Writing Adventure Games on the Amstrad CPC 464/CPC 664

Mike Lewis & Simon Price

WRITING ADVENTURE GAMES ON THE AMSTRAD

WRITING ADVENTURE GAMES ON THE AMSTRAD

Mike Lewis and Simon Price

MELBOURNE HOUSE
PUBLISHERS

IN THE UNITED KINGDOM —
Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

IN AUSTRALIA —
Melbourne House (Australia) Pty Ltd
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

ISBN 0 86161 196 9

Printed and bound in Great Britain by
Mackays of Chatham Ltd, Chatham, Kent

Edition	7 6 5 4 3 2 1
Printing:	F E D C B A 9 8 7 6 5 4 3 2 1
Year:	90 89 88 87 86 85

AKNOWLEDGEMENTS

To Andy Tyson and Doug Walker for playtesting
To Lesley Boxer for the artwork
and Tim Harrison — the "Printer Driver".

CONTENTS

SECTION 1
ADVENTURE GAMES

1
INTRODUCTION

The arcade shoot-'em-up style of computer games is the one most associated with home computers. They are games of skill, involving fast reflexes and good hand-to-eye coordination. While there are some elements of strategy in arcade games, it is more along the lines of "which baddy do I shoot first?" than any complex planning. These are games of action; adventure games are games of thought, of planning, of strategy. Fast reactions will not help you in a standard adventure game — you must use your brain, not your joystick.

There are many different forms of adventure game, as we shall see in the next chapter, but they can all be considered to ultimately be the same type of game. They are all linked at the basic level by a game format which involves the player interacting with the computer in order to solve puzzles, collect objects, and perhaps kill monsters. The player takes the part of a character, who is free to roam around an imaginary world, within his computer, and whose actions are controlled by the player through the computer. The best adventure games give you the opportunity to get lost in their worlds, to take on roles which allow you to escape from mundane, ordinary life and become an adventurer, an explorer. Adventure games can give you the same feeling of enjoyment and involvement which is available from a well-constructed, well-paced novel; or from a well-run roleplaying game session. Of course, not all adventure games are good, far from it, many lack atmosphere, ideas or even a vaguely logical plot. They can reduce the whole adventure playing process to a simple game of guessing which word to use, which object to pick up, etc rather than an exploration of the author's world. Playing a poorly designed adventure game is mechanical at best, if you even bother to play it, that is.

WHY READ THIS BOOK?

With the growth in popularity of adventure games, and the realisation by computer games companies that a good adventure game can sell as well as a good arcade game, there has been a boom in the number of adventures being produced. There has also been a comparative boom in utilities such as The Quill, which allows you to write your own adventures without knowing any programming techniques. This has led to a lot of Quilled adventures, some of which are very good — a lot of which are quite dire. There seems to be a similar expression to "Everyone has at least one novel in them" — with computers, everyone seems to have at least one adventure in them. These utilities allow them to produce their one (or even more) adventures.

There has also been a boom in adventure game books, which purport to show you how to write an adventure game easily and simply. Unfortunately, beyond giving you some form of programming knowledge, which can be picked up by typing in listings, these books have very little to offer. The standard format is to show how to write one adventure game, which is produced by directly coding the adventure in the form of BASIC, and to then claim that this will enable you to write your own adventures. Sadly, it doesn't do this at all; because of the adventure format chosen. The adventures presented as examples are directly coded programs which only apply to their adventure game alone; if you wish to write your own adventure, you have to code your game in the same way, from scratch. What is more, you will have to code each adventure game you write subsequently, in the same way, programming it directly. Far from allowing you to write adventures easily and quickly, this technique wastes a lot of your time, and produces fairly standard adventures, as well as teaching you very little.

That is the conclusion which we came to, after tiring of reading the same material time and again in adventure game books and articles. There are so many computing techniques and methods which can be applied to adventure games, and they seem to remain the secrets of large software houses. Well, this book sets out to outline some of the techniques which you can employ to write your own adventures, some of them fairly advanced; techniques which will actually teach you something new about computing. We have also set out to provide a complete adventure generating system which can be used to design any number of adventure games, without direct coding, or any programming being necessary. The Adventure Kernel System is easier to modify and more flexible to use than most commercial adventure designers as well, as it enables you to see the structure of your adventure game, in a way which menu systems cannot.

WHY WRITE ADVENTURE GAMES?

You may be wondering why anyone would want to write their own adventure games. The simple, and short, answer is for enjoyment and a feeling of satisfaction. A large adventure game is very similar to a novel, in the way it must be designed, plotted and finally written; and there is the same feeling of achievement when you finally finish an adventure as you get from writing a book. Designing and implementing adventure games is fun, as well. Your imagination has complete freedom to produce whatever strange ideas and puzzles you want, and you are able to stamp your own personality and feelings on an adventure game in a way which is not possible with an arcade game. There is also the enjoyment which can be had from watching other people play your adventure game, trying to puzzle through all those problems you carefully designed. In fact, it is often more fun watching people play your game than playing other peoples' adventures — an inside view gives you a wonderful perspective on the player's actions.

Besides the enjoyment derived from writing an adventure game, you can learn new programming techniques, something which isn't possible with a novel! An adventure game is easier to construct and write than a novel, as well, which may be why so many adventure games writers appear to be frustrated novelists! If you are creative and as much of a technophile as we are, it also gives a chance to do something useful and constructive using your treasured computer for a change. No longer will people be able to say that you are wasting your time playing silly games — you are creating them instead!

2
THE HISTORY OF ADVENTURE GAMES

THE DEVELOPMENT OF ROLEPLAYING

Roleplaying games grew out of the hobby of wargaming in the late sixties and early seventies. Wargames take the form of battles fought on table tops with miniature figures. Each figure represents ten, twenty or more men and the gamers move the armies according to a complex series of rules which govern movement, terrain, and the like — with combat being resolved through the use of dice; wargames recreate many different periods of warfare, from ancient Rome through Napoleonic battles to the conflicts of the 20th century. Wargames themselves developed from chess and the military strategic games played by the Prussian Military Command at the end of the 19th century.

The first factor which influenced the development of Roleplaying Games, (RPGs as they are known today) was without doubt the publishing of JRR Tolkein's LORD OF THE RINGS in paperback in 1967. This fired the imagination of a vast audience of young people, the idea of recreating the battles between men, dwarves, elves and orcs appealing to the fantasy fan and wargamer alike. Because of the demand for rules which could cope with the magic and creatures depicted in the books, the publishers of medieval wargames rulebooks had appendices added to include the use of fantasy elements such as Dragon Fire and magic swords.

In Lake Geneva, Wisconsin, USA a small group of wargamers published a book for medieval combat called Chainmail through Gary Gygax's small press games company Tactical Studies Rules, which

had previously published other wargames rules. While this essentially covered the medieval period, it also had a large fantasy element, with giants, spells, trolls and dragons. The game was reasonably popular due to these added fantasy elements.

From here, the game grew with Dave Arneson — a member of the group — creating a dungeon beneath the castle in his campaign. Here individual characters adventured, governed by the rules in CHAINMAIL. The new concept of playing a unique character proved very popular and this idea was developed by Arneson and Gygax into the game Dungeons and Dragons (D&D) which was published by Gygax's Tactical Studies Rules company.

D&D sold extremely well and vast numbers of supplements were soon added to the game to cover the inconsistencies present and the problems that 'combat according to the rules in CHAINMAIL' (as the original rules stipulated) caused.

While the game of D&D was the first, it was obviously incomplete and in many ways had flaws as a games system. There were a number of individuals around who felt that they could do better and soon several rival roleplaying games sprung up. These were games such as Tunnels and Trolls, a very simple level game developed by Ken St. Andre, and published by Flying Buffalo with different rules for magic and combat but still drawing on the fantasy background of elves, dwarves and trolls as inspiration for adventures.

THE GROWTH OF ADVENTURE GAMES

Back in the dim and distant days of 1973, when roleplaying games were first invented, computers were still immense mainframes filling rooms, and while computer games did exist they consisted of Noughts and Crosses and games like Star Trek. While many computer programmers played D&D, they never thought of implementing the game on a computer system. Roleplaying and computers remained very separate entities until two bright students by the names of Crowther and Woods created ADVENTURE.

The game ADVENTURE was created on one of these large mainframes and coded entirely in FORTRAN and Crowther and Woods were clearly D&D fans, for ADVENTURE contains many of the elements found in roleplaying games. It is set in a vast underground complex of caves which are populated with monsters to be fought, puzzles to solve and treasure to win. The player moves his 'character' around the caves by giving two word instructions describing his actions. These consist of a verb and a noun pair such as 'GO NORTH', 'KILL DWARF', etc. Thus he is able to manipulate objects, attack monsters and solve puzzles via simple commands. The game was tremendously innovative at the time, and doubtless many hours of hideously expensive computer time

were spent trying to unravel its mysteries. The player receives a score out of 350 when he dies and the aim of the adventure is to win by completing all the actions possible and to thus score 350.

ADVENTURE remained unavailable to all but a select few due to the scarcity of mainframe facilities to those other than students or computing specialists. The game might have remained a minor diversion for computer scientists, but for one person — Scott Adams. Scott had played the game at work, and had found it fascinating, he wanted to be able to show it to his friends, but couldn't take them into work. So, the obvious solution was to bring the adventure to them! This he did by programming the game in BASIC on the 16K TRS-80 Model 1, despite his colleagues' assurances that it couldn't be done!

Once it had been shown that you could reproduce an adventure game on a microcomputer in just 16k (the original adventure used over 64k of main store) other programmers caught on, and soon other adventures began to appear.

Adventure games consisted of the same basic elements: the two word input, puzzle solving and limited character interaction until relatively recently, when the Infocom games appeared. These grew out of experiments with a parsing system, and as such allow very complex input and output. Zork was the first of the new breed of games, and it provides a surprisingly user-friendly game with some very intricate and subtle puzzles.

Unfortunately, due to the large amounts of memory and disk space required to run ZORK it is only implemented on a few machines. However, some of the techniques shown in ZORK can be reproduced in any adventure games system extremely effectively and easily.

Nowadays, adventure games are a very common piece of computer software, and there are several different types available on all of the small home computers, the Amstrad having its fair share of innovative adventures.

While all adventure games have essentially the same structure — in that they are composed of an interaction between the player and the game, during which the player attempts to solve puzzles and overcome obstacles — they can be sub-divided. These divisions can be based on the style of input and output and how these two are linked through the game.

DIFFERENT ADVENTURE TYPES

Text Adventures

This is the original type of adventure game, and the text adventures available on the Amstrad differ very little in format from the original ADVENTURE on a mainframe computer. The game consists of plain

text input and output on a normal screen with no graphics, sound and normally no use of colour.

The original Crowther and Woods Adventure is available on the Amstrad as Colossal Caves from Level 9, which not only includes the original 200 locations, but adds 70 more! Quite a programming feat for just a small home computer!

The game includes all the elements of the original and more. You start the game on a road near a building, with a forest to the north and a valley to the south. Commands are entered by typing in a two word command consisting of a verb and a noun. Thus, "Enter building" will take you into the building, where the program describes the location as:

> You are in a small building with a well in the middle of the only room. A rusty ladder leads down the well into darkness.
> There is a bunch of keys here.
> There is a small brass lamp here.
> There is an empty bottle here.

Typing "Get Lamp" will enable you to pick up the lamp and typing "Inventory" will list all the objects you are carrying. The adventure recognises the standard abbreviations for directions, such as "S" for "South", "D" for "Down", "E" for "East" and so on. The two word input is limited compared to some games that allow full sentence input, but it is sufficient to play the game (and indeed to play most adventures).

Graphics Adventures

Text adventures with added pictures showing what the locations looked like have been around for a while on computers such as the Apple — the use of discs enabling fast access to picture data. The graphics adventure boom on small computers, however, was really sparked off by the release of The Hobbit adventure from Melbourne House. This led to a lot of other graphics adventures, which attempt to add atmosphere to the game by showing each of the locations you can visit in glowing colour.

The major problem with any graphics adventure is that the addition of graphics to the program eats up the available memory at a ferocious rate! This means that there is less space for locations, objects and actions, and thus the adventure has to be smaller with fewer puzzles and they generally prove to be less of a challenge. While the graphics do add a certain feeling of atmosphere to the game, they can never be detailed enough to accurately represent your location, and thus add little to the adventure element of the game. Most "committed" adventure games players tend to prefer text based games, as they offer a greater challenge.

Arcade Adventures

This type of adventure game bears the least resemblance to the original ADVENTURE, as it is entirely graphical in nature, with very little (if any) text. The character in the game is controlled by a joystick, or via cursor keys, and is moved around the screen collecting objects and fighting monsters. The screen generally depicts a set of corridors or a maze, which must be negotiated, and the monsters are overcome by firing at them — as in arcade games.

Because the action is purely graphical and the standard of animation of the characters in the game has to be high, the graphics will take up even more memory than with the Graphical Adventures. Thus, the actual adventure element of the game tends to be reduced to just picking up or dropping objects, fighting monsters and so on. All things which require skill on the joystick and good reflexes rather than the complex thought and calculation required by the traditional puzzle adventure. However, Arcade adventures may appeal to people who like a little more storyline to their blasting of aliens than in the standard shoot 'em up.

Adventure Simulations

Although The Hobbit falls into this category in many ways, with its interactive characters and their unpredictable behaviour, as do the Infocom games, the only real program to live up to the name is Valhalla from Legend (sadly, only available on the Spectrum and Commodore 64 at the time of writing.).

The world of Valhalla is that of Norse legend and myth, with the gods and goddesses, giants and dwarves, wolves and dragons. You take the part of a character in this world and you interact with the other characters present. Each of these characters has a unique personality and acts with complete independence from you or the other characters around them. The thing that makes Valhalla very different from The Hobbit or any other game available is that all this action takes place graphically on the screen. When you type in the command "Drink Wine", one of the little characters on the screen will raise a wine bottle to his lips and take a drink!

Each location is shown as a fairly detailed and colourful picture, with the terrain varying from plains to marsh or forest, and with castles and huts dominating the skyline. The characters are shown as little figures who walk about the central strip of the screen, drop food, pick up weapons, fight, etc. All this while you can stand and watch.

In fact, one of the fascinations when you start to play the game is just to sit and watch the other 36 characters in the game interact with one another while you don't do a thing! You can join in this world through a fairly complex sentence input which enables you to ask the

other characters for things, ask them to do things and move around the world.

Due to the graphics again, the adventure element is rather limited because of the sheer complexity of handling the animation and independence of the characters. You are really limited to just eating/drinking (a vital necessity to avoid dying of hunger), buying/selling, fighting and handling objects. The commands for such actions can be very complex in structure though, such as: "Sell the axe to Thor for 3Ø crowns".

The purpose of the game is to find the six magical items scattered throughout the world, which must be collected in order. To achieve this you will need the help of the other characters and this can only be gained by impressing them with your prowess at fighting etc.

Quite clearly, the adventures currently available on home computers are a massive improvement over the original Adventure, both in terms of complexity and playability, while still owing their format to the original game.

MACHINE REQUIREMENTS AND PROGRAMMING LANGUAGES

The resources required by an adventure game really depend on the type of game (as classified above) and on the aims of the game. All adventure games require a reasonable size of computer memory and they can be greatly enhanced if some form of disk storage is available.

Ideally an adventure game should have at least 48K available for the data and the driving routines. It is possible to get away with less than this, especially if some form of text compression is used, but the adventure game which can be fitted into a 16K machine pales into insignificance beside a 48K game! When available memory is limited, you are forced into using compression techniques and machine code, to create a useable adventure. This distracts you from the game itself, as far more time is devoted to perfecting coding techniques than developing the adventure game. There is a lot to be said for using a high-level language rather than machine code, and getting on with the game itself.

Which high-level language is most suitable for writing adventure games? Well, the major task which an adventure game performs is the manipulation of large amounts of text, usually in the form of strings; so any language which provides good string handling functions can be used. The original ADVENTURE was written in FORTRAN, simply because that was the only language available, however it is really designed for scientific number crunching, not for text manipulation. The most commonly used language for writing adventure games on micro-

computers is, of course, BASIC, Most BASICs have good string handling, it is a relatively simple language to learn to program in, and it is widely available, all good reasons for using it.

The major problem with using BASIC is that the larger the program, the slower it will run. For most adventure games this is not too much of a problem, as the response times to the player's input are still fast enough to be acceptable. The response delay only becomes a real problem when you are attempting to produce a more advanced adventure game, which will allow complex sentence input. The parsing of sentences takes time, and the delays can easily become totally unacceptable. No player wants to wait thirty seconds between inputting a command and the program responding! Thus, as the games become more ambitious and complex, you are forced to abandon BASIC in favour of machine code, or a compiled language.

As we have already mentioned, resorting to machine code will slow down the development of the adventure game by a significant factor. Machine code takes longer to learn initially, as it requires a totally different programming approach to a high-level language, and this can put the development of an adventure game even further back. Once you have written your adventure game in machine code on one machine, you are then faced with the problems of implementing the same game on a different machine. With a BASIC adventure game, transferring the game to a new machine is simply a matter of translating the program into the new dialect of the BASIC. Despite the lack of standards in BASIC, this is a fairly straight forward task, and it will certainly take you far less time than rewriting the game from scratch on the new machine. If your game is written in machine code, then this is exactly what you will have to do — rewrite the whole game! Even assuming that the new machine uses the same machine language. (e.g. both the Amstrad and the Spectrum use Z8Ø machine code), you cannot simply copy the program across. The routines for printing out text to the screen, inputting commands from the keyboard, drawing pictures, all the things BASIC does for you, will have to be completely rewritten.

BASIC has the advantages of making your adventure game portable, so it can run on another machine with minimal changes, yet it is not the only language you can use. We mentioned compiled languages, and PASCAL is such a language, which is becoming widely available on a lot of micros — the Amstrad included — for a relatively small price. The advantages of compiling a language are that you can write the game in a high-level language which is as easy to use as BASIC, yet it will run at almost the speed of machine code! Unfortunately, standard PASCAL is not very suitable for adventure games programming, as it lacks even elementary string handling functions, making text manipulation a difficult and complex problem. Fortunately, the

people developing the new micro-based versions of PASCAL have realised that the language does have some severe limitations, and they have taken steps to overcome them. The most common addition to the language, and the one feature we really need, is string handling, so you can manipulate text in the same way as BASIC.

If you write your adventure game in PASCAL, implementing it on a new machine is simply a case of putting the same program on the new machine and then compiling it, using that machine's version of PASCAL. Your adventure is then ready to run, without any alteration! There are disadvantages to using a compiled language like PASCAL, as it is not as fast as machine code, and the compiled code is not as compact as machine code; but these are outweighted by the ease of use, the portability, and the fast development time.

The other problem, of course, is being able to afford a PASCAL compiler! While they are becoming more widely available, not everyone can afford to buy one, and thus we will stick with BASIC in this book — every Amstrad has BASIC built into it!

THE FUTURE OF ADVENTURE GAMES

As all of computer technology and computer games are constantly developing, so is the adventure games genre. There are always new ideas to be tried and new complexities of programming to be reached. The future of adventure gaming looks very healthy, with a wealth of new technology and knowledge to draw on.

We have already seen the use of video disks in the computer games field with arcade games such as M.A.C.H. 3 and the adventure game Dragon's Lair (DL). In these games, the computer projects images from the video-disk which correspond to the players actions. Thus, in Dragon's Lair, you control Dirk the Daring, a fearless fighter, and all the action is shown on screen in the form of an animated cartoon. Unfortunately, games such as DL bear little resemblance to the adventure games we know, because the action is so limited. The computer cannot access frames continuously from the disk, in response to the players actions, and thus the game comprises of a number of scenes, which are "jumped" between. In DL the player moves Left, Right or waves his sword at the appropriate point. There is very little interaction.

However, the technology is coming here, and it should soon be possible to have a fully interactive use of Video. The adventure would now take the form of a live "film" in which the player takes an active part. The characters in the film will respond to you, and the character representing you will act out the actions you dictate. Thus it will be like Valhalla in some ways, but far more realistic and with the storage capacities of Video Disk, the true adventure element can be preserved.

Developments in the field of Artificial Intelligence research will also have a great effect on future adventure games. There is not only the concept of using the logic and knowledge processing techniques that have appeared in games such as Sherlock from Melbourne House and the Infocom games, where the other characters in the game appear to be intelligent, and thus you can order them around, hold conversations with them, etc. There is also the work in the field of natural language processing which has obvious applications to adventuring. One of the most frustrating aspects of playing an adventure game is the limited vocabulary available to you, and the problems often encountered when trying to find just the right word for a desired action. However, if you could type your commands into the computer in your natural language, in English, there would be no problem; you are freed from the restraints of an artificial language and able to concentrate on the adventure and absorb its atmosphere, without distraction.

Typing in commands is always a problem, especially when you want to use a long sentence and are a poor typist. Here, Artificial Intelligence research can help as well, in the field of Speech recognition. A truly interactive film could be produced if you were able to physically speak to the characters in the game, and to hold a spoken conversation with them!

Multi-user adventures are already starting to appear with the most famous of these being MUD, which is run on a small minicomputer. The acronym standing for Multi-User-Dungeon. MUD enables the player to not only enter a large, and complex adventure world, but to do this in the company of other players! Thus, the characters you meet while playing the game will not obey simple rules devised by the programmer — these characters have exactly the same potential for unexpected behaviour as you do as their controllers are human. While MUD is a fairly limited adventure game, bigger and better versions are already being worked upon with hundreds of locations, objects and players! The concept of multi-user games, either on large mainframes, where the players take part via a modem and a phone line, or via a large multi-user local network system, each player using one system terminal, offers almost infinite possibilities for expanding the present day adventure game and increasing the realism.

Fairly obviously, most of these ideas are a long way from becoming reality, yet they do show that there are many areas of computing which can be applied to adventure games and adventure game programming; areas which are on the forefront on computing research. Considering the growth of computing and computing techniques over just the last ten years, they could be here sooner than you think.

3
PLOTTING AN ADVENTURE

THE IMPORTANCE OF A GOOD PLOT

When first confronted by a new adventure game what is it that attracts a player? Is it the style of presentation, the colour the text is printed in or even the packaging? Most adventures on the computer market bear a very close resemblance to each other, especially with text adventures; there is only so much you can do with plain text output and input. Packaging may affect a buyer's choice, but it cannot hide a terrible adventure game beneath it.

When you consider any adventure game and the intitial appeal it has to a player, indeed the whole appeal of playing it — you realise there is one over-riding factor. The plot. Above everything else the adventure game must have a good plot. The plot idea is the game, all else is just trimmings to improve presentation rather than contents. Yet, it isn't enough to just have a good plot, even a well thought out plot with multitudes of twists and turns can bore a player very quickly — if he's seen it all before.

The plot is the very heart of your adventure and as such should be strong and well-defined. The best adventure games have plots which lead the player in stages through the game, until the eventual climax. If the plot meanders along, the player is going to be left wondering what he is supposed to be doing, and is going to lose interest in the whole thing. There must always be a firm goal in the player's mind as to what the adventure game is about, and what he is trying to achieve — of course, he may find out that he is totally wrong! But that is just one possible twist in the adventure.

The plot of an adventure game is composed of two elements, the actions and puzzles which make up the plotline and the background to the adventure — the setting in which everything takes place. We'll

look at the background in a moment, but for now let's consider the basic elements of an adventure game plot.

(a) The Map

This is the very lowest basis of any adventure game and it will show the area over which the player can move and where all the action takes place. While it may seem that the map in an adventure game is really only a list of locations and their connections which has no direct bearing on the plot, this really is not true. As the map is the basis of your adventure game so it is the basis of your plot, and a good, well thought out map can improve an adventure immensely.

The basic map will contain all the locations necessary for the action in your game to take place, each location might be used for an object, or an action to take place or even just as a red herring. The problem when designing an adventure game is deciding on how detailed the map should be — should you describe each room in a house, or simply have one location representing the whole house? Should you represent a road on your map as a series of (similar) locations, or just ignore it with the player moving from one end of the road to the other in one turn?

These difficulties can only be overcome by considering just how important each of these locations is in your game, and how each relates to the main plot and purpose of your game. If the only part the house plays in your game is to provide somewhere for a knife to be found, then why bother with more than one location — or perhaps two with a kitchen? The extra rooms in the house serve no useful purpose and will either bore the player or side track him from the main part of the game if he tries to find out what use they are. In any adventure game the number of irrelevant locations should be kept to a minimum — you do need them to keep the atmosphere of the game flowing properly. If your player is on foot, making locations long distances apart will only make the game seem unrealistic — either provide him with a means of transport or separate the locations with intermediate ones.

Making your locations and their descriptions interesting is an important part of the adventure writing process. Think about the background and basis for your game and come up with locations which match the atmosphere and style of your game. A mystery/horror adventure is best set in a spine-chilling mansion, not in the centre of town! Location descriptions should be long and evocative as well, two liners like "You are in the hall near the stairs and the kitchen" hardly convey a rich atmosphere.

(b) The objects

Many of the points already raised about locations apply equally well to objects. Try to avoid having too many objects in a game which are

not useful. Just because a kitchen usually has pots and pans in it doesn't mean you have to provide them if there is no use for them. The objects and locations should tie together in some way, without appearing too contrived, and without the objects seeming out of place.

(c) Puzzles

These are the major component of an adventure game and are what make the game a challenge to play. Poorly thought out puzzles can make a game far too easy to solve and thus bore the player or make the game impossible to complete, thus frustrating the player. The trick is to balance your puzzles somewhere between the two extremes, something which is not easy to do!

If you have difficulty thinking up puzzles of your own, then it is possible to adapt puzzles from other adventure games and disguise them by altering their circumstances and the objects used to solve them. For instance, in the original ADVENTURE, you must capture a bird in order to get past a large green snake. When you approach the snake, releasing the bird causes it to drive the snake away. In other games this has been translated into throwing Egyptian bird statues at a snake god while exploring a pyramid, and so on! Disguising a puzzle is not always easy and you cannot really rely on other adventure games for all your puzzles!

A puzzle should be both logical and yet hard to see unless you strike on the correct sequence of events. There is nothing more infuriating in an adventure than totally illogical puzzles which have no basis on any reality for their ideas. There must always be some way of solving a puzzle other than wildly guessing which objects to use and in which order and in which room! Because of this, it is important to try out your puzzle ideas on other people, just to see that you haven't made a logical jump from the solution to the problem which is impossible to make in the correct order. Not everyone will think in the same way as you, in fact very few people might have your knowledge about particular situations and events, so make puzzles fairly general. A solution which involves knowing a complex mathematical formula and how to apply it is not going to be solvable by most people unless you give a lot of clues!

BACKGROUND

The major pitfall that new adventure game designers (and several experienced ones) fall into is the reliance on the same old standard plot lines for their adventures. Game after game allows you to take the role of a heroic fighter whose mission is to save someone, or thing, from a horde of monsters. Just cast your mind over the games you have played or seen which are based on quasi-fantasy lands populated

with Orcs, Trolls, Dwarves and Wizards — all eager to get at your treasure. A lot of them, aren't there? All these games draw on the same background, that of the fantasy roleplaying games such as Dungeons and Dragons. Even the original ADVENTURE took this as an influence, and other programmers have been doing it ever since!

So, the first aim behind any adventure game we design is that the plot should attract the player into wanting to play the game, and avoid giving him a feeling of deja vu as he reads the instructions! This means constructing an original and thought provoking plot by using an unconventional background for our game or by using a standard plot in a new and unusual way. What we must not do is fall back on the same old ideas culled from the adventure games that we've seen — there is no point in recreating some one else's game and ideas!

If the fantasy genre is out, what can be used as a basis for the adventure plot? The answer is everything else really! There are so many sources for adventure game ideas that it would be impossible to even begin to list them all! The following are a few areas that are worth considering:

Historical

There are a large number of periods in history which could be drawn upon in a successful and entertaining adventure game. The major problems with using a historical period as the basis for your plot is that of accuracy. You must stick to the period in detail, and avoid any inconsistencies which a player might pick up, thus ruining the game. For example, you cannot introduce modes of transport such as cars or trains before the period they were available.

There is also the problem of recreating the atmosphere of the period, so that the player feels as though they actually are playing a game set in Victorian England, or wherever, rather than just a standard adventure with a few oddities thrown in. The secret here is to use the small details from the period to reinforce the feeling and mood of the adventure, for example, the music from the period overheard in the street, details of people's clothing, perhaps even the styles of characters' speech can all add to the adventure's atmosphere.

Some periods are obviously more suitable than others for an adventure game, as there is more happening, or a more interesting background for the adventure to take place against. Possibly the player could take part in an historical occasion, where only their actions enable history to come out as it should have done. Or perhaps they can alter history, playing the part of a famous historical figure.

It is far harder to write a consistent and believable historical adventure game than almost any other type, because of the little details needed to create the atmosphere, but it is, or can be, one of the most rewarding to write and play.

Fictional

Basing the plot for your adventure game on a book may seem a very good idea — your plot has been written for you! All you have to do is translate the book into an adventure — far simpler than creating a plot from scratch. Indeed, there are a lot of books which would make quite excellent adventure games and some have been used (for example The Hobbit) as adventure games already.

