
    GOTO 5000 ELSE IF A2$="HEL" THEN GOTO 6000 ELSE IF
    A2$="QUI" THEN GOTO 7000 ELSE IF A2$="JUM" THEN
    GOTO **** ELSE IF A2$="SWI" THEN GOTO ****
```

What we have done here is to cover all the one-word commands, Inventory, Score, Help, Quit, Jump and Swing. Having laid down our line numbering plan, we know the verb routines run from lines 1000—29998, and we shall start the routine for each verb on line numbers in increments of 1000. The most important and frequently used verbs are GO, TAKE and DROP, and as such we should place them before other verb routines, to reduce the time Basic takes to scan the program for the GOTO line. If we start each routine on line increments of 1000, then our next available start is 4000. Thus the Inventory routine will start at 4000, Score at 5000, etc. When we get to the end of the list of 'common' commands, we come to the verbs JUMP and SWING. These are intertwined in the depths of the plot, and it will pay us to write those lines in as we write each verb routine. If this sounds complicated, don't worry about it for the time being, suffice to say that we won't commit ourselves to the line number at which each of these routines will commence. To remind us that they are still to be written, we'll use asterisks in place of those numbers for now. We will surely not be able to forget about those words, for we have a syntax error sitting there waiting to remind us!

The next move is to check the verb to see if it is valid. However, since the most frequently used verb is likely to be GO, to save the scan time of the verb string, we can make GO a special case and deal with it first.

```
430     IF A$="GO" THEN GOTO 1000
```

and of course:

```
999     REM ***** GO *****
```

Having thus despatched GO to line 1000, we can conveniently forget about it for the time being.

The rest of the verbs can take their place in the string search, a subroutine that is the heart of the whole program. First we must search the string WV$ to see if it contains the first three letters of the verb entered by the player, i.e. A2$. Then we will have to search for the noun.

Let us find a spot to tuck this subroutine away. A look at our line-number plan suggests that 35000 might be a suitable line number, so we will start there.

```
34999  REM ***** STRING SEARCH SUBROUTINE *****
```

Since we are using a subroutine here we will have to ensure that all calls to it will have the same variable names for the search string and the string to be searched. We will call the string to be searched X$, and the string we are looking for Y$.

```
35000  FOR I%=1 TO LEN(X$) : IF Y$=MID$(X$,I%,LEN(Y$))
       THEN LET J%=I% : LET I%=LEN(X$)
35010  NEXT : RETURN
```

This little subroutine will now start searching string X$, starting at the first character. (Note that we started the loop at I%=1 rather than I%=0 which would have given us an error as there is no character at position zero of a string.) As we are going to use different strings as X$, i.e. the variables we have already set containing the recognition letters of the valid words, we have used LEN(X$) for the end of the loop, since its length will vary. The subroutine goes character by character down the string X$, looking for a set of letters of length LEN(Y$) that match Y$. If it finds one, the position of the beginning of the group of letters that match Y$ in the string X$, is assigned to variable J% and the FOR-NEXT loop is exited legally by equating I% to its end value. When the search is complete, the subroutine returns control to the main program which then uses the numeric information returned in the variable J% to identify the found word.

This is all very well, but there are three weaknesses. Firstly, we are checking every set of three letters, so that if the string X$ was ABCDEF, although we would only be looking to see if either ABC or DEF were present; in fact we would also be checking BCD and CDE. There is thus some ambiguity here, and with a bit of bad luck, a verb could be returned as valid when it should not be. Also, by checking unnecessarily the intermediate combinations precious time is being wasted. So we can dispense with these two problems by simply amending the loop:

```
    35000   FOR I%=1 TO LEN(X$) STEP LEN(Y$) : . . . .
```

By adding the STEP we have eliminated the difficulties and speeded things up for every search except an input of the first valid word in the target string.

The interpretation will rely on J% being returned as 0 if the verb is not found. As things stand, it will remain at the value to which it was last set unless it is initially set to zero. Once again we must modify the line:

```
    35000   LET J%=0 : FOR I%=1 TO LEN( . . . .
```

and we are there!

A word here about INSTR. This is a feature of some Basics, and provides a built-in string search facility.

For example, LET J%=INSTR(1,X$,Y$) would return in J% the start position of string Y$ in string X$, searching string X$ starting at position 1. J% would have a value of 0 if the substring was not found. If this feature is available in your Basic, then it could be used in place of the subroutine developed above, but its use brings with it some complications. A problem arises if the player types in 'rubbish' which happens to correspond to a combination of letters across word boundaries, since INSTR will search from every position, not just in jumps of length Y$. The only way to ensure the value returned in J% is correct to insert a foreign character to separate the groups in the string, e.g.:

```
    LET WV$="TAK*DRO*LOO* . . etc
```

A shifted character is ideal as it is less likely to be accidentally keyed by the player. If you do use this method, you will have to modify the line that detects the sequence number of the valid word.

Alternatively, you can insist that a valid value in J% is a multiple of the length of the string being searched for, plus 1

Now we can return to the place at which we left the main program and call the subroutine. First we need to check the valid verb string WV$ to see if A2$, the first three letters of the player's verb, is present. So WV$ must be named as X$, the variable searched in the subroutine, and A2$ as Y$, the variable searched for.

```
    440     LET X$=WV$ : LET Y$=A2$ : GOSUB 35000
```

If A2$ isn't found in WV$, then J% will be returned from the subroutine as zero, and this means that the verb isn't 'known'. Therefore we can add at the end of this line:

```
    440     . . GOSUB 35000 : IF J%=0 THEN LET Q$(2)="I DON'T
            KNOW HOW TO "+A1$ : GOTO 100 ELSE LET
            K1%=(J%-1)/3+1
```

If the verb isn't known, the reply variable Q$(2) will say as much, and execution returns to the start of the main program loop at line 100. Note that here we are using the '+' sign indicating the concatenation or stringing together of the text within the quotes plus the whole of the first word of the player's input, A1$. The ELSE will be executed if the IF statement is false, that is, if J% is not equal to zero, and the verb is a 'known' verb. At this stage, for the benefit of those without an ELSE statement, let us look at those lines re-written:

```
    440     LET X$=WV$ : LET Y$=A2$ : GOSUB 35000 :
            IF J%=0 THEN LET Q$(2)="I . . . . ." + A1$ : GOTO 100
    445     LET K1%=(J%-1) / 3 + 1
    450     . . . . as before . . .
```

You will see that it has been necessary to introduce a new line, 445, which illustrates the advantage of leaving spare numbers between lines! The statement following ELSE in the original line 440 has simply been placed on the new line, which will only be executed when the IF test in line 440 fails, causing the rest of that line — now minus an ELSE statement, to be ignored.

Thus if J% is returned to the program not equal to zero, we will need to find its position in terms of the word recognition groups from the start of the string WV$. At present we have only it's actual character position in the string. With a little arithmetic juggling, we can make K1% the value we need, and use J% again in our next search. The equation K1%=(J%-1)/3+1 may look strange, but try picking a couple of positions in the string WV$ which may be returned from the subroutine, (1, 4, 7 etc.), and you will see that it gives the number we need!

Our next task is to find a value for K2%, the number assigned to the noun or second word. Simple! Just do as we did with the verb. We have already saved ourselves the trouble of searching for the noun if the verb isn't valid, by passing back a message to avoid searching further. We know that the verb is good, so —

```
    450    LET X$=WN$ : LET Y$=A4$ : GOSUB 35000
```

and off we go on the search again! As it's a noun, this time if it can't be found the message will be slightly different:

```
    460    IF J%=0 THEN LET Q$(2)="WHAT IS A "+A3$+"?" :
           GOTO 100
```

and now the player will really begin to think that the machine understands English, even though it's vocabulary is somewhat limited! Back to our juggling act, this time using K2% as the noun number:

```
    470    LET K2%=(J%-1)/3
```

You will notice that the computations for arriving at values for K1% and K2% are:

K1% = (J% − 1) / 3 + 1
K2% = (J% − 1) / 3

K1% has been given a value one greater than K2% relative to the position of the word it represents in the string searched. You can see that the range of possible values for K1% starts from 1 whilst that for K2% starts from zero. There is a good reason for this.

K2% represents a noun. The first nouns in the string WN$ being searched represent objects in the object array, which are numbered from zero according to their subscript numbers in the array 0$. Thus we can use the value of K2% directly with the object-associated arrays.

K1% on the other hand, holds a number representing a verb. Verbs are present only in the string WV$, and so there is no need to associate them with any array subscript numbers. More importantly we will be writing a routine specific to each verb, and the value contained in K1% will be used to direct execution of the program to the appropriate routine. We will achieve this by means of an ON K1% GOTO . . . statement. This statement will not permit a value of zero in K1%. Therefore we have increased the value of K1% by 1, ensuring that no zero values appear.

It is worth mentioning that any reader whose computer has array elements starting from 1 and not zero, would have to add 1 to the value of K2% as well as to K1%.

Thus we have found two good words. They may be nonsense used together — that we shall sort out later — but at least we know they are both valid words.

Since the verb is 'known', we now have a limited number of possibilities to deal with, and can go sailing off to a routine dealing specifically with the verb entered:

```
    490    ON K1% GOTO 2000,3000, . .
```

So far we have decoded nouns as distinct from directions. A direction will only be valid following a verb that requests movement from one location to another, so we will treat directions in a slightly different way, as you will see shortly.

CHAPTER 10

On the move

The verb GO as used in an Adventure is in a different category from most of the others. In the example game here, it is unique. This is because GO alone (in this game) requests movement of the player from one location to another. The single words SWING and JUMP will do this in certain circumstances, but they have been isolated as single word commands. It is only the word GO in combination with a direction, that changes the value of PN% (the player's current location number).

Of course, it is quite likely that other moving verbs are desirable, such as SWIM, CLIMB, RUN etc., but for simplicity I have used only GO for our game; the same principle can easily be expanded to cover other words.

We left the word following GO out of our string search of nouns. This was done purposely because GO implies that it will be followed by a direction as distinct from an object-type noun. We thoughtfully placed the valid directions in a string of their own, separated from the common or garden nouns. By doing this, we have ended up with two short strings to search, WG$ and WN$ (see Chapter 8), rather than one long combined string. Once the verb has been decoded, we can direct the string search to one or other of these two strings, and will never need to search both, for if GO is detected we know we must be looking only for an exit, and must search WG$, not WN$. Thus the maximum length of the string search is reduced.

As all direction words will follow GO the searching of the 'direction noun' string can be carried out in the GO verb routine rather than in the main program. So line 1000 will read:

```
     1000   LET X$=WG$ : LET Y$=A4$ : GOSUB 35000
```

This time if zero is returned in J%, we will make use of a standard reply in the 40000 range:

```
    1000     . . . : IF J%=0 THEN GOTO 40010 ELSE LET
             X$=E$(PN%) : LET Y$=MID$(WD$,(J%−1)/3+1,1) :
             GOSUB 35000
    40010    LET Q$(2)="I CAN'T GO "+A3$ : GOTO 100
```

But what's this second search of E$ all about? If you cast your mind back, you will remember the exit codes were listed in Table 2. Having searched and found a valid exit word, we must check to see if it is a legal exit from the player's current location. We can find the code for the exit in WD$, in the same relative position as the 3-letter exit word group in WG$. So using MID$, we now take the exit code character corresponding to the exit number returned from the search of WG$, and search for its occurrence in the exit code string for the current location, E$(PN%). That position is given by (J%−1)/3+1, (our old friend!), and hence the expression following MID$ with which we are endowing our faithful Y$.

The search is now on for a string of length 1, and you can see why the subroutine was made able to handle a substring of any length. An all-purpose tool!

Again, if J%=0 the same message applies as it did if the exit wasn't recognised at all, so off we go to 40010 (we have used it twice, which was why we put it there rather than repeat it locally!)

```
    1000     . . IF J%=0 THEN GOTO 40010 ELSE
             LET PN%=VAL(MID$(D$(PN%),(J%−1)*2+1,2) : GOTO
             40020
    40020    LET Q$(2)="OK" : GOTO 100
```

If the exit found was legal for the current location, then we must take that exit and place the player in a new location. If you look at the way that the strings have been structured you can see that the location numbers in the string D$(PN%) are arranged in positions corresponding to those of the legal exits in E$. Since location numbers can consist of up to two digits, (being spaced one position apart for single digits) we must multiply the relative position of the exit code in its string by 2 to find the corresponding destination, i.e. the new location. The expression in Appendix 8 is thus incorporated in line 1000, where J% now represents the position in E$.

Having found this location number, remember that it is in string form. The location numbers themselves are held in numeric form, so we must apply the VAL function to the string found, to obtain a numeric value. This we can assign as the new value of PN%, the player's current location after moving. For this reason, the single digit location numbers were 'padded' with asterisks to the right of the numbers, as the VAL function will usually return a value of zero from a string containing a leading non-numeric character. If unfamiliar with the VAL function, it may be worth checking the way it operates on your computer to ensure you get the intended result.

Thus, the player's location is changed, a standard reply 'OK', is set in line 40020, and when the computer responds to the player's command it will reveal the details of the new location, L$(PN%).

That is all there is to interpreting the player's input. The driving force behind what, to the player, seems like the computer's ability to hold a conversation our little subroutine at line 35000, and the rest a little arithmetic juggling with the value returned acting upon the data we typed in right at the start.

CHAPTER 11
Screen presentation

So far we have been unable to test the programming we have already completed because we have written nothing that will display any information. We have now enough material on which to work to enable us to decide how to lay out the screen, and how we will effect the display.

Let us consider what information the player will need presented to him. He will need a description of his location, what objects are contained at that locality, and possibly the exits — or some of them — leading from it. He will obviously need to know the computer's reply to his most recent command, and he may quite often become so involved that he forgets what his last command was. It will therefore be an advantage to ensure this is also shown on the screen.

The simplest way to achieve this is to use the normal scroll mode of the screen. The advantage of this method is that quite a number of previous commands and replies will remain on the screen before disappearing off the top. A disadvantage is that the details of the location will also disappear. This can be overcome by a LOOK command, to cause the redisplay of L$(PN%), but it does require the player to use this command fairly frequently as an aide-memoire. Since one can normally see one's surroundings in real life, the player may feel that he is playing 'blind'.

Another method is to use a "whole screen" approach, whereby the screen is cleared and redisplayed after each command. By this means, the location can be re-displayed each time around, and the screen can have a tidier look — the way the information is set out can be controlled to a greater extent. The disadvantage is the loss of the player's commands, but this can be alleviated by using the variable A$, and redisplaying it.

A third method is a combination of both — a fixed display at the top of the screen, and a scrolling section below it, displaying the conversation between player and computer. This is the well-known split-screen method adopted by Scott Adams, and results in a very clear and recognisable presentation of information, combining as it does the best features of the two methods described above. Unfortunately, to perform this in Basic usually results in a rather slow and jerky display. It is best to use machine code if attempting this feat!

Since we are restricted to Basic, my own choice of method is the second one described, and that is the one we will adopt. We must now examine how we can best produce this display, and see if we can lay it out in a way approaching that used with a split screen.

The first task will be to clear the screen, and then to display the player's current location. The location descriptions held in array L$ are phrased in such a way that they can all be prefaced by the words "I AM":

```
330     CLS : PRINT:"I AM   ";L$(PN%)
```

Note the space between the M and the quotes. By not incorporating "I AM " in the variable we save space, since it appears only once as a PRINT statement, instead of in every location description variable.

Next, we will print some exits. We won't print them all, so that the player has to do a bit of guessing. The ones we will print will just be the straightforward ones, North, South, East, West, Up, Down and Out. For the exits we look to the exits string E$(PN%) associated with the location L$(PN%). We could write the program like this:

```
340     PRINT "SOME EXITS ARE :";: FOR I%=1 TO
        LEN(E$(PN%))
350     IF MID$(E$(PN%),I%,1)="N" THEN PRINT "NORTH.";
360     IF MID$(E$(PN%),I%,1)="S" THEN PRINT "SOUTH.";
```

etc., but we won't. Before we discuss why, let us go on to consider the visible objects.

To print a list of the objects that are visible, we need to examine the value of each element in the array P%, and print only those whose value is equal to PN%. So we could write the code to do this as follows:

```
410     PRINT "I CAN SEE :";: FOR I%=0 TO 10
420     IF P%(I%)=PN% THEN PRINT O$(I%);".";
430     NEXT I%
```

but we won't do that either!

There are two difficulties that present themselves. Firstly, when scanning the strings and checking the results with IF tests, those that are true will occur at random intervals in the search. The information will be displayed as it is found. Unless the Basic interpreter is particularly fast, this method of printing will tend to lead to a jerkiness in the display, particularly if the object list is long. The computer will look as if, having found one value to print, it is now having to think hard to find the next one. That is, of course, precisely what it is doing, but it makes the computer look indecisive, giving the player an uneasy feeling that the program is about to crash. Far better for the computer to gather all the information it

has to display together, and put it out on the screen in one go. It is easier on both the eyes and the nerves of the player.

The second problem is one of screen width. There is no knowing how many objects may be present simultaneously in one location. Some silly fool of a player might decide to collect everything he possibly can, and dump it all in a central spot. The chances then are that the list of objects when printed, will exceed the width of the screen — even if 80 columns! To control the wrap-around, so that the break occurs between whole words, is extremely cumbersome with the method described above. We can alleviate it by displaying the message "I CAN SEE :" on a separate line, giving us 11 more columns for the object list, but that is unlikely to be a complete cure for the problem. If each object is displayed on a separate line we will be scrolling the location description off the top pretty soon — not an answer either!

So the method we will use is to set up variables containing the objects and exits. We can more easily control their lengths, and can also display them quickly once they are assigned, eliminating any jerkiness in the response.

We will go one step further, and gather up all the display information into variables before we even clear the screen. Thus, the player will not be staring at a blank screen, and he is less likely to wonder if anything is ever going to happen. When it does it will be fast and smooth.

Back to the drawing board with the exits then! We will set up a variable EX$ which will contain all the valid exits strung together:

```
200     FOR I%=1 TO LEN(E$(PN%))
210     IF     MID$(E$(PN%),I%,1)="N"     THEN     LET
        EX$=EX$+"NORTH. " etc. . . (see full listing)
280     NEXT I%
```

This method is satisfactory in the demonstration game for screens of 32 column width upwards. If your screen is less than 32 columns, then you can either reduce the number of displayable exits, or adopt a method similar to the following, used to set the display variables for the visible objects.

Objects present a more difficult problem, as the number and range present at any location is determined by the player, who can cart them about willy nilly. Let us take it line by line:

```
290     LET II%=0 : FOR I%=0 TO 10 : IF P%(I%)=PN% THEN
        LET OS$=O$(I%) ELSE NEXT I% : GOTO 330
```

Here, we are scanning the object location array O$, and if an object is found in the current location PN%, its description is copied in to variable OS$ (Object Seen). If that happens, the NEXT I% and GOTO 330 is ignored, and control passes to the next line, otherwise the scan is continued. The variable II% will be used shortly, and it is initialised before the FOR-NEXT loop is entered.

Having found an object that is present in PN%, we put it into a member of the array V$ (Visible), whose length we can control:

```
310    IF LEN(V$(II%))+LEN(OS$)<61 THEN
       LET V$(II%)=V$(II%)+OS$+". " : LET OS$="" ELSE
       LET II%=II%+1 : GOTO 310
```

We are now using the array V$ to build up a series of objects descriptions. Each element of this array will have a length no greater than (width of screen — 3) to allow a full stop and space to be tacked on to the end without scrolling to the next line occurring. In this case the screen is 64 columns, so the figure of 61 has been used. This figure should be changed to suit the width of your own screen. Every time OS$ is set by an object present in location PN%, the potential length of V$(0) with the new OS$ added is tested against (width — 3). If it is less than (width — 3) then OS$ is added to the end of V$(0) and a full stop and space added. OS$ is made null, and then control passes to the next line:

```
320    NEXT I%
```

to continue the scan of objects. If the proposed length of V$(0) is too great, and the test fails, the ELSE part of the line executes and instead of tacking OS$ to the end of V$(0), II% is increased by 1, and control sent back to the beginning of the line. The test is now made against V$(1), which of course has no length yet, and thus starts to build up a list of more objects. And so on.

The original FOR-NEXT loop is continued in line 320, repeating the search until the object array is exhausted, when control passes to line 330 from 290. If the search ends in line 320, then of course, there is no need for the GOTO 330, as 330 is the next line in sequence.

The dimensioning of the array V$ must be reckoned by calculating how many objects may possibly be dropped in any one location, and working out how many elements of V$ these would require. Remember when making this calculation, that the objects will be added into an element of V$ in the order in which they appear in the object array O$. The dimension of 8 is far greater than the requirements for this game for a 64 column screen, and should be ample for the narrowest screen used.

All these manipulations with strings tend to use up string space, and the more objects and exits displayed in a given location, the more string space will be used. When we come to the actual display, we may well find that string space is so short that there is a hiccup in the display whilst the computer sorts itself out. Alternatively, we may run out of string space altogether, with the result that execution stops with an 'out of string space' error. In either case, more string space will have to be cleared. In a large game, memory restrictions may make this difficult, in which case a juggling act must be performed to see how much string space can be cleared without resulting in an 'out of memory' error. The more string space that can be cleared, the faster, on the whole, the program will run, so it is adviseable to grab hold of as much as you can.

The dimensioning of V$ can be reduced below its true requirement by setting the final element to say "AND MANY OTHER THINGS" if too many objects to be accomodated are dropped at any given location. This is most likely to occur at the location designated as the treasure store (assuming that there are a number of treasures in the game). Since the objective of the game will be to drop these treasures at the location, the last thing we really want is for other objects to be dropped here as well. If this location proves a problem to you, you may like to try preventing anything except treasures being dropped at the treasure store, and we will come back to that subject when we cover the verb DROP.

Now for the crunch! We can start to display things:

```
330    CLS : PRINT"I AM ";L$(PN%) : PRINT : PRINT"SOME
       EXITS ARE :" : PRINT EX$
340    PRINT"I CAN SEE :" : FOR I%=0 TO 8 : IF V$(I%)<>""
       THEN PRINT V$(I%)
350    NEXT I%
```

We now have the conversation between the player and the computer to consider. We have already set some computer replies in elements of the array Q$. We will treat that array in a similar way to the V$ array, and print only the non-null elements. First we will display the command to which Q$ is the response. We have that command in the variable A$ which was set with the player's input, and so we can print the command and reply, and await input:

```
360    PRINT A$ : FOR I%=1 TO 3 : IF Q$(I%)<>"" THEN
       PRINT Q$(I%)
370    NEXT I%
380    INPUT A$
```

This won't do as it stands. We must tell the player what the information is all about, and set it out on the screen in an attractive way. We could start off by printing a complete line under the list of objects, to separate that from the conversation and give the appearance of a split screen. But let us not try to simulate something that we're not actually doing; instead we will make it obvious that each display is new, and make it as easy to follow and as pleasing on the eye as possible.

First we'll provide a degree of separation between the object list and the conversation by leaving a line blank. All that we need to do here is insert a straight PRINT statement. Then we will add to that separation by printing a short line leading to some words explaining to what A$, when displayed, refers:

```
360    PRINT : PRINT "→YOU SAID ";A$ :
       FOR I%=1 TO 3 :
```

etc. The replies will print as shown in the first version of line 360, whilst we can ask for input in line 380 using another short line, thus providing a boundary at each end of the conversational part of the text:

```
380     PRINT"→WHAT NOW ";INPUT A$
```

Depending upon the number of lines displayable on your screen, and the length and number of the location descriptions, exits and objects that must be displayed, you may find a more pleasing appearance is obtained by inserting a few more PRINT statements between lines, to spread it all out a bit. I have done this, as you can see in the full listing.

A most important point to remember now, is that when we set the display variables, we assumed they were null to start with — in fact we relied on this, because if null, they were not printed. So we must at some stage set these to null, before the next time around as it were. We can be a little cunning here! Setting all these variables to null will take time — not really noticeable in isolation, but we still have a lot more to get through, and it all adds up! The time delay that the player will notice occurs between his pressing ENTER, and the screen clearing to redisplay. For at least half a second following the new display, our player is going to be checking what he now sees before starting to type input. This is the place to null our variables — after the display and before the input. Providing it doesn't take so long that he is typing away without any effect, now having received the prompt, the player will never notice the short delay! So we will amend line 380 by removing the input request:

```
380     PRINT"→WHAT NOW ";
```

and do the necessary instead in line 390:

```
390     FOR I%=0 TO 8 : LET V$(I%)="" : LET Q$(I%)="" :
        NEXT I% : LET A$="" : LET A1$="" : LET A2$="" : LET
        A3$="" : LET A4$="" : LET EX$="" : INPUT A$
```

and while he quietly reads the displayed messages, the computer will be toiling away in the background setting the stage for the next command!

Having finally got it all on the screen, at the start of the game, it should now look as shown in Appendix 9. This is almost the moment when we can start testing our progress so far, to see if it works as planned!

CHAPTER 12
Let's GO!!

Now that we have managed to put something out onto the screen, we can test our progress so far. Run the program, and you should find that you can move about, as we have already covered the verb GO. You can't? Of course not, we have got to pull that damned wallpaper off to get out of the first room! Well, at least it proved that part of the game works! It's just as well the attempt at leaving the room was thwarted, as we have not yet written line 100, the start of the program cycle. As we haven't got to the stage of deciding exactly what to put in line 100, in order to avoid an Undefined Line error, we had best insert a REM to keep us going for the moment:

```
100     REM ***** FIRST LINE OF MAIN LOOP GOES HERE *****
```

and since this will be replaced eventually by something more useful, a more permanent REMark should be placed on line 99:

```
99      REM ***** START OF MAIN PROGRAM LOOP *****
```

We now have a choice between proceeding with yet more programming to let us out of the room, and taking a short cut by temporarily changing our start location. The latter is the safer course, or we may end up compounding our errors. So let's position ourselves in the hallway. We can either press BREAK and from the command mode type:

```
>LET PN%=1 : GOTO 100
```

which should take us back into the game at the hallway, or change the value of PN% in line 50 from 0 to 1. Having freed ourselves of Room 0, we can GO about part of the network, so recap on the plot, and try going places that should be both accessible and inaccessible. It is just as important to check that invalid exits can't be used as it is to ensure the player can move where an exit should exist.

At this point, you should find that you can't cross the moat, and that although you can get back into the first room, you still cannot get out! That is because no attempt can be made to re-enter the room in normal play until the exit has been cleared from within, so there is no need to restrict movement in the opposite direction.

A limited amount of exploration can be carried out in the castle, so having checked out the cottage and forest, press BREAK and type:

```
>LET PN%=5 : GOTO 100
```

and try going up and down ladders and stairs.

If you end up in the wrong location, or the exits and lack of them don't work in the correct sense, then there are some variables set that can be questioned during these tests. Press BREAK, and type:

```
>PRINT PN% : PRINT E$(PN%) : PRINT D$(PN%) : PRINT WG$ :
 PRINT WD$
```

If you were in location 1, then the screen should look something like Appendix 10, and you can check the values through to see where the problem is.

Now we can see what is happening, as each section of the program is written, it should be tested, and if not working correctly the appropriate variables can be checked from the command mode in this way.

By now, you're getting used to the screen layout, so stop to consider any improvements that might be made. Spread it out a bit more? Change the lines of dashes to some other more fancy character at your disposal? You know exactly where the lines are that set up the screen, so it is an easy job to experiment until you get the right 'look' for your machine. After editing, of course, you will need to type RUN rather than GOTO 100, since the values of the variables will be lost, on most micros. Here the Spectrum in once again an exception and lines may be edited and the program resumed at line 100 with the variables intact.

Assuming that all is well with the game thus far, we can continue with the coding. Each new verb we add will open up parts of the game until the whole is integrated into an Adventure capable of solution.

CHAPTER 13
Take it or leave it

The verb GO, we decided, is in a special category because alone it moves the player around the network. There is yet another special category, TAKE, DROP, and INVENTORY, which are unique in that they involve possession, and will, under normal circumstances, only affect or list the player's inventory. Using these verbs, he can carry objects around from one location to another where he can put them to good use. Rarely will the means of carrying out a task be provided at the location in which the work must be done!

Before an object can be DROPped, it must be TAKEn, so logic dictates that we cover that verb first. Although there are not too many takeable objects (i.e. those we defined with a positive value for C%), it would be unrealistic to assume that the player can go on picking things up ad infinitum. We can set a limit to the number of items the player is capable of carrying, but must ensure that this maximum will not prevent the game from being completed. In this game it would be reasonable to limit the inventory count to 4.

We set up a count of the player's inventory, IN%, in line 50, and set it to 1, to account for the axe already in the player's hands.

Although we have 11 objects in the game, there are also nouns that are not objects, and are added on to the end of the object list in the noun recognition string WN$. We will have to check that an object player seeks to take is in the same location as the player, i.e. PN%. To do this we will take the value in the variable K2%, (the position of the object found in the string WN$), and check the value of the K2%th element in the object location array P%. Thus we will be looking at the value of P%(K2%). If the noun used is valid but not an object, the position of it in WN$ will be beyond those of the real objects, and K2% will return a value greater than 10. A mention of P%(K2%) when K2% is greater than the size of the object list will cause the computer to splutter with an error caused by a subscript beyond the dimensioned range. Remember, P% was dimensioned to 10 to include 11 objects. So our first task will be to filter out values of K2% greater than 10. Since we know that if K2% is greater than 10 it can't be taken anyway as it is not a real object, we can send such occurences off to a standard reply to that effect. So far then, we can write:

```
1999   REM **** TAKE ****
2000   IF K2%>10 THEN GOTO 40030 . . . .
40030  LET Q$(2)="DON'T BE ABSURD!" : GOTO 100
```

Next we should check whether the player is already carrying the object he has requested to take. If he is, then P%(K2%) will have a value of 55:

```
2000   . . ELSE IF P%(K2%)=55 THEN GOTO 40040 . . .
40040  LET Q$(2)="I'M ALREADY CARRYING IT!" : GOTO 100
```

If the object is not present in the location:

```
2000   . . . ELSE IF P%(K2%)<>PN% THEN GOTO 40050 . .
40050  LET Q$(2)="I DON'T SEE IT HERE" : GOTO 100
```

Taking a look at the flags that control objects, we decided that if the C% value was equal to −2, it could never be taken, whilst if it was −1, it could be taken at some later stage. Therefore continuing the same line:

```
2000   . . ELSE IF C%(K2%)=−2 THEN GOTO 40000 ELSE IF
       C%(K2%)=−1 THEN GOTO 40060 . .
```

We have already written 40000 as "IMPOSSIBLE", and the other line can be:

```
40060  LET Q$(2)="I CAN'T — YET!" : GOTO 100
```

giving the player some light at the end of the tunnel. Anything that gets through these conditions is a prime candidate for being taken, but one more check to see if the inventory is already full:

```
2010   IF IN%>4 THEN LET Q$(2)="I'M OVERLOADED
       ALREADY" : GOTO 100 . . .
```

If that succeeds, the object can be taken. We must increment the inventory count, place the object in the player's hands, and reply that all is well:

```
2010   . . ELSE LET IN%=IN%+1 : LET P%(K2%)=55 : GOTO
       40020
```

Line 40020, as you will no doubt recall, sets the reply to "OK" and sends control back to line 100.

Before testing our new-found ability to go around grabbing things, it will be as well to provide the facility for dropping them as well, otherwise testing TAKE will become very tiresome as the inventory limit is reached.

To drop something, we must again check that the noun entered is within the dimensions of the object array. Other than the obvious — that the player must be carrying it, there are no further restrictions we need apply:

```
2999   REM **** DROP ****
3000   IF K2%>10 THEN GOTO 40070 ELSE IF P%(K2%)<>55
       THEN GOTO 40070 . . .
40070  LET Q$(2)="I'M NOT CARRYING IT!" : GOTO 100
```

Notice here that although both tests, if true, result in a GOTO pointing at the same line, we cannot OR them. That is to say, we cannot write IF K2%>10 OR P%(K2%)<>55, and the reason for this is the same reason we got rid of anything with a K2% value greater than 10 in the first place. If the two conditions were ORed in this way, a subscript error would still occur. By despatching non-object nouns elsewhere first, there is no chance of K2% being greater than 10 when applied to P%(K2%). As the reply is the same in each case, line 40070 is the target both times.

Continuing with line 3000, we can now decrement the inventory count:

```
3000   . . ELSE LET IN%=IN%−1 . . .
```

Before wrapping this routine up, let's build a trap for the player, so that if he drops anything from the top of the castle, it will fall into the moat, never to be seen again! If dropped elsewhere, the object will land back in the player's location:

```
3000   . . : IF PN%=7 THEN LET P%(K2%)=88 : LET Q$(2)= "IT
       FELL OFF INTO THE MOAT" : GOTO 100 ELSE LET
       P%(K2%)=PN% : GOTO 40020
```

An object dropped from a great height now goes to a non-existant location, 88, and, search as he may, the player will not be able to find it! Eventually he may have to start playing the game again, particularly if he has lost something important! At any other location the objects are assigned to the current value of PN%, and line 40020 tells the player "OK".

If you were writing a game with a large number of objects and treasures, and, as I suggested earlier, wanted to restrict the objects dropped at the treasure store location, then you should arrange your treasures to be the last objects grouped together, in the object array. After the first two conditions in line 3000, you could then insert the following:

```
. . ELSE IF PN%=(n) AND K2%>(m) THEN GOTO xxxxx . .
```

where n is the location number of the treasure store, m is the highest object number which is not a treasure, and xxxxx is a line number in the standard reply range saying:

```
XXXXX LET Q$(2)="ONLY TREASURES HERE" : GOTO 100
```

There is one final point to cover here that is specific to the game under discussion. The plot calls for a tree to be felled and taken to the moat, where it will form a bridge into the castle. It would not be realistic to suppose it could be carried, so we will have to overcome that problem in another way. The C% flag for the tree is initially set at −1, providing for the standard reply "I CAN'T — YET!". When the tree has been felled, we will change its C% value to 3, and can now tack one final condition onto the end of line 2000:

```
     2000    .. ELSE IF C%(K2%)=3 THEN GOTO 40080
```

and

```
     40080   LET Q$(2)="YOU MUST BE JOKING!" : GOTO 100
```

Objects can now be taken and dropped! Run the program, and test for these verbs thoroughly. Check that when the inventory limit is reached the right reply is received, check than an object can't be taken if it is not in the right place, check that stairs and ladders and trees cannot be pocketed, and try the trap built in to location 7.

We now need to be able to list what the player is carrying, to provide a response to his INVENTORY command:

```
     3999    REM **** INVENTORY ****
```

We will build up the inventory list in a similar manner to the way in which we built up the visible objects list, since again, this could be quite long. This time however, we will make use of the reply array Q$ to hold the list. We will use J% to hold the subscript number of Q$ that we are currently filling, and then scan the array P$ for values of 55:

```
     4000    LET Q$(2)="I AM CARRYING:" : LET J%=2 : FOR I%=1
             TO 10
     4010    IF P%(I%)=55 THEN IF LEN(Q$(J%) )+LEN(0$(I%) ) >
             62 THEN LET J%=J%+1 : GOTO 4010 ELSE LET
             Q$(J%)=Q$(J%)+0$(J%)+". "
     4020    NEXT : GOTO 100
```

The inventory list is now built up, and the first words in the first element of Q$ to be displayed will say "I AM CARRYING".

By now you are probably wondering why we have avoided using that first element, Q$(1) for any of the replies we have framed so far. That is because, as the program progresses, we may want to print a spontaneous message, generated by a specific combination of circumstances in the state of play. Since such a message may be generated irrespective of the particular command from the player, conflict between the reply to the command and the message must be avoided, or one or other message will be overwritten. So Q$(1) has been reserved for these spontaneous messages, and all replies to commands start from Q$(2). More on this in a while.

CHAPTER 14 Some other common commands

In this game the scoring is very simple. The object is to find and safely get the treasure (object no.6) back to location 0, where the player first started out. The SCORE command routine has only to check if P%(6)=0. If it does, then the player has succeeded, a suitable message is displayed and the game ended.

In an Adventure with more than one treasure, things get more complicated. One way is to arrange, as I suggested when describing how to limit DROPping non-treasures, for the treasures to be grouped at one end of the object array, and to use a FOR-NEXT loop on that section of the P% array to see how many are in the correct store location.

It might be, however, that you adopt a method of modifying existing object descriptions. An example in this game is the tree, which when chopped down carries the same object number, but whose description changes to "tree trunk". If this sort of modification is used to generate treasures from objects that were not originally treasures, then the fact that they are all grouped together at one end of the object array will not solve the problem, as there will be no check as to whether or not the conversion has taken place. The non-treasure version might have been the one that was dropped in the treasure store! But grouping the treasures will still help by reducing the length of the search.

If the treasures are highlighted by a special character such as an asterisk to make them recognisable, we can use that asterisk to test for 'treasurability'! The asterisk will be an aid both to the player and computer in recognising which objects are treasures. Our SCORE routine might run something like this:

```
     5000    FOR I%=n TO m : IF P%(I%)=y AND LEFT$(0$,I%)="*"
             THEN LET SC%=SC%+1
     5010    NEXT I%
```

with a suitable message to display. Here, n is the start element of the treasures in the object array, and m the last. Y is the location number of the treasure store. SC% is introduced to hold the score, and would have to be made zero either at the beginning of the score routine, or after the score had been displayed.

The detection of the an asterisk in this way is an alternative method that can be used to prevent non-treasures being dropped in the store.

Another standard command is HELP, the author's way of making things a little easier for the baffled Adventurer. It is usually possible to predict most of the problems with which the player will need assistance, and these will tend to be dependent upon which location he is in. Therefore, the HELP command can be serviced by a fairly simple routine giving a set reply for each location:

```
5999   REM **** HELP ****
6000   ON PN%+1 GOTO 6010,6040,6040,6040,6020,6030 . . .
6010   LET Q$(2)="ISN'T THE WALLPAPER LOVELY?" :
       GOTO 100
```

etc. Line 6010 draws attention to the wallpaper in the hope that this will help the player overcome his likely problem in location 0, to get out of the room. In line 6000 it is necessary to add 1 to the location number, since this series starts at 0, whilst ON (variable) GOTO will not respond to a value of 0. The popularity in line 6000 of line 6040:

```
6040   LET Q$(2)="EXAMINE THINGS & LOOK AROUND" :
       GOTO 100
```

is because this is a reply which does not really give a special clue, it being decided that there are likely to be no major problems in the locations using this line. The complete list of HELP lines is shown in the full listing of the game.

QUIT is the last resort of the desperate player, and quite simply, ends the game. As a consolation we can do the decent thing and tell the player how well or badly he has fared, by displaying the score before we sign off. It could be, of course, that the player has fulfilled the objective of the game without realising it, not having thought to type SCORE! In this game, with only one treasure, we can grade the player's effort by detecting whether he has found the treasure, even though unable to get it home:

```
6999   REM **** QUIT ****
7000   CLS : IF P%(6)=0 THEN PRINT . . . (success message)
       ELSE IF P%(6)<>99 THEN PRINT . .(moderate success)
       ELSE PRINT "BETTER LUCK NEXT TIME!"
7010   END
```

remembering that 99 was the fictitious location assigned to the treasure to keep it from view until discovered.

Another common verb is LOOK. This is usually synonymous with EXAMINE, except in some scrolling-screen games, where LOOK on its own will recall the location details. We will make LOOK and EXAMINE synonymous for this game. The first thing we must do is to add the two verbs to the string of valid verbs, WV$. Edit line 60, and it should look like this:

```
60     LET WV$="TAKDROLOO"
```

We anticipated that LOOK would be high on our list, and inserted it in the string. Now to that list, add EXA:

```
60     LET WV$="TAKDROLOOEXA"
```

When the necessary searches have been made, LOOK will return a value of 3 to K1%, and EXAMINE will return 4. Now we must point to the routine we are about to write, so edit line 490:

```
490    ON K1% GOTO 2000,3000
```

We have used lines 4000 — 7000 for the verbs already covered, so our next available line for a verb routine will be 8000. This must be added twice to line 490 if we are to have one routine for the two new verbs:

```
490    ON K1% GOTO 2000,3000,8000,8000
```

and having set things up, we can start on the routine itself:

```
7999   REM **** LOOK/EXAMINE ****
8000   IF K2%<>11 THEN GOTO 8110 ELSE ON PN%+1
       GOTO 8010,8110,8110,8110,8020,8030 . . .
```

In 8000 we are taking LOOK AROUND as a separate case from a command that is to LOOK or EXAMINE a specific object. If AROUND is used, then K2% will have a value of 11, and the ON . . GOTO will operate, based on the location of the player. LOOK AROUND is very similar to HELP, and hidden things will be revealed in different locations. If K2% doesn't have a value of 11, then control will pass to line 8110, and in this game, the only object examined that will elicit further information is the packed lunch (object no.2). The player doesn't know it, but it's poisoned, and we will give him a clue, should he be prudent enough to check it out!