The major problem with this approach is if your player has already read the book when he comes to play your adventure. If the adventure bears any resemblance to the book, it is going to be relatively easy to solve the adventure as all the actions and their correct sequence, are already known. If the book and adventure are different in content and style it may make the game more of a challenge, but you are going to lose the atmosphere of the book, which the player will be expecting.

A better approach is to base the adventure on the style and feel of one of your favourite books, but without the exact plotline. This makes the atmosphere of the game easier to put across and the game easier to write — if you are happy with your game and enjoy the book, you will find it easier to recreate that atmosphere than with an unknown subject. Pick perhaps the best and most amusing bits from the book and include these if you wish, slightly altered so that a reader will pick up the reference without making the game too easy for him.

There are also the mercenary advantages of not basing your adventure game directly on a book — if it should prove to be a saleable product, you will need permission from the book's author/publishers to use it. This means paying them a royalty and less money for you! If you are writing adventures purely for fun, however, this won't really bother you.

Modern Day

There seem to be very few adventures based on modern day situations as most people prefer to escape from the present not experience it on their computers! For the adventure writer, though, the idea of setting an adventure in the present is not only a challenge, it may prove to be easier to write than you think. The advantages of such a plot is that the player can be assumed to be familiar with the situation he finds himself in which makes the explanation and background work you have to put into the game so much easier and shorter!

While there may not seem much you can do with an adventure game based on today, there is a whole world of excitement going on around you! We are not suggesting that the adventure should be based on travelling to work, etc — no one wants to play through the humdrum things of everyday life. What about foiling terrorist plots, freeing kidnapped people, etc? There is still plenty of scope to create novel and

interesting plots even with the limitations of an up-to-date and modern setting.

SUPPORT MATERIAL

Adventure games can never create a complete atmosphere which holds the attention of the player, and totally suspends his belief, because of their limited format. Any graphical adventure game will be let down by the limited graphics available on home computers. This is where adventures can be improved through the provision of additional, non-computer based material.

Rather than waste vast amounts of memory on computer graphics which generally fail to convey any real atmosphere, why not provide a separate set of pictures tied in to each location. This idea has been used by several commercial adventures, and enables the adventure game to be more complex and absorbing by utilising the extra space, as well as improving the quality of the art enough to allow extra clues to be hidden in the picture information.

Another idea is to provide a map with the adventure game showing the general area in which the adventure game takes place. This enables the player to move around the area he is supposed to know, thus making the game more realistic, while not giving away just which locations appear in the actual adventure game, or how they are connected. A separately produced map can be made far easier to read and more attractive than any produced on the computer screen. It is also far more practical if the player has to refer to the map while he is playing!

There is no reason why you should limit the support material provided with a game to just the graphical elements from an adventure. If the player starts with a number of items in his pocket then provide actual physical counterparts for these items! You have to take a practical approach to this of course, but such things as bus tickets, cinema passes, small fragments of map and so on are easy to produce and add so much more to the atmosphere of an adventure. Providing the actual items cuts down on the detailed descriptions needed and enables the player to examine them in far more detail than is possible within the program.

Other ideas for source material which will improve and expand on a simple adventure game include character portraits, if you have characters in the game, these will enable the player to visualise just who he is interacting with; object illustrations, which will show far more detail than it is possible to include when the player examines the object. Obviously this is only really necessary for important objects, rather than the mundane, everyday things such as knives, etc.

The support material provided with Witch Hunt, the example scenario detailed in later chapters should give you some idea of the type of material which is suitable for an adventure game. One pitfall to try and avoid is going over the top on support material, to the detriment of the adventure itself. If your support material contains far more information and details than are in the actual adventure game, you are going to reduce the players' enjoyment of the game, rather than enhancing it. Once this happens, the adventure becomes less of a computer based game, and more of a computer assisted game, where the computer adds to the main adventure, rather than controlling it.

4

THE STRUCTURAL ELEMENTS OF AN ADVENTURE GAME

As we have already seen in chapter 3, an adventure game breaks down into elements such as locations, objects and puzzles. Each of these structural elements can be implemented in a number of different ways, and in this chapter we will describe some of the methods which are possible, though we will leave the actual programming implementation details until we discuss AKS in later chapters.

A. LOCATIONS

Let's leave the methods of storing and accessing the location text until later, when we will consider all the text the game requires as a whole. For the moment, the major thing we are concerned with is how to link each of the locations with its neighbours and how to tie these links in with the commands the player will use to move around. We will use the small adventure map shown below as an example. Each of the locations is shown as a box, and the directions the player can move are indicated by the arrows.

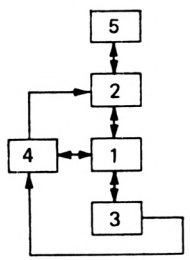


FIGURE 4.1

Each of the locations in an adventure is connected to others via a link, and the structure we use to represent this is known, reasonably enough, as a link map. The simplest form of a link map is a straight forward table. The link map for our example is shown below:

LOCATION NUMBER	DIRECTIONS			
	North	South	East	West
1	2	3	Ø	4
2	5	1	Ø	Ø
3	1	Ø	4	Ø
4	2	Ø	1	Ø
5	Ø	2	Ø	Ø

Each of our locations is represented by a number, these running down the table. For each location we then show which location we will be taken to if we take one of the four directions. Thus, in the above example, assuming we start at location 1, moving South, then East will take us to location 4. Note, if there is a Ø shown for the direction it indicates that we cannot move from that location in that direction. e.g. there is nowhere to go to East of location 1. Obviously, there is no reason why we should limit ourselves to just the four basic directions, and most adventures include the other compass directions NE, NW, SE, SW as well as UP and DOWN.

Using this method, we just need to store a series of numbers for each location, giving details of the links to other locations, and these are checked each time a movement command is entered. If it is possible to move in that direction, the player's location is altered to become that of the link entry, otherwise the "You cannot move in that direction" message is printed.

While the basic design of link map is fine for simple adventures, it limits what you can actually do during the adventure to fixed movements. The possible directions you can move in from on location never vary. However, there will be times when you want the player to perform an action before he is allowed to go in a certain direction. To return to the snake and the bird problem mentioned earlier, the player's way is blocked by a giant snake unless he has the bird and releases it to drive off the snake. Once he has solved this particular puzzle, he is free to move in the direction the snake blocked, and onto a new location. Thus, we need some method of representing this type of conditional move; the answer is to add a link condition to the link map.

This is done by adding a condition to each directions information, which must be true before the player can move in the direction. In a lot of cases the condition will simply be TRUE, meaning the player is able to move in that direction any time he wishes to; for some it will be

FALSE, meaning there is nowhere to go in that direction, while a few will have condition to be evaluated. In the case of the snake and bird puzzle, the player can only move once he has released the bird and there are a couple of methods of testing this. We could simply test if the bird is present in this location, if it is, the player has dropped it and we have already dealt with the snake. This presents problems, as if the player picks the bird up again, he will meet the snake again! A better solution is to use a flag — a variable which can be TRUE or FALSE — and test this to see if the player has released the bird yet. It will have an initial value of FALSE, so the player will not be able to get past the snake, when the bird is released, it is set to TRUE, and the way is now free.

The use of conditions can also apply to location descriptions, and it enables the adventure to produce different location descriptions depending on which condition is true. Most adventures use this facility to provide a long description the first time you visit a location, with a shorter, compact description for subsequent visits. The conditions need not be limited to flags alone, and they can include tests to see if you've visited a certain location, are carrying an object, wearing an object, or an object is present at the location.

B. OBJECTS

The first requirement for objects is to have some form of Object Characteristics, which will describe each object, and its details. The objects have to be manipulated by the player, and thus we will need some way of limiting the number of objects a player can carry at any one time. One simple and commonly used technique is simply to limit the number of objects to a fixed total, usually around 4 or 6 objects. Yet, this is unrealistic, as objects will have different properties and weights, and the player should be able to carry more of one type of object than another due to these properties. The Object Characteristics Vector is a simple table with an entry for each object, which gives the value for that object for each of the designer's chosen properties. Let us just consider one property, that of weight: each object can be assigned a weight value, which will represent a proportion of the total weight a player can carry. Thus, deciding if the player can carry an object is a simple matter of comparing the object's weight with the weight the player is capable of carrying.

This approach allows the player to carry a variety of objects and forces him to balance objects against each other — do you carry around one heavy object you think you need soon, or a number of lighter ones? It is also possible to stop players picking up objects which should be immovable, simply by setting their weight to more than the

total carrying capacity of the player. If they attempt to lift them, they are then told that the object is too heavy!

Only certain actions can be performed on each object, after all you can't ride a brick or eat a mirror, or whatever, and we need some way to represent these facts! The solution is to use Action Suitability codes for each of the objects, which explicitly state which actions can be performed on each of the objects. Thus, the entry for a hat might allow the actions: GET, DROP, WEAR, REMOVE, and EXAMINE to be performed on it. If the player attempts to say something like "EAT HAT", he will just get a standard "YOU CAN'T EAT THAT!" reply!

As well as needing to know how much objects weigh, and what actions the player can perform on them, we also have to know just where they are! The object's location is represented by a number, which simply shows at which location the object is present. When the location description is printed, the program then scans the Object Location Pointers and prints out the description for all the objects whose location matches the current player location.

There are a few special location numbers which can be used to show where objects are when they are not in a physical location. Objects may also be carried by the player, or possibly worn or they might not even exist yet! This requires the program to have a standard set of numbers to represent this. For example, we might use:

```
  Ø = object does not exist
254 = object carried
255 = object worn
```

Thus, if a player picks up an object, its location pointer is altered from pointing to the current location, to being 254 to indicate it is being carried. Occasionally, when an object has been used by the player to solve a puzzle, we might want that object to be used up, or if the player eats some food, we have to take the food off him. This is done by setting the object's location pointer to Ø, which "destroys" the object. Resetting an object's location pointer from Ø has the effect of "creating" it.

When programming, it is more efficient and safer to use all data structures in a consistent way — programs are easier to debug and understand. The use of special location numbers is a "dirty" technique, in this respect, as we are using a straight forward data structure in a messy way. If you can avoid this approach (and it is really a hang-over from the early days of adventure games programming) then so much the better. The alternative is to have an array representing the objects worn, which point to the object descriptions.

C. ACTORS

These are the player himself and the non-player characters which may inhabit the adventure, and with whom the player can interact. There are several ways in which to implement actors, some of which involve very complex and time-consuming techniques. One of the simplest ways of dealing with actors is to regard them as a type of pseudo object. If this approach is taken, the player can be moved from one location to another and manipulated in exactly the same way as any ordinary object. The difference being that the pseudo objects are not described along with the normal objects when the location description is given. Thus, the pseudo objects may only be manipulated by actions from within the program, not by the player himself. This avoids the problems inherent in the player picking himself up!

One slightly more advanced way of implementing non-player characters (NPCs) is to regard each of them as a player in their own right, with a predefined set of commands and desires. Thus, interrupts are used to share the processing time between all the actors, with each taking their turn to perform an action if they desire. This enables the NPCs in the game to have an appearance of true independence and lets them perform all of the actions the player is capable of performing. The programming implementation of such a system is harder to produce, and it requires careful calculations on timings to ensure that the player cannot be "locked out" of the game by hyper-active NPCs!

D. ACTIONS AND EVENTS

Actions are the result of commands from the player and they cause objects to be moved around and things to occur. An example of this is the movement verbs, which will cause the player to change location according to the links in the link map for the current location.

Events are actions which are triggered independently of a command from the player. This is usually implemented using a condition, which is tested each time the player inputs a command, to see if it has been triggered yet. When the condition evaluates to TRUE, the relevant event is set into motion. This approach allows you to implement events which will occur after a set number of turns have passed. For example, if the player drinks some poison, he may have a number of turns in which to find the antidote, after which the poisoned event will trigger and he will be informed he is dead! Alternatively, events can be used to regularly trigger an action after every few turns. This might be used to implement hunger, for example, so the player becomes hungry after every 20 turns have passed and then has 5 turns in which to find food; but the effect wears off and thus he has to find food again, and so on.

E. VOCABULARY

The vocabulary of an adventure game consists of the commands available to the player, which can be used while interacting with the adventure game. It is not only the actions and objects which make up the vocabulary (as in "GET LAMP"), even in a simple game. Sherlock has a vocabulary of 8ØØ words, and the Infocom games have a similar range of words, which allow you to construct very complex sentences such as "Get all the boxes except the yellow one and put the blue box under the bed"!

While the storage of vocabulary as data statements is fairly straight forward, a decision has to be made about where the player can access that part of the vocabulary. Often, during an adventure game, the player will reach a location where a special command is needed to solve a puzzle; this command only works in the one location and it has no effect elsewhere. If the vocabulary is stored globally, that is all the words are available throughout the adventure, the entire vocabulary will have to be searched each time a command is input. Even if the vocabulary is reasonably simple, searching all of it is going to take a fair amount of time, and reduce the response time to the player's commands even further. Obviously, the number of words to be searched through can be reduced considerably if the "special" commands are stored separately.

Within each location, we store a list of special commands which can be used here, and when a command is input, these are checked before the global vocabulary. While saving time, this approach does mean that words will generate very different messages depending on which location you are in, as the "special" commands will result in a standard "YOU CANNOT DO THAT" message unless you are at the pertinent location. The error reporting could be greatly enhanced if messages such as "YOU NEED THE WAND TO DO THAT" could be output. It is a question of balancing the user-friendly responses with the reaction time you want.

Once we have decided where to store our text, we have to consider the method of storage. The easiest approach to implement and understand is that of a serial list. This is usually just a series of data statements which contain the commands available, one after the other, with details of which actions to perform for each command. Thus, if we want to have a command PICK UP, and we already have an action GET, we might define it as:

```
IF ACTION = PICKUP THEN PERFORM GET, PRINT OKAY
```

which would perform a GET and then print out the message "OKAY". We will see more about this when we examine AKS in later chapters.

While the serial list approach is fine for small vocabularies, if the vocabulary is large it will require a long time to search through it all, especially if the word to be matched occurs near the end of the list. A more effective approach, but requiring more memory, is to store the vocabulary alphabetically, with pointers to the start of each letter of the alphabet. While this could be implemented using simple data statements, the most efficient method uses LINKED LISTS and what is known as an ILIFFE vector. For those of you who have never come across list structures, we'll explain them before demonstrating how they can be used in adventure games.

Linked Lists

A normal list is a straight forward, sequential list, which starts at one end and which can be followed to the other. This can be implemented using a normal one-dimensional array, as shown below:

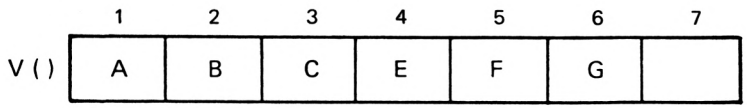


FIGURE 4.2

The array V has seven elements, of which six contain a letter of the alphabet. The seventh element is empty. If we wish to search through the list V for a letter, we simply start at the first element V(1), and continue through the elements until we reach V(7), or find what we are looking for. If we wanted to add a new element, D to the array, we can either place it in V(7), or if we wish to retain our alphabetical order, we have to place it in V(4), which means moving the elements E to G along. With a small array, as in our example, having to move the elements to make room for a new entry is no problem, but if you have an array which has several hundred elements it is time consuming!

Let's now look at simple linked lists. These are similar to normal lists, except that each of the elements has a pointer to the element after it in the list. A pointer is simply a variable which contains a value corresponding to a location in the array. Thus, a linked list version of V would look like this:

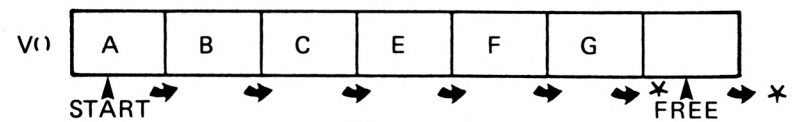


FIGURE 4.3

So, each element has a pointer to the element which immediately follows it. We also have two special pointers, outside of the array. These are a pointer to the start of information in the array, and a pointer to

the start of the free space — i.e. the first element of the array which is empty.

If we wish to add the new element D to the linked list V, there are a number of steps to be followed. Firstly, we put the new element into the first free space, as indicated by the Free Space pointer, this gives us:

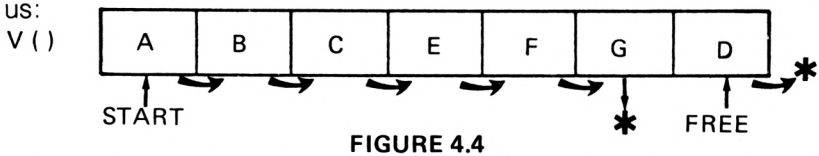


FIGURE 4.4

We now have to find out where in the list D should be put, this is done by searching down the list from the element the Start Pointer points at. We then follow the pointers from each element to its successor, until we find that D should go between C and E. We now have to insert D so that it is pointed to by C and it points to E.

We set the D's pointer to point to E, which will give us:

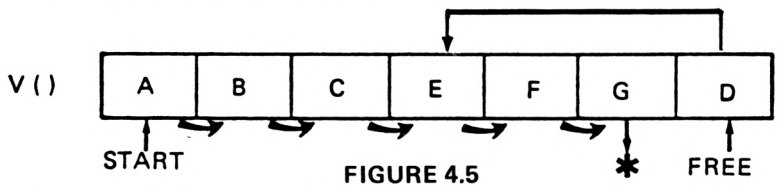


FIGURE 4.5

We can now set C's pointer to point to D.

FIGURE 4.4

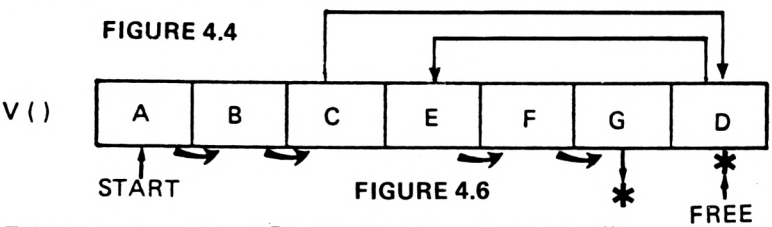


FIGURE 4.6

This has now linked D into the list, without shuffling any elements around. If you follow the pointers from the start, you can see we will go from A to B, to C, to D, and E and so on. The last thing we must do is adjust the Free Space Pointer, otherwise we will overwrite D if we add a new element. So, we set the Free Space Pointer to the next element after D. In our example, D is the last element of the array, and thus the Free Space Pointer will be set to a * indicating that the list is now full.

That is only a brief overview of linked lists, a very useful technique for manipulating data, and one which can be much more complex with two-way lists, circular lists, etc. If you are interested in more examples and explanations of how linked lists work, we recommend you look at any computing book on data structures or algorithms.

Right, you now know what a linked list is, and must have already realised how useful they can be in adventure games for storing data. If you wish to add an element to a list, you simply need to rearrange a couple of pointers, and avoid the problem of shifting large amounts of data to keep everything in order. For storing vocabulary in such a way, that it can be accessed quickly and simply, we require 26 linked lists, one for each letter, and a method of pointing to them. The pointer to each linked list is known as an IFLIFFE Vector, and is simply a linked list whose elements point to other linked lists. The only major problem with this method being the overheads of storage needed for all our pointers. Thus, the arrangement we will have is shown in FIG 4.7.

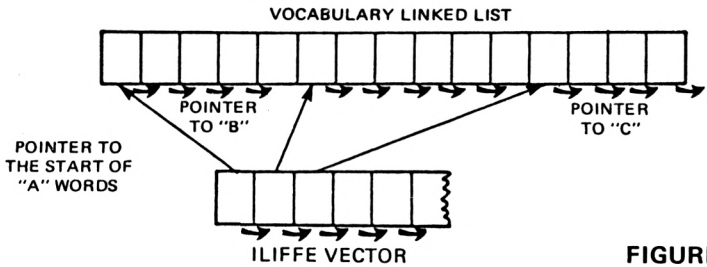


FIGURE 4.7

Now, when we have a command we wish to find, we take the first letter of the command and then follow the appropriate pointer from the ILIFFE Vector to the start of the commands beginning with that letter. For example, if the command was "KILL", we would take the K, which is the 12th letter of the alphabet. This means that the 12th element of the ILIFFE Vector points to the start of the commands beginning with K. We follow the pointer and can then scan down the list of "K" words. This ability to go directly to the section of the alphabet we require makes the list scanning very much faster, and thus the response time quicker.

The ILIFFE Vector arrangement shown in FIG 4.7 is relatively simple, for very big vocabularies, there is no reason why you shouldn't have an ILIFFE Vector which points to a series of ILIFFE Vectors, which then point to the word list. This enables you to split the words up even more, using the first letter to find the appropriate ILIFFE Vector, and then the second letter to find the start of such words in the command list! It really depends on the space you have available and the size of your vocabulary.

Simple Parsing Techniques

Now that we have considered how to store your vocabulary, the next step is to look at methods of interpreting the input you receive from the player, and matching this up with the vocabulary. The process of analysing an input string and breaking it down to the individual commands

is known as parsing, and we'll look at a few basic methods of achieving this.

The simplest, most basic form of command input is the one used by the original ADVENTURE and a lot of adventure games since — the two word command. This consists of a verb, which denotes an action, followed by a noun, usually an object. Thus commands along the lines of "GO SOUTH", "GET LAMP", "ENTER BUILDING", etc are all that are accepted by the program.

The two word command does have a lot going for it, in that it is extremely simple to parse. The command string can be split easily into the two words, by taking the space (or spaces) between the two words as a separator, and assuming that the first word is an action and the second an object. The action is compared with the list of possible actions, and control is passed to the action routine it matches with. This routine will then compare the object section of the command string with the list of objects it expects, and will perform the action if it matches. Error reporting is fairly straight forward, if the action is not found, the program replies with a "I DON'T UNDERSTAND."; if the object cannot be found, it simply says "YOU CANNOT action WITH object". Very basic, but effective.

The first complication which can be built into the parser is to make it accept "and", "then", "the" and the use of commas. This allows the player to say things like:

"GET THE APPLE, GO SOUTH AND THEN THROW THE APPLE."

It looks a fairly complex sentence, as the player is performing three actions with just one command. Yet, it is really just three actions which follow each other. The parser can ignore all uses of the word "the", as this is just a flowery addition to the sentence. The words "and", "then" and "," are simply command separators. The player could have typed in the above sentence as:

"GET APPLE"
"GO SOUTH"
"THROW APPLE"

with exactly the same effects. The parser simply has to scan through the input string from left to right, building up a list of commands. It will find "GET APPLE" first, and this is stored; the comma indicates the end of one command and the start of the next so it can be ignored. "GO SOUTH" is a straight forward command and can be stored as such. The only complication comes with the use of "AND THEN" where the player has to use two connectives. The parser has to take "AND" as a connective between two commands, and then realise that "THEN" is another connective, and thus ignore it, leaving "THROW APPLE" as the final command. Very complex sentences can be parsed using this

method, and it speeds the game up, as the player doesn't have to wait for the results of each input before typing in the next one.

Very complex parsing techniques can be used, which allow the user to use adjectives, adverbs, etc in his commands. These techniques require some very advanced Natural Language processing, a subject which is a field of Artificial Intelligence Research in its own right! Even in the most advanced Natural Language systems, these techniques have not been fully implemented yet.

5
SAVING SPACE

Adventure games can be developed very successfully in BASIC; the Adventure Kernel System described in the next section is a good example. It is possible to create a game which contains most of the elements of the original adventure or the commercial adventures available today. That is, you can create small games which contain these elements — the moment you try to create a game the size of Adventure (which had over 200 locations) — you are going to run into problems. The home computers available today simply don't have enough memory to contain that big an adventure. Even the Amstrad, with 64K available, would be hard pushed to hold more than 100 locations, and even then, the descriptive text would have to be kept to a minimum.

So, how can you produce an adventure game with vast amounts of flowing prose, and a large number of locations, puzzles and objects? You can resort to programming your game in machine code, but this will only save a small amount of space. The major part of any adventure game is not the driving routines, the actual program, but the data which makes up the adventure. Even using machine code, your data is going to occupy the same amount of space. However, it is possible to produce large, complex adventures on even a small machine, as Melbourne House has proved with the Hobbit. So, how do they fit all that information into the machine? Well, there are a couple of techniques employed by commercial adventure games which can be used by anyone to produce large adventure games. These are using discs as a backing storage, and using text compression. We'll look at each of the methods in detail.

DISC ADVENTURES

As well as providing a fast storage medium for the initial loading of the adventure, discs can be used during the game, to load in different sections of the adventure game, as the player encounters them, thus

abolishing the problems of fitting the whole adventure into the computer at the same time.

In an adventure game, the player is only ever at one location at one time — fairly obviously! This means that the location data for all the other locations is not needed, and thus does not need to be in memory at all. When the player moves from the current location to another one, the new location data is loaded in from disc.

The technique of loading data in from disc, when it is required, is known as PAGING, and is used by large computers to simulate a much larger memory capacity than they actually have. In the adventure game situation, we can divide memory up into several FRAMES (typically 256 bytes long) which can each hold a PAGE of data from disc. All the adventure data is stored in the form of pages, including objects, puzzles and commands, and the relevant sections are loaded in when needed. The driving routine is the only part of the data to be permanently in memory, and it searches all the frames of data, for the action, object or location data it requires. If the data is not currently present in the computer, then a PAGE FAULT is generated which causes the required page of data to be loaded.

If we just loaded in new pages each time we needed the new data, without considering the pages we were replacing, or even which page we should replace, then we would soon run into problems. If a page has been altered in some way, and we do not store the alterations, the next time the page is loaded, those alterations will have been forgotten by the adventure. In an adventure game, which relies on manipulating object data and the player's location, this would prevent us doing anything! The answer lies in developing a PAGE REPLACEMENT POLICY, which will enable us to save pages which have been altered, and to decide just which page to replace. The latter point is a very important one, if we replace a page which is being accessed all the time by the adventure — perhaps it contains the data for a command the player is using a lot — then the number of page faults will increase dramatically; slowing down the adventure game's response as it constantly loads in new data from disc.

The best approach to the problem is to consider the data to be split into three distinct types:

A. Resident Data

This is the data which will remain in the computer all the time the program is running, and it comprises the main driving routines, and sometimes the most important variables and counters in the game.

B. Pure Page Data

This is data which cannot be altered by the program, and thus it will never need to be written back to disc before replacing it with a

new page. An example of pure data is the location data, where the descriptions and connections never alter during the game.

C. Impure Page Data

This is the data which may be altered, and thus must be written back to disc before it can be replaced in memory. Each page contains a special marker, which is set when the page is altered in any way. When the page comes to be replaced, the marker is checked, and only if it is set is the page saved to disc. This prevents time from being wasted by saving pages which have not been altered.

The impure data consists of all the volatile, changeable parts of the adventure game, and it is here that the object pointers, flags, counters and variables would be stored. This also allows the adventure game to be saved during a game by saving the impure data alone, and to be restored by reloading the same data.

As well as saving pages before replacing them, we must consider which pages to actually replace, as we have already noted, and this has created a large number of very complex PAGE REPLACEMENT ALGORITHMS within the world of mainframe computing. We won't consider them all here, just take a brief look at one possible approach to the problem. Fairly obviously, we do not want to replace a page which is shortly to be used again, as this only means reloading it from disc. But, how do we tell when a page is in use? Or whether it is no longer required by the adventure game? There is little point in keeping location data for a location the player can no longer reach, for example, or object data for a puzzle the player has solved and no longer requires.

Well, in the case of impure page data, we can tell if the page has been used by looking at the marker which says if it has been written to; if this marker is set, then the page is in use. Thus, we replace the pages which have not been accessed before the pages which have been written to. This is not an infallible approach, however, and useless when it comes to considering pure page data. During the course of an adventure game, data tends to be altered very rarely, it is read by the program far more times than it is written to; thus we need some way of telling if a page has been accessed by the program and when, since all loaded pages will have been asked for by the program. This requires the addition of a reference marker, which contains the last time the page was accessed. This could be done by putting in the time — if the computer has a real-time clock, or simply by storing which turn of the adventure the page was looked at.

When we come to replace a page, we now only have to find the page which hasn't been accessed for the longest time; if two pages contain the same access date, we choose to replace the page which doesn't have to be saved first. The overhead of a reference and written-to

markers in extra storage is offset by the time which can be saved, and the reduction in page faults made possible by using them.

TEXT COMPRESSION

There are a number of text compression techniques, all of which are designed to reduce the amount of space required to store text, by coding it in some way. We will take a look at just two of the more popular methods in this chapter. The advantage of text compression is that it offers a method of reducing the size of your adventure data base, thus allowing you to develop a much larger adventure; or for the whole adventure to fit into the machine at once, rather than loading sections in from disc.

The major disadvantage of text compression is that it requires you to encode all your adventure data using a special encoding routine. This can be time-consuming and drastically reduces the development time if the adventure is so large you have to encode the data during the testing stages, and every time you wish to alter part of the adventure.

3:2 Byte Compression

This technique compresses three bytes worth of information into two bytes — hence its name! Before we can begin to understand how the technique works, let's look at how conventional text is stored on the Amstrad.