```
8110   IF K2%<>2 THEN LET Q$(2)="I SEE NOTHING
       SPECIAL" : GOTO 100 ELSE LET Q$(2)="LOOKS A BIT
       EVIL TO ME!" : GOTO 100
```

Since line 8110 is the end of the routine for these verbs, we could have rephrased that to avoid the repetition of the GOTO 100, in this way:

```
8110   IF K2%<>2 THEN LET Q$(2)="I SEE NOTHING
       SPECIAL" ELSE LET Q$(2)="LOOKS A BIT EVIL TO
       ME!"
8120   GOTO 100
```

but the line overhead on memory would be more than the saving in code, so on balance it's not worth it.

The LOOK AROUND replies will be accessed via the ON . . GOTO, and the first of these is for location 0.

```
8010   IF C%(1)<0 THEN LET Q$(2)="THE WALLPAPER'S
       PEELING" : GOTO 100 ELSE GOTO 8110
```

Notice that all the time we must check the relevant conditions so as to keep the replies logical. If the wallpaper has been pulled off, then it will have become TAKEable, but must not be allowed to fall off the wall again. C%(1) will have a value of 2. In that case, the GOTO 8110 will result in a reply "I SEE NOTHING SPECIAL", but if the player still hasn't caught on and pulled the paper, C%(1) will be −1, and giving the clue.

Turn to lines 8020 to 8060 in the listing. Line 8020 is for a look around by the moat. First a check is made to see if the tree is there. If it is, it will form a bridge, and the clue is given, but no tree and a different message results! At present we have no way of taking the tree to the moat, or even of chopping it down, but because we have detailed the full plot, we know we will be having to cater for this occurrence later — a good example of why the whole plot must be complete to the last detail before the program is written!

When the player has crossed the bridge, a return across the moat is made impossible to force him to solve the problem concerning the rope. How do we account to the player for this? Line 8030 has the LOOK AROUND answer.

```
8030   LET Q$(2)="TREE HAS SLIPPED INTO MOAT" : GOTO
       100
```

And so on. Have a further perusal of these lines, and follow the logic through before pressing on with the next chapter.

CHAPTER 15
Trees and Paper

Having written the program code covering the basic verbs, the framework for the game is complete, and now it's just a matter of working steadily through the plot to put the flesh on the bones.

Tackling this in a methodical way, we will put ourselves in the position of the player, and provide him with the verbs in the order in which he needs them. The first thing he must do is to get out of the wallpaper room. He can already LOOK AROUND and get the clue, and cry HELP for another clue. He can ask for INVENTORY which will tell him he's got an axe which will hopefully persuade him to try hacking his way out first. The first verb he will require to proceed from the room is PULL, to get rid of the wallpaper and uncover the hidden door. Before we rush off into the coding we must ascertain if there will be any other uses for the verb PULL. Since we have still to get over the problem of moving the felled tree from the forest to the moat, which we decided could not be TAKEn, we had better let our player PULL it along.

Let us get clear in our minds under what conditions we can use PULL to effect. The only objects that are PULLable are the TREE and the WALLPAPER, so the value of K2% will have to be either 3 or 1, the object numbers of the tree and wallpaper respectively. As we will be moving the tree around without carrying it, if TREE is PULLED it must be in the same location as the player. Also, of course, it must first have been chopped down. When this has been completed we will arrange for its C% flag to have a value of 3. For the WALLPAPER to be pulled, the player must be in location 0, and the wallpaper must still be in position on the wall, so it's C% flag must be −1. Off we go then:

```
8999   REM **** PULL ****
```

```
9000  IF K2%=3 AND C%(3)=3 AND P%(3)=PN% THEN
      GOTO 9010 ELSE IF PN%=0 AND K2%=1 AND
      C%(1)=-1 THEN GOTO 9020 ELSE GOTO 40090
```

If these tests both prove false, then control passes to a standard reply:

```
40090 LET Q$(2)="OK — NOTHING HAPPENS" : GOTO 100
```

but if either test is true, then we go on to line 9010 or 9020.

As far as the wallpaper is concerned, when pulled we must make it TAKEable by changing the value of its C% flag. This will automatically render pulling it again ineffective. We must make the door appear, and provide an exit through it, and then return with a suitable message:

```
9020  LET C%(1)=2 : LET P%(8)=0 : LET E$(0)="A" : LET
      Q$(2)="IT JUST FELL OFF!" : GOTO 100
```

and that sorts out the wallpaper!

Concentrating next on the tree, if it is pulled when in the moat location, we can make it form a bridge, otherwise we can set things up so that PULL causes it to follow the player if he changes location on his next turn. We will have to do this by setting a new flag PC%, signifying whether the tree is to follow the player. It's value will be 1 if the tree is to follow the player, (because the player has just commanded PULL TREE), and 0 if it is to remain in its present locaton (no PULL command immediately prior to a GO). Once the tree is a bridge, it will be fixed and an exit across the moat provided:

```
9010  IF PN%=4 THEN LET C%(3)=-2 : LET O$(3)="TREE
      TRUNK CROSSING MOAT" : LET E$(4)=E$(4)+"J" :
      LET PC%=0 ELSE LET PC%=1 : GOTO 40020
```

The C% flag controlling TAKEability is set negative and the "follow" flag is set to zero if the current location is 4. Otherwise the follow flag is set to 1, and an OK message despatched from line 40020. We can't do anything in the PULL routine to ensure that the tree follows the player, because he will have to use GO next, for that to happen. So we must put in a line that will be checked every time round the main loop, and will move the tree on if it sees the follow flag set. This is where our dummy line 100 becomes a real one! Certain events such as this are independent from any verb routine, and must be actioned after the verb routine has been executed:

```
100   IF PC%>0 THEN LET PC%=PC%+1 : LET P%(3)=PN%
      : IF PC%=3 THEN LET PC%=0
```

and that is sufficient to do it! The reason for increasing the value of the PC% flag if it is set, is that this line will be executed directly after the command PULL has been entered. If this is the case, PC% will already be set at 1, so it will then be increased to 2. The tree is already in the player's location, (line 9010 saw to that), so LET P%(3)=PN% will not have any noticeable effect. If the player next commands GO SOUTH, then PC% is increased to 3, the tree is assigned to the new location, and

the PC% flag reset to zero, forcing the player to use the PULL command again to get the tree moving!

To access and test the verb, add PULL to the end of WV$ in line 60, and add ',9000' to the end of the list of ON K1% GOTO in line 490. As each new verb is added you should extend WV$ and line 490 in this way. From now on I will not repeat this reminder for each verb, but rely on you to remember to do it! The building up of these lines is best carried out as each verb is written, as to complete them at the outset will complicate things if unforeseen changes to the vocabulary or plot are found necessary as the programming proceeds.

Enough pulling — on to our next verb. The player, out of the house, now needs to CHOP the tree down, before he is able to move it by pulling.

For a successful CHOP, the player must first be carrying the axe:

```
9999  REM **** CHOP ****
10000 IF P%(0)<>55 THEN LET Q$(2)="NO CHOPPER!" :
      GOTO 100
```

That settles that one. Let's now sneak in a guess that the player will try to hack his way out of the first room — it's nice to keep one ahead:

```
10010 IF PN%=0 THEN LET Q$(2)="YOU WON'T GET OUT
      BY FORCE!" : GOTO 100
```

and now to deal with the real thing, the object must be TREE, the location must be where it grows, 9, and it must not be already felled, so C%(3) must be −1:

```
10020 IF K2%=3 AND PN%=9 AND C%(3)=-1 THEN LET
      C%(3)=3 : LET O$(3)="TREE TRUNK" : GOTO 40020
```

which gives the player a new description, confirming something has happened, as well as an OK. If the tree isn't in the location, we must tell the player:

```
10030 IF K2%=3 THEN GOTO 40050
```

We know that the location can't be 9 with the tree growing, or the previous line would have trapped the command. What shall we do if the player, axe in hand, decides to chop something different? Well, whatever he chops, it will be a recognisable object, since otherwise the word would have been rejected by the string search routines. Let's simply say:

```
10040 GOTO 40020
```

giving him an OK which might worry him a bit! When the game is complete, here is some scope for you to enhance it with a few nasties!

One other thing that should be done here which we decided upon in the last chapter: having chopped down the tree and pulled it into position, we must bar the exit from the castle once the tree-bridge has been crossed, thereby forcing the player to solve the rope problem. As soon as the player crosses the bridge, we can arrange for the tree to slip into the moat. If the tree is forming a bridge, its C% flag will be −2; that is one condition. Another condition is that the player must be in the castle, location 5. We will be kind to the player, and replant the tree in the forest, enabling him to use it again if he desires. Hopefully he will think it is an entirely different tree, after all, it is a forest! But really it will be the same old tree that apparently fell into the moat! This will be achieved with another special condition line:

```
110    IF C%(3)=−2 AND PN%=5 THEN LET P%(3)=9 :
       LET O$(3)="TREE" : LET C%(3)=−1 :
       LET Q$(3)="OOPS! TREE JUST FELL IN MOAT!"
```

There is no need to change the exit string for location 5 because it never existed in the first place! The player is now trapped in the castle, unaware that the tree is available to be chopped down again.

Although it is hard to envisage any circumstances within the scope of this plot when the player might need to re-enter the castle, replanting the tree gives scope for further additions to the game, such as providing a key to the cupboard, which is hidden somewhere outside the castle, and perhaps has not been found by the player before his entry there. Then the new tree will allow him to re-enter with the key, and obtain the treasure.

CHAPTER 16

Opening and Closing

The treasure, if you can remember that far back, is secreted in a hidden cupboard on the first floor room of the castle. The player has to command LOOK AROUND when there, to reveal the cupboard. (See line 8030 of main listing). Our next task is to provide him with the means of opening the cupboard. But wait, there is another trap in store for the Adventurer, this time involving the umbrella. Famous and intrepid Adventurers never go around their stories sniffing and blowing their noses do they? An Adventurer with a common cold? Unheard of! Well, in this game the hero is going to have to look after his health! It is raining outside (whoever heard of rain in an Adventure?), and if he remains out of doors too long without using the umbrella, he will catch, literally, his death of cold! So let us cater for the opening of the umbrella as well as the cupboard:

```
10999  REM **** OPEN ****
11000  IF K2%=7 THEN GOTO 11020 . . .
```

umbrella problem now shelved for a couple of lines. Continuing:

```
11000  . . ELSE IF K2%<>13 THEN GOTO 40000 . . .
```

impossible to open anything else:

```
11000  . . ELSE IF PN%<>6 THEN GOTO 40050 . .
```

no cupboard anywhere except room 6 . .

```
11000  . ELSE IF P%(6)<>99 THEN LET Q$(2)="ALREADY
       OPEN" . .
```

if treasure (object 6) already moved from non-existant location 99 to room 6, i.e. if cupboard already open, and if none of these are true, everything points to it being in order to open the cupboard:

```
11000  . . ELSE LET P%(6)=6 :
       LET Q$(2)="LOOK WHAT I'VE FOUND!"
```

```
11010  GOTO 100
```

and the cupboard is open, the treasure is on display!

Now for the tricky bit with the umbrella. The player must be carrying it:

```
11020  IF P%(7)<>55 THEN GOTO 40070 . .
```

and it must not be open already. We will use a C% flag value of 4 to indicate open. We can't use 3 or it would have to be pulled around everywhere, rather than carried:

```
11020  . . ELSE IF C%(7)>2 THEN GOTO 40090 . .
```

otherwise there is no reason why it should not be opened:

```
11020  . .  ELSE  LET  C%(7)=4  :  LET  O$(7)="OPEN
       UMBRELLA" : GOTO 40020
```

There is now another obvious trap to build with the umbrella. It is considered unlucky to open one indoors and in this case it certainly will be! We will use another 'special condition' line for this one:

```
120    IF C%(7)=4 . .
```

if the umbrella is open . .

```
120    . . AND PN%<3 . .
```

and the player's location is inside the cottage . .

```
120    . . AND P%(7)=55 . .
```

and the player is carrying it . .

```
120    . THEN LET PN%=10 : LET Q$(2)="OPEN UMBRELLA
       INDOORS?": LET Q$(3)="DEAD UNLUCKY!"
```

and that will permanently put the player in DEAD. By placing this line after new locations and conditions had been set up via the verb routine last executed, any alterations to the reply, objects or locations occasioned by the 'special condition' specified will be altered before the display phase.

Now for the rain! Another special condition line. Let's get clear when we want it to operate. If the player

(a) isn't holding the umbrella, or the umbrella isn't open

AND

(b) the location is one of 3, 4 or 9

then we will increment a rain flag CR%, but if the player has been a good boy and used the umbrella, in case the rain flag is set, we will zero it:

```
130    IF (P%(7)<>55 OR C%(7)=2) AND
          (PN%=3 OR PN%=4 OR PN%=9) THEN
       LET CR%=CR%+1 ELSE LET CR%=0
```

We now need another line to make use of the information in the CR% flag. If CR% is set we will send out a message to say it is raining:

```
140    IF CR%>0 THEN LET Q$(1)="IT IS RAINING!" : . . .
```

Notice that we have used Q$(1), so that this unpredictable message won't obliterate an inventory list, or any other reply on it's way to the player. We don't want to kill him off just yet, only to make him worry a little, and wonder what to do. We can now consider escalating the message if, say four more moves have been made without the umbrella being opened:

```
140    . . IF CR%=5 THEN LET Q$(3)="I SHALL CATCH MY
       DEATH!"
```

This time we have used Q$(3), and hopefully the normal reply to the player will only occupy Q$(2), and he will get a response like:

```
→YOU SAID GO SOUTH
IT IS RAINING!
OK
I SHALL CATCH MY DEATH!
→WHAT NOW?
```

Finally we have got to put the player to death if he continues to ignore good advice for a further 2 moves:

```
150    IF CR%=7 THEN LET PN%=10 :
       LET Q$(2)="I CAUGHT MY DEATH!"
```

An open and closed case! At least, it will be when we have covered CLOSE. This verb is a bit of a luxury, in that solution of the game is possible without it. If the player needs to return to the cottage he can always leave the brolly outside, but for the sake of realism, we must allow him to close it.

The umbrella is the only object capable of being closed as it's not worth worrying about the cupboard — the player will hopefully lose interest in it when he has taken possession of the treasure within. So the object must be UMBRELLA, it must be carried by the player, and it must already be open:

```
11999  REM **** CLOSE ****
12000  IF K2%<>7 THEN GOTO 40000 ELSE IF P%(7)<>55
       THEN  GOTO  40070  ELSE  IF  C%(7)=2  THEN  GOTO
       40090  ELSE  LET  C%(7)=2  :  LET  O$(7)="ROLLED
       UMBRELLA" : GOTO 40020
```

and we have successfully closed and rerolled the umbrella!

That is another section of the plot complete. We can now do everything except leave the castle precincts to return the treasure to the treasure store — Location 0.

CHAPTER 17

Escaping from a Great Height

Having obtained the treasure from the cupboard, the player's next task is to climb to the top of the castle, and tie the rope down which he will climb to effect his escape.

The rope (object 5) is the only object that can be tied, and so:

```
12999  REM **** TIE ****
13000  IF K2%<>5 THEN GOTO 40000 . .
```

and of course, must be in the hands of the player:

```
13000  . . ELSE IF P%(5)<>55 THEN GOTO 40070 . .
```

and the only place in which the rope can be tied is location 7:

```
13000  . . ELSE IF PN%<>7 GOTO 40100 . .
```

Line 40100 is a standard reply we have not used yet:

```
40100  LET Q$(2)="IT SLIPS OFF" : GOTO 100
```

and if all these tests succeed the rope can be tied. It will have to remain permanently in position, unless we write a routine to UNTIE, and that is something else you can experiment with later. For our present purposes we will make the rope unTAKEable:

```
13000  . . ELSE LET C%(5)=-2 : LET P%(5)=7 : . .
```

and place it in location 7. We must not forget to decrement the inventory count:

```
13000  . . LET IN%=IN%-1 . . .
```

and finally indicate that the rope is providing an exit:

```
13000  . . LET O$(5)=O$(5)+" HANGING OVER PARAPET" :
       LET E$(7)=E$(7)+"H" : GOTO 40020
```

We have added the location code 'H' for ROPE to the list of exits available from location 7, and returned with a reply 'OK'.

But what if the player hasn't the rope with him, or can't work out what to do? Suppose he decides to jump for it? If he tries LOOK AROUND he will be told that jumping isn't advisable, but he might become desperate or even think this is a way to get to the bottom of the moat. It is, but not for any rewards! So we will provide him with the means to commit suicide:

```
13999  REM **** JUMP ****
14000  IF PN%=7 THEN GOTO 14010 ELSE IF PN%=8 THEN
       GOTO 14020 ELSE LET Q$(3)="BOUNCE! BOUNCE!" :
       GOTO 40020
```

We have now given him a harmless reply for any location other than 7 and 8. Location 7 is the one from which suicide is committed:

```
14010  LET PN%=10 : LET Q$(2)="SUCKED INTO THE
       MOAT'S EVIL SLIME" : GOTO 100
```

When the player has climbed down the rope, his only apparent exit is UP, giving him no escape route from the castle. However, he can jump from the rope. Since the rope is hanging over the moat, it may be expected that he will succumb to its slime in a similar way if he does this. But we will introduce a little subtlety here, and allow him to SWING on the end of the rope. Having swung, he could then be expected to have a chance of landing on firm ground if he were to jump the rope. Before we complete the verb JUMP, we had better see how we're going to takle SWING:

```
14999  REM **** SWING ****
15000  IF PN%<>8 THEN LET Q$(2)="YEAH MAN!" : GOTO
       100 . . .
```

a harmless enough reply if not on the rope. If he is, however:

```
15000  . . ELSE LET C%(5)=-3 : GOTO 40020
```

we set the C% flag for the rope to −3. We can now complete the routine for JUMP, looking at C%(5):

```
14020  IF C%(5)=-2 THEN GOTO 14010 . .
```

if he doesn't swing, the same fate awaits him as if he had jumped from the parapet, but:

```
14020  . . ELSE LET C%(5)=-2 : LET PN%=4 : LET
       Q$(3)="FAR ENOUGH TO LAND ON SAFE GROUND" :
       GOTO 40020
```

the swing has been corrected by returning C%(5) to −2, thus, should he attempt the trip again he must still say SWING before jumping. The location has been changed to 4, the ground by the moat, and the reply OK will be followed by the message set in Q$(3).

If we now test the verb SWING we will set a syntax error, since, if you remember, because SWING is a single word command it was treated in a different way to the other verbs and the line number of its routine was left as a row of asterisks. So edit line 405 and change those asterisks to "15000".

The game is now basically complete. Or is it? We placed a packed lunch on the table, and suggested in the LOOK AROUND reply that it wasn't altogether wholesome. Let's have one last attempt to kill the hero off!