Each character which makes up text has its own, unique character code, this varies from machine to machine, but the Amstrad uses the most popular form of coding — ASCII (American Standard Code for Information Interchange). Each character is stored in the form of a one byte number, which contains the code for that character. Thus, in ASCII, "A" is represented by the number 65, "B" by 66 and so on. The ASCII codes run from Ø to 255 (the maximum number you can fit into 8 bits), and offers letters, numbers, special control codes and graphics. We really don't need all of these symbols in an adventure game — indeed we need very few characters.

Adventure game text very rarely uses much in the way of punctuation, and really only requires the upper and lower case letters, space and perhaps "," and "." for punctuation. This is a mere 55 characters, rather than the 255 which ASCII provides — and we would only require 6 bits to store the codes, rather than the normal 8 in a byte, as 6 bits can represent the numbers Ø-63. This would allow you to reduce the text to three-quarters of its original size. Ah, but wait a moment, do we really need to have code for both upper case and lower case letters? Why not simply have a code which means switch into upper case, and a second code to switch into lower case? This would mean we only needed 26 + 3 + 2 codes, or 31. This can be represented by just 5

bits, in the form of the numbers Ø-31, allowing us one spare code in case we need it.

To understand how storing a character in just five bits instead of eight can save us space, let's consider an example, where we wish to store the word GOLDEN.

Figure 5.1 shows how the ASCII representation of GOLDEN's six letters would look in terms of six bytes.

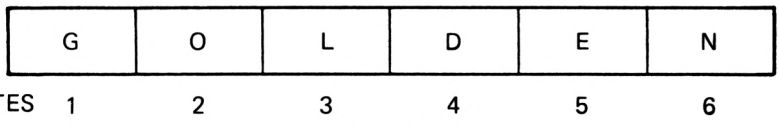


FIGURE 5.1

Now, if we assume that our 5-bit codes for our letters start at Ø to represent "A", the "G", "O", "L", "D", "E", "N" would be represented by the numbers 6, 14, 11, 3, 4, 13. The bit pattern for 6 can be stored in the first byte of storage, as shown in Figure 5.2.

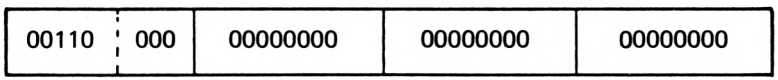


FIGURE 5.2

We can consider the bytes of storage to be continuous, so we next store the 5 bits which represent 14 in last three bits of the first byte, and the first 2 bits of the second byte. This is shown in Figure 5.3.

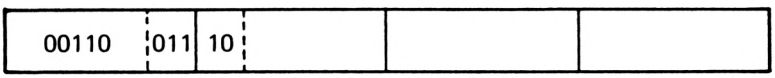


FIGURE 5.3

The bit pattern for the third letter, "L", is then stored in the next 5 bits available. This method of storage is then repeated for the last three letters of GOLDEN, and it produces the pattern of bits shown in Figure 5.4.

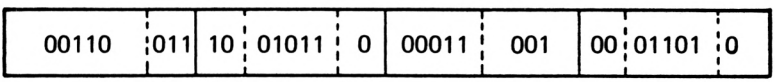


FIGURE 5.4

Notice that we do not use the last bit of the second byte, or the last bit of the 4th byte. If we did use this, it would cause problems when we came to decode the text, as it would radically alter the bits a letter could start at. In our example, a letter starts at bit 8 of the first byte, bit 2 of the first byte, and bit 5 of the second byte. This gives only three cases to deal with. Using the last bit of the second byte would mean

that a letter could start at any one of the bits in a byte, making decoding far more difficult.

Once your data is encoded using the 3:2 byte compression, you will obviously need a decoding routine to uncompress it! This simply has to step through the bit patterns, decoding the letters, translating them into ASCII, and then printing them out to the screen. This will have to be a machine code routine, as BASIC is far too slow at this type of thing, and doesn't provide the facilities to access individual bits like machine code does.

Huffman Coding

Huffman coding is a text compression technique which is based on the relative frequency of each letter. This means that if all the letters occur with the same frequency, there will be no space saved! This never occurs though, as the frequency of letters always varies in a piece of text — with "e" being fairly frequent and "z" fairly infrequent, for example.

The technique is to build up a dictionary tree, which can be used to decode the encoded text. The dictionary which allows you to decode the text is specific to one piece of text, and must be built up anew for each fresh set of text. This can be very slow, and time consuming, but the space savings which can be achieved by Huffman Coding more than make up for this disadvantage. Besides which, you would normally only compress your adventure text once — after each section has been completely tested.

The Huffman method represents different letters with a different number of bits, depending on their frequency. Thus, the most frequent letters will be represented by 1 or 2 bits, with the least frequent being represented by 8 or even more bits. Because the coding and decoding technique doesn't use a fixed number of bits, it must ignore byte boundaries, as the bits representing a letter can start in one byte and end in the next one.

The best way to explain the Huffman Coding technique is to give an example of its use. As the technique is quite long winded to do by hand (in practice, we would write a coding routine to scan through text and code it automatically), we will consider just one word, as an illustration of compression. In this case, we'll use minimum as the word we wish to code.

The first step is to find out which letters occur in the word, and the frequency of these letters. Using our example, we can build up the table below:

Letter	Frequency
i	2
m	3
n	1
u	1

Having done this, we search through the list of frequencies to find the two least frequent letters, and then pair these together, giving us:

i	2
m	3
n-u	2

We then repeat this pairing, with the next two lowest scoring letters, to give us:

i-(n-u)	4
m	3

And, finally, we pair up the last two frequencies, giving:

(m-(i-(n-u)))	7

Having built up this pairing of frequencies, we can now build up our dictionary tree, as shown in figure 5.5

FIGURE 5.5

When we wish to encode a character, we start at the top of the tree (known as the root — in computing, trees grow upside down!), and work our way down the tree until we reach the desired letter. For each time we take a left fork in the tree, we write down a Ø, and for every right fork, we write a 1. This means that the letters in minimum will be represented as follows:

m = Ø
i = 1Ø
n = 11Ø
u = 111

If you take the paths indicated by these numbers, you'll find that you end up at the letter they represent, and that is all there is to decoding a Huffman tree!

Now that we have calculated the dictionary tree for the word minimum, we can now code it into bits. The resultant code is shown in Figure 5.6.

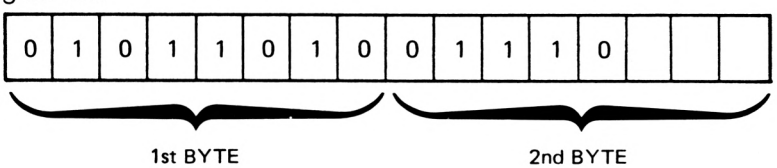


FIGURE 5.6

As can be seen from Figure 5.6, the representation for minimum is now a byte containing Ø1Ø11Ø1Ø and 5 bits containing Ø111Ø. Thus, the word now occupies just over one and half bytes. Using standard ASCII representation, it would normally occupy seven bytes (one for each letter). That is a reduction to just 23% of its original size — quite a saving! Obviously, the Huffman Coding technique works especially well with a word like minimum, due to the large number of repeated letters — the reason we chose it. The reductions in size for a normal piece of text won't be quite so spectacular, but should be in the region of 5Ø-6Ø% of original size at the worst.

One way to compress text even further, using Huffman Coding, is to use not just the frequency of letters, but also common phrases. Thus, you can represent phrases such as "A LARGE CAVERN", which may be quite common using just a few bits. This technique is used by a lot of the commercial software houses, as it enables adventures with several hundred locations to be fitted into most of the home computers currently available.

SECTION 2

THE ADVENTURE KERNEL SYSTEM

6
WHAT IS AKS?

This section of the book describes the use of AKS — the Adventure Kernal System. This has been designed as a data driven adventure games system which enables you to change the adventure scenario data without needing any knowledge of programming and without having to modify the main driving routines. Yet, it has been written in BASIC so that you can understand and modify the data simply and easily, if you so desire.

What can you use AKS for, and why would you want to use it? What are the advantages over writing your own adventures? Well, we'll try and give you an idea of the wide range of possibilities available with AKS, and the uses to which it can be put.

The first and obvious use for the AKS system is to create and play your own adventure games. This section outlines the commands available in AKS and gives examples of each of them. You will find that very complex games can be created using the techniques and commands shown and that they can be developed quickly and easily, as there is no actual programming work involved. The advantage of not writing your own adventure from scratch is that we have done all the hard work for your already! There is little point in duplicating programs when you can use the time creatively to design adventure games!

The fact that AKS is based on a data-driven structure means that new adventures can be constructed simply by changing the actual scenario data, without having to modify the program, or rewrite any part of it. The approach most adventure game books have advanced in the past is that of specific coding — each action and event in the game is tied to a specific piece of code which deals with that one part of the adventure alone. This is extremely inefficient, and a criminal waste of time and effort as the basis of most adventure games is exactly the same, although with different data components. The specific coding approach requires you to rewrite the whole adventure game from

scratch each time you create a new scenario; the data-driven approach simply means coding your scenario data each time.

Apart from designing your own adventure games without resorting to programming, the AKS system can teach you a lot of useful techniques for adventure games writing, and careful study of the program will prove extremely helpful. After you have mastered designing adventure games using AKS, you can then move on to expanding the system, using your own programming skills. We will look at the expansion possibilities of AKS in a later chapter.

We have included an example AKS scenario WITCHHUNT in the book, and you might wish to buy the cassette tape version of AKS and WITCHHUNT before you read through the relevant chapters on the scenario, so that you can enjoy the game, and then study the plot details after you have finished playing it.

DATA REPRESENTATION IN AKS

There are basically two approaches to representing the actual scenario data for an adventure game. The first of these is to store the data as straight forward BASIC DATA statements as any ordinary program data might be stored; or to store the data in arrays, which are generated by a special data preparation program and loaded into the program from tape or disk each time it is run. There are advantages and disadvantages to both of these methods, and the method we have chosen to use with the Adventure Kernel System is the first approach. Let's look at the two methods and try to explain why we found our chosen style of game the most suitable for AKS.

Advantages of readability over coding

With all the scenario information stored directly in the program itself, all the data is immediately visible to the scenario designer, you can easily list the locations or objects you are interested in, without having to print out array values or load in separate programs. This advantage will be immediately evident to someone who has tried to design any adventure game using either commercial adventure creators, or other programming methods. The adventure game data can be seen as a whole using DATA statements, something which is not possible otherwise; and if you want to check just what object 16 is, there is nothing more annoying than having to go through several menus, or printing out an array.

Coding the scenario within the program does mean that data cannot be easily edited or altered. Adding a location can mean having to change a large chunk of the program as well as just the locations array. Data statements mean the data can be easily added to and corrected using nothing more than the Amstrad BASIC editor, which allows you

to simply alter lines of scenario data in the same way you would alter any program line. This also avoids the necessity of having to write a special editor for the data, or having to learn to use one!

Not only can the data be entered and corrected more easily using this method of representation, but the whole process of debugging and developing the adventure game is speeded up. If the data is stored separately from the main adventure program and has to be loaded in to the machine each time, it is going to slow development work down enormously. Each time you edit and correct an invalid part of the adventure game, you will have to load in the data generator, load the data, edit the data, save the data, load the adventure driver, and finally load the edited data! With the AKS method you simply have to load the adventure program and your data is already there. Simply edit any data which is wrong and re-run the driver. The only time the program has to be loaded or saved is at the start and end of the design session. Clearly, the use of a separate data generator on a tape based machine is totally impractical, as most of the time will be spent swapping tapes!

The other major advantage of using BASIC and a very readable form of data is that the BASIC interpreter is already resident in the machine, and avoids the necessity of loading in yet another program for each session. This is a major failing with adventure design systems where you have to load the programming "language" and the database every time you wish to use them.

Disadvantages of readable data compared with coded data

Despite all the advantages listed above and the fact that we have chosen this method for the AKS system, it is not totally perfect and can cause some serious problems.

One problem on a lot of machines is that the data is in no way compacted. Using text compression techniques can reduce the size of the data by a significant amount. The problem is balancing the need for space against the need for readability and ease of development. Compressing text can take a lot of time, and you don't want to have to do that each time you test a new part of the adventure! On a machine with limited memory — say only 16 or 32K, the need for space is extremely pressing if the game is going to have more than just a few locations, or a reasonable number of locations and little descriptive text. In this case, text compression would probably be advisable, though it may be possible to test at least part of the adventure game before compressing the text for the final version. On a 64K machine such as the Amstrad, this is not such a problem and you are unlikely to find space a major concern while using AKS, unless you are trying to write a truly enormous adventure!

The use of large amounts of uncompressed text also results in a large program, which will slow the game down. The more data the

program has to scan through, the slower it will become and the use of DATA and RESTORE can slow the program by a significant amount.

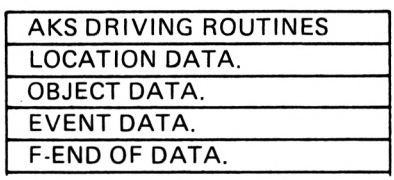
Adventure games are meant to be a challenge and they should keep the player occupied for at least several hours, if not days or months. However, players can often be driven to extremes by a puzzle they cannot solve and thus they may be tempted to "cheat". This is not possible on most games where the text messages and responses for the game are coded — in order to find the solution to a puzzle you will have to write a decoding routine of your own; and this can be a major puzzle in itself! However, with the AKS approach the whole idea is to have adventure puzzles, solutions, etc as visible and as easy to find as possible, so that the writer can easily debug and finish the game. This will allow any player to cheat, simply by listing the program and finding the pertinent DATA statement. On most machines this is a problem, but there is a solution on the Amstrad. Develop your game using the normal mode of files and tape handling. Once you are sure that the program is fully debugged, tested and finished, save it using the PROTECT option. This will allow other people to load in your adventure and play it, but they won't be able to break into the listing to find the solutions to problems. Keep an unprotected copy for yourself though, just in case there are any bugs left undetected!

Considering the overall advantages and disadvantages, it can be seen that the use of easily read, easily edited DATA statements is the better approach for the Amstrad AKS system. The problems are far outweighed by the advantages.

THE LAYOUT OF AKS

AKS is designed rather like a programming environment, in that each part of the program and data has a set place in memory. The diagram in FIG 6.1 is a memory map of the AKS system, and shows you the order of data.

FIGURE 6.1



MEMORY MAP OF AKS

Thus, all the data is located below the AKS system, and it should be presented in the order of Locations, Objects and then Events. The end of the data section is marked simply by an F. Within each section, the data is ordered by location, object or event number in numerical, increasing order.

7
ACTIONS IN AKS

The AKS actions are the parts of the vocabulary which cause an action to take place, there are nineteen separate commands, each of which is represented by a two letter abbreviation. A command line starts with an A, to indicate an action. The actions are only performed:

a) as the result of a trigger
or b) as the result of an event firing.

We'll take a look at each of the Actions in turn, together with an explanation of the effect of using each one. Later chapters will show how these commands are actually incorporated within an AKS database, as well as expressions and triggers.

Assign Flag (AF)

AKS provides a number of flags, which may be set to TRUE or FALSE by the user, and then tested in later operations. The assignflag command will set the specified flag to the specified state, and takes the form of:

AF,<fnum>,<flagstatus>
e.g. AF,11,T

where <fnum> is the specified flag and flagstatus is T or F.
NOTE: If flagstatus has a value other than T, then the flag will be set to FALSE, without an error being reported.

DRop (DR)

This will result in the object mentioned in the player's command line being dropped, providing it is being carried, and that it can be dropped. Otherwise, a suitable error message is produced.

EXamine (EX)

The object in the player's command line is tested, and if it is present and may be examined, the description is then printed.

GEt (GE)

As with the drop command, the object to be taken is checked for suitability, and to see if it is present. If all the conditions are met, the object is added to his inventory.

GO (GO)

This command causes the player to move in the direction given, moving him to a new location if a connection exists from the player's current location; otherwise printing "You cannot go that way". The command takes the form:

GO,<direction>
e.g. GO,N

where <direction> is usually one of N, S, E, W, U or D — indicating north, south, east, west, up or down respectively. The direction may be any word, providing it has been used in both the action and location definition.

HaltCounter (HC)

The AKS system provides a number of counters, which are decremented each move. The HC command will stop the specified counter from counting. The command's form is:

HC, <cnum>
e.g. HC, 11

IncScore (IS)

A scoring facility is provided by AKS and the IS command will add the score increment to the score. The command format is:

IS, <int>
e.g. IS, 25

The increment can be positive or negative.

InitialiseCounter (IC)

This will initialise the counter indicated to <int>, and starts the countdown of moves. When Ø is reached, the event <cnum> will fire. The command format is:

IC, <cnum>, <int>
e.g. IC, 11, 1Ø

INventory (IN)

Displays the objects which the player is currently carrying or wearing.

LOad (LO)

This reloads a SAved game from tape.

MoveObject (MO)

This will move the object specified to the location number specified. It takes the format of:

MO, <obj>, <loc>
e.g. MO, 12, 24

PutOn (PO)

This changes the status of the object referred to in the command line to show that it is now being worn by the player.

PRint (PR)

Prints the following string onto the screen. The print command takes the form of:

PR <string>
e.g. PR, you cannot eat that!

QUit (QU)

This will quit the game (note: without requesting a confirmation from the player), and then prompts "Another Game?".

SAve (SA)

This saves all the variables associated with the game to tape, so that the game can be reloaded later on.

SCore (SC)

Prints the player's current score on the screen.

TakeOff (TO)

The opposite of PutOn, fairly naturally, this changes the status of the object referred to in the command line from being worn, to being carried by the player.

ZapIn (ZI)

This takes an object from whatever location it is currently at, and brings it to the player's current location. The command form is:

ZI, <obj>
e.g. ZI, 11

ZapOut (ZO)

This takes an object and changes its location to "nowhere", effectively destroying it. It takes the form of:

ZO, <obj>

e.g. ZO, 11

RANGES

The parameters to the above actions have maximum and minimum values, which cannot be exceeded; an Amstrad Basic error "Subscript out of range" will occur if you do exceed the values. These are defined as follows:

<int> = BASIC integer (signed).
<fnum> = Ø..maxflag.
<string> = BASIC string. NB must be surrounded by quotes if the string contains commas.
<obj> = Ø..noofobjs (where Ø is the player).
<loc> = -1..nooflocs (Ø is the player, -1 is nowhere)
<cnum> = Ø..maxcount.

The upper limits to some of these ranges are set within the program, by the constants at the start of AKS. The initial (and arbitrary) values are:

maxflag = 3Ø
maxcount = 5
maxobj = 2Ø
maxloc = 3Ø

Their values can easily be changed using the normal BASIC editor.

8
TRIGGERS IN AKS

Actions in AKS can be activated in two ways, either directly from the player input, or indirectly inside the program. These are known as triggers and events respectively, and we will look at the former in this chapter.

The format of a trigger command is:

```
T, word list, condition
 A, action
```

The word list is simply a list of words, which will cause this trigger to become active if they appear in the player's input. These are followed by a condition, which governs the condition under which the actions may be performed. The trigger line is then followed by a list of actions which will be performed.

While triggers are easy to implement, it is important to pick the correct words for the word list, to ensure that you have covered all the possible player input, which should cause the actions to trigger. For example, if the player has to give an object to another character in the game, he might use "GIVE", but you should also include words such as "SHOW", "HAND", etc. Only using one direct phrase such as "GIVE THE COINS", results in a very limited vocabulary and causes great frustration to the player. You want players to enjoy your game — not tear their hair out while playing it!

There are two forms of trigger in AKS, local and global triggers. The first of these, as their name suggests, are triggers which can only be activated when the player is at the current location, and which are ignored at all other locations. The second type, global triggers, are active all the time, and can be triggered by player input from any location. The AKS system examines the local triggers before the global triggers are searched, which enables you to alter the behaviour of commands within different locations. Let's look at some examples to

see what we mean by this, and how to use triggers in your own adventures.

Firstly, the global triggers. These are entered in the database as part of location Ø. This is a special location, which does not appear in the actual adventure game — the player can never visit location Ø. It is here that we can declare all the synonyms for our global vocabulary. For example, we might want the player to be able to perform a "GET" action, enabling them to pick up objects. As well as "GET", it would be convenient to allow the player to type in "TAKE", "PICK UP", and so on. This is done by declaring a global trigger as follows:

```
T, get, take, pick up,*
  A, GE
```

So, if the player inputs any of the words in the trigger line, then the system performs a GET action. We can do the same for all the usual commands as well — if you look at the listing for WITCH HUNT, you will see that most of the global commands you'd ever want to use have been declared.

The local triggers are very similar to this, except that they are declared within the location description they apply to. They are declared in exactly the same way, so a declaration of:

```
T, feed ducks, feed duck,*
  A, PR, the ducks gobble up all the bread and leave.
  A, ZO, 5
```

would only trigger if it was at the player's current location. If this was true, and the player typed in "FEED DUCKS", then the system would print the appropriate message and then remove the ducks from that location (the ducks being object 5).

As well as declaring actions which manipulate objects, we can change the possible directions the player can move. If the current location has connections east and west, with a small building to the east, the player can move east or west by using the usual movement commands. However, most adventure players will expect to be able to go east to the building by typing "IN", not just "EAST". This can be allowed by declaring a trigger such as:

```
T, in, enter,*
  A, GO, E
```

which will move the player east when they type "IN" or "ENTER".

We can also alter the actions a command word had in each of the locations. For example, suppose we have declared this global trigger:

```
T, rub amulet, stroke amulet, *C5
  A, PR, nothing happens.
```

If the player rubs the amulet (object 5) the system will simply inform him that "NOTHING HAPPENS". In one location, however, we want rubbing the amulet to actually cause a very different action. This can be implemented by declaring a similar trigger in the desired location. This might take the form of:

```
T, rub amulet, stroke amulet,*C5
A, PR, The portcullis rises up and out of sight!
A, AF, 1Ø, T
```

This means that rubbing the amulet causes a message to be printed and the flag 1Ø to be set to TRUE. This is possible because, as noted before, AKS scans the current location for triggers before scanning the global triggers. When a trigger is found, it is activated, and the trigger search is terminated. This ensures that only one trigger will be activated for each of the player's inputs.

9
LOCATIONS, OBJECTS AND EVENTS IN AKS

An AKS scenario is based around the structures of Locations, Objects and Events, as these govern which actions take place where, which objects can be manipulated and so on. In this chapter we will take a look at how the Actions and Triggers we have already considered fit into the Location and Object structures.

The ordering of different data types is important in AKS, as the memory map in chapter 6 shows, and thus we will deal with these differing data types in the order they will appear in an AKS scenario.

LOCATIONS

Locations must be declared in an AKS database in ascending order of location number. Location zero is the special global location, as we have already mentioned in the previous chapter while looking at Triggers. Each location has a very similar header, which might look like this:

```
L, 15
  D, *, In the dark woods.
  C, N, *, 16
  C, S, *, 18
```

This declares location 15, with a description of "In the dark woods", which has no condition attached. The location is connected to location 16 to the north, and location 18 to the S. There are no conditions attached to the player moving to either of these two locations. There

can be as many descriptions (each with an associated condition) as you wish, which enables us to simulate darkness, or to abbreviate the description if the player has already visited this location. We simply include these facts in the relevant condition; see the next chapter for a detailed description of the conditions available in AKS, and their use.

The above location declaration is very basic, as it does not include any triggers, these are simply added in before or after the connection information. An example location might be:

```
L, 15
  D, *, In the dark woods.
  C, N, *, 16
  C, S, *, 18
  T, climb tree, go tree, *
    A, PR, you cannot climb trees.
  T, plant acorn, plant, bury, * C7
    A, PR, the planted acorn grows into an oak tree
    A, ZO, 7
    A, ZI, 8
```

This adds triggers which will detect attempts to climb trees, and plant acorns!

OBJECTS

Objects must be declared in numerical order, any out of sequence objects being reported by the AKS system when you attempt to execute the adventure. In the same way that location Ø is special, so is object Ø, as it represents the player! This does mean that object Ø can be manipulated in the same way as other objects — so you can move the player around, but, the declaration for the player will normally only include an initial starting location and nothing else.

There are a number of elements to an object declaration, and we will look at each of these in turn. The best method is to consider the following example, and we can then look at what each element stands for:

```
O, 4
  D, *, A small rusty lamp.
  P, 19
  N, rusty lamp, torch, *
  S, GE, *
  S, DR, *
  S, EX, *
```

This declares object number 4. The first line of the declaration gives the object a description, which will be printed out with no conditions

attached. The "P, 19" says that the lamp's initial position at the start of the adventure will be location 19. We need to let the player refer to the lamp by different names in his command input, so we declare a set of names using the "N" command, which allows you to attach a condition to the player using that name.

Finally, the "S" construct lets us set the suitability of the object to be used with certain of the inbuilt AKS actions. In our example above, our lamp is suitable for GEtting, DRopping and EXamining; there is no limit to the actions which can be performed on an object (except commonsense of course!).

EVENTS

An event is acted upon whenever the counter associated with it reaches a value of zero, after it has been activated by another part of the database. This is where the Initialise Counter action comes in, as we can set the counter to a number of turns, after which the event associated with that counter will come into action.

An example event might be declared as follows:

```
E, O
  A, PR, You have run out of time . . .
  A, SC
  A, QU
```

This declares event Ø (which is not special in the way location and object Ø are) and lists a number of actions to be performed. In the case of our example, when the event is triggered, a message "You have run out of time . . . " is printed, the player's score displayed and then the game ended. This could be used if the player had a maximum number of moves to complete some action, or even the whole game, in. If he fails to do so, event Ø comes into action.

10
EXPRESSIONS IN AKS

At certain points in AKS you may wish to attach a condition to something. If the condition is true then some operation is performed otherwise it is not. When writing an adventure game scenario in Basic a condition can be expressed as an "IF condition THEN perform operation" statement. AKS aims to do away with the need to write in Basic in order to specify your scenario and so cannot use this technique. However, the need for some form of conditional testing when specifying an adventure game cannot be overlooked. How could the puzzle with the big green snake barring the way of the player be implemented if the scenario were unable to test if this puzzle had been solved. Obviously, AKS must have conditional testing. A glance at the scenario definition for WITCHHUNT will reveal the presence of a large number of asterisks at various positions in the definition statements. Each of these may be followed by a string of characters terminated by the end of the line or by a comma. The absence of this string represents the absence of any condition — i.e. the associated operation is unconditional.