```
15999  REM **** EAT ****
16000  IF K2%<>2 THEN GOTO 40030 ELSE IF P%(2)<>55
       THEN GOTO 40070 ELSE LET PN%=10 : LET
       Q$(3)="IT WAS POISONED!" : GOTO 40020
```

and he's a gonner!

Let me remind you once more to test out each verb routine as it is completed. Also, another reminder that as each verb is added to the vocabulary, its recognition letters must be added on to the end of the string WV$ in line 60, and line 490 extended to include the line number of its routine.

That now completes the programming of the demonstration game. I hope that if you have followed it through, with 'hands on' your own computer, you have not only got a working version, but have understood the method sufficiently to write your own complete Adventure from scratch. If you propose to do this, do not commence until you have finished reading this book, for there is one more very important task to do.

The program must now be examined critically to see if any improvements can be made. And some always can!

CHAPTER 18
The Finished Game

I hope that what you have read has enabled you to put your own ideas for an Adventure game into practice without having to spend hours devising a programming method which will work satisfactorily. Using this method, you should be able to write an Adventure game with quite a complex plot, having 30 or more locations and about the same number of objects, using a micro with 16k of RAM. The final size of the game that you can fit in will depend on which micro you are using. For example, the BBC micro will use more memory than the TRS-80 because of all those $ and % signs that can't be pre-defined. The BBC handles its strings in a rather uneconomical way, duplicating DATA strings in memory. On the other hand, it will respond so fast that you may consider replacing locations strings with print statements, each a subroutine accessed by a suitable "ON PN% GOSUB" statement in the display phase. A VIC-20, without an ELSE statement, will use more memory due to the line overhead for the additional lines required to accomodate the same logic. A Spectrum will use more memory for all those obligatory words like LET, GOTO and the loop variable after NEXT, optional on most micros.

From experience, using each machine in the way most suitable to its BASIC and the way it handles its memory, I would say that 32K on a TRS-80 is about equivalent to 48K in a Spectrum, and about 40K in Mode 7 on a BBC. Since the BBC's memory is stuck at 32K, an Adventure game of this size would have to be substantially abbreviated.

When you have written your Adventure, what can you do if you have run so close to the memory limit of your system that you have not left enough room for the game to RUN? With the plot determined, and the program code written, you will find it quite difficult to make major changes. Use as many as you can of the methods I earlier described under "saving space". Look through the listing and see if you can eliminate any lines by combining some. When doing this be careful you don't remove a line that has a GOTO pointing towards it, and ensure that you maintain the logic of IF/THEN tests. Abbreviate the displayable text down to the absolute minimum. Check carefully to see if you have repeated any code that could be more economically incorporated into a subroutine. Cut down the number of standard replies, making some of them a little more general so one can be used in place of two. Above all, eliminate any optional features of your Basic, spaces, LETs etc.

If happily you should find yourself with space to spare, then it will not take you long in hindsight, to see where a few extra traps or more humorous replies could be included without restructuring the whole program.

If you have a micro on which you can program sound, you may find a sound effect or two, or snatches from a tune that fits the theme of a reply, are worth adding in. If you decide to incorporate sound in this way, then the ideal sequence in the listing is between lines 380 and 390. Use a variable and set it when setting the Q$ reply with a number which can be used in an ON/GOSUB line between lines 380 and 390 to call an appropriate sound subroutine. The sound subroutines can directly follow the DATA statements starting at line 50030.

If you have colour, and space still permits, add a dash of colour to the text displayed, to enhance the visual effect.

Having written your masterpiece, the first Adventurer to receive his cassette from the software house starts to play. Eventually he is confronted by a problem which you considered to be one of your more masterly pieces of brilliance! However, it is too subtle for your player, and until he types in the necessary sequence of commands, he can go no further. What you assumed to be an obvious solution given some thought, is too obscure for your player, and he is now hopelessly stuck.

What shall he do? Write to Computer & Video Games for help? He may do that if the Adventure is so intriguing that it continues to beg a solution, but his other alternative is to put it to one side, and advise his friends that it is so hard as to be not worth buying! How can you, the Author, avoid this situation arising?

When the game is written, you will no doubt have played it yourself, and given correct commands to prove it was soluble. Perhaps you have thought to sit down and give it the wrong commands, to check out that all the traps and replies operate correctly. If not, you certainly should. You should try anything you can think of, however silly, to try and get the program to crash or do something that is contrary to the intended logic of the plot.

All this effort helps, but is not the final proof that the program is perfect. There never will be that sort of proof — only proof that it isn't perfect! However, the chances of finding the latter must be reduced as much as possible. Your own efforts to break the program down are not sufficient.

Since you are the author, you are by now so familiar with the plot and the commands, that you are thinking along fairly set lines with regard to the game. What you must do is get some fresh minds on to it.

Get family and friends to try the game — and don't give them any clues! Sit them down one at a time in front of the computer, and watch and listen! Make notes about their problems, watch to see if they type in a word that logic dictates should be recognised, but is one which you failed to spot. Perhaps instead you made do with a synonym, needed anyway for another part of the game.

As a result of these 'trials', you may decide you will have to add a whole new verb routine or a couple of extra clues for HELP or LOOK AROUND. You might very well have to correct a Syntax Error, because there are so many alternative paths to take through an Adventure program it is very doubtful whether you will have caused every part of every line to be executed however thoroughly you test it!

The demonstration game in this book is no different from any others — except perhaps that it is shorter! The first person to play it, my wife, was very upset about the wallpaper in the first room. She examined it and was told it is beginning to peel so she typed PEEL PAPER. As you now know, it has to be PULLed off. PULL seemed quite a logical alternative to me, and I knew the word was going to be needed for the tree trunk! But she felt that PEEL was the response most people would give, and became quite baffled when it wouldn't work! I have deliberately left this part of the program in its original form for two reasons. You may care to test this out on someone to see if you get the same reaction, or perhaps get some practice in by adding the verb PEEL to the vocabulary!

Another question in my own mind, is whether the recognition word for the tree after it has been felled should be changed to TRUNK, rather than left as TREE?

A number of other errors in the program have come to light whilst writing the chapters describing how the program works, when I have had to look very closely at the listing. For example, in line 120, I had inadvertently typed a 3 instead of a 4, in

```
IF C%(7)=4 AND . . .
```

This would have meant the "taking the open umbrella indoors" trap didn't work. This error wouldn't have been serious from the point of view of the player, who would merely have been unaware of the existance of the trap. But it would have meant one less feature in the plot. The reason I had not found the problem during testing, was that I was too keen to prove the whole game worked, forgetting this part was there to check out!

Another point of which to take note when watching other people play, is any comment they might express such as "I thought I would be able to fish in the moat!" A quite innocent comment like this might lead to a major improvement in the game! Perhaps the axe was too easily available, placed as it was in the hands of the player — he might have said "CHOP TREE" without realising he needed an axe, let alone being aware that he was already carrying one! So why not provide him with a fishing rod in the kitchen, and let him fish the moat and end up catching a slime-covered axe!

Before any improvments like this are put into effect, give careful consideration to whether the scale of changes required to the program is balanced by the degree of resulting improvements to the game. The two are not necessarily in direct relationship!

You should look critically at the speed of the game, the delay between pressing ENTER and receiving the prompt for more input. If this is tiresomely long, then make sure you have used all the devices suggested in the section on "saving time". Try defining the most common variables first — I haven't deliberately set out to do this in the game listing — experiment to see whether it makes a noticeable difference on your particular computer. All that you will need to do to change the order in which the variables are defined, is to insert a line at 15, and in the order of the frequency of their use, set each variable equal to zero if numeric, null if string. Move the string search subroutine and any others that are frequently called up to the beginning of the program, and put in a GOTO to bypass it into the main program. Again, redundant or optional code will slow the execution down, however minutely — so zap it!

If you are still not satisfied with the speed, then give consideration to writing a machine-code subroutine to replace the Basic string search subroutine. This subroutine can be called up to 3 times for every command the player gives, so a small saving here would pay dividends.

When you are satisfied that you can improve no more, then is the time to make a polished version. Keep what you already have in case you need at any time to go back to it, load it in the machine, and go through the program deleting all the remarks. Look for line numbers ending with a 9, and zap them! As we pointed to the line numbers immediately after the remarks, no damage should be done by deleting them. If you have the facility of a renumber utility now you can renumber the program, preferably in increments of 1, starting at 10. This in itself will test for Undefined Line errors and report back to you on where they are. If any are found, look at the resulting program, decide where the problem lies, and go back to correct your original version before proceeding further.

Now for the finishing touch — use the spare line numbers up to 10 to add your own titles and instructions to the game! If you are too short on memory to do this, then providing you have a DELETE command in your Basic — all is not lost! The last line of text in the instructions can read:

```
9     PRINT"TYPE 'RUN' AND PRESS 'ENTER' TO
      CONTINUE" : DELETE 1 – 9
```

effectively clearing the space taken up by the titles!

Another way of providing titles and instructions if memory is short, is to write these as a short separate program ending with:

```
  . . . PRINT"KEEP CASSETTE ON. TYPE <RUN> AND PRESS
      <ENTER> WHEN PROMPT APPEARS" : CLOAD
```

Some micros will do this automatically, so there is no need for the PRINT statement. If a Spectrum program is saved to LINE nnn, then LOAD at the end of the instructions will automatically load and execute it. The BBC "CHAIN" statement will do the same for that micro, without the need for special saving.

As soon as the instructions have been paged through, the CLOAD (or whatever keyword your machine uses to load a program from tape) will automatically start the tape running again, this time loading the main program, which, of course, you must record directly after the introduction program. Be careful to include a warning, if necessary, for those without motor control on their cassette recorders.

If you now have a game popular with friends and family, you might consider the possibility of getting it published. Before you do so, ask for some really honest opinions about it in comparison with other Adventure games available commercially. If the answers are encouraging, then give it a try! Making and distributing your own copies, and the necessary advertising involved will tie you, your computer and your finances up. Although you will make a greater proportion of the selling price than through royalties, the easier way to start out is to find a publisher. Pick a reliable software house that carries games programs for your system and advertises well, and write off with a brief synopsis of your game, requesting, if they are interested, details of the percentage of the retail price payable as Royalties. (Be sure to subtract VAT from the retail price before calculating how much this is worth per copy sold!) Royalties should be upwards of 10%. If your ambitions don't stretch as far as this, try submitting a listing of the program for publication in a popular computing magazine.

I hope that what I have written has given you the programming insight to enable you to write an Adventure. I hope also you have gained sufficient inspiration to go ahead and write your own. I wish you the best of luck in your efforts, look forward to playing and possibly reviewing your Adventures in the future!

APPENDIX 1

Demonstration Program Listing

```
5 CLEAR 500 : REM CLEARS STRING SPACE.  CHECK IF NEEDED.
6 REM DEFSTRD,E,L,O,Q,V,W : DEFINTI-K,C,P,S : REM IF USED OMIT %
 AND $ SIGNS FROM VARIABLE NAMES
10 DIML$(10),E$(10),D$(10),O$(10),P%(10),C%(10),Q$(8),V$(8)
20 FOR I%=0 TO 10 : READ L$(I%),E$(I%),D$(I%) : NEXT I% : FOR I%
=0 TO 10 : READ O$(I%),P%(I%),C%(I%) : NEXT I%
50 LET PN%=0 : LET IN%=1 : LET CT%=0 : LET Q$(2)="HOW DO I GET O
UT OF HERE ?" : LET A$="RUN"
60 LET WV$="TAKDROLOOEXAPULCHOOPECLOTIEJUMSWIEAT"
70 LET WG$="NORSOUEASWESUP DOWOUTDOOSTALADENTCOTROPTRE"
80 LET WD$="NSEWUDOAFGBCHJ"
90 LET WN$="AXEPAPLUNTREENTROPCROUMBDOOSTALADAROCUPCASDOWMOA"
99 REM **** START OF MAIN PROGRAM LOOP *****
100 IF PC%>0 THEN LET PC%=PC%+1 : LET P%(3)=PN% : IF PC%=3 THEN
LET PC%=0
110 IF C%(3)=-2 AND PN%=5 AND P%(3)=4 THEN LET P%(3)=9 : LET O$(
3)="TREE" : LET C%(3)=-1 : LET Q$(3)="OOPS! TREE JUST FELL IN MO
AT!"
120 IF C%(7)=4 AND PN%<3 AND P%(7)=55 THEN LET PN%=10 : LET Q$(1
)="OPEN UMBRELLA INDOORS?" : LET Q$(3)="DEAD UNLUCKY!"
130 IF (P%(7)<>55 OR C%(7)=2) AND (PN%=3 OR PN%=4 OR PN%=9) THEN
 LET CR%=CR%+1 ELSE LET CR%=0
140 IF CR%>0 THEN LET Q$(1)="IT'S RAINING!" : IF CR%=5 THEN LET
Q$(3)="I'LL CATCH MY DEATH!"
150 IF CR%=7 THEN LET PN%=10 : LET Q$(2)="I CAUGHT MY DEATH"
200 FOR I%=1 TO LEN(E$(PN%))
210 IF MID$(E$(PN%),I%,1)="N" THEN LET EX$=EX$+"NORTH. "
220 IF MID$(E$(PN%),I%,1)="S" THEN LET EX$=EX$+"SOUTH. "
230 IF MID$(E$(PN%),I%,1)="E" THEN LET EX$=EX$+"EAST. "
240 IF MID$(E$(PN%),I%,1)="W" THEN LET EX$=EX$+"WEST. "
250 IF MID$(E$(PN%),I%,1)="U" THEN LET EX$=EX$+"UP. "
260 IF MID$(E$(PN%),I%,1)="D" THEN LET EX$=EX$+"DOWN. "
270 IF MID$(E$(PN%),I%,1)="O" THEN LET EX$=EX$+"OUT. "
280 NEXT I%
290 LET II%=0 : FOR I%=0 TO 10 : IF P%(I%)=PN% THEN LET OS$=O$(I
%) ELSE NEXT I% : GOTO 330
310 IF LEN(V$(II%))+LEN(OS$)<61 THEN LET V$(II%)=V$(II%)+OS$+".
" : LET OS$="" ELSE LET II%=II%+1 : GOTO310
320 NEXT I%
330 CLS : PRINTL$(PN%) : PRINT : PRINT"SOME EXITS ARE :" : PRINT
EX$ : REM IF SCREEN WIDTH > 32 INSERT <"I AM ";> AFTER 1ST PRINT
340 PRINT"I CAN SEE :" : FOR I%=0 TO 8 : IF V$(I%)<>"" THEN PRIN
TV$(I%)
350 NEXT I%
360 PRINT : PRINT"------>YOU SAID ";A$ : PRINT : FOR I%=1 TO 3 :
 IF Q$(I%)<>"" THEN PRINT Q$(I%)
370 NEXT I%
380 PRINT : PRINT"-------->WHAT NOW";
```

```
390 FOR I%=0 TO 8 : LET V$(I%)="" : LET Q$(I%)="" : NEXT I% : LE
T A$="" : LET A1$="" : LET A2$="" : LET A3$="" : LET A4$="" : LE
T EX$="" : INPUT A$
400 IF LEN(A$)<3 THEN 40000 ELSE LET A2$=LEFT$(A$,3)
405 IF A2$="INV" THEN GOTO 4000 ELSE IF A2$="SCO" THEN GOTO 5000
 ELSE IF A2$="HEL" THEN GOTO 6000 ELSE IF A2$="QUI" THEN GOTO 70
00 ELSE IF A2$="JUM" THEN GOTO 14000 ELSE IF A2$="SWI" THEN GOTO
 15000
410 LET J%=0 : FOR I%=1 TO LEN(A$) : IF MID$(A$,I%,1)=" " THEN J
%=I%
420 NEXT I% : IF J%=0 THEN GOTO 40110 ELSE LET A1$=LEFT$(A$,J%-1
) : LET A3$=RIGHT$(A$,LEN(A$)-J%) : LET A4$=LEFT$(A3$,3)
430 IF A1$="GO" THEN GOTO 1000
440 LET X$=WV$ : LET Y$=A2$ : GOSUB 35000 : IF J%=0 THEN LET Q$(
2)="I DON'T KNOW HOW TO "+A1$ : GOTO 100 ELSE LET K1%=(J%-1)/3+1
450 LET X$=WN$ : LET Y$=A4$ : GOSUB35000
460 IF J%=0 THEN LET Q$(2)="WHAT IS A "+A3$+"?" : GOTO 100
470 LET K2%=(J%-1)/3
489 REM **** FOUND BOTH WORDS - GOTO VERB ROUTINE ****
490 ON K1% GOTO 2000,3000,8000,8000,9000,10000,11000,12000,13000
,14000,15000,16000
999 REM ****  GO  ****
1000 LET X$=WG$ : LET Y$=A4$ : GOSUB 35000 : IF J%=0 THEN GOTO 4
0010 ELSE LET X$=E$(PN%) : LET Y$=MID$(WD$,(J%-1)/3+1,1) : GOSUB
 35000 : IF J%=0 THEN GOTO 40010 ELSE LET PN%=VAL(MID$(D$(PN%),(
J%-1)*2+1,2)) : GOTO 40020
1999 REM **** TAKE  ****
2000 IF K2%>10 THEN GOTO 40030 ELSE IF P%(K2%)=55 THEN GOTO 4004
0 ELSE IF P%(K2%)<>PN% THEN GOTO 40050 ELSE IF C%(K2%)=-2 THEN G
OTO 40000 ELSE IF C%(K2%)=-1 THEN GOTO 40060 ELSE IF C%(K2%)=3 T
HEN GOTO 40080
2010 IF IN%>4 THEN LET Q$(2)="I'M OVERLOADED ALREADY!" : GOTO 10
0 ELSE LET IN%=IN%+1 : LET P%(K2%)=55 : GOTO 40020
2999 REM **** DROP  ****
3000 IF K2%>10 THEN GOTO 40070 ELSE IF P%(K2%)<>55 THEN GOTO 400
70 ELSE LET IN%=IN%-1 : IF PN%=7 THEN LET P%(K2%)=88 : LET Q$(2)
="IT FELL OFF INTO THE MOAT" : GOTO 100 ELSE LET P%(K2%)=PN% : G
OTO40020
3999 REM ****  INVENTORY  ****
4000 LET Q$(2)="I AM CARRYING: " : LET J%=2 : FORI%=0 TO 10
4010 IF P%(I%)=55 THEN IF LEN(Q$(J%))+LEN(O$(I%)) > 61 THEN LET
J%=J%+1 : GOTO 4010 ELSE LET Q$(J%)=Q$(J%)+O$(I%)+". "
4020 NEXT I% : GOTO 100
4999 REM ****  SCORE  ****
5000 IF P%(6) = 0 THEN CLS : PRINT"CONGRATULATIONS!" : PRINT"YOU
 HAVE COMPLETED YOUR QUEST!" : END ELSE LET Q$(2)="NO SCORE YET!
" : LET Q$(3)="RETURN WITH THE TREASURE!" : GOTO 100
5999 REM ****  HELP  ****
6000 ON PN%+1 GOTO 6010,6040,6040,6040,6020,6030,6040,6050,6060,
6040,6070
6010 LET Q$(2)="ISN'T THE WALLPAPER LOVELY?" : GOTO 100
6020 LET Q$(2)="A BRIDGE COULD PROVE USEFUL" : GOTO 100
6030 LET Q$(2)="MUST BE ANOTHER WAY BACK . ." : GOTO 100
6040 LET Q$(2)="EXAMINE THINGS & AND LOOK AROUND" : GOTO 100
6050 LET Q$(2)="THE ONLY WAY SEEMS TO BE DOWN" : GOTO 100
6060 LET Q$(2)="TRICKY ISN'T IT?" : GOTO 100
6070 LET Q$(2)="TRY <BREAK> AND <RUN> !" : GOTO 100
6999 REM ****  QUIT  ****
```

```
7000 CLS : IF P%(6)=0 THEN PRINT"YOU HAVE COMPLETED THE QUEST" E
LSE IF P%(6)<>99 THEN PRINT"YOU WERE NEARLY THERE!" ELSE PRINT"B
ETTER LUCK NEXT TIME!"
7010 END




7999 REM **** LOOK/EXAMINE ****
8000 IF K2% <> 11 THEN GOTO 8110 ELSE ON PN%+1 GOTO 8010,8110,81
10,8110,8020,8030,8040,8050,8020,8110,8060
8010 IF C%(1) < 0 THEN LET Q$(2)="THE WALLPAPER'S PEELING" : GOT
O 100 ELSE GOTO 8110
8020 IF P%(3) <> PN% THEN LET Q$(2)="MOAT IS FULL OF POISONOUS S
LIME" ELSE LET Q$(2)="THE TREE MAKES AN IDEAL BRIDGE"
8021 GOTO 100
8030 LET Q$(2)="TREE HAS SLIPPED INTO THE MOAT" : GOTO 100
8040 IF C%(6)=2 THEN GOTO 8110 ELSE LET C%(6)=2 : LET L$(6)=L$(6
)+" WITH A CUPBOARD" : LET Q$(2)="I JUST NOTICED SOMETHING" : GO
TO 100
8050 LET Q$(2)="I DON'T FEEL LIKE JUMPING . ." : GOTO 100
8060 LET Q$(2)="I'M IN AN INFINITY OF MISERY" : GOTO 100
8110 IF K2%=2 THEN IF P%(2)<>55 THEN GOTO 40070 ELSE LET Q$(2)="
LOOKS A BIT EVIL TO ME" : GOTO 100
8120 IF K2%=13 AND (PN%=6 OR PN%=4) THEN LET Q$(2)="I'D DROWN IN
 IT" : GOTO 100
8130 IF K2%=14 AND PN%=6 THEN LET Q$(2)="A FALL WOULD BE FATAL"
: GOTO 100
8140 LET Q$(2)="I SEE NOTHING SPECIAL" : GOTO 100
8999 REM **** PULL ****
9000 IF K2%=1 THEN GOTO 9020 ELSE IF K2%<>3 THEN GOTO 40090 ELSE
 IF P%(3)<>PN% THEN GOTO 40050 ELSE IF C%(3)=3 THEN GOTO 9010 EL
SE GOTO 40090
9010 IF PN%=4 THEN LET C%(3)=-2 : LET O$(3)="TREE TRUNK CROSSING
 MOAT" : LET E$(4)=E$(4)+"J" : LET PC%=0 : GOTO 40020 ELSE LET P
C%=1 : GOTO 40020
9020 IF PN%<>0 OR C%(1)<>-1 THEN GOTO 40090 ELSE LET C%(1)=2 : L
ET P%(8)=0 : LET E$(0)="A" : LET Q$(2)="IT JUST FELL OFF!" : GOT
O 100
9999 REM **** CHOP ****
10000 IF P%(0)<>55 THEN LET Q$(2)="HAVEN'T GOT A CHOPPER" : GOTO
 100
10010 IF PN%=0 THEN LET Q$(2)="YOU WON'T GET OUT BY FORCE!" : GO
TO 100
10020 IF K2%=3 THEN IF PN%=9 AND C%(3)=-1 THEN LET C%(3)=3 : LET
 O$(3)="TREE TRUNK" : LET Q$(3)="TIMBER!" : GOTO 40020 ELSE GOTO
 40090
10030 IF K2%=3 THEN GOTO 40050
10040 GOTO 40020
10999 REM **** OPEN ****
11000 IF K2%=7 THEN GOTO 11020 ELSE IF K2%<>12 THEN GOTO 40000 E
LSE IF PN%<>6 THEN GOTO 40050 ELSE IF P%(6)<>99 THEN LET Q$(2)="
ALREADY OPEN" ELSE LET P%(6)=6 : LET Q$(2)="LOOK WHAT I'VE FOUND
!"
```

```
11010 GOTO 100
11020 IF P%(7)<>55 THEN GOTO 40070 ELSE IF C%(7)>2 THEN GOTO 400
90 ELSE LET C%(7)=4 : LET O$(7)="OPEN UMBRELLA" : GOTO 40020
11999 REM **** CLOSE ****
12000 IF K2%<>7 THEN GOTO 40000 ELSE IF P%(7)<>55 THEN GOTO 4007
0 ELSE IF C%(7)=2 THEN GOTO 40090 ELSE LET C%(7)=2 : LET O$(7)="
ROLLED UMBRELLA" : GOTO 40020
```

```
12999 REM **** TIE ****
13000 IF K2%<>5 THEN GOTO 40000 ELSE IF P%(5)<>55 THEN GOTO 4007
0 ELSE IF PN%<>7 THEN GOTO 40100 ELSE LET C%(5)=-2 : LET P%(5)=7
 : LET IN%=IN%-1 : LET O$(5)=O$(5)+" HANGING OVER PARAPET" : LET
 E$(7)=E$(7)+"H" : GOTO 40020
13999 REM **** JUMP ****
14000 IF PN%=7 THEN GOTO 14010 ELSE IF PN%=8 THEN GOTO 14020 ELS
E LET Q$(3)="BOUNCE! BOUNCE!" : GOTO 40020
14010 LET PN%=10 : LET Q$(2)="SUCKED INTO MOAT'S EVIL SLIME" : G
OTO 100
14020 IF C%(5)=-2 THEN GOTO 14010 ELSE LET C%(5)=-2 : LET PN%=4
 : LET Q$(3)="OVER SAFE GROUND" : GOTO 40020
14999 REM **** SWING ****
15000 IF PN%<>8 THEN LET Q$(2)="YEAH MAN!" : GOTO 100 ELSE LET C
%(5)=-3 : GOTO 40020
15999 REM **** EAT ****
16000 IF K2%<>2 THEN GOTO 40030 ELSE IF P%(2)<>55 THEN GOTO 4007
0 ELSE LET PN%=10 : LET Q$(3)="IT WAS POISONED!" : GOTO 40020
34999 REM **** INSTRING SUBROUTINE   ****
35000 LET J%=0 : FOR I%=1 TO LEN(X$) STEP LEN(Y$) : IF Y$=MID$(X
$,I%,LEN(Y$)) THEN J%=I% : LET I%=LEN(X$)
35010 NEXT I% : RETURN
39999 REM **** STANDARD REPLIES *****
40000 LET Q$(2)="IMPOSSIBLE!" : GOTO 100
40010 LET Q$(2)="I CAN'T GO "+ A3$ : GOTO 100
40020 LET Q$(2)="OK" : GOTO 100
40030 LET Q$(2)="DON'T BE ABSURD!" : GOTO 100
40040 LET Q$(2)="I'M ALREADY CARRYING IT!" : GOTO 100
40050 LET Q$(2)="I DON'T SEE IT HERE" : GOTO 100
40060 LET Q$(2)="I CAN'T - YET!" : GOTO 100
40070 LET Q$(2)="I'M NOT CARRYING IT!" : GOTO 100
40080 LET Q$(2)="YOU MUST BE JOKING!" : GOTO 100
40090 LET Q$(2)="OK - NOTHING HAPPENS" : GOTO 100
40100 LET Q$(2)="IT SLIPS OFF" : GOTO 100
40110 LET Q$(2)="HUH?" : GOTO 100
50000 DATAIN A SMALL ROOM,,1*,IN A DIMLY LIT HALLWAY,SWO,0*2*3*,
IN THE KITCHEN OF A COTTAGE,E,1*,OUTSIDE A FOREST COTTAGE,NEC,9*
4*1*,BY THE MOAT OF A CASTLE,W,3*5*
50010 DATAIN A CRUMBLING CASTLE,UF,6*6*,IN A TOWER ROOM,DFUG,5*5
*7*7*,ON A PARAPET AT TOWER TOP,D,6*8*,HANGING ON ROPE ABOVE MOA
T,U,7*,IN THE FOREST,NESW,9*9*3*9*,DEAD,NESWUD,101010101010
50020 DATAAXE,55,2,WALLPAPER,0,-1,PACKED LUNCH,2,2,TREE,9,-1,ENT
RANCE,5,-2,ROPE,0,2,*PRICELESS CROWN*,99,-2,ROLLED UMBRELLA,1,2,
DOOR,99,-2,STAIRS,5,-2,IRON LADDER,6,-2
```

APPENDIX 2

List of Variables Used in the Listing

Note: Some of the string variable names have been changed for the Spectrum version. See Chapter 8 for details. All Spectrum arrays are dimensional to 1 higher than shown here.

String Variables — Simple

A$	—	Player's input command
A1$	—	1st word of player's command
A2$	—	1st 3 letters of 1st word of player's command
A3$	—	2nd word of player's command
A4$	—	1st 3 letters of 2nd word of player's command
EX$	—	List of exits from a location, for display
OS$	—	Temporary storage of visible object for display
WD$	—	String of exit codes
WG$	—	String of 1st 3 letters of valid directions
WN$	—	String of 1st 3 letters of valid objects and nouns
WV$	—	String of 1st 3 letters of valid verbs
X$	—	Target string in search subroutine
Y$	—	String searched for in search subroutine

String Variables — Arrays (dimensions shown in brackets)

D$(10)	—	Location numbers of possible destinations from a location, expressed in a string.
E$(10)	—	Codes for possible exits from a location
L$(10)	—	Location descriptions
O$(10)	—	Object descriptions
Q$(8)	—	Computer replies to player
V$(8)	—	Used to display visible objects

Integer Variables — Simple

CT%	—	Count of number of turns
I%	—	Loop counter
II%	—	Temporary use for display of objects
IN%	—	Number of items in player's inventory
J%	—	Returns value from string search subroutine
K1%	—	Sequence number of valid verb
K2%	—	Sequence number of valid noun
PC%	—	Flag to indicate if PULL TREE has been commanded
PN%	—	Current location number of player

Integer Variables — Arrays

C%(10)	—	Object flag controlling takeability of object
P%(10)	—	Current location of each object
N%	—	Spectrum only — text length of objects.

APPENDIX 3

Line Numbers and Blocks

Note: The BBC and Spectrum have maximum line numbers of 32767 and 9999 respectively. See listings for those micros to determine line layout.

Block 1.
Lines 5-10

Laying out the ground for the program to operate in:

CLEAR string space.	(on micros requiring space to be cleared)
DEFine variable types.	(string, integer etc. where the particular micro has the facility.)
DIMension arrays.	(tell the computer how many elements there will be in each array)

Block 2.
Lines 20-98

READ in DATA statements and/or directly assign variables.

Block 3.
Lines 100-399

This is the start of the main program loop, and communication with the player:
Check for special conditions.
Clear Screen.
PRINT screen display.
Reset input/output variables to null.
Await INPUT.

Block 4.
Lines 400-998

Interpret player's communication with the computer:

Decode verb and noun.
If either are invalid singly or in combination set reply accordingly and return to block 3.
ELSE GOTO block 5.

Block 5.
Lines 1000-34998

Execute the plot:

This block comprises a number of routines, one for each valid verb. Each routine may alter game variables, and either sets a reply and returns to block 3, or if the reply is a common one, goes to block 6.

Block 6.
Lines 3500-39998

String search subroutine.

Block 7.
Lines 40000-49998

Sets standard replies.

Block 8.
Lines 50000-

DATA statements for locations and objects.

APPENDIX 4

Spectrum Listing

```
   1 BORDER 6: PAPER 6: INK 2: CLS : GO TO 7
   7 LET o$=""
   8 LET oc=0
  10 DIM l$(11.32): DIM e$(11.6): DIM d$(11.12): DIM o$(11.32):
DIM p(11): DIM c(11): DIM n(11): DIM q$(9.64): DIM v$(9.32)
  20 FOR i=1 TO 11: READ l$(i).e$(i).d$(i): NEXT i: FOR i=1 TO 1
1: READ o$(i),p(i),c(i),n(i): NEXT i
  50 LET pn=0: LET in=1: LET ct=0: LET q$(2)="How do I get out o
f here?": LET a$="run"
  60 LET u$="takdrolooexapulchoopeclotiejumswieat"
  70 LET f$="norsoueaswesup dowoutdoostaladentcotroptre"
  80 LET t$="nsewudoafgbchj"
  90 LET z$="axepapluntreentropcroumbdoostaladarocupcasdowmoa"
  99 REM **Start of main program loop**
 100 IF pc>0 THEN  LET pc=pc+1: LET p(4)=pn: IF pc=3 THEN  LET p
c=0
 110 IF c(4)=-2 AND pn=5 AND p(4)=4 THEN  LET p(4)=9: LET o$(4)=
"Tree": LET n(4)=4: LET c(4)=-1: LET q$(4)="Oops! Tree just fell
 in moat!": LET e$(5)="w"
 120 IF c(8)=4 AND pn<3 AND p(8)=55 THEN  LET pn=10: LET q$(2)="
Open umbrella indoors?": LET q$(4)="Dead unlucky!"
 130 IF (p(8)<>55 OR c(8)=2) AND (pn=3 OR pn=4 OR pn=9) THEN  LE
T cr=cr+1: GO TO 140
 135 LET cr=0
 140 IF cr>0 THEN  LET q$(2)="It's raining!": IF cr=5 THEN  LET
q$(4)="I'll catch my death!"
 150 IF cr=7 THEN  LET pn=10: LET q$(3)="I caught my death"
 200 FOR i=1 TO LEN (e$(pn+1))
 210 IF e$(pn+1,i)="n" THEN  LET q$=q$+"North."
 220 IF e$(pn+1,i)="s" THEN  LET q$=q$+"South."
 230 IF e$(pn+1,i)="e" THEN  LET q$=q$+"East."
 240 IF e$(pn+1,i)="w" THEN  LET q$=q$+"West."
 250 IF e$(pn+1,i)="u" THEN  LET q$=q$+"Up."
 260 IF e$(pn+1,i)="d" THEN  LET q$=q$+"Down."
 270 IF e$(pn+1,i)="o" THEN  LET q$=q$+"Out."
 280 NEXT i
 290 LET n=0: LET ii=0: FOR i=0 TO 10: IF p(i+1)=pn THEN  GO TO
310
 300 NEXT i: LET n=ii+1: GO TO 330
 310 IF n+n(i+1)<29 THEN  LET v$(ii+1)=v$(ii+1)(1 TO n)+o$(i+1)(
1 TO n(i+1))+". ": LET n=n+2+n(i+1): GO TO 320
 315 LET n=0: LET ii=ii+1: GO TO 310
 320 NEXT i: LET n=ii+1
 330 CLS : PRINT  INK 0;l$(pn+1)''"Some exits are :"; INK 2;'g$
 340 PRINT  INK 0;'"I can see :": FOR i=1 TO n: PRINT ;v$(i)
 350 NEXT i
 360 PRINT '' INK 0;"------>You said "; INK 2;a$'': FOR i=1 TO 3
: IF q$(i+1)<>"" AND q$(i+1,1)<>" " THEN  PRINT  INK 2;q$(i+1)
 370 NEXT i
 380 PRINT  INK 0;'"--------->"; FLASH 1;"What now? ";
 390 FOR i=1 TO 9: LET v$(i)="": LET q$(i)="": NEXT i: LET a$=""
: LET h$="": LET i$="": LET j$="": LET k$="": LET g$="": BEEP .2
,20: INPUT a$: PRINT a$
 400 IF LEN a$<3 THEN  GO TO 4000
 401 LET i$=a$( TO 3)
 403 IF i$="inv" THEN  GO TO 900
 404 IF i$="sco" THEN  GO TO 1000
 405 IF i$="hel" THEN  GO TO 1100
 406 IF i$="qui" THEN  GO TO 1200
 407 IF i$="jum" THEN  GO TO 2000
 408 IF i$="swi" THEN  GO TO 2100
 410 LET j=0: FOR i=1 TO LEN a$: IF a$(i)=" " THEN  LET j=i
 420 NEXT i: IF j=0 THEN  GO TO 4110
 425 LET h$=a$( TO j-1): LET j$=a$(j+1 TO ): IF LEN j$>=3 THEN
LET k$=j$( TO 3)
 427 IF LEN j$<3 THEN  LET k$=j$( TO )
 428 IF k$="" THEN  GO TO 4110
 430 IF h$="go" THEN  GO TO 600
 440 LET x$=u$: LET y$=i$: GO SUB 3500: IF j=0 THEN  LET b$=q$(3
): GO SUB 2: LET b$="I don't know how to "+h$: LET q$(3)=b$: GO
TO 100
 445 LET k1=(j-1)/3+1
 450 LET x$=z$: LET y$=k$: GO SUB 3500
 460 IF j=0 THEN  LET q$(3)="What is a "+j$+"?": GO TO 100
 470 LET k2=(j-1)/3
 489 REM **Found both words-GOTO  verb routine**
 490 IF k1=1 THEN  GO TO 700
```