As in Basic, an AKS condition may have the value of true (T) or false (F). If the condition is true the operation is performed. An IF statement which only allowed you to say "IF T THEN . . . " or "IF F THEN . . . " would be useless. To be of use the condition must be allowed to be something which is evaluated to either T or F. At the simplest level this could be a flag variable. For example, a flag variable could be set aside to indicate whether or not the snake puzzle has been solved. AKS has flags called FØ, F1, F2 . . . and so on. Supposing we have allocated F1Ø to represent the snake puzzle then we can write the condition as *F1Ø. AKS initialises all flags to F at the start of a game. When the player solves the puzzle the scenario assigns the value T

to F1Ø. The AKS coding for the snake puzzle could be implemented by defining the appropriate location (18) as follows :

```
DATA L,18
DATA   D, *, You are in the Hall of the Mountain King.
DATA   D, * F1Ø, A large green snake bars your way ahead!
DATA   T, release bird, *
DATA     A, PR, The bird drives the snake away.
DATA     A, AF, 1Ø, T
```

When the player first arrives at this location he will be greeted by the following description:

```
You are in the Hall of the Mountain King.
A large green snake bars your way ahead!
```

The player can now drive the snake away by entering the command "RELEASE BIRD" to which AKS will respond:

```
The bird drives the snake away.
```

The AssignFlag (AF) command then sets F1Ø to T indicating that the puzzle has been solved. Now the description of the location will appear as just:

```
You are in the Hall of the Mountain King.
```

This is fine but there is nothing there which states that the player must be carrying the bird in order to release it. Another test is required to replace the unconditional indicator at the end of the trigger line. One possibility would be to set a flag to T when the player catches the bird and test this as we did for the snake puzzle. However, you then have to remember to reset the flag to F if the player drops the bird. A neater solution is to have a function which tests to see if the object is being carried by the player and becomes T or F accordingly. This type of function is called a predicate and can be tested in a similar way to a flag. So that CØ, C1, C2, . . . indicate whether objects Ø, 1, 2, . . . are being carried. Therefore, if the bird is object number 3, the snake puzzle coding can be updated to:

```
DATA L, 18.
DATA   D, *, You are in the Hall of the Mountain King.
DATA   D, * F1Ø, A large green snake bars your way ahead!
DATA   T, release bird, * C3
DATA     A, PR, The bird drives the snake away.
DATA     A, AF,1Ø, T
```

AKS supports six different types of predicates and flags for use in expressions. With the exception of the flag type discussed earlier, the

AKS program maintains the necessary information to return a value of T or F for each of these tests. The flag variables FØ, F1, F2 . . . are only altered by actions in the scenario definition and their meaning is decided by the scenario designer. The AKS predicates and flags are listed below:

Cx ..... Carrying object x
Fx ..... Flag x
Lx ..... at Location x
Ox ..... Object x at current location
Wx ..... Wearing object x
Vx ..... Visited location x

Although these flags and predicates allow a number of tests to be performed it is often useful to be able to invert the result of a test. The operator "NOT" allows you to do this in Basic. The "NOT" of true is false and vice versa. AKS uses a minus sign to perform the same operation. For example, -C3 can be used to test for the bird not being carried and to print an appropriate message if you try to release it:

```
DATA L, 18
DATA  D, *, You are in the Hall of the Mountain King.
DATA  D, *F1Ø, A large green snake bars your way ahead!
DATA  T, release bird, *C3
DATA   A, PR, The bird drives the snake away.
DATA   A, AF, 1Ø, T
DATA  T, release bird, *-C3
DATA   A, PR, Good idea, but you don't have it.
```

In addition to the "NOT" operator, Basic conditional expressions allow the use of "AND" and "OR" to construct complex tests. AKS uses the symbols "." for "AND" and "/" for "OR". So far there is nothing in the example scenario location to stop the player passing the snake even though he has been told his way is barred. This requires that all the connections from this location except up (U) which is the way back out of the location, only be opened when F1Ø is T:

```
DATA L, 18
DATA D, *, You are in the Hall of the Mountain King.
DATA  D, *F1Ø, A large green snake bars your way ahead!
DATA  T, release bird, *C3
DATA   A, PR, The bird drives the snake away.
DATA   A, AF, 1Ø, T
DATA  T, release bird, *-C3
DATA   A, PR, Good idea, but you don't have it.
DATA   C, N, *F10, 17
DATA   C, E, *F10, 12
```

```
DATA    C,W, *F1Ø, 25
DATA    C, U, *, 5
```

This has defined the connections north, east and west to locations 17, 12 and 25 to be open only when the snake has been driven off. Now, let us introduce another problem the player is faced with at this location. Once the snake is disposed of, the player is free to explore the connected locations and is able to find three treasures. However, certain of these treasures are too large or heavy to carry up the stairs from the Hall of the Mountain King together. If the player is carrying the gold bar (object 4) and the axe (object 5) or alternatively, the gold bar and the jewels (object 6) and wearing or carrying the armour (object 7) then he may not go up. This can be expressed by:

```
DATA L, 18
DATA   D, *, You are in the Hall of the Mountain King.
DATA   D, *F1Ø, A large green snake bars your way ahead!
DATA   T, release bird, *C3
DATA     A, PR, The bird drives the snake away.
DATA     A, AF, 1Ø, T
DATA   T, release bird, *-C3
DATA     A, PR, Good idea, but you don't have it.
DATA   C, N, *F1Ø, 17
DATA   C, E, *F1Ø, 12
DATA   C,W, *F1Ø, 25
DATA   C, U, *-((C4.C5)/(C4.C6.W7/C7)), 5
```

AKS evaluates conditional expressions starting with the highest priority operators first unless brackets specify otherwise, as does Basic. The priority of operators in descending order are - (not), / (or), . (and). So in the above example the order of evaluation is:

```
<a> (C4. C5)
<b> W7/C7      "/" is higher than "."
<c> C4. C6
<d> result of <c> . result of <b>
<e> result of <a> / result of <d>
<f>  - result of <e>
```

SECTION THREE

IMPLEMENTING AKS ON THE AMSTRAD

11
PROGRAMMING TECHNIQUE

AKS has been written in a highly structured format with an emphasis on readability and robustness of the Basic code. The techniques adopted in the programming of AKS to achieve this structured format are described below.

1. PROCEDURAL APPROACH

The problem tackled by AKS has been divided into several sub-problems and each of these divided into sub-problems and so on until each sub-problem becomes trivial. These trivial sub-problems are then coded as Basic Subroutines. It is important that the interaction between these subroutines is tightly controlled and made clear in the listing. To this end, the assignment of a value to a variable which is to be used in another subroutine is made on the same line as the 'GOSUB' statement (e.g. 'loc =Ø:GOSUB 123Ø:REM *isobjatloc*'). If a subroutine returns a value in a variable then where possible any testing of that value, or any use of that value is performed immediately after returning from that routine.

2. SENSIBLE VARIABLE USAGE

The most obvious point here is the use of meaningful variable names to increase readability. Amstrad Basic allows multi-character variable names for this reason. In addition to enhancing readability, the use of long variable names reduces the chance of conflicting use of a variables (i.e. attempting to use the same variable for different things at the same time). All variable names are in lower case to make them stand out from the upper case Basic keywords.

AKS initialises several constants at the start of the program (e.g. the number of scenario locations is set by 'rnaxloc =3Ø'). As their name suggests these 'variables' are not changed anywhere else in the program, even though it is strictly possible to do so. This has two effects. Firstly, it increases readability and secondly, it makes it easy to modify the program because the value need only be changed in one place.

3. COMMENTING

Comments are used in an orderly and consistent way. Lines containing just a colon are used extensively to space the program out and divide it into logical blocks. In the main part of the program a logical block is a group of instructions with a similar function. For the rest of the program, a logical block is a subroutine. In addition, every subroutine is headed by two 'REM' statements. The first of these has the name adopted throughout AKS for that routine bracketed by three asterisks (e.g. 'REM *** resetflags ***'). The second is just a blank 'REM' statement to highlight the routine title.

Any subroutine call is followed by a comment containing the name of the routine bracketed by single asterisks (e.g. 'GOSUB 5ØØØ:REM *resetflags*'). To maintain this standard some 'IF condition THEN gosubsomewhere ELSE gosubsomewhereelse' statements have been split into two statements (i.e. 'IF condition THEN gosubsomewhere' and 'IF NOT (condition) THEN gosubsomewhereelse') to allow the 'GOSUB's to be followed by a comment.

Although the game is written to run in the 4Ø column MODE 4, the commenting has been written for viewing in the 8Ø column MODE 2. Where possible, comments about the operation of program lines are on the same line and use the single quote character instead of ':REM . . .' to start the comment. This helps to distinguish normal comments from subroutine labels on 'GOSUB's. Should the comment be too long to fit on the same screen line or if it is syntactically invalid to have a comment on the same line as that type of statement then the comment is placed, where possible, on the line preceding the statement. In this case a normal 'REM' is used instead of a single quote to make the comment stand out from the Basic statements. Unless emphasising something, comments are in lower case to make the upper case Amstrad Basic keywords easy to see.

4. CAREFUL CONTROL OF FLOW

Something which explicitly controls the flow of a program is potentially very dangerous. Apart from having the potential to make a program totally unreadable, incorrect use of flow controlling instructions can result in corruption of the Basic stacks. The most obvious control flow

instruction is 'GOTO'. Much has been written about the use and misuse of 'GOTO', but few will deny that it is difficult to program in Basic without it. Amstrad Basic goes some way towards removing the need to use 'GOTO' by allowing the use of 'WHILE..WEND' loops as well as the normal 'FOR..NEXT' loops. AKS makes much use of the 'WHILE..WEND' construction. Another common use of 'GOTO' is to implement multiline 'IF..THEN..ELSE..' statements. AKS avoids using 'GOTO' for this by the use of multistatement lines instead. However, AKS does make use of 'GOTO' as a quit instruction to skip over the rest of a 'WHILE' loop's instructions and go directly to the 'WEND'. This technique of jumping to 'WEND' is used to terminate the loop without corrupting the Basic stack. When AKS does this, a comment is used to identify the 'GOTO' as a quit instruction.

Another frequently misused control flow instruction is 'GOSUB'. To make a subroutine easily understandable, a subroutine should have only one entry point. In the AKS listing the entry point to each subroutine is the line immediately following the title 'REM'. It is a good idea not to 'GOSUB' or 'GOTO' a 'REM' statement, as these are often removed from the runtime version of the program; thereby causing a 'line does not exist' error. The last instruction in every AKS subroutine is a 'RETURN'. The practice of using 'GOTO' to jump in and out of subroutines is avoided. However, it is often desirable to skip the remaining instructions in a subroutine. This could be done using 'GOTO' to jump to the 'RETURN' instruction. A cleaner solution is to use another 'RETURN' instruction instead of the 'GOTO'. Even here, AKS comments the premature subroutine exits as quits.

DIRTY TRICKS

Examination of the first few lines of the AKS listing will reveal that everything is not perfect. To implement AKS in Amstrad Basic a significant problem must be overcome. AKS works by interpreting a scenario database specified in 'DATA' statements at the end of the program. Each statement occupies one 'DATA' line and AKS must be able to find the start of any given statement. Unfortunately, Amstrad Basic does not allow use of the 'RESTORE' statement with a variable line number (i.e. 'RESTORE fred' is illegal). Instead, it insists on a literal line number (e.g. 'RESTORE 5ØØ'). Totally legitimate use of 'RESTORE' in AKS would require a ridiculously time consuming search from the first 'DATA' statement, reading a string (not a line) at a time, until the desired line is found. Therefore, AKS resorts to a 'dirty technique' which 'POKE's the value of a variable into the space occupied by the literal line number part of a 'RESTORE' statement IN THE BASIC PROGRAM. This explains the existence of the first few lines of the program. For example, to 'RESTORE' to line 37Ø, AKS would perform 'lin =37Ø:GOSUB

2Ø:REM *restorelin*'. This fudge routine is placed at the front of the program along with comments giving a warning of the dangers inherent in this technique. Obviously, changing this routine will result in the wrong part of the program being altered and may corrupt the code. Things are further complicated by the fact that the 'RESTORE' statement's line number must be set to a non-existent line number before attempting to 'RENUM'ber the program. The small 'ENTER' key is programmed to do this. AKS requires the 'DATA' statements in the scenario to be numbered in increments of the constant 'lineinc'. The normal value of 'lineinc' will be 1Ø — which is the default line number increment used by Basic 'RENUM' and 'AUTO'.

In addition to knowing the line number increment, AKS must know the line number of the first 'DATA' statement of the scenario definition. It is unreasonable to expect a programmer working on AKS to find this out and set a variable to the value everytime he adds a new line of Basic. For this reason, a second 'dirty technique' is used. To find this value, AKS jumps to a purposely included invalid program line that immediately precedes the first 'DATA' statement of the scenario definition. AKS traps the error generated using 'ON ERROR..' and assigns 'datastart' the line number of the line after the invalid line (i.e. 'ERL + lineinc'). Although not as dangerous as the first 'dirty technique', this technique is also heavily commented.

12
STRUCTURAL OVERVIEW OF AKS

The overall structure of the AKS driving routines can be broken down into six different sections, which we will consider and briefly outline in this chapter. In order to understand how the routines work in detail, you are advised to study the full listing of AKS in Appendix A, as it is fully commented, with meaningful variable and routine names. To detail the program within this section, in the same way, would simply duplicate the information in the Appendix. This chapter is merely considering the structural elements of AKS, and the way these elements relate to the overall program.

INITIALISATION

The first stage of the program requires it to initialise all the variables it is going to use. This is done by the three routines Initlocations, Initobjects, and Initevents. As their names suggest, they initialise all the variables associated with the locations, the objects, and the events respectively. We'll look at each of these separately.

Initlocations

This routine starts at the beginning of the data, with location Ø, and it searches through the data, until it encounters a data declaration for an object, event or the F end marker. While it is scanning through these data declarations, the location numbers declared in the "L, <locnum>" command are noted, and any missing or out of order locations are reported. The array element "locline(loc)" is set to the program line at which the data for the location "loc" begins. This process of scanning

continues until a non-location declaration is found, at which point "nooflocs" is set to the number of locations found, and the routine returns to the main initialisation.

Initobjects

This routine works in a similar way to Initlocations, except that it deals with Objects, and it scans until it finds a declaration for an Event or the "F" end marker. During this scanning process, the array element "objline(obj)" is set to the data line which begins the declaration for the object "obj", in the same way as "locline(loc)" is used. Any out of order, or missing object numbers are reported by the system, and the scanning is stopped.

This initial scanning process also allows us to find the start locations for each of the objects and to set their position in the "objloc(obj)" array where all the object positions are stored. When the scanning search for objects has finished, the "noofobjs" can be set, and the routine returns to the main initialisation section.

Initevents

This initialisation routine deals with any events which may have been declared at the end of the data block, and sets the array "eventlin(cnt)" to mark the data line at which each event begins. The search for events finishing when the "F" end marker is encountered.

After initialising the locations, objects and events, all the program flags are initialised by Resetflags, which simply sets all of them to the value FALSE. This then completes all the initialisation the program has to do.

MAIN PROGRAM LOOP

This is a very simple WHILE loop, which runs continually until the flag "eogame" becomes TRUE, whereupon the game ends. This flag is obviously only set by the player losing, winning or quitting the game.

Thus, the loop consists of five different operations:

(a) Describing the current location.
(b) Noting that the current location has been visited.
(c) Inputting a command from the player.
(d) Processing this command line.
(e) Updating any countdowns which are currently active.

The second action is simply a case of setting a flag corresponding to the current location in the visited array to TRUE, which is very straightforward. The other routines are a little more complex, so we will consider these separately.

DESCRIBELOC

This is the routine which prints out the appropriate description for the current location, in a neat and formatted form. The first step is to find the data line at which the player's current location is defined, and then to call DescribeIn, to print out the current description. We then test to see if any objects are at the current location. If there are objects here, we search through all the objects, printing out the object descriptions for all objects at this location. This done, we return to the main loop.

DescribeIn

This routine searches through the data statements for the current location, looking for description data (beginning with a "D"), until it reaches the start of another location, object or event. Once it finds a line of description, the condition attached to that description has to be evaluated, and this is done with a call to the expression evaluator (which is covered in great detail in the next chapter). If the condition evaluates to TRUE, we can then print the description by calling Printdescr, and carry on searching for the next one.

Printdescr

This is the routine which performs the actual act of printing the location description to the screen. This is not a simple matter of printing out the text in a straightforward fashion. If we did this, some words would overlap the edge of the screen, being split across two lines. This would not only be difficult for the player to read, but it would give our adventure games a very untidy, messy appearance. A lot of the appeal of the adventure game is the way in which it is presented to the player. A slapdash, untidy presentation only a discourages the player from bothering with the game.

The printdescr routine gets around this problem by making sure that words are not split over two lines, and that punctuation is not put in at the beginning of a line. This is done by taking the description string a screen width's worth at a time, e.g. if our screen width is 8Ø characters, and we have already printed the first 1Ø characters on the current screenline, we consider the next 8Ø characters of the description. This is then checked to see if the end of the string occurs in the middle of a word, if it does, we search backwards through the string to find the end of the previous word. The string can now be printed up to that point. We then consider the next 8Ø characters from this point and so on, until the string left is less than (or equal) to the screen width when we simply print it, and return from the routine.

This printing routine is very general-purpose and not just specific to the AKS program, so you could easily use it to present neat, word-wrapped output in your own programs.

GETCOMLINE

This is a very short and simple routine, which prompts the player with the question "What now?". The player's command line is then input into the variable "in$". Obviously, we cannot provide for all the possible combinations of upper and lowercase which the player might type in, so we use LOWER$ to convert the input string to be totally lowercase. This does mean that all the commands, and object name which you include in the object and location declarations must all be in lowercase also. But, this is small price to pay in return for faster processing of the command line.

PROCESSCOMLINE

This is the most important routine in the program, in many ways, as it checks the player input against the database and causes actions to be performed if matches have been found. The first stage is to search for triggers in current location, and then in the global location; matching the input string against the trigger phrases. It is worthwhile noting at this point that it is important to order trigger phrases in the correct order — substrings after the main string. By this, we mean that "FEED THE DUCKS" should come before "FEED", "WIND UP THE CLOCK" should come before "WIND UP" and so on. If you fail to do this, then the main phrases will never be activated, due to the search method that the Triggertest routine uses. If no triggers are found then the "Sorry I do not understand that" message is printed and control returns to the main loop. If the command is recognised, then the Actions routine is called to carry out the required action.

Triggers

This routine steps through the current location data (until it reaches a new declaration for location, object or event) searching for Trigger commands. If a command is found, it is tested by the routine triggertest.

Triggertest

This routine attempts to match the command line with the trigger phrases for the current trigger. This is done by scanning the trigger line until the end marking "*" is found, which marks the start of the condition. A test is made for each of the trigger phrases to see if it occurs within the command line (using the string function INSTR). If a match is made, the condition for that trigger is evaluated. Only if there is a match and the condition evaluates to TRUE, is success reported back to Triggers and PROCESSCOMLINE.

Actions

This routine simply scans down the list of actions which follow a Trigger

command. The type of action is read in, and then the appropriate action routine is called, depending on the action. When all the actions have been read in, the routine returns to the main loop. Each of these actions then performs the required tests and manipulations to carry out their function, before returning to the main action routine. If you wished to add more actions to the AKS system, it is simply a matter of adding a new test on "act$" inside this routine's WHILE loop, with a GOSUB to your new action routine.

UPDATECOUNTDOWNS

This routine steps through all the counters possible and tests to see if they are currently counting. If a counter is active, then it is decremented by one. If the counter value is still above Ø, nothing further is done, and the routine returns to the main program loop.

However, if the counter has reached a value of zero, the counter is reset to a non-counting state and the appropriate event is activated. This is done by setting the current dataline to the start of that event, and then calling the Actions routine to step through and perform the required actions. Once this is complete, the main program loop is resumed.

That completes the main program structure. You should find all the above routine descriptions relatively easy to follow, using the comments in the program as a guideline. Many of the routines described in this chapter can quite easily be used in other programs, not just adventure games. The pretty printing abilities of Printdescr being just one such example, along with the whole method of data searching employed by AKS.

13
IMPLEMENTING THE EXPRESSION EVALUATOR

The behaviour of the AKS expression evaluator has already been discussed. This chapter explains how the AKS Basic program arrives at a result of true or false for a conditional expression. This description of the expression evaluator will be invaluable should you decide to alter the terminology, implement additional operators or allow the use of further flags and predicates. In addition, the expression evaluator in AKS illustrates some fundamental programming techniques and data structures.

The Basic code corresponding to the expression evaluator is contained in the subroutine *evalthis* and the subroutines it invokes. On entry to this routine, the expression string should be stored in the variable 'expr$'. On exit, the variable 'res' holds the result of true or false. The integer values representing true or false are -1 and Ø respectively. This is the same as the internal notation used by Amstrad Basic when it evaluates conditional expressions in IF statements. Consequently, the AKS program can test the result returned by *evalthis* as follows:

```
IF res THEN . . .
```

Thus to evaluate an expression, AKS copies the expression string into expr$ and calls *evalthis*. Often, the AKS program wishes to read in the next string and then evaluate it. This is reflected in the presence of the subroutine *evalnext*, which reads the next string in the DATA into 'expr$' and then calls *evalthis* to evaluate it.

Considering that the function of *evalthis* is to return something which can be tested by a Basic IF statement, it may seem pointless to go to the trouble of writing a conditional expression evaluator to do the same thing as the existing Amstrad Basic conditional expression evaluator. Take for example, the AKS condition string (having already stripped off the preceding '*'):

```
-(W3/C3).V12
```

It is not difficult to see how, by substituting 'NOT' for '-', 'OR' for '/' and 'AND' for '.', this could be transformed into the string:

```
NOT (W3 OR C3) AND V12
```

Now if you replaced all occurrences of sequences of digits by '(', the sequence of digits itself and ')', the resulting string appears to become a Basic conditional expression:

```
NOT (W(3) OR C(3)) AND V(12)
```

Given that the arrays W, C and V existed and contained true/false values (-1 or Ø), it would be very convenient if Amstrad Basic could then be made to evaluate this for us. Unfortunately, this information is locked inside a string variable and Amstrad Basic is unable to evaluate a string variable. For example, it is INVALID to write in a program:

```
IF expr$ THEN . . .
```

Some Basics have a command to overcome this problem and force the Basic interpreter to evaluate a string. A very 'dirty' way in which Amstrad Basic could be made to evaluate a string would be to convert the string into Amstrad Basic's internal representation and POKE this into the program between an IF and a THEN and then execute this line. However, AKS expressions are variable lengths and so other parts of the program would need to be adjusted to create exactly the right number of spaces between the IF and the THEN. An alternative method of evaluation must be used.

Therefore, the AKS program is forced to do a step by step evaluation of 'expr$'. Ignoring the trivial case of 'expr$' being just '*' where *evalthis* returns true, the evaluation process can be divided into three stages:

1. Substitution of flags and predicates (eg. C3,W3,V12) in expr$ for 't' or 'f' representing their true or false values.
2. Reorganisation of the expression according to operator priority and bracketing (ie. '-' before '.' and '.' before '/' unless brackets dictate otherwise). The now redundant brackets are discarded.
3. Evaluation of operators and their 't' and 'f' operands in the order established in stage 2.

Although the evaluation takes place in three logical stages, in practice stage 1 can be performed during the same scan through the expression string as stage 2. A brief glance at *evalthis* reveals that it has two subroutines performing the three stages:

1. *converttoRP* — stages 1 and 2.
2. *evaluateRP* — stage 3.

The letters 'RP' stand for Reverse Polish. An expression written in Reverse Polish notation requires no brackets or operator prededence as the ordering of the expression precisely represents the order of evaluation. To achieve this, Reverse Polish notation places an operator after its operands, giving rise to the alternative name of postfix notation. The normal notation used in mathematics and Basic is called infix notation and places an operator in between its operands. The scenario writer is allowed to write AKS expressions in infix notation for the sake of readability. It would be unacceptable to force the scenario writer to learn Reverse Polish before he could use AKS. However, anyone who has done a lot of programming in the language Forth, which uses Reverse Polish all the time, may be happier writing expressions in Reverse Polish. If this is the case, the *converttoRP* subroutine can be replaced by a subroutine which just performs stage 1 of the evaluation process. A slight increase in the execution speed would also be achieved. To help convince you to stick to infix notation, some examples of infix AKS expressions and their corresponding Reverse Polish versions are given below:

INFIX	**REVERSE POLISH**
W3/C3	W3C3/
-(W3/C3)	W3C3/-
-(W3/C3).V12	W3C3/-V12.
(C1.C2)/(C3.C4.C5.C6)	C1C2C3C4C5C6..../

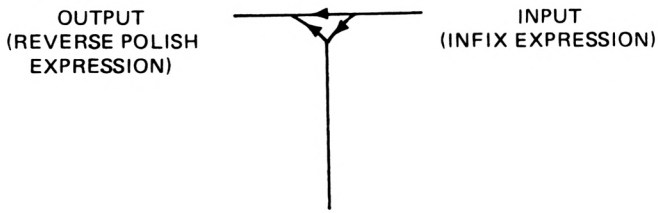
The subroutine *converttoRP* converts the infix expression held in 'expr$' into the Reverse Polish equivalent which is returned in 'revpol$'. It repeatedly calls *getlex* to get the next operator or operand from 'expr$'. It is the subroutine *getlex* which performs the substitution of flags and predicates for either 't' or 'f' to perform 1 of the evaluation process. When *getlex* encounters an operator (ie. next character is '/', '.', '-', '(' or ')'), it simply returns it to *converttoRP* in 'dat$'. However, if it encounters a flag or predicate (ie. next character is 'F', 'V', 'W', 'C', 'L' or 'O') then it calls *evalflag* to determine a value of true or false and set 'dat$' to 't' or 'f' accordingly.

Having performed stage 1 of the evaluation, *converttoRP* must reorder the lexical units returned by calls to *getlex* to form Reverse Polish. There are several algorithms which could be used to do this

reordering and removal of brackets. All of these algorithms require a means of determining the priority of operators shown below:

OPERATOR	PRIORITY
-	highest
.	
/	lowest

The algorithm used in AKS is often referred to as the Shunting Algorithm. This name arises from an analogy with a simple railway network, shown in the diagram below.



Let us ignore brackets for the moment. The algorithm takes carriages (lexical units) from 'input' (infix expression) one at a time. If the carriage is type-A (operand) then it passes straight across to 'output' (Reverse Polish). However, if the carriage is type-B (operator) it goes into the 'siding'. Each carriage has a priority (operator priority). When a new carriage approaches the 'siding' it allows carriages already in the 'siding' to go to 'output' one at a time until a carriage of lower priority is at the front of the 'siding'. The new carriage then takes its place at the front of the 'siding'. Eventually, there are no more carriages at 'input' and all the carriages in the 'siding' are allowed to continue to 'output'. The Reverse Polish expression is now at 'output'. An example conversion using this algorithm is given below. For clarity, flags and predicates are shown as their identifiers instead of their actual 't' or 'f' values from stage 1.

INPUT	SIDING	OUTPUT
C5.-V7/F4	empty	empty
.-V7/F4	empty	C5
-V7/F4	.	C5
V7/F4	-.	C5
/F4	-.	C5V7
F4	/	C5V7-.
empty	/	C5V7-.F4
empty	empty	C5V7-.F4/

By a simple extension, this algorithm can be made to remove brackets after they have served their purpose. Firstly, the symbol ')' must be considered as the highest priority operator. When a '(' is encountered it goes into 'siding' obeying the same rules as the other operators. The algorithm then continues as normal until a ')' is encountered. This causes all operators in 'siding' to be released to 'output' until a '(' is reached. This bracket pair can now be discarded. For example:

INPUT	SIDING	OUTPUT
-(C3/W3).V12	empty	empty
(C3/W3).V12	-	empty
C3/W3).V12	(-	empty
/W3).V12	(-	C3
W3).V12	/(-	C3
).V12	/(-	C3W3
.V12	-	C3W3/
V12	.	C3W3/-
empty	.	C3W3/-V12
empty	empty	C3W3/-V12

In computing terms, a structure called a stack embodies the idea of the 'siding'. Machine code programmers will undoubtedly be very familiar with the operation of the Amstrad's hardware stack. The AKS program is unable to use this stack freely because Amstrad Basic is using it as the program is running. Therefore AKS maintains its own software stack. Whether it is implemented in hardware or in software, a stack has the same logical structure. There are two operations associated with a stack, adding an item and removing an item. In Z8Ø assembler these operations are known as PUSH and POP respectively. AKS uses somewhat the more readable names of *stack* and *unstack* for the subroutines performing these operations. The important thing to remember about a stack is the Last In First Out (LIFO) rule, which means last item in will be the first item out. This is the reason for the analogy of the Shunting Algorithm where the last carriage in is always the first carriage out. The internal mechanism by which the status of a stack is maintained is by a pointer to the next free element of the stack. The AKS stack elements are held in an array called 'stack' and the pointer to the next free element of 'stack' is 'stacktop'. At the start of the program 'stacktop' is initialised to point to the first element of 'stack'. From then on, 'stacktop' is only changed by the *stack* and *unstack* subroutines. Stacking or unstacking a variable is done by storing the value of the variable in 'dat' and calling *stack* or *unstack* respectively.

Having converted the infix string into Reverse Polish it must be evaluated. This can be done very simply by the use of a stack. As *converttoRP* has finished using the software stack, *evaluateRP* can make use of the same stack. The algorithm scans through 'revpol$' one character at a time from left to right. If the character is an operand ('t' or 'f') then stack it. If the character is an operator then take the required number of operands off the stack, perform the operation and stack the result of 't' or 'f'. The binary operators, '.' and '/' will unstack two operands whereas the unary operator '-' will just unstack one operand. When all the characters in 'revpol$' have been processed, a single value remains on the stack. If this value is 't', 'dat' is set to 'true' otherwise 'dat' is set to 'false'. The expression has now been fully evaluated.

However, what happens if the original 'expr$' was incorrect? Fortunately, this method of expression evaluation allows simple checks to be made at each stage of the evaluation process to detect the validity of the expression. Firstly, if any invalid characters (ie. not in "-./()FVWCLO" or a digit) are included then these are recognized during the initial scan through the string — stage 1 of the evaluation process. Missing numbers after a flag or predicate character (eg. '(C/W3)') are detected when the evaluator attempts to substitute 't' or 'f' for the flag/predicate — stage 2. Finally, an incorrectly structured expression (eg. '(C3/W3)V12') will be detected after the Reverse Polish expression has been evaluated because the stack will hold more than just the result — at the end of stage 3.

14
EXTENDING AKS

While the basic AKS system does provide a complete adventure game generator, and one which can be used to create some very complex and large adventures, obviously there are additions which could be made. Everyone has their own ideas for adventure games, and for the types of actions and situations they would like to include in their own games. In this chapter we will detail some of the additions we have thought of, as well as methods for adding your own additions and extensions to the basic AKS format. We hope that you will experiment with AKS, and add your own new actions, commands and so on; you will learn a lot more about programming and adventure games writing in this way than from any commercial games designer.

EXTRA ACTIONS

This is the simplest addition to AKS, and the easiest to carry out. The actions included in the basic system cover all the usual adventure game requirements, but you may well wish to add other actions which cannot be constructed from several existing ones. Possible new actions might be EAT, DRINK or even SLEEP! These can be fitted into the system by inserting a new subroutine which deals with the objects the new action affects. This new subroutine is then accessed by adding a new line into the Actions routine, which tests "act$" for the new value. An example might be:

```
IF act$ ="SL" THEN GOSUB 6000 : REM *SLeep*
```

where the subroutine dealing with the sleep action is located at line 6000.