```
 495 IF k1=2 THEN  GO TO 800
 500 IF k1=3 OR k1=4 THEN  GO TO 1300
 505 IF k1=5 THEN  GO TO 1500
 510 IF k1=6 THEN  GO TO 1600
 515 IF k1=7 THEN  GO TO 1700
 520 IF k1=8 THEN  GO TO 1800
 525 IF k1=9 THEN  GO TO 1900
 530 IF k1=10 THEN  GO TO 2000
 535 IF k1=11 THEN  GO TO 2100
 540 IF k1=12 THEN  GO TO 2200
 599 REM **go**
 600 LET x$=f$: LET y$=k$: GO SUB 3500: IF j=0 THEN  GO TO 4010
 610 LET x$=e$(pn+1): LET y$=t$((j-1)/3+1): GO SUB 3500: IF j=0
THEN  GO TO 4010
 615 IF pn=10 THEN  GO TO 4020
 620 LET pn=VAL (d$(pn+1)((j-1)*2+1 TO (j-1)*2+2)): GO TO 4020
 699 REM ***Take***
 700 IF k2>10 THEN  GO TO 4030
 705 IF p(k2+1)=55 THEN  GO TO 4040
 710 IF p(k2+1)<>pn THEN  GO TO 4050
 715 IF c(k2+1)=-2 THEN  GO TO 4000
 720 IF c(k2+1)=-1 THEN  GO TO 4060
 725 IF c(k2+1)=3 THEN  GO TO 4080
 730 IF in>4 THEN  LET q$(3)="I'm overloaded already!": GO TO 10
0
 735 LET in=in+1: LET p(k2+1)=55: GO TO 4020
 799 REM **Drop**
 800 IF k2>10 THEN  GO TO 4070
 810 IF p(k2+1)<>55 THEN  GO TO 4070
 820 LET in=in-1: IF pn=7 THEN  LET p(k2+1)=88: LET q$(3)="It fe
ll off into the moat": GO TO 100
 830 LET p(k2+1)=pn: GO TO 4020
 899 REM **Inventory**
 900 LET q$(3)="I am carrying: ": LET n=15: LET j=2: FOR i=0 TO
10
 910 IF p(i+1)<>55 THEN  GO TO 920
 915 IF n+n(i+1)<29 THEN  LET q$(j+1)=q$(j+1)(1 TO n)+o$(i+1)(1
TO n(i+1))+". ": LET n=n+2+n(i+1): GO TO 920
 917 LET j=j+1: LET n=0: GO TO 915
 920 NEXT i: GO TO 100
 999 REM **score**
 1000 IF p(7)=0 THEN  CLS : PRINT "Congratulations!"'"You have co
mpleted your quest!": STOP
 1010 LET q$(3)="No score yet!": LET q$(4)="Return with the treas
ure!": GO TO 100
 1099 REM **Help**
 1110 IF pn+1=1 THEN  LET q$(3)="Isn't the wallpaper lovely?"
 1120 IF (pn+1)>=2 AND pn+1<=4) OR pn+1=7 OR pn+1=10 THEN  LET q$(
3)="Examine things & look around"
 1130 IF pn+1=5 THEN  LET q$(3)="A bridge could prove useful"
 1140 IF pn+1=6 THEN  LET q$(3)="Must be another way back.."
 1150 IF pn+1=8 THEN  LET q$(3)="The only way seems to be down"
 1160 IF pn+1=9 THEN  LET q$(3)="Tricky isn't it?"
 1170 IF pn+1=11 THEN  LET q$(3)="Try `quit` and `RUN` !"
 1180 GO TO 100
 1199 REM ***Quit***
 1200 CLS : IF p(7)=0 THEN  PRINT "You have completed the quest":
 GO TO 1250
 1210 IF p(7)<>99 THEN  PRINT "You were nearly there!": GO TO 125
0
 1220 PRINT "Better luck next time!"
 1250 STOP
 1299 REM ***Look/examine***
 1300 IF k2<>11 THEN  GO TO 1410
 1302 IF (pn+1)>=2 AND pn+1<=4) OR pn+1=10 THEN  GO TO 1410
 1304 IF (pn+1)>=5 AND pn+1<=8) THEN  GO TO ((pn-2)*10)+1300
 1306 IF pn+1=9 THEN  GO TO 1320
 1308 IF pn+1=11 THEN  GO TO 1360
 1310 IF c(2)<0 THEN  LET q$(3)="The wallpaper's peeling": GO TO
100
 1315 GO TO 1410
 1320 IF p(4)<>pn THEN  LET q$(3)="Moat is full of poisonous slim
e": GO TO 100
 1325 LET q$(3)="The tree makes an ideal bridge": GO TO 100
 1330 LET q$(3)="Tree has slipped into the moat": GO TO 100
 1340 IF c(7)=2 THEN  GO TO 1410
 1342 LET c(7)=2: LET l$(7)="In a tower room with cupboard": LET
q$(3)="I just noticed something": GO TO 100
 1350 LET q$(3)="I don't feel like jumping..": GO TO 100
 1360 LET q$(3)="I'm in an infinity of misery": GO TO 100
 1410 IF k2=2 THEN  IF p(3)<>55 THEN  GO TO 4070
 1415 IF k2=2 THEN  LET q$(3)="Looks a bit evil to me": GO TO 100
```

```
 1420 IF k2=13 AND (pn=6 OR pn=4) THEN  LET q$(3)="I'd drown in i
t": GO TO 100
 1430 IF k2=14 AND pn=6 THEN  LET q$(3)="A fall would be fatal":
GO TO 100
 1440 LET q$(3)="I see nothing special": GO TO 100
 1499 REM **pull**
 1500 IF k2=1 THEN  GO TO 1520
 1502 IF k2<>3 THEN  GO TO 4090
 1504 IF p(4)<>pn THEN  GO TO 4050
 1506 IF c(4)=3 THEN  GO TO 1510
 1508 GO TO 4090
 1510 IF pn=4 THEN  LET c(4)=-2: LET o$(4)="Tree trunk crossing m
oat": LET n(4)=24: LET e$(5)="wj": LET pc=0: GO TO 4020
 1515 LET pc=1: GO TO 4020
 1520 IF pn<>0 OR c(2)<>-1 THEN  GO TO 4090
 1530 LET c(2)=2: LET p(9)=0: LET e$(1)="a": LET q$(3)="It just f
ell off!": GO TO 100
 1599 REM **Chop**
 1600 IF p(1)<>55 THEN  LET q$(3)="Haven't got a chopper": GO TO
100
 1610 IF pn=0 THEN  LET q$(3)="You won't get out by force!": GO T
O 100
 1620 IF k2=3 THEN  IF pn=9 AND c(4)=-1 THEN  LET c(4)=3: LET o$(
4)="Tree trunk": LET n(4)=10: LET q$(4)="Timber!": GO TO 4020
 1625 IF k2=3 THEN  IF pn<>9 AND c(4)<>-1 THEN  GO TO 4090
 1630 IF k2=3 THEN  GO TO 4050
 1640 GO TO 4020
 1699 REM **open**
 1700 IF k2=7 THEN  GO TO 1720
 1702 IF k2<>12 THEN  GO TO 4000
 1704 IF pn<>6 THEN  GO TO 4050
 1706 IF p(7)<>99 THEN  LET q$(3)="Already open": GO TO 1710
 1708 LET p(7)=6: LET q$(3)="Look what I've found!"
 1710 GO TO 100
 1720 IF p(8)<>55 THEN  GO TO 4070
 1730 IF c(8)>2 THEN  GO TO 4090
 1740 LET c(8)=4: LET o$(8)="Open umbrella": LET n(8)=13: GO TO 4
020
 1799 REM ***Close***
 1800 IF k2<>7 THEN  GO TO 4000
 1810 IF p(8)<>55 THEN  GO TO 4070
 1820 IF c(8)=2 THEN  GO TO 4090
```

```
1830 LET c(8)=2: LET o$(8)="Rolled umbrella": LET n(8)=15: GO TO
 4020
1899 REM ***Tie***
1900 IF k2<>5 THEN  GO TO 4000
1910 IF p(6)<>55 THEN  GO TO 4070
1920 IF pn<>7 THEN  GO TO 4100
1930 LET c(6)=-2: LET p(6)=7: LET in=in-1: LET o$(6)="Rope hangi
ng over parapet": LET n(6)=25: LET e$(8)="dh": GO TO 4020
1999 REM ***Jump***
2000 IF pn=7 THEN  GO TO 2010
2005 IF pn=8 THEN  GO TO 2020
2007 LET q$(4)="Bounce! Bounce!": GO TO 4020
2010 LET pn=10: LET q$(3)="Sucked into moat's evil slime": GO TO
 100
2020 IF c(6)=-2 THEN  GO TO 2010
2030 LET c(6)=-2: LET pn=4: LET q$(4)="Over safe ground": GO TO
4020
2099 REM ***Swing***
2100 IF pn<>8 THEN  LET q$(3)="Yeah man!": GO TO 100
2110 LET c(6)=-3: GO TO 4020
2199 REM ***Eat***
2200 IF k2<>2 THEN  GO TO 4030
2210 IF p(3)<>55 THEN  GO TO 4070
2220 LET pn=10: LET q$(4)="It was poisoned!": GO TO 4020
3499 REM **Instring subroutine**
3500 LET j=0: FOR i=1 TO LEN x$ STEP LEN v$: IF v$=x$(i TO i+((L
EN v$)-1)) THEN  LET j=i: LET i=LEN x$
3510 NEXT i: RETURN
3999 REM ***Standard replies***
4000 LET q$(3)="Impossible": GO TO 100
4010 LET q$(3)="I can't go "+j$: GO TO 100
4020 LET q$(3)="OK": GO TO 100
4030 LET q$(3)="Don't be absurd!": GO TO 100
4040 LET q$(3)="I'm already carrying it!": GO TO 100
4050 LET q$(3)="I don't see it here": GO TO 100
4060 LET q$(3)="I can't - yet!": GO TO 100
4070 LET q$(3)="I'm not carrying it!": GO TO 100
4080 LET q$(3)="You must be joking!": GO TO 100
4090 LET q$(3)="OK - nothing happens": GO TO 100
4100 LET q$(3)="It slips off": GO TO 100
4110 LET q$(3)="Huh?": GO TO 100
```

```
9500 DATA "In a small room","","1 ","In a dimly lit hallway","sw
o","0 2 3 ","In the kitchen of a cottage","e","1 ","Outside a fo
rest cottage","nec","9 4 1 ","By the moat of a castle","w","3 5
"
9501 DATA "In a crumbling castle","uf","6 6 ","In a tower room",
"dfuq","5 5 7 7 ","On a parapet at tower top","d","6 8 ","Hangin
g on rope above moat","u","7 ","In the forest","nesw","9 9 3 9 "
,"Dead","neswud","101010101010"
9502 DATA "Axe",55,5,3,"Wallpaper",0,-1,9,"Packed lunch",2,2,12,
"Tree",9,-1,4,"Entrance",5,-2,8,"Rope",0,2,4,"*Priceless crown*"
,99,-2,17,"Rolled umbrella",1,2,15,"Door",99,-2,4,"Stairs",5,-2,
6,"Iron ladder",6,-2,11
```

APPENDIX 5

BBC Listing

```
10DIML$(10),E$(10),D$(10),O$(10),P%(10),C%(1
0),Q$(8),V$(8)
   20 FOR I%=0 TO 10 : READ
L$(I%),E$(I%),D$(I%) : NEXT : FOR I%=0 TO 10
: READO$(I%),P%(I%),C%(I%) : NEXT
   50PN%=0 : IN%=1 : CT%=0 : PC%=0 : EX$=""
: A$=""
60WV$="TAKDROLOOEXAPULCHOOPECLOTIEJUMSWIEAT"
   70 WG$="NORSOUEASWESUP
DOWOUTDOOSTALADENTCOTROPTRE"
   80 WD$="NSEWUDOAFGBCHJ"
90WN$="AXEPAPLUNTREENTROPCROUMBDOOSTALADAROC
UPCASDOWMOA"
   99 REM **** START OF MAIN PROGRAM LOOP
****
  100 IF PC%>0 THEN PC%=PC%+1 : P%(3)=PN% :
IF PC%=3 THEN PC%=0
  110 IF C%(3)=-2 AND PN%=5 AND  P%(3)=4
THEN P%(3)=9 : O$(3)="TREE" : C%(3)=-1 :
Q$(3)="OOPS! TREE JUST FELL IN MOAT!"
  120 IF C%(7)=4 AND PN%<3 AND P%(7)=55 THEN
PN%=10 : Q$(1)="OPEN UMBRELLA INDOORS?" :
Q$(3)="DEAD UNLUCKY!"
  130 IF (P%(7)<>55 OR C%(7)=2) AND (PN%=3
OR PN%=4 OR PN%=9) THEN CR%=CR%+1 ELSE CR%=0
  140 IF CR%>0 THEN Q$(1)="IT'S RAINING!" :
IF CR%=5 THEN Q$(3)="I'LL CATCH MY DEATH!"
  150 IF CR%=7 THEN LET PN%=10 : LET
Q$(2)="I CAUGHT MY DEATH"
  200 FOR I%=1 TO LEN(E$(PN%))
  210 IF MID$(E$(PN%),I%,1)="N" THEN
EX$=EX$+"NORTH. "
  220 IF MID$(E$(PN%),I%,1)="S" THEN
EX$=EX$+"SOUTH. "
  230  IF MID$(E$(PN%),I%,1)="E" THEN
```

```
EX$=EX$+"EAST. "
  24Ø IF MID$(E$(PN%),I%,1)="W" THEN
EX$=EX$+"WEST. "
  25Ø IF MID$(E$(PN%),I%,1)="U" THEN
EX$=EX$+"UP. "
  26Ø IF MID$(E$(PN%),I%,1)="D" THEN
EX$=EX$+"DOWN. "
  27Ø IF MID$(E$(PN%),I%,1)="O" THEN
EX$=EX$+"OUT. "
  28Ø NEXT
  29Ø II%=Ø : FOR I%=Ø TO 1Ø : IF P%(I%)=PN%
THEN OS$=O$(I%) ELSE NEXT : GOTO 33Ø
  31Ø IF LEN(V$(II%))+LEN(OS$)<61 THEN
V$(II%)=V$(II%)+OS$+". " : OS$="" ELSE
II%=II%+1 : GOTO31Ø
  32Ø NEXT
  33Ø CLS : PRINT"I AM ";L$(PN%) : PRINT :
PRINT "SOME EXITS ARE :" : PRINTEX$ : REM IF
SCREEN WIDTH > 32 INSERT <"I AM ";> AFTER
1ST PRINT (BBC SCREEN WIDTH=4Ø SO INCLUDE)
  34Ø PRINT"I CAN SEE :" : FOR I%=Ø TO 8 :
IF V$(I%)<>"" THEN PRINT V$(I%)
  35Ø NEXT
  36Ø PRINT : PRINT"______>YOU SAID ";A$ :
PRINT : FOR I%=1 TO 3 : IF Q$(I%)<>"" THEN
PRINT Q$(I%)
  37Ø NEXT
  38Ø PRINT : PRINT"_______>WHAT NOW ";
  39Ø FOR I%=Ø TO 8 : V$(I%)="" : Q$(I%)=""
: NEXT : A$="" : A1$="" : A2$="" : A3$="" :
A4$="" : EX$="" : INPUT A$
  4ØØ IF LEN(A$)<3 THEN 4ØØØ ELSE
A2$=LEFT$(A$,3)
  4Ø5 IF A2$="INV" THEN 4ØØØ ELSE IF
A2$="SCO" THEN 5ØØØ ELSE IF A2$="HEL" THEN
6ØØØ ELSE IF A2$="QUI" THEN 7ØØØ ELSE IF A2$
="JUM" THEN 14ØØØ ELSE IF A2$="SWI" THEN
15ØØØ
  41Ø J%=Ø : FOR I%=1 TO LEN(A$) : IF
MID$(A$,I%,1)=" " THEN J%=I%
  42Ø NEXT : IF J%=Ø THEN 2Ø1ØØ ELSE
A1$=LEFT$(A$,J%-1) :
```

```
A3$=RIGHT$(A$,LEN(A$)-J%) : A4$=LEFT$(A3$,3)
  43Ø IF A1$="GO" THEN 1ØØØ
  44Ø X$=WV$ : Y$=A2$ : GOSUB 25ØØØ : IF
J%=Ø THEN Q$(2)="I DON'T KNOW HOW TO "+A1$ :
GOTO1ØØ ELSE K1%=(J%-1)/3+1
  45Ø X$=WN$ : Y$=A4$ : GOSUB 25ØØØ
  46Ø IF J%=Ø THEN Q$(2)="WHAT IS A
"+A3$+"?" : GOTO1ØØ
  47Ø K2%=(J%-1)/3
  489 REM **** FOUND BOTH WORDS _ GOTO VERB
ROUTINE ****
  49Ø ON K1% GOTO
2ØØØ,3ØØØ,8ØØØ,8ØØØ,9ØØØ,1ØØØØ,11ØØØ,12ØØØ,1
3ØØØ,14ØØØ,15ØØØ,16ØØØ
  999 REM **** GO ****
  1ØØØ X$=WG$ : Y$=A4$ : GOSUB 25ØØØ : IF
J%=Ø THEN 2ØØ1Ø ELSE X$=E$(PN%) :
Y$=MID$(WD$,(J%-1)/3+1,1) : GOSUB 25ØØØ : IF
J%=Ø THEN 2ØØ1Ø ELSE
PN%=VAL(MID$(D$(PN%),(J%-1)*2+1,2)) : GOTO
2ØØ2Ø
  199Ø REM **** TAKE ****
  2ØØØ IF K2%>1Ø THEN 2ØØ3Ø ELSE IF
P%(K2%)=55 THEN 2ØØ4Ø ELSE IF P%(K2%)<>PN%
THEN 2ØØ5Ø ELSE IF C%(K2%)=-1 THEN 2ØØ6Ø
ELSE IF C%(K2%)=3 THEN 2ØØ8Ø
  2Ø1Ø IF IN%>4 THEN Q$(2)="I'M OVERLOADED
ALREADY!" : GOTO 1ØØ ELSE IN%=IN%+1 :
P%(K2%)=55 : GOTO 2ØØ2Ø
  2999 REM **** DROP ****
  3ØØØ IF K2%>1Ø THEN 2ØØ7Ø ELSE IF
P%(K2%)<>55 THEN 2ØØ7Ø ELSE IN%=IN%-1 : IF
PN%=7 THEN P%(K2%9=88 : Q$(2)="IT FELL OFF
INTO THE MOAT" : GOTO 1ØØ ELSE P%(K2%)=PN% :
GOTO2ØØ2Ø
  3999 REM **** INVENTORY ****
  4ØØØ Q$(2)="I AM CARRYING: " : J%=2 :
FORI%=Ø TO 1Ø
  4Ø1Ø IF P%(I%)<>55 THEN 4Ø2Ø ELSE IF
LEN(Q$(J%))+LEN(O$(I%)) > 61 THEN J%=J%+1 :
GOTO 4Ø1Ø ELSE Q$(J%)=Q$(J%)+O$(I%)+". "
  4Ø2Ø NEXT : GOTO 1ØØ
```