REAL TIME ACTION

Although the AKS system already provides a system of counters which can simulate a timed event, where the time is measured in turns, this may not be enough. In many games, the adventure is played in real-time, with time ticking away between turns (while the player is making his decisions and typing in a command) as well as while the computer is actively engaged in processing a command. In machine code, this is fairly easy to achieve, using interrupts to update a clock. Indeed, on some machines a realtime clock is provided by the computer's hardware. In the case of the Amstrad, we are fortunate that the Basic itself provides for interrupts, and allows us to create real time events from Basic.

The way such a feature could be implemented is to alter the Initialise Counter subroutine so that it sets an interrupt timer in progress. Thus, we would use the Amstrad AFTER statement to say that a certain subroutine would execute after a certain amount of time. This would enable us to set an event going, and then the basic hardware would interrupt whatever process was going on in order to execute our event after the elapsed time. Once the event has occurred, we can return to the main program at the point we left it.

It would also be possible to implement an event which occurred every so often using the basic command EVERY, instead of AFTER, as in the above example. This would cause our event subroutine to execute at regular intervals, instead of just once. One possible use for this would be intelligent characters, who could move through the game of their own accord, with their position updated after every three minutes or so.

The Amstrad manual gives full details of the interrupt facilities available in Chapter 10.

DON'T CARE MATCHES

At the moment, we have to match all of our command input against a set of trigger phrases, with a match for each word in the trigger phrase being necessary. Wouldn't it be better if we didn't always have to match every word, but could just make sure that some of the input phrase was correct, while not caring about the rest? This is where what is known as a "don't care" indicator comes in. This enables us to say that we are not bothered about what value some part of the phrase has, just that we want that part of the phrase to be present. To add this feature to AKS, we simply have to add a "don't care" facility into the routine triggertest, which compares the phrases with the command line input. This could be a symbol such as "£", which would represent a "don't care" word. For instance, in:

T,eat £,*

we are simply testing to see if the player wants to eat an object, we are not bothered about which object he is trying to eat!

This facility also allows us to set unconditional triggers which will activate whenever the player inputs a command at their location, regardless of what that command is. This is done by using the trigger:

T,£,*

which will ignore the player's input completely. This might be useful if we wish to cause an action the first time the player tries to do something in a location, or perhaps to simulate something like the player being drunk, where it will trap any of the player's input!

A RANDOM FACTOR

Although, personally, we do not like random happenings in adventure games, as they can spoil an otherwise excellent adventure by taking away any skill factor, some people do like them. If you wish to add a random facility to AKS, it would be possible by adding the random facility to the conditions part of a trigger. Thus, the trigger:

T,jump,*R25

would only happen 25% of the time, where the "R25" condition indicates the random 25% chance.

SKILLS

With the growth of roleplaying games and the increasing interest in such games, there are more and more adventure games being produced which attempt to emulate a roleplaying game. This means adding player character skills to the adventure. In AKS, this would require a character generating section, which would enable the player to create his character. You could then test these skills in the normal conditional checks to see if the player is capable of performing an action, or to see if the player is affected by an event. Again, skills do tend to add to the random elements in the adventure game, and produce a poorer adventure in our opinion. There is no way a computer can substitute for a human games master!

SECTION FOUR

WITCH HUNT — AN EXAMPLE AKS SCENARIO

15
WITCH HUNT PLOT DESIGN

Designing the plot for an adventure game is very similar to designing the plot of a novel. You begin with basic ideas about the situations and characters you want to deal with, often just in the form of short, disconnected notes. These must be brought together, and interlinked, to form a continuous storyline. Within the adventure context, you do not have to bother quite so much about detail, or even realism, but you are faced with several other problems. In a novel, the main character or characters, stick to the storyline you have set out — they don't have a mind of their own; in an adventure game, your central character is the player, and they can think for themselves! In this chapter we will be looking at how an adventure game is designed, and how to develop the basic ideas into a fully-fledged plotline.

The example plot and adventure game we will be considering is Witch Hunt. This is a relatively simple and fairly short scenario which has been implemented using AKS. As such, we are able to break down the whole scenario into sections, and show just how each of these sections connects with each other and the overall plot. If you have not yet played through Witch Hunt, and would like to puzzle out the game by yourself, we suggest that you do so before reading this chapter. The following pages discuss Witch Hunt in detail, and thus give away most of the solutions to the scenario.

BASIC PLOT IDEAS

The basic idea behind Witch Hunt was to develop a short scenario which offered a chance to interact with some characters, and which had an overall goal. We also wanted to get away from the very overworked theme of magicians, knights and dragons — in fact the whole fantasy milieu has been done to death.

Thus, we settled on an approximation of Middle Ages England, with the player's task requiring him to find the identity of a witch in a small village. With both of us being firm fans of Monty Python's Holy Grail film this was a fairly natural choice! Once we had decided on the theme of the adventure, we had to decide on some of the basic elements of the plot.

The Setting

In order to keep the adventure within a reasonable size, and thus keep down the number of locations, we decided to base the adventure in and around a small country village. Once this had been fixed, it was immediately possible for us to work on the map of the adventure game. As the game was relatively small, we needed only show the major buildings and sites of interest during the game, thus people's houses could be left out, along with miles of road, fields, etc. This enabled us to produce the map shown in Figure 15.1, which is a rough sketch map of the area the adventure takes place in. Note, that at this stage we are not bothered about how each location is connected, or what each location contains, just with what locations we have; all the connection, etc information comes later, when we start to produce the actual scenario data.

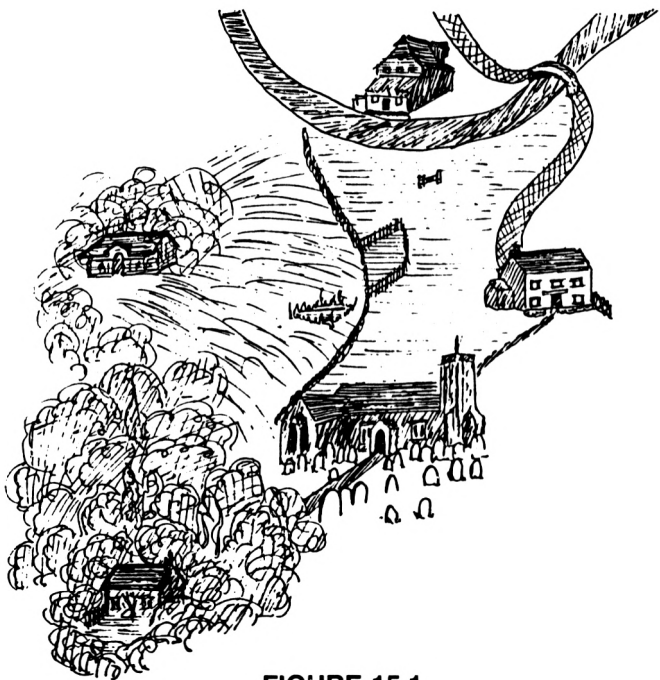


FIGURE 15.1

The Purpose of the Game

While we have already said that the player's mission is to find out which person in the village is the witch; just how does he do that? We came up with, and discarded, several ideas before hitting on the concept used in the Witch Hunt scenario. The player finds a hat which belongs to the witch in question, and thus, he must find the person this hat fits (shades of Cinderella!).

Characters

If the player is looking for a witch in his village, then there must be some inhabitants for him to try the hat on! Thus, we will need some non-player characters for the player to interact with. Well, we have already drawn up the map, and this shows a number of important locations within the village. Obviously, the owners of the various premises within the village can be used as characters, as this has the advantage of tying them down to a logical, fixed location. This gives us a woodcutter, innkeeper, blacksmith, priest, goatherd and miller.

The Objects and Puzzles

The player has to try the witch's hat he finds on everyone in the village, without them realising what he is doing, that is the major puzzle, but there has to be more than this in the game. We have to introduce some puzzles (and objects associated with them) that will hinder him in his task, and prevent the player just breezing through the game.

One limitation is to set a time limit on the game — either he solves the mystery and finds the witch by midday, or he is burnt! This gives the game a sense of desperation, but doesn't really stop the player completing the quest simply and easily. What else is there associated with a witch? Black cats, of course, and thus the black cat in the church yard enters the game. This sets another time limit on the player, but requires him to solve this puzzle (how to lose the cat, before he is burned as a witch), rather quickly; it also adds extra puzzles which must be solved to achieve this aim. All these puzzles use the locations we have already put on the map, and objects which would naturally be there.

Having developed a few rough ideas for our adventure, we can connect these into a complete plot, for the whole adventure. The best approach to plotting the adventure is to consider the actions the player must take to solve the game as a short narrative — thus we write them out as a short story. This enables us to spot inconsistencies in actions, and helps us check that it is possible to perform the actions in the order we require. The plot line for Witch Hunt, our example game is written out below:

WITCH HUNT: THE PLOT

Introduction

You are a simple village lad, who works for the local miller, fetching and carrying grain for him. Your life was reasonably happy and settled, very little disturbs the tranquil lifestyle of your small country village. That is, nothing disturbed it until recently; there have been some strange goings on! The crops have been turning bad, the corn at the mill has been plagued with rats (previously unseen locally) and the goatherd has vanished. These occurrences would be worrying by themselves, even if they didn't relate to you; unfortunately, the villagers have decided that your working at the mill has something to do with the corn, and the fact you found a secret crypt near the church (and were discovered there by the priest) has led them to accuse you of being the witch! You have protested your innocence, of course, to no avail, and they have given you until noon today to prove your innocence by finding the "real" witch — if there is one!

Plot

The game starts on the village green, at the centre of the village. Where can you go to find the witch, or even to find out if there is one? The first clue comes from the village pond, here you find a small toad, which seems strangely afraid of water! So, you pick up the toad and drop it into the pond, and to your surprise there stands the missing goatherd. He is wet and confused, but manages to tell you that he was attacked and turned into a toad in the woods. You also discover that the ducks on the pond are guarding something, and they appear to be hungry enough to object to you reaching it.

Once you are in the woods, your surroundings all look very similar, and you find yourself lost. Whilst wandering through the maze of trees, you hear someone running away from you, and there on the ground is a witch's hat! This must belong to the witch — and you quickly realise that whoever it fits will be shown to be the witch! You now have a way of identifying your quarry. You slip the hat onto your own head — it doesn't fit!

In the centre of the woods is a small clearing, and here you see the woodcutter, he is breathing heavily and looks hot and bothered. He shades his eyes from the sun and then notices the hat you are wearing. Commenting that it will shade him from the sun, he tries it on for size — it doesn't fit him. That is one suspect you can strike off your list.

Coming out of the woods, you pass through the churchyard, and as you do so, a black cat steps out in front of you. You almost trip over it, but manage to keep your feet, only to find that it follows you. You quickly realise that if the villagers find you with the witch's hat and the black cat, they are not going to believe in your innocence for very long!

You have to find some way of getting rid of the cat — food seems a good idea, and then you remember the rats at the mill. You will need to catch them, and that will require cheese, so you head for the inn.

Unfortunately, the innkeeper is in the inn kitchen, and he prevents you from taking his cheese, or his last loaf of bread. Luckily, however, you are able to steal the cheese when his back is turned. So, armed with some bait you (and the cat!) head for the mill. Once inside, you can see some movements on the rafters, and dropping the cheese brings a mouse scurrying down. This you catch and give to the cat, hastily leaving while its attention is diverted! While you are up at the mill, you wonder if the miller is the witch — he might have spoilt his own corn as a cover! When you find the miller he is busy carrying sacks of flour, and he looks thirsty and tired. An idea strikes you, and you fill your witch's hat from the stream, offering it to the miller. He thanks you, grudgingly, and drinks some of the water, before pouring the rest over his head. You can see that the hat will not fit him, as he does this. Another suspect can be crossed off the list.

As the miller takes the hat, he drops a sack of flour, and you remember that the inn-keeper wanted one, which would enable you to get the bread to feed the ducks. You head back to the inn, where the innkeeper accepts the sack and lets you have the loaf of bread. As you leave the kitchen, you notice that the innkeeper is continually running backwards and forwards between the main part of the inn and the kitchen. This gives you an idea for testing the hat against the innkeeper's head. You carefully balance the hat on the door between the two rooms, and then call in the inkeeper — the hat falls from the door onto his head! It doesn't fit, but he is so furious with you that he throws you and the hat out of the inn! Ah well, you can't go there again but at least you know the innkeeper isn't the witch, as the hat didn't fit him.

The next place to try is the church, where the priest will be. He isn't in sight when you enter, so you climb the belfry to look for him. Once by the bell, you cannot resist the urge to strike it, and so you ring the bell. There is a shout below you, and you see the priest run into the nave of the church, and stand looking up directly below you! This is too good an opportunity to miss, and so you drop the hat, which sails gently down and over the priest's head. And over his shoulders as well! You hurry down to the nave and pull the hat from his head, he seems annoyed and hurries back to the crypt. You follow him, and find the strange carvings on the floor of the crypt that you saw earlier, as well as torches lining the walls. Strangely, the brackets holding the torches onto the wall are made of iron — an expensive method! That reminds you — you haven't been to see the blacksmith yet!

The blacksmith is very unhelpful and very unfriendly, when you try and approach him. Perhaps he is trying to hide something? Like the blacksmith's ducks are? It is time to find out just what the ducks are

hiding, and so you feed them the bread. They scatter and reveal the gold which they have been guarding. The woodcutter appears and claims the gold as the money which was stolen from him — obviously by the blacksmith! This is a fairly serious offence, and so the blacksmith is put into the stocks on the green.

This gives you the opportunity to try the hat on the blacksmith, now that he is immobile, and . . . it fits! The blacksmith is the witch, and you have proved your innocence! You celebrate that evening with the burning of the blacksmith, glad that it isn't you who is being burnt!

Obviously, when you are constructing your own plot lines, you do not need to write them out in such a detailed and structured way as the one above. That is written out like this, to make it interesting to read and follow for other people — you are the only one who has to understand your plot notes.

The next stage in preparing your adventure is to go from the plot notes to the actual data statements required for coding the adventure using AKS. The next chapter shows how this is done, with a full break down of the Witch Hunt plot described above. If you study both the chapters together, you will see how a plot idea can be translated in a straight forward way into AKS statements.

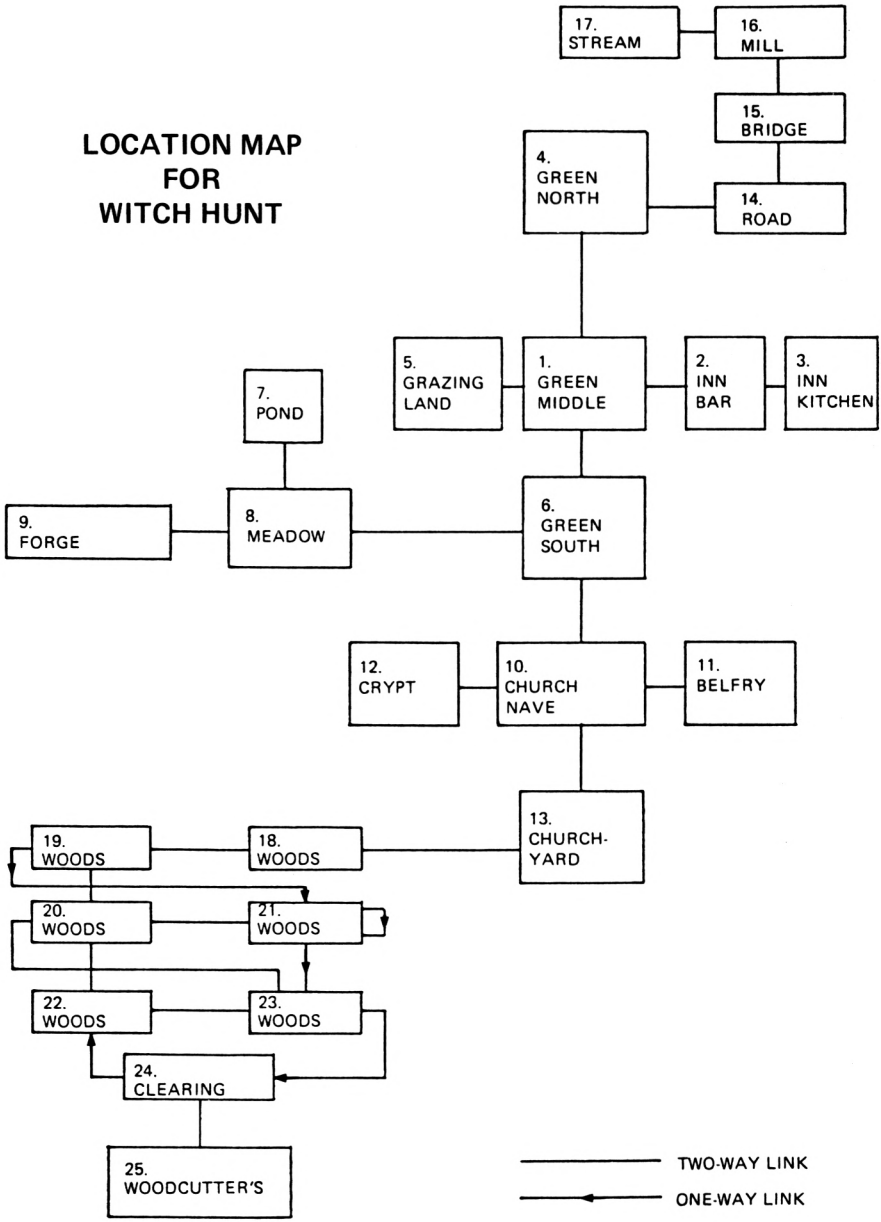
16
BREAKDOWN OF WITCH HUNT

This chapter consists of a detailed breakdown of each location, object and event in the example scenario Witch Hunt, explaining how they can be implemented using the AKS system. The breakdown consists of the following:

1) The location map of AKS, showing how each of the locations is connected.
2) Tables listing each of the flags used, the scores given, the location names, and the objects with their start locations.
3) Locations.
4) Objects.
5) Events.

FLAGS

Flag no.	Meaning
Ø	found hat
1	rung church bell
2	hat dropped on priest's head
3	put toad in pond
4	being followed by cat
5	hat balanced on inn door
6	banned from inn
7	fed mouse
8	hat full of water
9	given miller water
1Ø	blacksmith in stocks
11	started counting game moves


LOCATION MAP
FOR
WITCH HUNT
17.
STREAM
16.
MILL
15.
BRIDGE
4.
GREEN
NORTH
14.
ROAD
5.
GRAZING
LAND
1.
GREEN
MIDDLE
2.
INN
BAR
3.
INN
KITCHEN
7.
POND
9.
FORGE
8.
MEADOW
6.
GREEN
SOUTH
12.
CRYPT
10.
CHURCH
NAVE
11.
BELFRY
13.
CHURCH-
YARD
19.
WOODS
18.
WOODS
20.
WOODS
21.
WOODS
22.
WOODS
23.
WOODS
24.
CLEARING
25.
WOODCUTTER'S
TWO-WAY LINK
ONE-WAY LINK

SCORES

Puzzle	Score
Trying hat on woodcutter	1Ø
Trying hat on priest	1Ø
Trying hat on innkeeper	1Ø
Trying hat on miller	1Ø
Trying hat on blacksmith	2Ø
Getting blacksmith in stocks	2Ø
Getting rid of the cat	1Ø
Freeing goatherd from spell	5
Finding gold	5
Loading a saved game	−1

Maximum score is 100 points.
If you save the game, you only get 99 points.

LOCATIONS

Location no.	Location
Ø	global location/player
1	green middle
2	inn bar
3	inn kitchen
4	green north
5	grazing land
6	green south
7	pond
8	meadow
9	forge
1Ø	church nave
11	church belfry
12	church crypt
13	churchyard
14	road
15	bridge
16	mill
17	stream
18	woods
19	woods
20	woods
21	woods
22	woods
23	woods
24	clearing
25	woodcutter's

OBJECTS

Object no.	Object	Initial location
Ø	player	green middle (1)
1	hat	woods (23)
2	toad	meadow (8)
3	ducks	pond (7)
4	cheese	inn kitchen (3)
5	bread	inn kitchen (3)
6	mouse	mill (16)
7	bell	church belfrey (11)
8	goats	grazing land (5)
9	priest	church crypt (12)
1Ø	sack of flour	nowhere (−1)
11	woodcutter	woodcutter's (25)
12	cat	nowhere (−1)
13	innkeeper	inn kitchen (3)
14	gold	nowhere (−1)
15	miller	mill (16)
16	blacksmith	forge (9)
17	stocks	green north (4)

LOCATION Ø — THE GLOBAL LOCATION

Many of the definitions in this location will be standard for most AKS scenarios. Commands which are usually explicitly coded in the Basic program of other adventure games, such as movement, inventory, get and drop, are defined in the AKS scenario global location. The explicit coding in AKS is associated with the action (A) statements not with the keywords which cause their execution. Witch Hunt defines the action of going east as:

```
T,e,east,*
 A,GO,E
```

This associates the keywords 'e' and 'east' with the action 'GO,E' so that when the player's command line contains either then word 'e' or the word 'east' AKS will perform the action of moving the player east. Supposing you wished to adapt Witch Hunt to print a message on the screen confirming the action, it is a simple matter to add an extra action to this trigger as follows:

```
T,e,east,*
 A,PR,Going eastwards
 A,GO,E
```

No modification of the AKS program itself is required. In addition to

allowing modification of the fundamental actions of an adventure game scenario, actions may be deliberately omitted from a scenario by omitting their definition from the global location.

The way Witch Hunt implements the cat following the player around shows the flexibility of AKS triggers. From the above example, it can be seen that adding actions to triggers allows extension of an action. A brief examination of the Witch Hunt global location definition will reveal that it has two versions of every movement command. Both versions contain the same keywords but have different trigger conditions and actions. The first version will fire when flag 4, the 'player being followed by cat' flag, is set to true (ie. '*F4' evaluates to true). The second version will fire unconditionally and so if the first version does not fire the second version will. The first version, which fires when flag 4 is set to true, results in the player being moved by the 'GO' action and the object 12 (the cat) being Zapped In to the player's new location. The second version, which fires when version one fails to fire, just does a normal move player using 'GO'. To cause the cat to start or stop following the player merely requires setting flag 4 to true or false respectively.

The global location in Witch Hunt also contains triggers for actions not fundamental to adventure games in general but which may be performed at any location in the Witch Hunt scenario. An example is the action of eating something. Commands containing 'eat loaf' or 'eat bread' will cause 'Yummy yummy!' to be printed and the loaf (object 5) to be Zapped Out to nowhere; provided, of course, that the player is carrying the loaf (ie. '*C5' is true). A similar trigger exists for eating the cheese. However, if the player enters 'eat church' then neither of these triggers will fire and AKS will reach the 'T,eat,*' trigger and print 'No thanks'. The addition of these extra scenario specific commands helps to give the scenario credibility.

The global location cannot be reached by the player and so requires no description or connection statements. Triggers which may occur in any location must be defined here. However, remember that triggers defined in the player's current location (local triggers) have priority over triggers defined in the global location (global triggers). If a local trigger is satisfied then the global trigger will never even be tested. This is illustrated in Witch Hunt by the global trigger 'T,n,north,*' and the local trigger defined in location 13, 'T,n,north,church,*-F4.012'. If the player is at location 13 and enters the command 'go north' then AKS try to match the local trigger first. The keyword 'north' will match and so the condition '*-F4.Ø12' will be evaluated. When this evaluates to false, the trigger is not satisfied and so AKS continues to try the remaining local trigger statements; only when all these have been tried are the global triggers tried resulting in the firing of the unconditional 'T,n,north,*'. In the case where '*-F4.012' evaluates to true, the actions immediately fol-

lowing the local trigger statement are executed and no further trigger testing is done; hence AKS does not reach the global trigger statement. In this way, the same keyword can be made to have different effects in different locations. Here, 'north' is made to cause an entirely different action in location 13 to the normal global one.

```
7690  DATA L,0
7700  DATA  T,n,north,*F4
7710  DATA   A,GO,N
7720  DATA   A,ZI,12
7730  DATA  T,n,north,*
7740  DATA   A,GO,N
7750  DATA  T,e,east,*F4
7760  DATA   A,GO,E
7770  DATA   A,ZI,12
7780  DATA  T,e,east,*
7790  DATA   A,GO,E
7800  DATA  T,s,south,*F4
7810  DATA   A,GO,S
7820  DATA   A,ZI,12
7830  DATA  T,s,south,*
7840  DATA   A,GO,S
7850  DATA  T,w,west,*F4
7860  DATA   A,GO,W
7870  DATA   A,ZI,12
7880  DATA  T,w,west,*
7890  DATA   A,GO,W
7900  DATA  T,u,up,*F4
7910  DATA   A,GO,U
7920  DATA   A,ZI,12
7930  DATA  T,u,up,*
7940  DATA   A,GO,U
7950  DATA  T,d,down,*F4
7960  DATA   A,GO,D
7970  DATA   A,ZI,12
7980  DATA  T,d,down,*
7990  DATA   A,GO,D
8000  DATA  T,wear,put on,*
8010  DATA   A,PO
8020  DATA  T,remove,take off,*
8030  DATA   A,TO
8040  DATA  T,get,take,pick up,catch,*
8050  DATA   A,GE
8060  DATA  T,drop,put,throw,release,*
8070  DATA   A,DR
8080  DATA  T,ex,exam,examine,look at,*
8090  DATA   A,EX
8100  DATA  T,i,inv,inventory,*
8110  DATA   A,IN
8120  DATA  T,score,*
8130  DATA   A,SC
8140  DATA  T,load,*
8150  DATA   A,LO
8160  DATA   A,IS,-1
8170  DATA  T,save,*
8180  DATA   A,SA
8190  DATA  T,eat bread,eat loaf,*C5
8200  DATA   A,PR,yummy yummy!
8210  DATA   A,ZO,5
8220  DATA  T,eat cheese,*C4
8230  DATA   A,PR,Far too mouldy!
8240  DATA  T,eat,*
```

```
8250  DATA   A,PR,No thanks.
8260  DATA  T,feed cat,drop mouse,release mouse,*C6.012
8270  DATA   A,PR,The cat takes both mouse and cheese and the
      n runs off.
8280  DATA   A,IS,10
8290  DATA   A,AF,4,F
8300  DATA   A,ZO,6
8310  DATA   A,ZO,12
8320  DATA   A,HC,0
8330  DATA  T,hit,kill,attack,*
8340  DATA   A,PR,Sorry. No violence is allowed in this game.

8350  DATA  T,show,give,accuse,feed,*
8360  DATA   A,PR,Not interested.
8370  DATA  T,quit,*
8380  DATA   A,SC
8390  DATA   A,QU
```

LOCATION 1 — GREEN MIDDLE

This is the first real game location in the scenario definition. Witch Hunt starts the player off at this location. This location has two description definitions (D statements). The description the player is given will always start with the first of these which has no condition attached. The second description is only given when the player has not spent one game turn at this location (ie. not visited location 1, '*-V1', is true). Being as the player starts at this location he will only receive the second description once — before his first move. In this way, Witch Hunt implements the usual introduction to an adventure game (ie. as the second description).

The player is only allowed 100 turns in which to complete Witch Hunt. The counting of moves is done using one of the AKS counters (counter 1). When the counter reaches zero event 1 will fire automatically and the actions defined in event 1 will be executed. These actions are described later in this chapter. However, for a countdown to be started, the player's first move must result in an initialise counter (IC) action. For this reason there is a trigger for each possible direction of movement from this location (N,S,E,W). In addition to the normal effect of moving the player (by 'GO,N', 'GO,S', etc.) each of these triggers initialises counter 1 to 100 (by 'IC,1,100'). Although this would have the desired effect on the player's first move, if the player later returns to this location and moves off again then the counter will be reinitialised to 100. This would be fine if you wanted to implement something like a player holding his breath as he leaves a diving bell and having to return to the bell within 100 moves or drown, but we don't. Therefore a flag (flag 11) is used to indicate whether the initialisation has been done or not. AKS automatically initialises all flags to false (F) at the start of a game. The movement triggers test for this flag not being true (by '*-F11') before firing and once fired they assign a value of true to flag 11 (by 'AF,11,T'). Counter 1 will only be initialised by the player's first movement as these triggers will never fire again.

Travelling east from this location will take the player to location 2 — the inn. The trigger keyword 'inn' allows the player to enter commands like 'go inn' or 'enter inn'. However, if the player has been banned from the inn (ie. flag 6 is set to T) then the trigger 'T,inn,*-F6' will not fire. Instead, the 'T,e,east,inn,*F6' trigger will fire and print a message saying that the player may not enter. The latter is another example of the local trigger having priority over the global trigger.

```
8400  DATA L,1
8410  DATA  D,*,In the middle of the village green.
8420  DATA  D,*-V1,Beyond the north end of the green you can
      see your home - the mill. To east is the inn. The church
       where all this trouble began is to the south. You can h
      ear the goats making goaty noises to the west.
8430  DATA  T,inn,*-F6
8440  DATA   A,GO,E
8450  DATA  T,e,east,inn,*F6
8460  DATA   A,PR,The innkeeper refuses to let you enter the
      inn.
8470  DATA  T,e,east,inn,*-F11
8480  DATA   A,GO,E
8490  DATA   A,AF,11,T
8500  DATA   A,IC,1,100
8510  DATA  T,n,north,*-F11
8520  DATA   A,GO,N
8530  DATA   A,AF,11,T
8540  DATA   A,IC,1,100
8550  DATA  T,w,west,*-F11
8560  DATA   A,GO,W
8570  DATA   A,AF,11,T
8580  DATA   A,IC,1,100
8590  DATA  T,s,south,*-F11
8600  DATA   A,GO,S
8610  DATA   A,AF,11,T
8620  DATA   A,IC,1,100
8630  DATA  C,N,*,4
8640  DATA  C,S,*,6
8650  DATA  C,E,*,2
8660  DATA  C,W,*,5
```

LOCATION 2 — INN BAR

The bar in the Melbourne Inn has an unconditional short description and a long description for the first visit to the location implemented in the same way as for location 1. The long description mentions that the kitchen door is slightly ajar thereby giving a clue to the way of trying the hat on the innkeeper. A third description is given when the player has balanced the hat on the door. Triggers exist to allow the hat to be placed on the door and to retrieve it should the player fail to complete the puzzle. Two versions of calling the innkeeper allow him to enter the room when the hat is either on or off the door (flag 5). Notice that the innkeeper object is never actually moved as the messages displayed make this seem to happen without having to genuinely move the object. When the innkeeper is called and the hat is on the door, the player discovers that the hat does not fit him, gains ten points for solving this

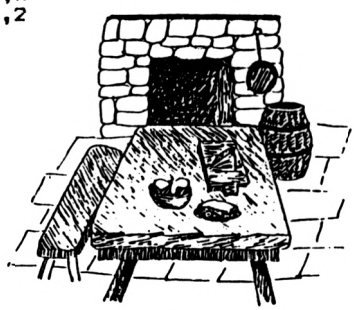
puzzle (by Increment Score — 'IS,1Ø'), gets thrown out of the inn with the hat and banned from returning (by 'AF,6,T').

```
8670  DATA L,2
8680  DATA  D,*,The bar in the Melbourne Inn.
8690  DATA  D,*F5,Some fool has balanced a pointed hat on top
       of the kitchen door!
8700  DATA  D,*-V2,The inn is closed at the moment due to the
       unexplained spoiling of the ale. The kitchen door sits
      slightly ajar at the east end of the room.
8710  DATA  T,balance hat,put hat,place hat,*C1
8720  DATA   A,PR,You manage to balance the hat on top of the
       kitchen door.
8730  DATA   A,AF,5,T
8740  DATA   A,ZO,1
8750  DATA  T,e,east,kitchen,*F5
8760  DATA   A,PR,The hat falls to the floor.
8770  DATA   A,ZI,1
8780  DATA   A,AF,5,F
8790  DATA   A,GO,E
8800  DATA  T,get hat,*
8810  DATA   A,ZI,1
8820  DATA   A,AF,5,F
8830  DATA   A,GE
8840  DATA  T,call,shout,*-F5
8850  DATA   A,PR,"The▲innkeeper▲comes▲into▲the▲room,▲but▲see
      ing▲you▲he▲crosses▲himself▲and▲dashes▲back▲into▲the▲kitc
      hen."
8860  DATA  T,call,shout,*F5
8870  DATA   A,PR,The innkeeper pushes the kitchen door open
      and steps in the room.
8880  DATA   A,PR,PLOP! The hat lands on his head. It looks l
      ike a thimble on a giants head. He throws you and your h
      at out of the inn.
8890  DATA   A,IS,10
8900  DATA   A,GO,W
8910  DATA   A,ZI,1
8920  DATA   A,AF,5,F
8930  DATA   A,AF,6,T
8940  DATA  T,leave,out,*
8950  DATA   A,GO,W
8960  DATA  C,E,*,3
8970  DATA  C,W,*,1
```

LOCATION 3 — INN KITCHEN

The kitchen in the Melbourne Inn is the home for two puzzles — the cheese and the bread. The innkeeper will not allow the player to take either of these objects. Provided that the objects are present, local triggers will trap an attempt to 'get cheese' or any mention of the words 'loaf' and 'bread'. Instead of performing the normal get action, a message is printed telling the player that the innkeeper has prevented his action. Both messages contain clues to a way of obtaining the appropriate object. In the case of the cheese, the clue is in the words ". . . openly trying to take . . .". The solution is to not try and take the cheese openly — 'steal' it. In the case of the bread, the clue is in the words ". . . last loaf . . ." and in the fact that on entering the inn bar the player is told that the innkeeper is broke. Giving the innkeeper a sack of flour from the mill allows him to bake some new loaves and so he gives the player the old one. This is implemented by the trigger "T,give,*C10.05" which may fire when the player is carrying the flour (object 10) and when the innkeeper has the loaf to give away (object 5 is at this location). Although dishonest enough to accept stolen flour, he will not be interested in receiving money from the player because of the danger of being accused of stealing the woodcutter's gold.

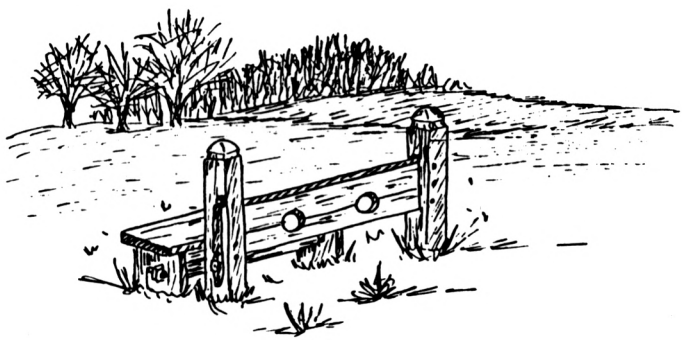
```
8980  DATA L,3
8990  DATA  D,*,Kitchen in the Melbourne.
9000  DATA  D,*-V3,It seems surprisingly empty. Perhaps the i
      nnkeeper is a little short of money at the moment.
9010  DATA  T,steal cheese,*
9020  DATA   A,GE
9030  DATA  T,cheese,*04
9040  DATA   A,PR,GET OFF MY CHEESE! What do you mean openly
      trying to take my food!
9050  DATA  T,give,*C10.05
9060  DATA   A,PR,The innkeeper says "Oh▲goody▲goody.▲I▲can▲b
      ake▲some▲more▲loaves▲now." and grabs your sack of flour.
       "You▲can▲have▲this▲loaf."
9070  DATA   A,MO,5,0
9080  DATA   A,ZO,10
9090  DATA  T,loaf,bread,*05
9100  DATA   A,PR,GET OFF MY BREAD! What do you mean trying t
      o take my last loaf!
9110  DATA  T,leave,out,bar,*
9120  DATA   A,GO,W
9130  DATA  C,W,*,2
```



LOCATION 4 — GREEN NORTH

The north end of the village green is the location where the game can be completed. With the blacksmith in the stocks here for stealing the woodcutter's gold (flag 1Ø is set to T), the hat can be tried on the woodcutter and found to fit. The final twenty points are awarded for doing this (by 'IS,2Ø') and the player clears himself.

```
9140  DATA L,4
9150  DATA  D,*,At the north end of the village green.
9160  DATA  D,*-V4,You remember coming here in the past to th
      row rotten food (and the odd brick) at people in the sto
      cks. This is where they always burnt witches. You rememb
      er throwing on wood... Those were good times.
9170  DATA  T,put hat,place hat,try hat,*C1.F1Ø
9180  DATA   A,PR,It fits!
9190  DATA   A,PR,A crowd of villagers gather. "The▲blacksmit
      h▲is▲the▲witch!" screams the priest. "Burn▲him!▲Burn▲him
      !▲Burn▲him!". And they do. Everyone has a real good time
      ... roasting chestnuts and potatoes in the fire. You hav
      e cleared yourself.
9200  DATA   A,IS,2Ø
9210  DATA   A,SC
9220  DATA   A,QU
9230  DATA  T,road,*
9240  DATA   A,GO,E
9250  DATA  C,S,*,1
9260  DATA  C,E,*,14
```



LOCATION 5 — GRAZING LAND

This location is a red herring. It is a very good way of losing bread and cheese to the goats. If the player is carrying either the cheese or the bread and tries to feed these (or any other object) to the goats then a goat will help itself to either cheese or bread. The order of triggers makes the goats prefer cheese to bread. In addition to this, any attempt

to perform an action using either cheese or bread will result in the goats snatching one or the other.

```
9270  DATA L,5
9280  DATA  D,*,An area of grazing land.
9290  DATA  T,feed,cheese,*C4
9300  DATA   A,PR,A goat snatches your cheese and eats it.
9310  DATA   A,ZO,4
9320  DATA  T,feed,bread,loaf,*C5
9330  DATA   A,PR,A goat snatches your bread and eats it.
9340  DATA   A,ZO,5
9350  DATA  C,E,*,1
```

LOCATION 6 — GREEN SOUTH

```
9360  DATA L,6
9370  DATA  D,*,At the southern end of the village green.
9380  DATA  D,*-V6,A foul odour drifts from the direction of
      the meadow at the west of the green.
9390  DATA  T,meadow,*
9400  DATA   A,GO,W
9410  DATA  T,church,*
9420  DATA   A,GO,S
9430  DATA  C,W,*,8
9440  DATA  C,N,*,1
9450  DATA  C,S,*,10
```

LOCATION 7 — POND

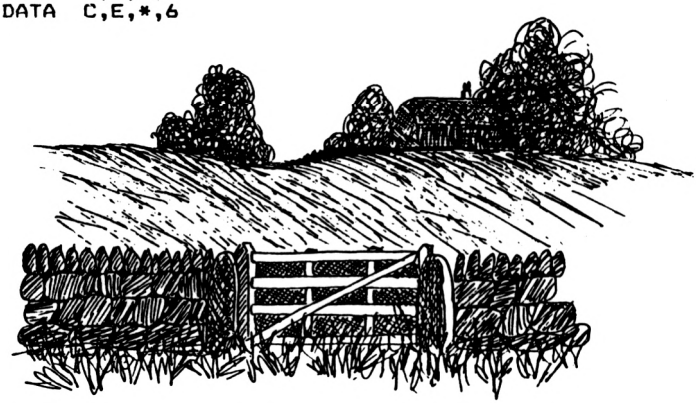
The water in the pond is stagnant and may not be reached by the player. The pond does provide a means of releasing the goatherd from the spell which has turned him into a toad. Throwing the toad into the pond undoes the spell and so he turns back into a boy. Five points are awarded for this and a clue from the boy tells the player to try exploring the woods. The changing of the toad to a boy is only a change in the description of the toad object (object 2) brought about by setting a flag (flag 3) to true. This technique is explained in the description of the toad object later in this chapter.

Feeding the ducks the bread from the inn makes the ducks move away from their nest and reveal the woodcutter's gold. Unlike the toad to boy change, the ducks to gold change involves a genuine substitution of objects. The ducks object (object 3) is Zapped Out to nowhere and the gold object (object 14) is Zapped In from nowhere. The reason for this alternative approach is that the gold object can be manipulated by the player whereas the boy object can not. Were the description switching technique used here, the player could create strange effects by referring to the gold as ducks. While not doing any harm in terms of the scenario, this would make the game appear rather stupid to the player.

```
9460  DATA L,7
9470  DATA  D,*,At the edge of the village pond.
9480  DATA  D,*-V7,The pond is stagnant and smells foul.
9490  DATA  T,put toad,drop toad,throw toad,release toad,*C2.
      -F3
9500  DATA   A,DR
9510  DATA   A,PR,Splash...! The toad turns into a small boy.
       "Where▲am▲I?▲What▲happened?" he mutters as he climbs ou
      t of the pond. "Last▲thing▲I▲remember▲I▲was▲strolling▲in
      ▲the▲woods!"
9520  DATA   A,AF,3,T
9530  DATA   A,IS,5
9540  DATA  T,feed duck,feed ducks,*C5
9550  DATA   A,PR,The ducks gobble up the bread and leave. Th
      ere was something hidden in their nest.
9560  DATA   A,ZO,3
9570  DATA   A,ZO,5
9580  DATA   A,ZI,14
9590  DATA   A,IS,5
9600  DATA  T,fill,water,pond,*
9610  DATA   A,PR,It is impossible to get at the water.
9620  DATA  C,S,*,8
```

LOCATION 8 — MEADOW

```
9630  DATA L,8
9640  DATA  D,*V8,In the meadow.
9650  DATA  D,*-V8,In the middle of a meadow. To the north th
      e ground becomes marshy. At the western end of the meado
      w you can see a wooden building.
9660  DATA  C,N,*,7
9670  DATA  C,W,*,9
9680  DATA  C,E,*,6
```



LOCATION 9 — FORGE

The description the player is given on the first visit to this location says that the blacksmith is well known for his all round handyman skills. This might give the player a clue as to who carved the intricate markings on the floor of the secret crypt.

Showing the blacksmith anything other than gold coins, or accusing him of something when the coins (object 14) are not carried results in the player being told to go away. However if the player is carrying the coins the blacksmith will confess to stealing the woodcutter's gold and is dragged away by the villagers. The player is not told where they take him — to the stocks on the north of the green (by 'MO,16,4'). A flag is set (flag 1Ø) to indicate that the blacksmith is in the stocks. Twenty points are awarded for completing this puzzle.

```
9690  DATA L,9
9700  DATA  D,*,At the blacksmith's forge.
9710  DATA  D,*-V9,You remember coming here often in the past
      . The blacksmith always repairs everyones tools. He is a
       pretty good all round handyman.
9720  DATA  T,show gold,give gold,show coins,give coins,accus
      e,*C14
9730  DATA   A,PR,The blacksmith bursts into tears. "Alright.
      ▲I▲confess▲-▲I▲stole▲the▲woodcutter's▲gold." he blurts o
      ut.
9740  DATA   A,PR,A crowd of villagers rush up and drag the p
      leading blacksmith away.
9750  DATA   A,IS,20
```

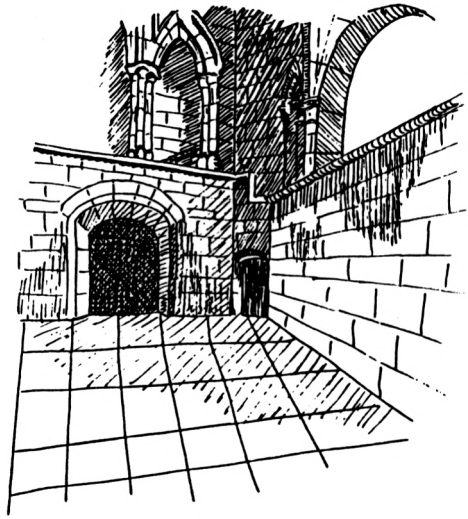
```
9760  DATA   A,AF,10,T
9770  DATA   A,MO,16,4
9780  DATA  T,show,give,accuse,forge,anvil,*
9790  DATA   A,PR,He shouts "Be▲off▲with▲you!▲Little▲witch.".

9800  DATA  C,E,*,8
```

LOCATION 10 — CHURCH NAVE

Once the player has managed to get the hat to land over the priest's head and shoulders as described in the next location explanation, the hat must be retrieved. The trigger which traps the keyword 'hat' when the hat is on the priest's head Zaps Out the priest (object 9), Zaps In the hat (object 1) and then does a Get to pick it up.

```
9810  DATA L,10
9820  DATA  D,*,In the church nave.
9830  DATA  D,*-V10,The secret entrance to the crypt you foun
      d lies open. High above you is the belfry balcony. The f
      ront door is to the north. In the south wall is a small
      door.
9840  DATA  T,hat,*F2.09
9850  DATA   A,PR,Tug...
9860  DATA   A,ZI,1
9870  DATA   A,AF,1,F
9880  DATA   A,ZO,9
9890  DATA   A,GE
9900  DATA   A,PR,The priest storms off.
9910  DATA  T,crypt,*
9920  DATA   A,GO,D
9930  DATA  T,belfry,*
9940  DATA   A,GO,U
9950  DATA  C,N,*,6
9960  DATA  C,S,*,13
9970  DATA  C,D,*,12
9980  DATA  C,U,*,11
```



LOCATION 11 — CHURCH BELFRY

The puzzle at this location is a two stage one, given that the player has already found the hat. The first stage is to ring the bell which causes the priest (object 9) to come from the crypt and stand in the nave (location 1Ø) directly below you (by 'M0,9,1Ø'). Setting flag 1 to true causes the priest's description to change to say he is looking up at the belfry. The second stage of the puzzle involves dropping the hat from the belfry onto the priest's head (indicated by setting flag 2 to T) to reveal that it is too large for the small priest and gain ten points. If the bell has not been rung when the hat is dropped (flag 1 is F) then it just falls onto the nave floor, as a result of the second 'drop hat' trigger statement.

```
9990  DATA L,11
10000 DATA  D,*,In the belfry.
10010 DATA  D,*-V11,Far below you can see the church nave.
10020 DATA  T,ring bell,*-(F1/F2)
10030 DATA   A,PR,The bell tolls and nearly deafens you!
10040 DATA   A,M0,9,10
10050 DATA   A,PR,You see the priest run into the church dire
      ctly below you.
10060 DATA   A,AF,1,T
10070 DATA  T,ring bell,*
10080 DATA   A,PR,Ding dong...!
10090 DATA  T,drop hat,throw hat,*C1.F1
10100 DATA   A,PR,The hat drops from the belfry and lands ove
      r the priest's head.
10110 DATA   A,IS,10
10120 DATA   A,AF,2,T
10130 DATA   A,ZD,1
10140 DATA  T,drop hat,throw hat,*C1.-F1
```

```
10150 DATA   A,PR,Weeee.... it falls from the belfrey.
10160 DATA   A,MO,1,10
10170 DATA  T,nave,leave,*
10180 DATA   A,GO,D
10190 DATA  C,D,*,10
```

LOCATION 12 — CHURCH CRYPT

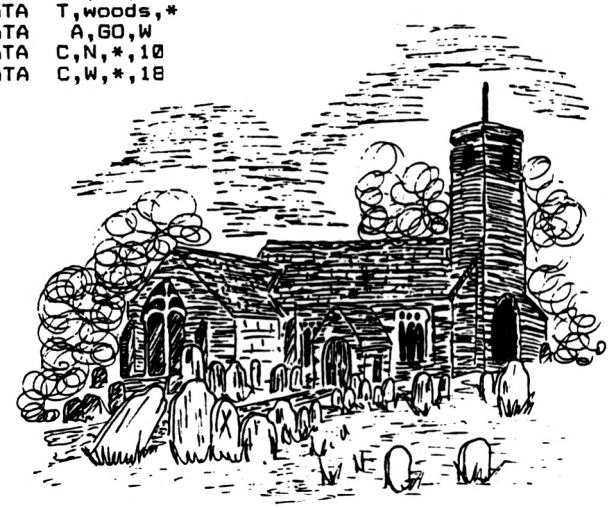
This is the location discovered by the player (before the game started) and where he was caught by the priest. The function of the crypt in the Witch Hunt scenario is to provide the player with clues to the identity of the witch, without providing any hard and fast evidence. The clues are obtained by examining the location. Examination of the markings on the floor reveals nothing of their meaning but the player is told that they are well carved. When the player first visits the blacksmith, he is reminded that the smith is well known for his all round handyman skills. This clue is a little vague whereas the clue given by examining the torches on the wall is very informative. Just mentioning torches says that they are firmly attached to the wall by brackets. The unwary player may take this to only mean that the torches may not be taken. However, this should prompt the player to examine the brackets and discover that they are made of iron. This points very strongly at the blacksmith's involvement. Notice that the markings, torches and brackets are not objects and so their examination requires explicit triggers in the crypt location definition; whereas objects have their examination details embedded in the object definition.

```
10200 DATA L,12
10210 DATA  D,*,In the  secret crypt.
10220 DATA  D,*-V12,The air is icy cold. The floor is intrica
      tely carved with strange markings. On the wall are lit t
      orches.
10230 DATA  T,torches,torch,wall,walls,*
10240 DATA   A,PR,A couple of torches are firmly attatched to
       the wall by brackets.
10250 DATA  T,bracket,brackets,*
10260 DATA   A,PR,Just plain iron brackets.
10270 DATA  T,floor,markings,*
10280 DATA   A,PR,You can make no sense of the markings but y
      ou can see they are well carved.
10290 DATA  T,leave,*
10300 DATA   A,GO,U
10310 DATA  C,U,*,10
```

LOCATION 13 — CHURCHYARD

When the player goes north from here he trips over a black cat which will then follow him around for 20 moves before a crowd of villagers notice he has a black cat familiar and decide to burn him. The counting of moves is done using counter Ø and the villagers burning the player is event Ø. The local trigger which traps an attempt to move north starts the countdown by the action 'IC,Ø,2Ø'. In addition to this, a flag is set to indicate that the player is being followed by the cat (flag 4). This prevents the same trigger from firing again when the player tries to go north again and it allows firing of the alternative set of movement triggers defined in the global location. These alternative movement triggers move both player and cat. The way in which the player may get rid of the cat is described in the explanation of the mill location definition (location 16). It should be noted that the cat will appear again should the player return here and go north again after having got rid of the cat at the mill. However, the objects to get rid of the cat have been destroyed (Zapped Out to nowhere) at this stage in the game so the cat can not be disposed of again.

```
10320 DATA L,13
10330 DATA  D,*,In the churchyard.
10340 DATA  T,dig,grave,headstone,tomb,*
10350 DATA   A,PR,Careful... they bury grave robbers around h
      ere!
10360 DATA  T,n,north,church,*-F4
10370 DATA   A,PR,You trip over a black cat which appears fro
      m behind a headstone!
10380 DATA   A,ZI,12
10390 DATA   A,AF,4,T
10400 DATA   A,IC,Ø,2Ø
10410 DATA  T,church,*
10420 DATA   A,GO,N
10430 DATA  T,woods,*
10440 DATA   A,GO,W
10450 DATA  C,N,*,1Ø
10460 DATA  C,W,*,18
```



LOCATION 14 — ROAD

To prevent the player wasting many hours trying to give a hat full of stream water to characters other than the miller, a local trigger will fire as the player moves westwards carrying the hat of water. This trigger tells the player that he tripped and spilt it. The flag indicating that the hat is full of water (flag 8) is reset to F; thereby emptying the hat.

```
10470 DATA L,14
10480 DATA  D,*,The road.
10490 DATA  D,*-V14,You know this road well. To the north it
      passes by the mill on its way to town.
10500 DATA  T,w,west,*F8
10510 DATA   A,PR,WHOOPS! You tripped and spilt the water.
10520 DATA   A,AF,8,F
10530 DATA  T,follow road,along road,*
10540 DATA   A,GO,N
10550 DATA  C,W,*,4
10560 DATA  C,N,*,15
```

LOCATION 15 — BRIDGE

The player may not fill the hat at this location. If he attempts this then the trigger will say that he is unable to fill it here. A careless adventurer may take this to mean that the hat may not be filled instead of 'not filled here'.

```
10570 DATA L,15
10580 DATA  D,*,On a bridge over a stream.
10590 DATA  T,water,stream,fill,down,d,*
10600 DATA   A,PR,The stream is totally inaccessable from her
      e.
10610 DATA  C,S,*,14
10620 DATA  C,N,*,16
```

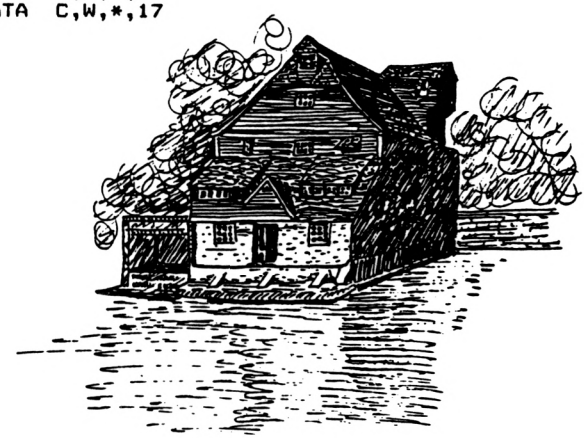
LOCATION 16 — MILL

There are two problems to be overcome at the mill. Firstly, the mouse must be caught. However, the player is not automatically told that there is a mouse here. Instead he is told that something is moving around in the rafters. Examination of the rafters or listening in this location will inform the player that he thinks that there is a rat in the rafters. If the player then mentions the keyword 'rat' or 'rats' he is told that it is a mouse not a rat. These triggers only fire when the mouse is in the rafters (ie. flag 7 is F). The mouse can be tempted from the rafters by feeding it the cheese or just dropping the cheese. When this happens, the cheese object is Zapped Out (by 'Z0,4') and flag 7 is set to T indicating the mouse is no longer in the rafters. The description of the mouse changes as a result of setting flag 7 and becomes a 'piece of cheese with a mouse attached to it'. An attempt to get the cheese when flag 7 is set to T results in a message saying that the mouse refuses to let go of it. If the player has the hat he may catch the mouse in it otherwise the mouse will slip through his fingers. From hereon the mouse and cheese may be manipulated as one object.