```
 4999 REM **** SCORE ****
 5000 IF P%(6) = 0 THEN CLS :
PRINT"CONGRATULATIONS!" : PRINT"YOU HAVE
COMPLETED YOUR QUEST!" : END ELSE Q$(2)="NO
SCORE YET!" : Q$(3)="RETURN WITH THE
TREASURE!" : GOTO 100
 5999 REM **** HELP ****
 6000 ON PN%+1 GOTO
6010,6040,6040,6040,6020,6030,6040,6050,6060
,6040,6070
 6010 Q$(2)="ISN'T THE WALLPAPER LOVELY?" :
GOTO 100
 6020 Q$(2)="A BRIDGE COULD PROVE USEFUL" :
GOTO 100
 6030 Q$(2)="MUST BE ANOTHER WAY BACK . ." :
GOTO 100
 6040 Q$(2)="EXAMINE THINGS & LOOK AROUND" :
GOTO 100
 6050 Q$(2)="THE ONLY WAY SEEMS TO BE DOWN"
: GOTO 100
 6060 Q$(2)="TRY<BREAK> AND <RUN> !" : GOTO
100
 6999 REM **** QUIT ****
 7000 CLS : IF P%(6)=0 THEN PRINT"YOU HAVE
COMPLETED THE QUEST"ELSE IF P%(6)<>99 THEN
PRINT "YOU WERE NEARLY THERE!" ELSE PRINT
"BETTER LUCK NEXT TIME!"
 7010 END
 7999 REM **** LOOK / EXAMINE ****
 8000 IF K2% <> 11 THEN 8110 ELSE ON PN%+1
GOTO
8010,8110,8110,8110,8020,8030,8040,8050,8020
,8110,8060
 8010 IF C%(1) < 0 THEN Q$(2)=" THE
WALLPAPER IS PEELING " : GOTO 100 ELSE 8110
 8020 IF P%(3) <> PN% THEN Q$(2) = "MOAT IS
FULL OF POISONOUS SLIME" ELSE Q$(2)="THE
TREE MAKES AN IDEAL BRIDGE "
 8021 GOTO 100
 8030 Q$(2)="TREE HAS SLIPPED INTO THE MOAT"
: GOTO 100
```

```
 8040 IF C%(6)=2 THEN 8110 ELSE C%(6)=2 :
L$(6)=L$(6)+" WITH A CUPBOARD" : Q$(2)="I
JUST NOTICED SOMETHING" : GOTO 100
 8050 Q$(2)="I DON'T FEEL LIKE JUMPING . ."
: GOTO 100
 8060 Q$(2)="I'M IN AN INFINITY OF MISERY" :
GOTO 100
 8110 IF K2%<>2 THEN 8120 ELSE IF P%(2) <>
55 THEN 20070 ELSE Q$(2)="LOOKS A BIT EVIL
TO ME" : GOTO 100
 8120 IF K2%=15 AND (PN%=6 OR PN%=4) THEN
Q$(2) ="I'D DROWN IN IT" : GOTO 100
 8130 IF K2%=14 AND PN%=6 THEN Q$(2)="A FALL
WOULD BE FATAL" : GOTO 100
 8140 LET Q$(2)="I SEE NOTHING SPECIAL" :
GOTO 100
 8999 REM **** PULL ****
 9000 IF K2%=1 THEN 9020 ELSE IF K2%<> 3
THEN 20090 ELSE IF P%(3)<> PN% THEN 20050
ELSE IF C%(3)=3 THEN 9010 ELSE 20090
 9010 IF PN%=4 THEN C%(3)=-2 : O$(3)="TREE
TRUNK CROSSING MOAT" : E$(4)=E$(4)+"J" :
PC%=0 : GOTO 20020 ELSE PC%=1 : GOTO 20020
 9020 IF PN%<>0 OR C%(1)<>-1 THEN 20090 ELSE
C%(1)=2 : P%(8)=0 : E$(0)="A" : Q$(2)="IT
JUST FELL OFF!" : GOTO 100
 9999 REM **** CHOP ****
10000 IF P%(0)<>55 THEN Q$(2)="HAVEN'T GOT A
CHOPPER" : GOTO 100
10010 IF PN%=0 THEN Q$(2)="YOU WON'T GET OUT
BY FORCE!" : GOTO 100
10020 IF K2%<>3 THEN 20020 ELSEIF PN%=9 AND
C%(3)=-1 THEN C%(3)=3 : O$(3)="TREE TRUNK" :
Q$(3)="TIMBER!!" : GOTO 20020 ELSE 20090
10030 IF K2%=3 THEN 20050
10040 GOTO 20020
10999 REM **** OPEN ****
11000 IF K2%=7 THEN 11020 ELSE IF K2%<>12
THEN 20000 ELSE IF PN%<> 6 THEN 20050 ELSE
IF P%(6)<>99 THEN Q$(2)="ALREADY OPEN " ELSE
P%(6)=6 : Q$(2)="LOOK WHAT I FOUND!"
11010 GOTO 100
```

```
11020 IF P%(7)<> 55 THEN 20070 ELSE IF
C%(7)>2 THEN 20090 ELSE C%(7) =4 :
O$(7)="OPEN UMBRELLA" : GOTO 20020
11999 REM **** CLOSE ****
12000 IF K2%<>7 THEN 20000 ELSE IF P%(7)<>55
THEN 20070 ELSE IF C%(7) =2 THEN 20090 ELSE
C%(7)=2 : O$(7)="ROLLED UMBRELLA" : GOTO
20020
12999 REM **** TIE ****
13000 IF K2%<>5 THEN 20000 ELSE IF P%(5)<>55
THEN 20070 ELSE IF PN%<>7 THEN 20100 ELSE
C%(5)=-2 : P%(5)=7 : IN%=IN%-1 :
O$(5)=O$(5)+" HANGING OVER PARAPET" :
E$(7)=E$(7)+"H" : GOTO 20020
13999 REM **** JUMP ****
14000 IF PN%=7 THEN 14010 ELSE IF PN%=8 THEN
14020 ELSE Q%(3)="BOUNCE! BOUNCE!" : GOTO
20020
14010 PN%=10 : Q$(2)=" SUCKED INTO MOAT'S
EVIL SLIME" : GOTO 100
14020 IF C%(5)=-2 THEN 14010 ELSE C%(5)=-2 :
PN%=4 : Q$(3)="OVER SAFE GROUND" : GOTO
20020
14999 REM **** SWING ****
15000 IF PN%<>8 THEN Q$(2)="YEAH MAN!" :
GOTO 100 ELSE C%(5)=-3 : GOTO 20020
15999 REM **** EAT ****
16000 IF K2%<>2 THEN 20030 ELSE IF P%(2)<>55
THEN 20070 ELSE PN%=10 : Q$(3)="IT WAS
POISONED!" : GOTO 20020
20000 Q$(2)="IMPOSSIBLE!" : GOTO100
20010 Q$(2)="I CAN'T GO "+ A3$ : GOTO 100
20020 Q$(2)="OK" : GOTO 100
20030 Q$(2)="DON'T BE ABSURD!" : GOTO 100
20040 Q$(2)="I'M ALREADY CARRYING IT!" :
GOTO 100
20050 Q$(2)="I DON'T SEE IT HERE" : GOTO 100
20060 Q$(2)="I CAN'T - YET" : GOTO 100
20070 Q$(2)="I'M NOT CARRYING IT!" : GOTO
100
20080 Q$(2)="YOU MUST BE JOKING!" : GOTO 100
20090 Q$(2)="OK - NOTHING HAPPENS" : GOTO 100
```

```
20100 Q$(2)="IT SLIPS OFF" : GOTO 100
20110 Q$(2)="HUH?" : GOTO 100
24999 REM *** INSTRING SUBROUTINE ***
25000J%=0 : FOR I%=1 TO LEN(X$) STEP LEN(Y$)
: IF Y$=MID$(X$,I%,LEN(Y$)) THEN J%=I% :
I%=LEN(X$)
25010 NEXT : RETURN
30000DATAIN A SMALL ROOM,,1*,IN A DIMLY LIT
HALLWAY,SWO,0*2*3*,IN THE KITCHEN OF A
COTTAGE,E,1*,OUTSIDE A FOREST
COTTAGE,NEC,9*4*1*,BY THE MOAT OF A
CASTLE,W,3*5*
30010DATAIN A CRUMBLING CASTLE,UF,6*6*,IN A
TOWER ROOM,DFUG,5*5*7*7*,ON A PARAPET AT
TOWER TOP,D,6*8*,HANGING ON A ROPE ABOVE
MOAT,U,7*,IN THE
FOREST,NESW,9*9*3*9*,DEAD,NESWUD,1010101010101
0
30020DATAAXE,55,2,WALLPAPER,0,-1,PACKED
LUNCH,2,2,TREE,9,-1,ENTRANCE,5,-2,ROPE,0,2,*
PRICELESS CROWN*,99,-2,ROLLED
UMBRELLA,1,2,DOOR,99,-2,STAIRS,5,-2,IRON
LADDER,6,-2
```

APPENDIX 6

Commodore Listing

```
10 DIM L$(10),E$(10),D$(10),O$(10),P(10),C(10),Q$(10),V$(10)
20 FOR I=0 TO 10:READ L$(I),E$(I),D$(I):NEXT I
21 FOR I=0 TO 10:READ O$(I),P(I),C(I):NEXT I
50 LET PN=0 : LET IN=1 : LET CT=0 : LET Q$(2)="HOW DO I
   GET OUT OF HERE?"
51 LET A$="RUN"
60 LET WV$="TAKDROLOOEXAPULCHOOPECLOTIEJUMSWIEAT"
70 LET WG$="NORSOUEASWESUP DOWOUTDOOSTALADENTCOTROPTRE"
80 LET WD$="NSEWUDOAFGBCHU"
90 LET WN$="AXEPAPLUNTREENTROPCROUMBDOOSTALADAROCUPCASDOWMOA"
99 REM **** START OF MAIN PROGRAM LOOP ****
100 IF PC>0 THEN LET PC=PC+1:GOTO 103
101 GOTO 110
103 LET P(3)=PN
105 IF PC=3 THEN LET PC=0
110 IF C(3)=-2 AND PN=5 AND P(3)=4 THEN LET P(3)=9:GOTO 112
111 GOTO 120
112 LET O$(3)="TREE"
113 LET C(3)=-1
114 LET Q$(3)="OOPS! TREE JUST FELL IN MOAT! "
120 IF C(7)=4 AND PN<3 AND P(7)=55 THEN LET PN=10:GOTO 122
121 GOTO 130
122 LET Q$(1)="OPEN UMBRELLA INDOORS?" : LET
    Q$(3)="DEAD UNLUCKY!"
130 IF (P(7)<>55 OR C(7)=2) AND (PN=3 OR PN=4 OR PN=9)THEN
    CR=CR+1:GOTO 140
139 LET CR=0
140 IF CR>0 THEN LET Q$(1)="IT'S RAINING!" :
141 IF CR=5 THEN LET Q$(3)="I'LL CATCH MY DEATH!"
150 IF CR=7 THEN LET PN=10 : LET Q$(2)="I CAUGHT MY DEATH"
200 FOR I=1 TO LEN (E$(PN))
210 IF MID$(E$(PN),I,1)="N" THEN LET EX$=EX$+"NORTH. "
220 IF MID$(E$(PN),I,1)="S" THEN LET EX$=EX$+"SOUTH. "
230 IF MID$(E$(PN),I,1)="E" THEN LET EX$=EX$+"EAST. "
240 IF MID$(E$(PN),I,1)="W" THEN LET EX$=EX$+"WEST. "
250 IF MID$(E$(PN),I,1)="U" THEN LET EX$=EX$+"UP. "
260 IF MID$(E$(PN),I,1)="D" THEN LET EX$=EX$+"DOWN. "
270 IF MID$(E$(PN),I,1)="O" THEN LET EX$=EX$+"OUT. "
280 NEXT I
290 LET II=0
292 FOR I=0 TO 10:IF P(I)=PN THEN LET OS$=O$(I):GOTO 310
293 NEXT I
294 GOTO 330
310 IF LEN(V$(II))+LEN(OS$)<37 THEN LET
    V$(II)=V$(II)+OS$+".":OS$="":GOTO320
311 LET II=II+1 : GOTO 310
320 NEXT I
```

```
330 PRINT"⊐":PRINT L$(PN):PRINT:PRINT"SOME EXITS ARE
    :":PRINT EX$
331 REM IF SCREEN WIDTH > 32 INSERT <"I AM ";> AFTER 1ST PRINT
340 PRINT"I CAN SEE :":FOR I=0 TO 8:IF V$(I)<>"" THEN PRINT
    V$(I)
350 NEXT I
360 PRINT : PRINT"------>YOU SAID ";A$ :PRINT
361 FOR I=1 TO 3:IF Q$(I)<>"" THEN PRINT Q$(I):NEXT I
370 REM
380 PRINT : PRINT"------->WHAT NOW";
390 FOR I=0 TO 8:LET V$(I)="":LET Q$(I)="":NEXT I:LET
    A$="":LET A1$=""
391 LET A2$="":LET A3$="":LET A4$="":LET EX$=""
392 INPUT A$
400 IF LEN(A$)<3 THEN GOTO 40000
401 LET A2$=LEFT$(A$,3)
402 IF A2$="INV" THEN GOTO 4000
403 IF A2$="SCO" THEN GOTO 5000
404 IF A2$="HEL" THEN GOTO 6000
405 IF A2$="QUI" THEN GOTO 7000
406 IF A2$="JUM" THEN GOTO 14000
407 IF A2$="SWI" THEN GOTO 15000
410 LET J=0:FOR I=1 TO LEN (A$):IF MID$(A$,I,1)=" "THEN J=I
411 NEXT I
420 IF J=0 THEN GOTO 40110
421 LET A1$=LEFT$(A$,J-1)
425 LET A3$=RIGHT$(A$,LEN(A$)-J):LET A4$=LEFT$(A3$,3)
430 IF A1$="GO" THEN GOTO 1000
440 LET X$=WV$:LET Y$=A2$:GOSUB 35000
441 IF J=0 THEN Q$(2)="I DON'T KNOW HOW TO "+A1$ : GOTO 100
442 LET K1=INT((J-1)/3)+1
450 LET X$=WN$ : LET Y$=A4$ : GOSUB 35000
460 IF J=0 THEN LET Q$(2)="WHAT IS A "+A3$+" ?" : GOTO 100
470 LET K2=INT((J-1)/3)
489 REM **** FOUND BOTH WORDS - GOTO VERB ROUTINE ****
490 ONK1GOTO2000,3000,8000,8000,9000,10000,11000,12000,13000,
    14000,15000,16000
491 REM
999 REM **** GO ****
1000 LET X$=WG$ : LET Y$=A4$ : GOSUB 35000 : IF J=0 THEN GOTO
     40010
1001 LET X$=E$(PN):LET Y$=MID$(WD$,INT(((J-1)/3)+1),1) :
     GOSUB 35000
1002 IF J=0 THEN GOTO 40010
1003 LET PN=VAL(MID$(D$(PN),INT(J-1)*2+1,2)):GOTO 40020
1004 REM
1999 REM **** TAKE ****
2000 IF K2>10 THEN GOTO 40030
2001 IF P(K2)=55 THEN GOTO 40040
2002 IF P(K2)<>PN THEN GOTO 40050
2003 IF C(K2)=-2 THEN GOTO 40000
2005 IF C(K2)=-1 THEN GOTO 40060
2006 IF C(K2)=3 THEN GOTO 40080
2010 IF IN>4 THEN LET Q$(2)="I'M OVERLOADED ALREADY! ":GOTO 100
2011 LET IN=IN+1 :LET P(K2)=55:GOTO 40020
```

```
2999 REM **** DROP ****
3000 IF K2>10 THEN GOTO 40070
3001 IF P(K2)<>55 THEN GOTO 40070
3010 LET IN=IN-1 :IF PN=7 THEN LET P(K2)=88 :GOTO 3015
3011 GOTO 3016
3015 LET Q$(2)="IT FELL OFF INTO THE MOAT" : GOTO 100
3016 LET P(K2)=PN :GOTO 40020
3020 REM
3999 REM **** INVENTORY ****
4000 LET Q$(2)="I AM CARRYING: ":LET J=2
4001 FORI=0 TO 10
4010 IF P(I)=55 THEN GOTO 4012
4011 GOTO 4019
4012 IF P(I)=55 THEN IF LEN(Q$(J))+LEN(O$(I)) > 37 THEN LET
     J=J+1:GOTO 4010
4019 LET Q$(J)=Q$(J)+O$(I)+". "
4020 NEXT I  : GOTO 100
4999 REM **** SCORE CLOSE****
5000 IF P(6)=0 THEN PRINT"⊐": PRINT"CONGRATULATIONS!"
     :GOTO 5002
5001 GOTO 5003
5002 PRINT"YOU HAVE COMPLETED YOUR QUEST" :END
5003 LET Q$(2)="NO SCORE YET !"
5010 LET Q$(3)="RETURN WITH THE TREASURE!" : GOTO 100
5999 REM **** HELP CLOSE****
6000 ON PN+1 GOTO 6010,6040,6040,6040,6020,6030,6040,6050,
     6060,6040,6070

6010 LET Q$(2)="ISN'T THE WALLPAPER LOVELY ?" : GOTO 100
6020 LET Q$(2)="A BRIDGE COULD PROVE USEFUL " : GOTO 100
6030 LET Q$(2)="MUST BE ANOTHER WAY BACK. . " : GOTO 100
6040 LET Q$(2)="EXAMINE THINGS & LOOK AROUND ": GOTO 100
6050 LET Q$(2)="THE ONLY WAY SEEMS TO BE DOWN ": GOTO 100
6060 LET Q$(2)="TRICKY ISN'T IT ?" : GOTO 100
6070 LET Q$(2)="TRY <BREAK> AND <RUN> !" : GOTO 100
6999 REM **** QUIT ****
7000 PRINT"⊐" : IF P(6)=0 THEN PRINT"YOU HAVE COMPLETED THE QUEST
     ":GOTO 7010
7001 IF P(6)<>99 THEN PRINT"YOU WERE NEARLY THERE!" :GOTO 7010
7002 PRINT"BETTER LUCK NEXT TIME!"
7010 END
7999 REM **** LOOK/EXAMINE ****
8000 IF K2<>11 THEN GOTO 8110
8001 REM
8002 ON PN+1 GOTO 8010,8110,8110,8110,8020,8030,8040,8050,
     8020,8110,8060
8003 REM
8010 IF C(1)<0 THEN LET Q$(2)="THE WALLPAPER'S PEELING ":GOTO
     100
8011 GOTO 8110
8020 IF P(3)<> PN THEN LET Q$(2)="MOAT IS FULL OF
     POISONOUS SLIME ":GOTO 100
8021 LET Q$(2)="THE TREE MAKES AN IDEAL BRIDGE"
8022 GOTO 100
8030 LET Q$(2)="TREE HAS SLIPPED INTO THE MOAT" : GOTO 100
8040 IF C(6)=2 THEN GOTO 8110
8041 LET C(6)=2
8042 REM
```