The second problem faced by the player in this location is finding out if the hat fits the miller. This is solved by giving the miller the hat full of water. He will drink some of the water and then pour the rest over his head at which point the player sees that the hat would not fit the miller. Ten points are awarded for doing this. However, if the player has already done this once (ie. flag 9 is T) and tries a second time, the miller just smiles and says go away creep. When the miller took his drink, he put down the sack of flour (object 1Ø) he was carrying and did not pick it up again when he started working again. The sack can now be picked up by the player. Previously, any mention of flour or sacks caused a trigger to fire which said the miller growls at you.

```
10630 DATA L,16
10640 DATA  D,*,The mill.
10650 DATA  T,listen,*-F7
10660 DATA   A,PR,You can hear a faint scurrying noise. You w
      onder if it is a rat.
10670 DATA  T,rafters,*-F7
10680 DATA   A,PR,Something is making noises. You wonder if i
      t is a rat.
10690 DATA  T,rat,rats,*-F7
10700 DATA   A,PR,Rat! Pah! I'm a mouse you fool.
10710 DATA  T,feed mouse,drop cheese,*C4
10720 DATA   A,PR,The mouse scurries down and nibbles the che
      ese.
10730 DATA   A,AF,7,T
10740 DATA   A,ZO,4
10750 DATA  T,catch mouse,get mouse,take mouse,pick up mouse,
      *F7.C1.-(W1/F8)
10760 DATA   A,GE
10770 DATA   A,PR,You caught the mouse in the hat.
10780 DATA  T,catch mouse,get mouse,take mouse,pick up mouse,
      *F7
```

```
10790 DATA   A,PR,He slips through your fingers.
10800 DATA  T,cheese,*F7.(06/C6)
10810 DATA   A,PR,The mouse holds onto the cheese.
10820 DATA  T,flour,sack,grain,bag,mill,*-F9
10830 DATA   A,PR,The miller growls at you!
10840 DATA  T,give,*F8.-F9
10850 DATA   A,PR,"The miller puts the sack down, takes the h
      at and drinks some water.
10860 DATA   A,PR,He says "Ta Lad." and then pours the rest o
      ver his head. As he does this you notice that the hat is
       too small for him to wear.
10870 DATA   A,IS,10
10880 DATA   A,AF,8,F
10890 DATA   A,ZI,10
10900 DATA   A,AF,9,T
10910 DATA  T,give,*F8.F9
10920 DATA   A,PR,He smiles and says "Go away creep."
10930 DATA  C,S,*,15
10940 DATA  C,W,*,17
```



LOCATION 17 — STREAM

Unlike the bridge over the stream and the pond this location allows the player to fill the hat with water. To indicate the hat is full of water, flag 8 is assigned the value T. Should the player be foolish enough to forget to take the hat off (ie. 'W1' is T) before attempting to fill it, he is told why his action failed.

```
10950 DATA L,17
10960 DATA  D,*,By the stream.
10970 DATA  D,*-V17,A loud sploshing sound comes from the wat
      er wheel. The water looks as fresh and clear as ever.
10980 DATA  T,fill,*C1.-(W1/C6)
10990 DATA   A,PR,If you insist.
11000 DATA   A,AF,8,T
11010 DATA  T,fill hat,*W1
11020 DATA   A,PR,Gargle...gargle...bubble! You are unable to
       hold your breath any longer and take your head out of t
      he stream.
11030 DATA  T,mill,*
11040 DATA   A,GO,E
11050 DATA  C,E,*,16
```

LOCATION 18 — WOODS (a)

This location is the first WOODS location the player reaches. When the player arrives here for the first time, the second description is given in addition to the main description as a warning to the player.

```
11060 DATA L,18
11070 DATA  D,*,In the deep dark woods.
11080 DATA  D,*-V18,Careful! You might get lost.
11090 DATA  C,E,*,13
11100 DATA  C,W,*,19
```

LOCATION 19..22 — WOODS (b)

These locations all have the same description as location 18 making it difficult for the player to find his way around. To further complicate navigation of this maze, moving in a certain direction does not always take the player to the next physical location in that direction. Reference to the location map will quickly clarify this idea.

```
11110 DATA L,19
11120 DATA  D,*,In the deep dark woods.
11130 DATA  C,E,*,18
11140 DATA  C,S,*,20
11150 DATA  C,W,*,21
11160 DATA L,20
11170 DATA  D,*,In the deep dark woods.
11180 DATA  C,W,*,23
11190 DATA  C,N,*,19
11200 DATA  C,S,*,22
11210 DATA  C,E,*,21
11220 DATA L,21
11230 DATA  D,*,In the deep dark woods.
11240 DATA  C,W,*,20
11250 DATA  C,E,*,21
11260 DATA  C,S,*,23
11270 DATA L,22
11280 DATA  D,*,In the deep dark woods.
11290 DATA  C,N,*,20
11300 DATA  C,E,*,23
```

LOCATION 23 — WOODS (c)

This is another location in the maze but, as the player arrives for the first time a second description tells the player that he thought he saw someone run away. The player finds the hat here.

```
11310 DATA L,23
11320 DATA  D,*,In the deep dark woods.
11330 DATA  D,*-V23,As you arrived you thought you saw someon
      e run away.
11340 DATA  C,N,*,20
11350 DATA  C,E,*,24
11360 DATA  C,W,*,22
```

LOCATION 24 — CLEARING

The clearing in the woods exists to provide the player with a hint for getting the woodcutter to try the hat on. One part of the location description says that the clearing is sheltered from the wind and the sun seems very hot. Wearing the hat removes this part of the description. The player must be wearing the hat when he first moves south to the woodcutter's house. Seeing that the player is wearing the hat, the overheating woodcutter takes it and tries it on for size to see if it will shade him from the sun. It does not fit so he replaces it on the player's head. The player is awarded ten points for solving this part of the scenario.

```
11370 DATA L,24
11380 DATA  D,*,In a clearing in the woods.
11390 DATA  D,*-W1,It is sheltered from the wind here and the
       sun is uncomfortably hot.
11400 DATA  D,*-V24,You hear a loud chopping sound to the sou
      th.
11410 DATA  T,s,south,*-V25.W1
11420 DATA   A,IS,10
11430 DATA   A,GO,S
11440 DATA  C,W,*,22
11450 DATA  C,S,*,25
```

LOCATION 25 — WOODCUTTER'S

The woodcutter can be found here. Any mention of the woodcutter's hut results in the player being told off by the woodcutter.

```
11460 DATA L,25
11470 DATA  D,*,The wood cutter's hut.
11480 DATA  T,hut,*
11490 DATA   A,PR,The woodcutter bars your way. "I've already
      had my gold stolen. I'm not going to lose anything else
      . Keep out!"
11500 DATA  C,N,*,24
```

OBJECT Ø — THE PLAYER

In AKS the player is considered to be a special type of object. The player object is defined in a similar way to all the other objects although any description statements will never be printed by AKS. The initial position of the player can therefore be defined using the position statement. Witch Hunt starts the player off at location 1, the middle of the village green. Unlike other objects, it makes no sense to start the player object off at a special location (ie. either location Ø — carried, or location −1 : nowhere). The player object may be given names (using the N statement) and suitabilities (using the S statements) to allow reference to the player character in the game. It makes no sense to allow the player to get himself (by 'S,GE,*') or wear himself (by 'S,PO') whereas it may be desirable to allow the player to examine himself (by 'S,EX,*,description'). Witch Hunt, however, sticks to the more conven-

tional adventure game format and does not implement actions on the player by the player.

```
11510 DATA 0,0
11520 DATA  P,1
```

OBJECT 1 — HAT

The witch's hat can exist in two states depending on the value of the 'hat full of water' flag (flag 8). As far as the hat object is concerned these states are just different descriptions — one for each state of flag 8. The hat starts off at location 23 (position 23 — 'P,23'). The hat is fundamental to completion of Witch Hunt and the player can get an important hint by examining it. Note that the hat may not be put on when full of water ('S,PO,*-F8').

```
11530 DATA 0,1
11540 DATA  D,*-F8,a witch's hat.
11550 DATA  D,*F8,a witch's hat full of water.
11560 DATA  P,23
11570 DATA  N,hat,*
11580 DATA  S,EX,*,It has a label on the inside which says 'A
      CME Witch's Hat - SIZE 9'. You wonder who wears a size 9
       hat!
11590 DATA  S,GE,*
11600 DATA  S,DR,*-(F8/C6)
11610 DATA  S,PO,*-(F8/C6)
11620 DATA  S,TO,*
```

OBJECT 2 — TOAD

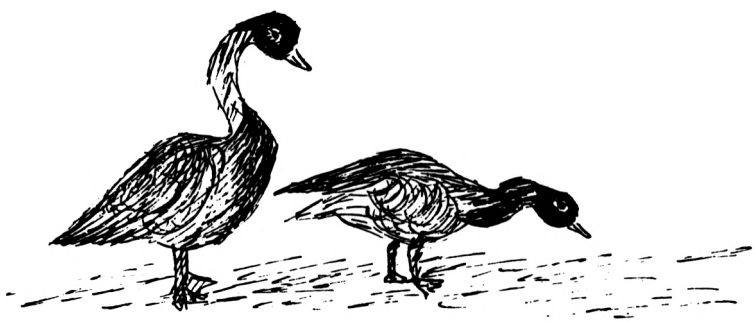
The toad has two states, toad and goatherd, implemented in an identical manner those of object 1. When in the goatherd state (flag 3 is T) this object may not be picked up ('S,GE,*-F3'). Names are defined for both states to allow the object to be examined in either state. The description given depends on the state of the object.

```
11630 DATA 0,2
11640 DATA  D,*-F3,a small wart-covered toad.
11650 DATA  D,*F3,a wet and frightened goat herd.
11660 DATA  P,8
11670 DATA  S,GE,*-F3
11680 DATA  S,DR,*
11690 DATA  N,toad,boy,herd,*
11700 DATA  S,EX,*-F3,a very human looking toad.
11710 DATA  S,EX,*F3,a very toady looking human.
```

OBJECT 3 — DUCKS

The ducks object can be referenced by the names 'duck' or 'ducks'. It is important not to make a scenario too fussy about small details of the vocabulary. In general a plural object should be made to recognise its singular form.

```
11720 DATA 0,3
11730 DATA  D,*,several ducks.
11740 DATA  P,7
11750 DATA  N,ducks,duck,*
11760 DATA  S,EX,*,They seem to be sitting on something.
```



OBJECT 4 — CHEESE

```
11770 DATA O,4
11780 DATA  D,*,a small piece of cheese.
11790 DATA  P,3
11800 DATA  N,cheese,*
11810 DATA  S,GE,*
11820 DATA  S,DR,*
11830 DATA  S,EX,*,Looks a bit cheesy!
```

OBJECT 5 — LOAF

```
11840 DATA O,5
11850 DATA  D,*,a loaf of bread.
11860 DATA  P,3
11870 DATA  N,loaf,bread,*
11880 DATA  S,GE,*
11890 DATA  S,DR,*
```

OBJECT 6 — MOUSE

The mouse object exists in three states:

1) mouse
2) mouse and cheese
3) mouse and cheese in hat

The current state and description of the object is controlled by only one flag, the 'mouse fed' flag (flag 7). If the mouse has not been fed the cheese this flag has the value F and the mouse is in state 1. On feeding the mouse this flag is set to T. After feeding the mouse, but before picking it up ('*F7.-C6), the object is in state 2. Once the object has been picked up ('*F7.C6'), it enters state 3. The sequence of transitions from state to state is fixed as 1-2-3. This object is known by the names cheese or mouse. Another cheese object already exists so a condition must be attached to the recognition of the name. The mouse may not be handled before it has been fed; by which time the real cheese object has been Zapped Out to nowhere. Therefore, when the mouse has been fed (flag 7 is T) it is valid to refer to the mouse object by the cheese name. This is implemented by the 'N,mouse,cheese,*F7' statement.

```
11900 DATA O,6
11910 DATA  D,*-F7,something moving around in the rafters.
11920 DATA  D,*F7.-C6,a piece of cheese with a mouse attatche
      d to it.
11930 DATA  D,*F7.C6,the mouse and the cheese in the hat.
11940 DATA  P,16
11950 DATA  N,mouse,cheese,*F7
11960 DATA  S,GE,*
11970 DATA  S,DR,*
```

OBJECT 7 — BELL

```
11980 DATA 0,7
11990 DATA  D,*,a large brass bell.
12000 DATA  P,11
12010 DATA  N,bell,*
12020 DATA  S,EX,*,a large church bell inscribed with the let
      ters 'ring me!'.
```

OBJECT 8 — GOATS

The goats object is a red herring. Examination of the goats will just confirm the genuine disappearance of the goatherd.

```
12030 DATA 0,8
12040 DATA  D,*,a lot of goats.
12050 DATA  P,5
12060 DATA  N,goat,goats,*
12070 DATA  S,EX,*,They are tethered to posts. Strange! They
      seem to have eaten all the good grass they can reach. Pe
      rhaps they have not been moved for a while?
```

OBJECT 9 — PRIEST

The priest may be described as doing one of three things depending on whether the bell has been rung (flag 1) and whether the hat is on the priest's head (flag 2). The sequence of state transitions is 1-2-3-1.

```
12080 DATA 0,9
12090 DATA  D,*-(F1/F2),a very small priest blessing the secr
      et crypt.
```

```
12100 DATA  D,*F1.-F2,a very agitated priest looking up at th
      e belfrey.
12110 DATA  D,*F2,a priest wearing a black hat over his head
      and shoulders!
12120 DATA  P,12
12130 DATA  N,priest,*
12140 DATA  S,EX,*,He looks very small to you.
```

OBJECT 10 — SACK OF FLOUR

Note that the initial position is 'nowhere' (location −1).

```
12150 DATA O,10
12160 DATA  D,*,a sack of flour.
12170 DATA  P,-1
12180 DATA  N,sack,bag,flour,*
12190 DATA  S,GE,*
12200 DATA  S,DR,*
12210 DATA  S,EX,*,It is labelled 'MegaMill Flour Co.'
```

OBJECT 11 — WOODCUTTER

When the player first encounters the woodcutter ('*-V25'), he is given the first description. If the player is also wearing the hat (object 1) on this first encounter ('*-V25.W1') the first two descriptions are given. The first description will appear in the object list, indented as normal. The second description will appear underneath the object list and will not be indented. This is another way implementing an event in AKS. Subsequent descriptions of this object will only give the third description.

```
12220 DATA O,11
12230 DATA  D,*-V25,an out of breath woodcutter resting on hi
      s axe.
12240 DATA  D,*-V25.W1,Suddenly the woodcutter snatches the h
      at and trys it on. "I▲wonder▲if▲this▲will▲shield▲me▲from
      ▲the▲sun▲?" he says. "Pity...▲not▲my▲size." he grumbles
      and replaces the hat on your head.
12250 DATA  D,*V25,the woodcutter hard at work.
12260 DATA  P,25
12270 DATA  N,woodcutter,*
12280 DATA  S,EX,*,A rather hot sweaty woodcutter.
```

OBJECT 12 — CAT

```
12290 DATA O,12
12300 DATA  D,*F7,a friendly black cat drooling around your a
      nkles.
12310 DATA  D,*-F7,a friendly black cat.
12320 DATA  P,-1
12330 DATA  N,cat,*
12340 DATA  S,EX,*,It looks friendly.
```

OBJECT 13 — INNKEEPER

```
12350 DATA O,13
12360 DATA  D,*,the innkeeper.
12370 DATA  P,3
12380 DATA  N,innkeeper,*
12390 DATA  S,EX,*,He is rather large.
```

OBJECT 14 — GOLD

```
12400 DATA O,14
12410 DATA  D,*,some gold coins!
12420 DATA  P,-1
12430 DATA  N,coins,gold,*
12440 DATA  S,GE,*
12450 DATA  S,DR,*
12460 DATA  S,EX,*,No. They are not sliced golden egg. They m
      ust have come from somewhere else.
```

OBJECT 15 — MILLER

Examination of the miller will give different descriptions before and after he has been given the hat full of water.

```
12470 DATA O,15
12480 DATA  D,*,The miller humping sacks about.
12490 DATA  P,16
12500 DATA  N,miller,*
12510 DATA  S,EX,*-F9,He looks hot and thirsty.
12520 DATA  S,EX,*F9,He looks wet.
```

OBJECT 16 — BLACKSMITH

The blacksmith object may be in one of two states depending on whether he has been put in the stocks or not (ie. flag 1Ø is T or F respectively).

```
12530 DATA O,16
12540 DATA  D,*-F1Ø,The blacksmith hard at work.
12550 DATA  D,*F1Ø,The blacksmith in the stocks.
12560 DATA  P,9
12570 DATA  N,blacksmith,smith,*
12580 DATA  S,EX,*-F1Ø,He looks rather hot.
12590 DATA  S,EX,*F1Ø,He looks stuck.
```

OBJECT 17 — STOCKS

The stocks have no function other than as something for the player to examine. The description given depends on flag 10 in the same way as object 16.

```
12600 DATA 0,17
12610 DATA  D,*,some stocks.
12620 DATA  P,4
12630 DATA  N,stocks,*
12640 DATA  S,EX,*-F10,There is a brass plaque with 'Made by
      OXO' engraved on it.
12650 DATA  S,EX,*F10,There seems to be a blacksmith in them!
      !!
```

EVENT 0

The actions in this event are executed when counter number 0 reaches zero. These actions kill the player because he has been followed by the cat for too long. The final score is printed and a quit actioned.

```
12660 DATA E,0
12670 DATA  A,PR,"▲"
12680 DATA  A,PR,A crowd of villagers gather round you. The p
      riest points at the cat and says "Look▲he▲has▲a▲black▲ca
      t▲familiar!▲That▲proves▲he▲is▲a▲witch.". They drag you a
      way and test your inflammability.
12690 DATA  A,SC
12700 DATA  A,QU
```

EVENT 1

This event fires when counter number 1 reaches zero indicating that the player has had 100 game turns. This is used to represent the midday deadline for the player clearing himself of being a witch. The optimal solution to the game requires nowhere near 100 game turns.

```
12710 DATA E,1
12720 DATA  A,PR,The church bell rings. It is midday. The vil
      lagers drag you away and burn you. It was a really jolly
       occasion and people came from miles around to see you.
12730 DATA  A,SC
12740 DATA  A,QU
12750 DATA F
```

APPENDIX A
THE AKS AND WITCH HUNT LISTING

This section contains the complete listing for the Adventure Kernel System and the Witch Hunt example scenario. The program is over 30k long, so it is a major typing task! In order to get AKS into your machine and fully functioning as easily as possible, we recommend that you follow a couple of guidelines. Firstly, leave in the comments and indentation as you type in the program. They may mean a lot more typing, but you'll be grateful for them if you are trying to debug a section of code! Secondly, pay particular care to the punctuation and format of the DATA statements. If you do find errors after you have typed in the program, this is a very likely place for them to occur.