```
8043 LET L$(6)=L$(6)+"WITH A CUPBOARD":LET Q$(2)="I JUST
     NOTICED SOMETHING "
8045 GOTO 100
8050 LET Q$(2)="I DON'T FEEL LIKE JUMPING . ." : GOTO 100
8060 LET Q$(2)="I'M IN AN INFINITY OF MISERY" : GOTO 100
8110 IF K2=2 THEN GOTO 8112
8111 LET Q$(2)="LOOKS A BIT EVIL TO ME" : GOTO 100
8112 IF P(2)<>55 THEN GOTO 40070
8115 LET Q$(2)="LOOKS A BIT EVIL TO ME" : GOTO 100
8120 IF K2=13 THEN GOTO 8123
8121 GOTO 8130
8123 IF (PN=6 OR PN=4) THEN LET Q$(2)="I'D DROWN IN IT"
     :GOTO 100
8130 IF K2=14 THEN GOTO 8133
8131 GOTO 8140
8133 IF PN=6 THEN LET Q$(2)="A FALL WOULD BE FATAL" : GOTO 100
8140 LET Q$(2)="I SEE NOTHING SPECIAL" : GOTO 100
8999 REM **** PULLCLOSE****
9000 IF K2=1 THEN GOTO 9020
9001 IF K2<>3 THEN GOTO 40090
9005 IF P(3)<>PN THEN GOTO 40050
9006 IF C(3)=3 THEN GOTO 9010
9007 GOTO 40090
9010 IF PN=4 THEN LET C(3)=-2:LET O$(3)="TREE TRUNK CROSSING
     MOAT ":GOTO 9013
9011 REM
9012 GOTO 9015
9013 LET E$(4)=E$(4)+"J" : LET PC=0 : GOTO 40020
9015 LET PC=1 : GOTO 40020
9020 IF PN<>0 OR C(1)<>-1 THEN GOTO 40090
9021 LET C(1)=2 : LET P(8)=0
9025 LET Q$(2)="IT JUST FELL OFF!" : GOTO 100
9999 REM **** CHOP ****
10000 IF P(0)<>55 THEN LET Q$(2)="HAVEN'T GOT A CHOPPER" :
      GOTO 100
10010 IF PN=0 THEN LET Q$(2)="YOU WON'T GET OUT BY FORCE!" :
      GOTO 100
10020 IF K2=3 THEN GOTO 10023
10021 GOTO 10025
10022 REM
10023 IF  PN=9 AND C(3)=-1 THEN LET C(3)=3
10024 LET O$(3)="TREE TRUNK":LET Q$(3)="TIMBER!":GOTO 40020
10025 GOTO 40090
10028 REM
10030 IF K2=3 THEN GOTO 40050
10040 GOTO 40020
10999 REM **** OPEN ****
11000 IF K2=7 THEN GOTO 11020
11001 IF K2<>12 THEN GOTO 40000
11002 IF PN<>6 THEN GOTO 40050
11003 IF P(6)<>99 THEN LET Q$(2)="ALREADY OPEN":GOTO 100
11009 LET P(6)=6 : LET Q$(2)="LOOK WHAT I'VE FOUND!"
11010 GOTO 100
11020 IF P(7)<>55 THEN GOTO 40070
11021 IF C(7)>2 THEN GOTO 40090
11025 LET C(7)=4 : LET O$(7)="OPEN UMBRELLA" : GOTO 40020
```

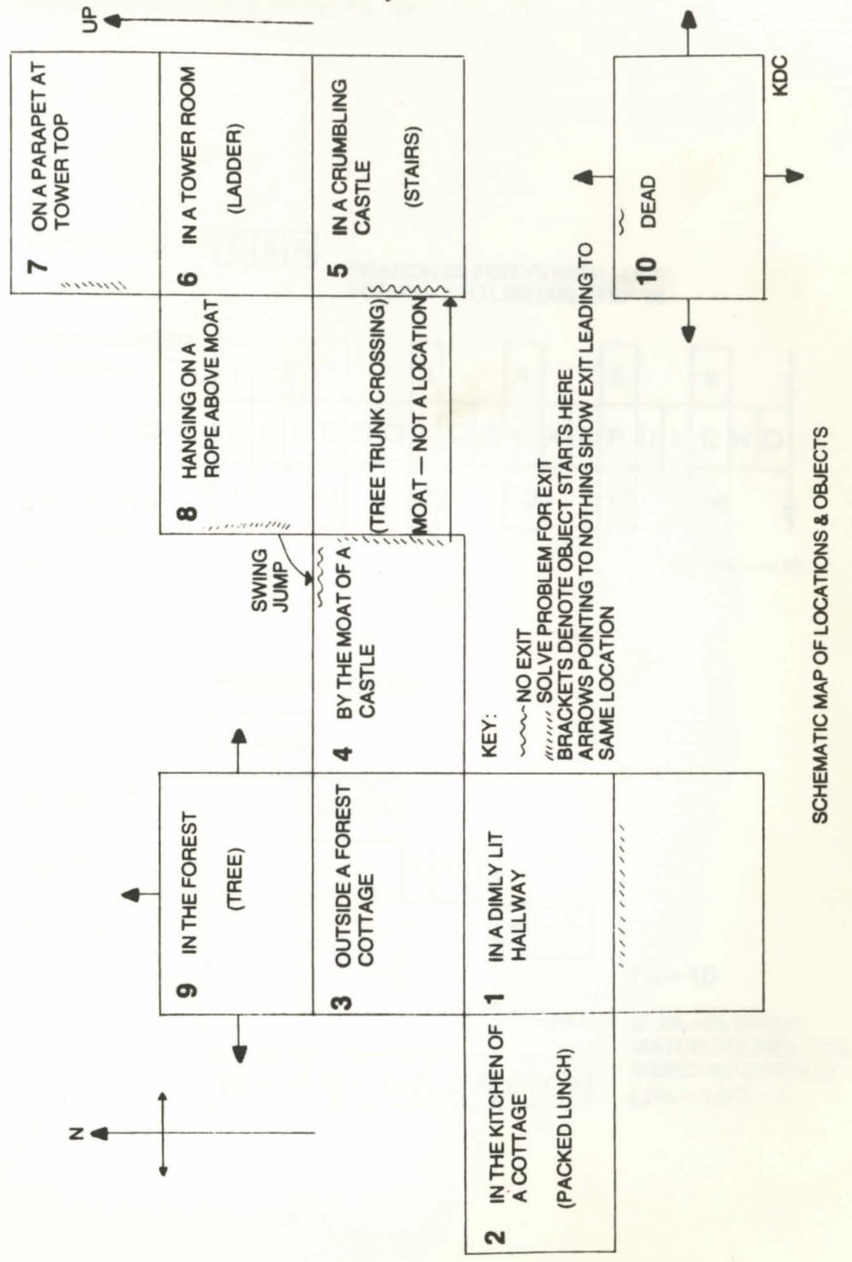
```
11999 REM **** CLOSE ****
12000 IF K2<>7 THEN GOTO 40000
12001 IF P(7)<>55 THEN GOTO 40070
12003 IF C(7)=2 THEN GOTO 40090
12005 LET C(7)=2:LET O$(7)="ROLLED UMBRELLA" : GOTO 40020
12999 REM **** TIE ****
13000 IF K2<>5 THEN GOTO 40000
13001 IF P(5)<>55 THEN GOTO 40070
13005 IF PN<>7 THEN GOTO 40100
13006 LET C(5)=-2 : LET P(5)=7
13007 LET IN=IN-1 : LET O$(5)=O$(5)+" HANGING OVER PARAPET "
13008 LET E$(7)=E$(7)+"H" : GOTO 40020
13999 REM **** JUMP ****
14000 IF PN=7 THEN GOTO 14010
14001 IF PN=8 THEN GOTO 14020
14005 LET Q$(3)="BOUNCE! BOUNCE!" : GOTO 40020
14010 LET PN=10 : LET Q$(2)="SUCKED INTO MOAT'S EVIL SLIME "
      : GOTO 100
14020 IF C(5)=-2 THEN GOTO 14010
14021 LET C(5)=-2 : LET PN=4
14022 LET Q$(3)="OVER SAFE GROUND" : GOTO 40020
14999 REM **** SWINGCLOSE****
15000 IF PN<>8 THEN LET Q$(2)="YEAH MAN!" :GOTO 100
15001 LET C(5)=-3 :GOTO 40020
15999 REM **** EAT ****
16000 IF K2<>2 THEN GOTO 40030
16001 IF P(2)<>55 THEN GOTO 40070
16002 LET PN=10 : LET Q$(3)="IT WAS POISONED!" : GOTO 40020
34999 REM **** INSTRING SUBROUTINE ****
35000 LET J=0
35001 FOR I=1 TO LEN(X$) STEP LEN(Y$)
35005 IF Y$=MID$(X$,I,LEN(Y$)) THEN J=I:  LET I=LEN(X$)
35010 NEXT I  : RETURN
39999 REM **** STANDARD REPLIES ****
40000 LET Q$(2)="IMPOSSIBLE!" : GOTO 100
40010 LET Q$(2)="I CAN'T GO "+ A3$ : GOTO 100
40020 LET Q$(2)="OK" : GOTO 100
40030 LET Q$(2)="DON'T BE ABSURD!" : GOTO 100
40040 LET Q$(2)="I'M ALREADY CARRYING IT!" : GOTO 100
40050 LET Q$(2)="I DON'T SEE IT HERE" : GOTO 100
40060 LET Q$(2)="I CAN'T - YET!" : GOTO 100
40070 LET Q$(2)="I'M NOT CARRYING IT!" : GOTO 100
40080 LET Q$(2)="YOU MUST BE JOKING!" : GOTO 100
40090 LET Q$(2)="OK - NOTHING HAPPENS" : GOTO 100
40100 LET Q$(2)="IT SLIPS OFF" : GOTO 100
40110 LET Q$(2)="HUH ?" : GOTO 100
50000 DATA IN A SMALL ROOM,"",1*,IN A DIMLY LIT
      HALLWAY,SWO,0*2*3*
50001 DATA IN THE KITCHEN OF A COTTAGE,E,1*,OUTSIDE A FOREST
      COTTAGE,NEC
50002 DATA 9*4*1*,BY THE MOAT OF A CASTLE,W,3*5*
50010 DATA IN A CRUMBLING CASTLE,UF,6*6*,IN A TOWER
      ROOM,DFUG,5*5*7*7*
50011 DATA ON A PARAPET AT TOWER TOP,D,6*8*,HANDING ON ROPE
      ABOVE MOAT,U,7*
50012 DATA IN THE FOREST,NESW,9*9*3*9*,DEAD,NESWUD,101010101010
50020 DATA AXE,55,2,WALLPAPER,0,-1,PACKED LUNCH,2,2,TREE,9,-1,
      ENTRANCE
```

```
 50021 DATA 5,-2,ROPE,0,2,*PRICELESS CROWN*,99,-2,ROLLED
       UMBRELLA,1,2
 50022 DATA DOOR,99,-2,STAIRS,5,-2,IRON LADDER,6,-2
 60000 DIM A$(100)
 60001 FOR I= 1 TO 10
 60010 READ A$(I)
 60020 PRINTA$(I);"  ";I
 60030 NEXT
READY.

READY.
```

APPENDIX 7

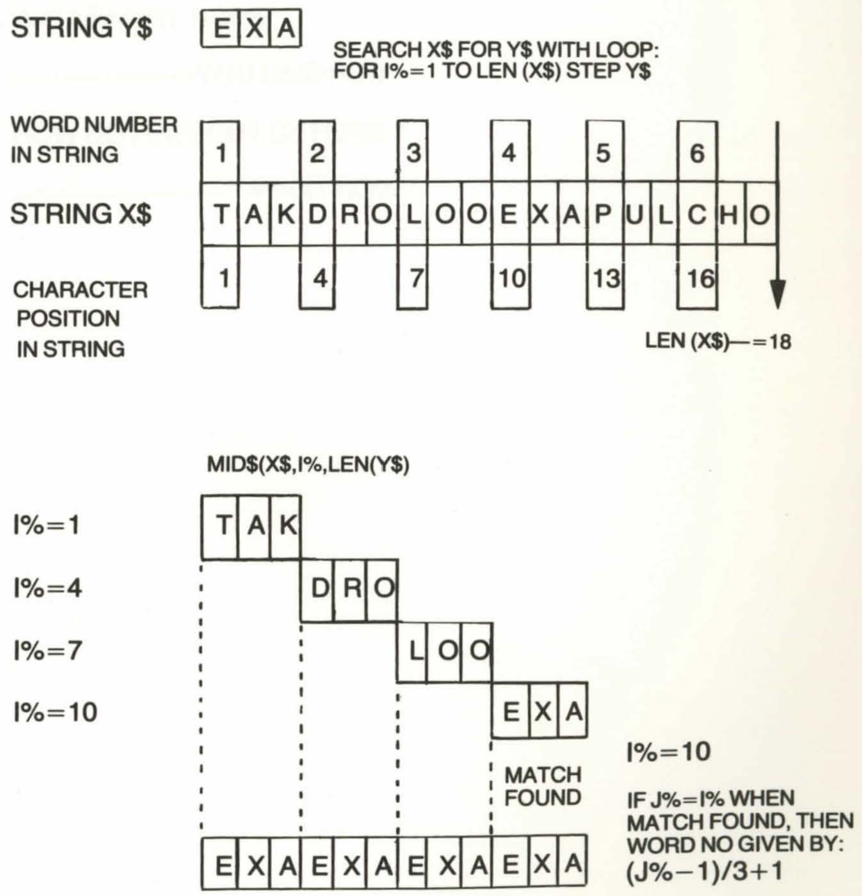
Schematic Map of Locations & Objects



UP
N
7 ON A PARAPET AT TOWER TOP
6 IN A TOWER ROOM (LADDER)
5 IN A CRUMBLING CASTLE (STAIRS)
10 DEAD
KDC
8 HANGING ON A ROPE ABOVE MOAT
(TREE TRUNK CROSSING)
MOAT — NOT A LOCATION
SWING JUMP
4 BY THE MOAT OF A CASTLE
9 IN THE FOREST (TREE)
3 OUTSIDE A FOREST COTTAGE
1 IN A DIMLY LIT HALLWAY
2 IN THE KITCHEN OF A COTTAGE (PACKED LUNCH)
KEY:
NO EXIT
SOLVE PROBLEM FOR EXIT
BRACKETS DENOTE OBJECT STARTS HERE
ARROWS POINTING TO NOTHING SHOW EXIT LEADING TO SAME LOCATION
SCHEMATIC MAP OF LOCATIONS & OBJECTS

APPENDIX 8

Method of Searching String X$ for String Y$



APPENDIX 9

Screen Display

```
IN A SMALL ROOM

SOME EXITS ARE :

I CAN SEE :
WALLPAPER. ROPE.

————————>YOU SAID RUN

HOW DO I GET OUT OF HERE ?

————————————>WHAT NOW?
```

APPENDIX 10

Screen Display

```
Break in 390
READY
>PRINT PN% : PRINT E$(PN%) : PRINT D$ (PN%) : PRINT WG$ :
  PRINT WD$
 0

1*
NORSOUEASWESUP          DOWOUTDOOSTALADENTCOTROPTRE
NSEWUDOAFGBCHJ
READY
>
```

APPENDIX 11

Location-Association Arrays

LOCATION NUMBER	ARRAY L$ LOCATION DESCRIPTION	ARRAY E$ EXIT CODES	ARRAY D$ DESTINATION NUMBERS
0	IN A SMALL ROOM	—	1*
1	IN A DIMLY LIT HALLWAY	SWO	Ø*2*3*
2	IN THE KITCHEN OF A COTTAGE	E	1*
3	OUTSIDE A FOREST COTTAGE	NEC	9*4*1*
4	BY THE MOAT OF A CASTLE	W	3*5*
5	IN A CRUMBLING CASTLE	UF	6*6*
6	IN A TOWER ROOM	DFUG	5*5*7*7*
7	ON A PARAPET AT TOWER TOP	D	6*8*
8	HANGING ON ROPE ABOVE MOAT	U	7*
9	IN THE FOREST	NESW	9*9*3*9*
10	DEAD	NESWUD	1Ø1Ø1Ø1Ø1Ø1Ø

APPENDIX 12

Exit Codes

EXIT RECOGNITION WORD	EXIT CODE
NORTH	N
SOUTH	S
EAST	E
WEST	W
UP	U
DOWN	D
OUT	O
DOOR	A
ENTRANCE	B
COTTAGE	C
STAIRS	F
LADDER	G
ROPE	H
TREE	J

KDC

APPENDIX 13

Object-Associated Arrays

OBJECT NUMBER	ARRAY O$ OBJECT DESCRIPTION	ARRAY P% OBJECT LOCATION	ARRAY C% OBJECT FLAG
0	AXE	55	2
1	WALLPAPER	Ø	−2
2	PACKED LUNCH	2	2
3	TREE	9	−1
4	ENTRANCE	5	−2
5	ROPE	Ø	2
6	*PRICELESS CROWN*	99	−2
7	ROLLED UMBRELLA	1	2
8	DOOR	99	−2
9	STAIRS	5	−2
10	IRON LADDER	6	−2

KDC

COMPUTER & VIDEO GAMES BOOK OF ADVENTURE

REGISTRATION CARD

Please fill out this page and return it promptly in order that we may keep you informed of new software and special offers that arise. Simply cut along the dotted line and return it to the correct address selected from those overleaf.

Which computer do you own? ....

Where did you learn of this product?

☐ Magazine. If so, which one? ....

☐ Through a friend.

☐ Saw it in a Retail Store.

☐ Other. Please specify ....

Which Magazines do you purchase?

Regularly: ....

Occassionally: ....

What age are you?

☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ Over 25

We are continually writing new material and would appreciate receiving your comments on our product.

How would you rate this book?

☐ Excellent	☐ Value for money
☐ Good	☐ Priced right
☐ Poor	☐ Overpriced

Please tell us what software you would like to see produced for your computer.

____

____

____

Name ____________________________________

Address __________________________________

____________________________ Code ________

PUT THIS IN A STAMPED ENVELOPE AND SEND TO:
In the United States of America return page to:
Melbourne House Software Inc., 347 Reedwood Drive,
Nashville TN 37217.

In the United Kingdom return page to:
Melbourne House (Publishers) Ltd., Church Yard, Tring,
Hertfordshire, HP23 5LU.

In Australia & New Zealand return page to:
Melbourne House (Australia) Pty. Ltd., Suite 4, 75 Palmerston Crescent,
South Melbourne, Victoria, 3205.

In this unique book, Keith Campbell will lead you through various facets of adventure games, including the history of adventure games, how to play them and a hall of fame.

He then presents a complete program listing, and explains an adventure game, including devising a plot, creating the environment and screen presentation.

Add to all of this, a clear explanation of programming techniques which will show you how to introduce objects, control space and time, interpret English input, move your player from one location to another and many more exciting skills.

Suitable for all microcomputers with specific listings for BBC, Spectrum, and Commodore 64.

ISBN 0-86161-143-8
MELBOURNE HOUSE