After the program has been fully debugged, then you can remove the REM statements and the indentation to create more space for scenario data. This will also speed up the response time of AKS.

```
10    GOTO 60
20    POKE 429,lin MOD 256
30    POKE 430,lin\256
40    RESTORE 10
50    RETURN 'from *** restorelin ***
60    REM That routine enables the program to RESTORE to a va
      riable value.
70    REM It does this by poking the line no. stored in "lin"
       into line 40.
80    REM
90    REM >>>IMPORTANT<<<
100   REM
110   REM i)  Do not change lines 10-40.
120   REM ii) Do not use the normal RENUM command.
130   REM     instead - press the <ENTER> key on the numeric
      key pad.
```

```
140   REM                  Ignore the error message.
150   REM
160   REM NB : the program must be RUN to initialise the <ENT
      ER> key...
170   KEY 139,"poke 429,0 : poke 430,0 : renum"+CHR$(13)
180   :
190   :
200   REM ************************************************
210   REM *                                              *
220   REM * A D V E N T U R E   K E R N E L   S Y S T E M *
230   REM *                  ( A K S )                   *
235   REM *                                              *
240   REM ************************************************
250   REM *                                              *
260   REM *AKS was written and developed by : Simon Price.*
270   REM *   WITCH HUNT was written jointly by  :       *
275   REM *   Mike Lewis & Simon Price.                  *
280   REM *                                              *
290   REM *   (C) Mike Lewis and Simon Price 1985.       *
300   REM *                                              *
310   REM ************************************************
320   :
330   :
340   MODE 1       'set 40 column screen mode
350   PRINT "Welcome to AKS."
360   PRINT : PRINT "Initialising... please wait"
370   :
380   REM declare all numeric variables as integers
390   DEFINT a-z
400   :
410   REM initialise constants
420   REM
430   lineinc=10   'BASIC line no. increment
440   maxloc=30    'maximum no. locations
450   maxobj=20    'maximum no. objects
460   maxflag=30   'no. of scenario defineable flags
470   maxcount=5   'no. scenario defineable counters
480   maxstack=20  'maximum size of expression evaluator stack

490   true=-1      'boolean values recognised by IF statement
500   false=0
510   linlen=40    'screen width for description output
520   :
530   REM find value of the constant 'datastart'
540   REM ie. the line no. of the first scenario DATA stateme
      nt
550   ON ERROR GOTO 570 'next but one line
560   GOTO 7680 'an erroneous line immediately preceding firs
      t DATA statement
570   IF ERR<>2 THEN ERROR ERR ELSE RESUME 590
580   REM report if not expected error type
590   datastart=ERL+lineinc
600   ON ERROR GOTO 0  'turn error trapping off
610   :
620   :
630   REM initialise variables
640   REM
650   :
660   DIM locline(maxloc) 'line no.s of start of object DATA
      definitions
670   GOSUB 1080 : REM *initlocations*
680   :
690   DIM objline(maxobj) 'line no.s of start of location DAT
      A definitions
```

```
700   DIM objloc(maxobj)  'initial location of objects
710   GOSUB 1320 : REM *initobjects*
720   :
730   DIM eventlin(maxcount) 'event action definition start l
      ines
740   GOSUB 1710 : REM *initevents*
750   :
760   DIM flag(maxflag)       'scenario defined flags
770   DIM worn(noofobjs)      'object worn flags
780   DIM visited(nooflocs)   'location visited by player flag
      s
790   DIM counting(noofcnts) 'countdown timer on flag
800   GOSUB 1920 : REM *resetflags*
810   :
820   DIM count(noofcnts)     'value of countdown timer
830   :
840   DIM stack(maxstack)     'expression evaluation stack
850   stacktop=0
860   :
870   score=0
880   :
890   REM main program body
900   REM
910   CLS
920   eogame=false
930   WHILE NOT(eogame)
940     GOSUB 2090 : REM *describeloc*
950     visited(objloc(0))=true
960     GOSUB 4300 : REM *getcomline*
970     GOSUB 4390 : REM *processcomline*
980     GOSUB 4790 : REM *updatecountdowns*
990   WEND
1000  PRINT
1010  INPUT "Another game ?",res$
1020  IF LOWER$(LEFT$(res$+"y",1))="y" THEN RUN
1030  GOTO 1030 'hang machine up
1040  :
1050  :
1060  REM *** initlocations ***
1070  REM
1080  lin=datastart
1090  loc=0
1100  type$="?" 'dummy value to force at least one iteration
      of WHILE
1110  WHILE INSTR("OEF",type$)=0
1120    GOSUB 20 : REM *restorelin*
1130    READ type$
1140    IF type$="L" THEN GOSUB 1230 : REM *initloc*
1150    lin=lin+lineinc
1160  WEND
1170  nooflocs=loc-1
1180  RETURN
1190  :
1200  :
1210  REM *** initloc ***
1220  REM
1230  READ defloc
1240  IF loc<>defloc THEN PRINT"loc out of sequence AT LINE "
      ;lin : END
1250  locline(loc)=lin+lineinc
1260  loc=loc+1
1270  RETURN
1280  :
```

```
1290  :
1300  REM *** initobjects ***
1310  REM
1320  obj=0
1330  lin=lin-lineinc
1340  WHILE INSTR("EF",type$)=0
1350    IF type$="O" THEN GOSUB 1460 : REM *initobj*
1360    lin=lin+lineinc
1370    GOSUB 20 : REM *restorelin*
1380    READ type$
1390  WEND
1400  noofobjs=obj-1
1410  RETURN
1420  :
1430  :
1440  REM *** initobj ***
1450  REM
1460  READ defobj
1470  IF obj<>defobj THEN PRINT"obj out of sequence AT LINE "
      ;lin : END
1480  objline(obj)=lin+lineinc
1490  type$="?" 'dummy value
1500  WHILE INSTR("OEF",type$)=0
1510    lin=lin+lineinc
1520    GOSUB 20 : REM *restorelin*
1530    READ type$
1540    IF type$="P" THEN GOSUB 1630 : REM *initobjloc*
1550  WEND
1560  lin=lin-lineinc 'adjust to suit *initobjects*
1570  obj=obj+1
1580  RETURN
1590  :
1600  :
1610  REM *** initobjloc ***
1620  REM
1630  READ loc
1640  IF loc<-1 OR loc>nooflocs THEN PRINT"obj loc out of ran
      ge AT LINE ";lin : END
1650  objloc(obj)=loc
1660  RETURN
1670  :
1680  :
1690  REM *** initevents ****
1700  REM
1710  noofcnts=0
1720  WHILE type$<>"F"
1730    GOSUB 20 : REM *restorelin*
1740    READ type$
1750    IF type$="E" THEN GOSUB 1830 : REM *initeventlin*
1760    lin=lin+lineinc
1770  WEND
1780  RETURN
1790  :
1800  :
1810  REM *** initeventlin ***
1820  REM
1830  READ cnt
1840  IF cnt<>noofcnts THEN PRINT"event out of sequence AT LI
      NE";lin
1850  eventlin(cnt)=lin+lineinc
1860  noofcnts=noofcnts+1
1870  RETURN
1880  :
```

```
1890  :
1900  REM *** resetflags ***
1910  REM
1920  FOR i=0 TO maxflag
1930    flag(i)=false
1940  NEXT
1950  FOR i=0 TO noofobjs
1960    worn(i)=false
1970  NEXT
1980  FOR i=0 TO nooflocs
1990    visited(i)=false
2000  NEXT
2010  FOR i=0 TO noofcnts
2020    counting(i)=false
2030  NEXT
2040  RETURN
2050  :
2060  :
2070  REM *** describeloc ***
2080  REM
2090  PRINT : PRINT : PRINT STRING$(40,"-")
2100  lin=locline(objloc(0)) : GOSUB 2220 : REM *describeln*
2110  loc=objloc(0) : GOSUB 2610 : REM *isobjatloc*
2120  IF NOT(res) THEN RETURN 'quit
2130  PRINT"There is/are : "
2140  FOR obj=1 TO noofobjs
2150    IF objloc(obj)=objloc(0) THEN PRINT"    "; : lin=objli
      ne(obj) : GOSUB 2220 : REM *describeln*
2160  NEXT
2170  RETURN
2180  :
2190  :
2200  REM *** describeln
2210  REM
2220  linefeed=true
2230  GOTO 2270
2240  REM *** describe ***
2250  REM
2260  linefeed=false
2270  type$="?" 'dummy value
2280  WHILE INSTR("LOEF",type$)=0
2290    GOSUB 20 : REM *restorelin*
2300    READ type$
2310    IF type$<>"D" THEN 2340 'go try next DATA line
2320    GOSUB 2700 : REM *evalnext*
2330    IF res THEN READ descr$ : GOSUB 2410 : REM *printdesc
      r*
2340    lin=lin+lineinc
2350  WEND
2360  RETURN
2370  :
2380  :
2390  REM *** printdescr ***
2400  REM
2410  WHILE LEN(descr$) > linlen-POS(#0)+1
2420    rlim=linlen-POS(#0)+1         'right hand side of scre
      en
2430    rhs=rlim+1                    '1 char beyond screen ri
      ght
2440    WHILE MID$(descr$,rhs,1)<>" "
2450      rhs=rhs-1    'skip backwards over rightmost word on
      line
2460    WEND
```

```
2470    rhs=rhs-1
2480    PRINT LEFT$(descr$,rhs);
2490    IF POS(#0)>1 THEN PRINT     'start new line if not at
    start of one
2500    descr$=RIGHT$(descr$,LEN(descr$)-rhs)
2510    WHILE LEFT$(descr$,1)=" "     'skip over leading space
    s
2520       descr$=RIGHT$(descr$,LEN(descr$)-1)
2530    WEND
2540  WEND
2550  PRINT descr$;
2560  IF linefeed THEN PRINT
2570  RETURN
2580  :
2590  REM *** isobjatloc ***
2600  REM
2610  res=false
2620  FOR obj=1 TO noofobjs
2630    IF objloc(obj)=loc THEN res=true
2640  NEXT
2650  RETURN
2660  :
2670  :
2680  REM *** evalnext ***
2690  REM
2700  READ expr$
2710  GOSUB 2770 : REM *evalthis*
2720  RETURN
2730  :
2740  :
2750  REM *** evalthis ***
2760  REM
2770  char$=LEFT$(expr$,1)
2780  IF char$<>"*" THEN PRINT"EXPR expected AT LINE ";lin :
    END
2790  IF LEN(expr$)=1 THEN res=true : RETURN 'quit
2800  expr$=RIGHT$(expr$,LEN(expr$)-1)
2810  GOSUB 2880 : REM *converttoRP*
2820  GOSUB 3780 : REM *evaluateRP*
2830  RETURN 'res is either true/false
2840  :
2850  :
2860  REM *** converttoRP ***
2870  REM
2880  revpol$=""
2890  dat=ASC("(") : GOSUB 4120 : REM *stackdat*
2900  WHILE LEN(expr$)<>0
2910    GOSUB 3050 : REM *getlex*
2920    IF INSTR("tf",dat$)<>0 THEN revpol$=revpol$+dat$ : GO
    TO 2980 'next lex
2930    REM dat$ is an operator
2940    dat=ASC(dat$)
2950    IF dat$="(" THEN GOSUB 4120 : REM *stackdat*
2960    IF dat$=")" THEN GOSUB 3430 : REM *closepar*
2970    IF INSTR("()",dat$)=0 THEN GOSUB 3540 : REM *comparep
    riority*
2980  WEND
2990  GOSUB 3430 : REM *closepar*
3000  RETURN
3010  :
3020  :
3030  REM *** getlex ***
3040  REM
```

```
3050  dat$=LEFT$(expr$,1)
3060  expr$=RIGHT$(expr$,LEN(expr$)-1)
3070  used=false
3080  IF INSTR("FVWECLO",dat$)<>0 THEN used=true : GOSUB 3150
       : REM *evalflag*
3090  IF NOT(used) AND INSTR("()/.-",dat$)=0 THEN PRINT"inval
      id expr AT LINE ";lin : END
3100  RETURN
3110  :
3120  :
3130  REM *** evalflag ***
3140  REM
3150  num=0
3160  isdigit=true : gotnum=false
3170  WHILE isdigit AND LEN(expr$)<>0
3180    char$=LEFT$(expr$,1)
3190    chval=ASC(char$)-ASC("0") 'convert to digit
3200    IF chval<0 OR chval>9 THEN isdigit=false ELSE expr$=R
      IGHT$(expr$,LEN(expr$)-1) : num=num*10+chval : gotnum=tr
      ue
3210  WEND
3220  IF NOT(gotnum) THEN PRINT"flag no. missing AT LINE ";li
      n : END
3230  GOSUB 3300 : REM *setbool*
3240  IF bool THEN dat$="t" ELSE dat$="f"
3250  RETURN
3260  :
3270  :
3280  REM *** setbool ***
3290  REM
3300  bool=false
3310  ON INSTR("FVWCLO",dat$) GOTO 3320,3330,3340,3350,3360,3
      370
3320  bool=flag(num) : RETURN      'F-flag
3330  bool=visited(num) : RETURN   'V-flag
3340  bool=worn(num) : RETURN      'W-flag
3350  IF objloc(num)=0 THEN bool=true : RETURN ELSE RETURN
            'C-flag
3360  IF objloc(0)=num THEN bool=true : RETURN ELSE RETURN
            'L-flag
3370  IF objloc(num)=objloc(0) THEN bool=true : RETURN ELSE R
      ETURN  'O-flag
3380  RETURN
3390  :
3400  :
3410  REM *** closepar ***
3420  REM
3430  releasing=true
3440  WHILE releasing
3450    GOSUB 4200 : REM *unstackdat*
3460    op$=CHR$(dat)
3470    IF op$="(" THEN releasing=false ELSE revpol$=revpol$+
      op$
3480  WEND
3490  RETURN
3500  :
3510  :
3520  REM *** comparepriority ***
3530  REM
3540  newop=dat 'save new operator code
3550  GOSUB 3720 : REM *priority*
3560  newpri=oppri
3570  releasing=true
```

```
3580  WHILE releasing AND stacktop<>0
3590    GOSUB 4200 : REM *unstackdat*
3600    GOSUB 3720 : REM *priority*
3610    IF newpri<oppri THEN revpol$=revpol$+CHR$(dat) : GOTO
      3650 'next
3620    releasing=false
3630    GOSUB 4120 : REM *stackdat*
3640    dat=newop
3650  WEND
3660  GOSUB 4120 : REM *stackdat*
3670  RETURN
3680  :
3690  :
3700  REM *** priority ***
3710  REM
3720  oppri=INSTR("(/.-",CHR$(dat))
3730  RETURN
3740  :
3750  :
3760  REM *** evaluateRP ***
3770  REM
3780  WHILE LEN(revpol$)>0
3790    dat$=LEFT$(revpol$,1)
3800    revpol$=RIGHT$(revpol$,LEN(revpol$)-1)
3810    IF INSTR("tf",dat$)<>0 THEN dat=ASC(dat$) : GOSUB 412
      0 : REM *stackdat*
3820    IF INSTR("tf",dat$)=0 THEN GOSUB 3940 : REM *evalop*
3830  WEND
3840  GOSUB 4180 : REM *unstackdat*
3850  IF dat=ASC("t") THEN dat=true
3860  IF dat=ASC("f") THEN dat=false
3870  res=dat 'dat may already have been boolean (0,-1)
3880  IF stacktop<>0 THEN PRINT"Invalid expr AT LINE ";lin :
      END
3890  RETURN
3900  :
3910  :
3920  REM *** evalop ***
3930  REM
3940  GOSUB 4180 : REM *unstackdat*
3950  IF dat=ASC("t") THEN op1=true ELSE op1=false
3960  IF dat$<>"-" THEN 4000 'not a unary operator
3970  dat=NOT(op1)
3980  GOTO 4050 'store result
3990  REM binary operators
4000  GOSUB 4180 : REM *unstackdat*
4010  IF dat=ASC("t") THEN op2=true ELSE op2=false
4020  IF dat$="." THEN dat=op1 AND op2
4030  IF dat$="/" THEN dat=op1 OR op2
4040  REM store result
4050  IF dat THEN dat=ASC("t") ELSE dat=ASC("f")
4060  GOSUB 4120 : REM *stackdat*
4070  RETURN
4080  :
4090  :
4100  REM *** stackdat ***
4110  REM
4120  stack(stacktop)=dat
4130  stacktop=stacktop+1
4140  IF stacktop>maxstack THEN PRINT"expr too large AT LINE
      ";lin : END
4150  RETURN
4160  :
```

```
4170  :
4180  REM *** unstackdat ***
4190  REM
4200  stacktop=stacktop-1
4210  IF stacktop<0 THEN PRINT"Incomplete expr AT LINE ";lin
      : END
4220  dat=stack(stacktop)
4230  RETURN
4240  :
4250  :
4260  :
4270  :
4280  REM *** getcomline ***
4290  REM
4300  PRINT : INPUT "What now ? ",in$
4310  PRINT
4320  comline$=LOWER$(in$)
4330  RETURN
4340  :
4350  :
4360  REM *** processcomline ***
4370  REM
4380  REM remember to ensure that strings come before substri
      ngs - eg. "take off" must be defined before "take" or it
       would never be reached.
4390  trig=false
4400  lin=locline(objloc(0)) : GOSUB 4480 : REM *triggers*
4410  IF NOT(trig) THEN lin=locline(0) : GOSUB 4480 : REM *tr
      iggers*
4420  IF NOT(trig) THEN PRINT "Sorry I do not understand that
      ." ELSE GOSUB 5430 : REM *actions*
4430  RETURN
4440  :
4450  :
4460  REM *** triggers ***
4470  REM
4480  type$="?" 'dummy value
4490  WHILE NOT(trig) AND INSTR("LOEF",type$)=0
4500    GOSUB 20 : REM *restorelin*
4510    READ type$
4520    IF type$="T" THEN GOSUB 4600 : REM *triggertest*
4530    lin=lin+lineinc
4540  WEND
4550  RETURN
4560  :
4570  :
4580  REM *** triggertest ***
4590  REM
4600  match=false
4610  trigsleft=true
4620  comlen=LEN(comline$) : xcomline$=" "+comline$+" "     'c
      onstants within loop
4630  WHILE NOT(match) AND trigsleft
4640    READ trig$
4650    IF LEFT$(trig$,1)="*" THEN trigsleft=false : GOTO 467
      0 'quit loop
4660    IF comlen >= LEN(trig$) THEN IF INSTR(xcomline$," "+t
      rig$+" ")<>0 THEN match=true
4670  WEND
4680  WHILE trigsleft
4690    READ trig$     'consume remaining triggers upto expr
4700    IF LEFT$(trig$,1)="*" THEN trigsleft=false
4710  WEND
```

```
4720  IF NOT(match) THEN res=false ELSE expr$=trig$ : GOSUB 2
      770 : REM *evalthis*
4730  trig=res
4740  RETURN
4750  :
4760  :
4770  REM *** updatecountdowns ***
4780  REM
4790  FOR i=0 TO noofcnts
4800    IF NOT(counting(i)) THEN 4860 'test next one
4810    count(i)=count(i)-1
4820    IF count(i)>0 THEN 4860 'no event yet
4830    counting(i)=false
4840    lin=eventlin(i) : GOSUB 20 : REM *restorelin*
4850    GOSUB 5430 : REM *actions*
4860  NEXT
4870  RETURN
4880  :
4890  :
4900  :
4910  :
4920  REM *** assignobj ***
4930  REM
4940  obj=1 'start with first real object (as obj0 is player)

4950  trig=false
4960  WHILE NOT(trig) AND obj<=noofobjs
4970    lin=objline(obj)
4980    GOSUB 5100 : REM *namesearch*
4990    obj=obj+1
5000  WEND
5010  IF NOT(trig) THEN PRINT "You can't do that." : RETURN
5020  obj=obj-1
5030  GOSUB 5220 : REM *suitability*
5040  IF NOT(res) THEN trig=false : PRINT"That is not possibl
      e."
5050  RETURN
5060  :
5070  :
5080  REM *** namesearch ***
5090  REM
5100  type$="?" 'dummy value
5110  WHILE NOT(trig) AND INSTR("OEF",type$)=0
5120    GOSUB 20 : REM *restorelin*
5130    READ type$
5140    IF type$="N" THEN GOSUB 4600 : REM *triggertest*
5150    lin=lin+lineinc
5160  WEND
5170  RETURN
5180  :
5190  :
5200  REM *** suitability ***
5210  REM
5220  lin=objline(obj)
5230  res=false
5240  type$="?" 'dummy value
5250  WHILE NOT(res) AND INSTR("OEF",type$)=0
5260    GOSUB 20 : REM *restorelin*
5270    READ type$
5280    IF type$="S" THEN GOSUB 5360 : REM *suittest*
5290    lin=lin+lineinc
5300  WEND
5310  RETURN
```

```
5320  :
5330  :
5340  REM *** suittest ***
5350  REM
5360  READ suit$
5370  IF suit$=actsuit$ THEN GOSUB 2700 : REM *evalnext*
5380  RETURN
5390  :
5400  :
5410  REM *** actions ***
5420  REM
5430  actline=lin   'maintain a seperate lin for this routine
5440  acttype$="?" 'dummy value
5450  WHILE INSTR("TLOEF",acttype$)=0
5460    lin=actline : GOSUB 20 : REM *restorelin*
5470    READ acttype$
5480    actline=actline+lineinc
5490    IF acttype$<>"A" THEN GOTO 5700 'no more actions for
      this trigger
5500    READ act$
5510    IF act$="SC" THEN GOSUB 5760 : REM *SCore*
5520    IF act$="IN" THEN GOSUB 5820 : REM *INventory*
5530    IF act$="QU" THEN GOSUB 5960 : REM *QUit*
5540    IF act$="IS" THEN GOSUB 6020 : REM *IncScore*
5550    IF act$="AF" THEN GOSUB 6090 : REM *AssignFlag*
5560    IF act$="PR" THEN GOSUB 6160 : REM *PRint*
5570    IF act$="GO" THEN GOSUB 6320 : REM *GO*
5580    IF act$="MO" THEN GOSUB 6250 : REM *MoveObj*
5590    IF act$="GE" THEN GOSUB 6580 : REM *GEt*
5600    IF act$="DR" THEN GOSUB 6660 : REM *DRop*
5610    IF act$="PO" THEN GOSUB 6740 : REM *PutOn*
5620    IF act$="TO" THEN GOSUB 6820 : REM *TakeOff*
5630    IF act$="EX" THEN GOSUB 6900 : REM *EXamine*
5640    IF act$="IC" THEN GOSUB 6990 : REM *InitCounter*
5650    IF act$="HC" THEN GOSUB 7070 : REM *HaltCounter*
5660    IF act$="ZI" THEN GOSUB 7140 : REM *ZapIn*
5670    IF act$="ZO" THEN GOSUB 7220 : REM *ZapOut*
5680    IF act$="LO" THEN GOSUB 7300 : REM *LOad*
5690    IF act$="SA" THEN GOSUB 7440 : REM *SAve*
5700  WEND
5710  RETURN
5720  :
5730  :
5740  REM *** SCore ***
5750  REM
5760  PRINT "You have scored ";score
5770  RETURN
5780  :
5790  :
5800  REM *** INventory ***
5810  REM
5820  loc=0 : GOSUB 2610 : REM *isobjatloc*
5830  IF NOT(res) THEN PRINT "You are not carrying anything."
       : RETURN 'quit
5840  PRINT"You are carrying : "
5850  FOR obj=1 TO noofobjs
5860    IF objloc(obj)<>0 THEN GOTO 5900 'this obj not carrie
      d
5870    PRINT "    ";
5880    lin=objline(obj) : GOSUB 2260 : REM *describe*
5890    IF worn(obj) THEN PRINT" (worn)" ELSE PRINT
5900  NEXT
5910  RETURN
```

```
5920  :
5930  :
5940  REM *** QUit ***
5950  REM
5960  eogame=true
5970  RETURN
5980  :
5990  :
6000  REM *** IncScore ***
6010  REM
6020  READ inc
6030  score=score+inc
6040  RETURN
6050  :
6060  :
6070  REM *** AssignFlag ***
6080  REM
6090  READ flagnum,bool$
6100  IF bool$="T" THEN flag(flagnum)=true ELSE flag(flagnum)
      =false
6110  RETURN
6120  :
6130  :
6140  REM *** PRint ***
6150  REM
6160  READ descr$
6170  linefeed=true
6180  GOSUB 2410 : REM *printdescr*
6190  PRINT
6200  RETURN
6210  :
6220  :
6230  REM *** MOve ***
6240  REM
6250  READ obj,loc
6260  objloc(obj)=loc
6270  worn(obj)=false
6280  RETURN
6290  :
6300  REM *** GO ***
6310  REM
6320  READ dir$
6330  lin=locline(objloc(0))
6340  match=false
6350  type$="?" 'dummy value
6360  WHILE NOT(match) AND INSTR("LOEF",type$)=0
6370    GOSUB 20 : REM *restorelin*
6380    READ type$
6390    IF type$="C" THEN GOSUB 6480 : REM *GO2*
6400    lin=lin+lineinc
6410  WEND
6420  IF NOT(match) OR (match AND NOT(res)) THEN PRINT"You ca
      n not go that way."
6430  RETURN
6440  :
6450  :
6460  REM *** GO2 ***
6470  REM
6480  READ defdir$
6490  IF dir$<>defdir$ THEN RETURN 'quit
6500  match=true
6510  GOSUB 2700 : REM *evalnext*
6520  IF res THEN READ objloc(0)
```

```
6530  RETURN
6540  :
6550  :
6560  REM *** GEt ***
6570  REM
6580  actsuit$="GE" : GOSUB 4940 : REM *assignobj*
6590  IF NOT(trig) THEN RETURN 'quit
6600  IF objloc(obj)=objloc(0) THEN objloc(obj)=0 : PRINT "Ta
      ken." ELSE PRINT"You can not see it here."
6610  RETURN
6620  :
6630  :
6640  REM *** DRop ***
6650  REM
6660  actsuit$="DR" : GOSUB 4940 : REM *assignobj*
6670  IF NOT(trig) THEN RETURN 'quit
6680  IF objloc(obj)=0 THEN objloc(obj)=objloc(0) : worn(obj)
      =false : PRINT"Dropped." ELSE PRINT"You do not have it."

6690  RETURN
6700  :
6710  :
6720  REM *** PutOn ***
6730  REM
6740  actsuit$="PO" : GOSUB 4940 : REM *assignobj*
6750  IF NOT(trig) THEN RETURN 'quit
6760  IF objloc(obj)=0 THEN worn(obj)=true : PRINT"Worn." ELS
      E PRINT"You do not have it."
6770  RETURN
6780  :
6790  :
6800  REM *** TakeOff ***
6810  REM
6820  actsuit$="TO" : GOSUB 4940 : REM *assignobj*
6830  IF NOT(trig) THEN RETURN 'quit
6840  IF worn(obj) THEN worn(obj)=false : PRINT"Removed." ELS
      E PRINT"You are not wearing it."
6850  RETURN
6860  :
6870  :
6880  REM *** EXamine ***
6890  REM
6900  actsuit$="EX" : GOSUB 4940 : REM *assignobj*
6910  IF NOT(trig) THEN RETURN 'quit
6920  IF NOT(objloc(obj)=0 OR objloc(obj)=objloc(0)) THEN PRI
      NT"You see nothing special." ELSE READ descr$ : linefeed
      =true : GOSUB 2410 : REM *printdescr*
6930  lin=objline(obj)
6940  RETURN
6950  :
6960  :
6970  REM *** InitCounter ***
6980  REM
6990  READ cnum,cval
7000  count(cnum)=cval
7010  counting(cnum)=true
7020  RETURN
7030  :
7040  :
7050  REM *** HaltCounter ***
7060  REM
7070  READ cnum
7080  counting(cnum)=false
```

```
7090  RETURN
7100  :
7110  :
7120  REM *** ZapIn ***
7130  REM
7140  READ obj
7150  objloc(obj)=objloc(0)    'move object to current locatio
      n
7160  worn(obj)=false
7170  RETURN
7180  :
7190  :
7200  REM *** ZapOut ***
7210  REM
7220  READ obj
7230  objloc(obj)=-1           'set object location to nowhere

7240  worn(obj)=false
7250  RETURN
7260  :
7270  :
7280  REM *** LOad ***
7290  REM
7300  GOSUB 7570 : REM *getfname*
7310  SPEED WRITE 1
7320  OPENIN file$
7330  INPUT #9,score,nooflocs,noofobjs,maxflag,noofcnts
7340  FOR l=0 TO nooflocs : INPUT #9,visited(l) : NEXT l
7350  FOR l=0 TO noofobjs : INPUT #9,objloc(l),worn(l) : NEXT
       l
7360  FOR l=0 TO maxflag : INPUT #9,flag(l) : NEXT l
7370  FOR l=0 TO noofcnts : INPUT #9,count(l),counting(l) : N
      EXT l
7380  CLOSEIN
7390  RETURN
7400  :
7410  :
7420  REM *** SAve ***
7430  REM
7440  GOSUB 7570 : REM *getfname*
7450  OPENOUT file$
7460  WRITE #9,score,nooflocs,noofobjs,maxflag,noofcnts
7470  FOR l=0 TO nooflocs : WRITE #9,visited(l) : NEXT l
7480  FOR l=0 TO noofobjs : WRITE #9,objloc(l),worn(l) : NEXT
       l
7490  FOR l=0 TO maxflag : WRITE #9,flag(l) : NEXT l
7500  FOR l=0 TO noofcnts : WRITE #9,count(l),counting(l) : N
      EXT l
7510  CLOSEOUT
7520  RETURN
7530  :
7540  :
7550  REM *** getfname ***
7560  REM
7570  IF LEN(comline$)<6 THEN file$="" : RETURN    'default na
      me
7580  comline$=MID$(comline$+STRING$(16," "),6,16)    'extract
       filename
7590  file$=""
7600  FOR l=1 TO LEN(comline$)    'remove any quote marks
7610     IF MID$(comline$,l,1)<>CHR$(34) THEN file$=file$+MID
      $(comline$,l,1) ELSE file$=file$+"*"
7620  NEXT l
```

```
7630  RETURN
7640  :
7650  :
7660  :
7670  :
7680  THIS LINE GENERATES AN ERROR
7690  DATA L,0
7700  DATA  T,n,north,*F4
7710  DATA   A,GO,N
7720  DATA   A,ZI,12
7730  DATA  T,n,north,*
7740  DATA   A,GO,N
7750  DATA  T,e,east,*F4
7760  DATA   A,GO,E
7770  DATA   A,ZI,12
7780  DATA  T,e,east,*
7790  DATA   A,GO,E
7800  DATA  T,s,south,*F4
7810  DATA   A,GO,S
7820  DATA   A,ZI,12
7830  DATA  T,s,south,*
7840  DATA   A,GO,S
7850  DATA  T,w,west,*F4
7860  DATA   A,GO,W
7870  DATA   A,ZI,12
7880  DATA  T,w,west,*
7890  DATA   A,GO,W
7900  DATA  T,u,up,*F4
7910  DATA   A,GO,U
7920  DATA   A,ZI,12
7930  DATA  T,u,up,*
7940  DATA   A,GO,U
7950  DATA  T,d,down,*F4
7960  DATA   A,GO,D
7970  DATA   A,ZI,12
7980  DATA  T,d,down,*
7990  DATA   A,GO,D
8000  DATA  T,wear,put on,*
8010  DATA   A,PO
8020  DATA  T,remove,take off,*
8030  DATA   A,TO
8040  DATA  T,get,take,pick up,catch,*
8050  DATA   A,GE
8060  DATA  T,drop,put,throw,release,*
8070  DATA   A,DR
8080  DATA  T,ex,exam,examine,look at,*
8090  DATA   A,EX
8100  DATA  T,i,inv,inventory,*
8110  DATA   A,IN
8120  DATA  T,score,*
8130  DATA   A,SC
8140  DATA  T,load,*
8150  DATA   A,LO
8160  DATA   A,IS,-1
8170  DATA  T,save,*
8180  DATA   A,SA
8190  DATA  T,eat bread,eat loaf,*C5
8200  DATA   A,PR,yummy yummy!
8210  DATA   A,ZO,5
8220  DATA  T,eat cheese,*C4
8230  DATA   A,PR,Far too mouldy!
8240  DATA  T,eat,*
8250  DATA   A,PR,No thanks.
```

```
8260  DATA  T,feed cat,drop mouse,release mouse,*C6.012
8270  DATA   A,PR,The cat takes both mouse and cheese and the
      n runs off.
8280  DATA   A,IS,10
8290  DATA   A,AF,4,F
8300  DATA   A,ZO,6
8310  DATA   A,ZO,12
8320  DATA   A,HC,0
8330  DATA  T,hit,kill,attack,*
8340  DATA   A,PR,Sorry. No violence is allowed in this game.

8350  DATA  T,show,give,accuse,feed,*
8360  DATA   A,PR,Not interested.
8370  DATA  T,quit,*
8380  DATA   A,SC
8390  DATA   A,QU
8400  DATA L,1
8410  DATA  D,*,In the middle of the village green.
8420  DATA  D,*-V1,Beyond the north end of the green you can
      see your home - the mill. To east is the inn. The church
       where all this trouble began is to the south. You can h
      ear the goats making goaty noises to the west.
8430  DATA  T,inn,*-F6
8440  DATA   A,GO,E
8450  DATA  T,e,east,inn,*F6
8460  DATA   A,PR,The innkeeper refuses to let you enter the
      inn.
8470  DATA  T,e,east,inn,*-F11
8480  DATA   A,GO,E
8490  DATA   A,AF,11,T
8500  DATA   A,IC,1,100
8510  DATA  T,n,north,*-F11
8520  DATA   A,GO,N
8530  DATA   A,AF,11,T
8540  DATA   A,IC,1,100
8550  DATA  T,w,west,*-F11
8560  DATA   A,GO,W
8570  DATA   A,AF,11,T
8580  DATA   A,IC,1,100
8590  DATA  T,s,south,*-F11
8600  DATA   A,GO,S
8610  DATA   A,AF,11,T
8620  DATA   A,IC,1,100
8630  DATA  C,N,*,4
8640  DATA  C,S,*,6
8650  DATA  C,E,*,2
8660  DATA  C,W,*,5
8670  DATA L,2
8680  DATA  D,*,The bar in the Melbourne Inn.
8690  DATA  D,*F5,Some fool has balanced a pointed hat on top
       of the kitchen door!
8700  DATA  D,*-V2,The inn is closed at the moment due to the
       unexplained spoiling of the ale. The kitchen door sits
      slightly ajar at the east end of the room.
8710  DATA  T,balance hat,put hat,place hat,*C1
8720  DATA   A,PR,You manage to balance the hat on top of the
       kitchen door.
8730  DATA   A,AF,5,T
8740  DATA   A,ZO,1
8750  DATA  T,e,east,kitchen,*F5
8760  DATA   A,PR,The hat falls to the floor.
8770  DATA   A,ZI,1
8780  DATA   A,AF,5,F
```

```
8790  DATA   A,GO,E
8800  DATA  T,get hat,*
8810  DATA   A,ZI,1
8820  DATA   A,AF,5,F
8830  DATA   A,GE
8840  DATA  T,call,shout,*-F5
8850  DATA   A,PR,"The innkeeper comes into the room, but see
      ing you he crosses himself and dashes back into the kitc
      hen."
8860  DATA  T,call,shout,*F5
8870  DATA   A,PR,The innkeeper pushes the kitchen door open
      and steps in the room.
8880  DATA   A,PR,PLOP! The hat lands on his head. It looks l
      ike a thimble on a giants head. He throws you and your h
      at out of the inn.
8890  DATA   A,IS,10
8900  DATA   A,GO,W
8910  DATA   A,ZI,1
8920  DATA   A,AF,5,F
8930  DATA   A,AF,6,T
8940  DATA  T,leave,out,*
8950  DATA   A,GO,W
8960  DATA  C,E,*,3
8970  DATA  C,W,*,1
8980  DATA L,3
8990  DATA  D,*,Kitchen in the Melbourne.
9000  DATA  D,*-V3,It seems surprisingly empty. Perhaps the i
      nnkeeper is a little short of money at the moment.
9010  DATA  T,steal cheese,*
9020  DATA   A,GE
9030  DATA  1,cheese,*04
9040  DATA   A,PR,GET OFF MY CHEESE! What do you mean openly
      trying to take my food!
9050  DATA  T,give,*C10.05
9060  DATA   A,PR,The innkeeper says "Oh goody goody. I can b
      ake some more loaves now." and grabs your sack of flour.
       "You can have this loaf."
9070  DATA   A,MO,5,0
9080  DATA   A,ZO,10
9090  DATA  T,loaf,bread,*05
9100  DATA   A,PR,GET OFF MY BREAD! What do you mean trying t
      o take my last loaf!
9110  DATA  T,leave,out,bar,*
9120  DATA   A,GO,W
9130  DATA  C,W,*,2
9140  DATA L,4
9150  DATA  D,*,At the north end of the village green.
9160  DATA  D,*-V4,You remember coming here in the past to th
      row rotten food (and the odd brick) at people in the sto
      cks. This is where they always burnt witches. You rememb
      er throwing on wood... Those were good times.
9170  DATA  T,put hat,place hat,try hat,*C1.F10
9180  DATA   A,PR,It fits!
9190  DATA   A,PR,A crowd of villagers gather. "The blacksmit
      h is the witch!" screams the priest. "Burn him! Burn him
      ! Burn him!". And they do. Everyone has a real good time
      ... roasting chestnuts and potatoes in the fire. You hav
      e cleared yourself.
9200  DATA   A,IS,20
9210  DATA   A,SC
9220  DATA   A,QU
9230  DATA  T,road,*
9240  DATA   A,GO,E
```

```
9250  DATA  C,S,*,1
9260  DATA  C,E,*,14
9270  DATA L,5
9280  DATA  D,*,An area of grazing land.
9290  DATA  T,feed,cheese,*C4
9300  DATA   A,PR,A goat snatches your cheese and eats it.
9310  DATA   A,ZO,4
9320  DATA  T,feed,bread,loaf,*C5
9330  DATA   A,PR,A goat snatches your bread and eats it.
9340  DATA   A,ZO,5
9350  DATA  C,E,*,1
9360  DATA L,6
9370  DATA  D,*,At the southern end of the village green.
9380  DATA  D,*-V6,A foul odour drifts from the direction of
      the meadow at the west of the green.
9390  DATA  T,meadow,*
9400  DATA   A,GO,W
9410  DATA  T,church,*
9420  DATA   A,GO,S
9430  DATA  C,W,*,8
9440  DATA  C,N,*,1
9450  DATA  C,S,*,10
9460  DATA L,7
9470  DATA  D,*,At the edge of the village pond.
9480  DATA  D,*-V7,The pond is stagnant and smells foul.
9490  DATA  T,put toad,drop toad,throw toad,release toad,*C2.
      -F3
9500  DATA   A,DR
9510  DATA   A,PR,Splash...! The toad turns into a small boy.
       "Where am I? What happened?" he mutters as he climbs ou
      t of the pond. "Last thing I remember I was strolling in
       the woods!"
9520  DATA   A,AF,3,T
9530  DATA   A,IS,5
9540  DATA  T,feed duck,feed ducks,*C5
9550  DATA   A,PR,The ducks gobble up the bread and leave. Th
      ere was something hidden in their nest.
9560  DATA   A,ZO,3
9570  DATA   A,ZO,5
9580  DATA   A,Z1,14
9590  DATA   A,IS,5
9600  DATA  T,fill,water,pond,*
9610  DATA   A,PR,It is impossible to get at the water.
9620  DATA  C,S,*,8
9630  DATA L,8
9640  DATA  D,*V8,In the meadow.
9650  DATA  D,*-V8,In the middle of a meadow. To the north th
      e ground becomes marshy. At the western end of the meado
      w you can see a wooden building.
9660  DATA  C,N,*,7
9670  DATA  C,W,*,9
9680  DATA  C,E,*,6
9690  DATA L,9
9700  DATA  D,*,At the blacksmith's forge.
9710  DATA  D,*-V9,You remember coming here often in the past
      . The blacksmith always repairs everyones tools. He is a
       pretty good all round handyman.
9720  DATA  T,show gold,give gold,show coins,give coins,accus
      e,*C14
9730  DATA   A,PR,The blacksmith bursts into tears. "Alright.
       I confess - I stole the woodcutter's gold." he blurts o
      ut.
9740  DATA   A,PR,A crowd of villagers rush up and drag the p
```

```
      leading blacksmith away.
9750  DATA    A,IS,20
9760  DATA    A,AF,10,T
9770  DATA    A,MO,16,4
9780  DATA   T,show,give,accuse,forge,anvil,*
9790  DATA    A,PR,He shouts "Be off with you! Little witch.".

9800  DATA  C,E,*,8
9810  DATA L,10
9820  DATA  D,*,In the church nave.
9830  DATA  D,*-V10,The secret entrance to the crypt you foun
      d lies open. High above you is the belfry balcony. The f
      ront door is to the north. In the south wall is a small
      door.
9840  DATA  T,hat,*F2.09
9850  DATA   A,PR,Tuq...
9860  DATA   A,Z1,1
9870  DATA   A,AF,1,F
9880  DATA   A,ZO,9
9890  DATA   A,GE
9900  DATA   A,PR,The priest storms off.
9910  DATA  T,crypt,*
9920  DATA   A,GO,D
9930  DATA  T,belfry,*
9940  DATA   A,GO,U
9950  DATA  C,N,*,6
9960  DATA  C,S,*,13
9970  DATA  C,D,*,12
9980  DATA  C,U,*,11
9990  DATA L,11
10000 DATA  D,*,In the belfry.
10010 DATA  D,*-V11,Far below you can see the church nave.
10020 DATA  T,ring bell,*-(F1/F2)
10030 DATA   A,PR,The bell tolls and nearly deafens you!
10040 DATA   A,MO,9,10
10050 DATA   A,PR,You see the priest run into the church dire
      ctly below you.
10060 DATA   A,AF,1,T
10070 DATA  T,ring bell,*
10080 DATA   A,PR,Ding dong...!
10090 DATA  T,drop hat,throw hat,*C1.F1
10100 DATA   A,PR,The hat drops from the belfry and lands ove
      r the priest's head.
10110 DATA   A,IS,10
10120 DATA   A,AF,2,T
10130 DATA   A,ZO,1
10140 DATA  T,drop hat,throw hat,*C1.-F1
10150 DATA   A,PR,Weeee.... it falls from the belfrey.
10160 DATA   A,MO,1,10
10170 DATA  T,nave,leave,*
10180 DATA   A,GO,D
10190 DATA  C,D,*,10
10200 DATA L,12
10210 DATA  D,*,In the  secret crypt.
10220 DATA  D,*-V12,The air is icy cold. The floor is intrica
      tely carved with strange markings. On the wall are lit t
      orches.
10230 DATA  T,torches,torch,wall,walls,*
10240 DATA   A,PR,A couple of torches are firmly attatched to
       the wall by brackets.
10250 DATA  T,bracket,brackets,*
10260 DATA   A,PR,Just plain iron brackets.
10270 DATA  T,floor,markings,*
```

```
10280 DATA   A,PR,You can make no sense of the markings but y
      ou can see they are well carved.
10290 DATA  T,leave,*
10300 DATA   A,GO,U
10310 DATA  C,U,*,10
10320 DATA L,13
10330 DATA  D,*,In the churchyard.
10340 DATA  T,dig,grave,headstone,tomb,*
10350 DATA   A,PR,Careful... they bury grave robbers around h
      ere!
10360 DATA  T,n,north,church,*-F4
10370 DATA   A,PR,You trip over a black cat which appears fro
      m behind a headstone!
10380 DATA   A,ZI,12
10390 DATA   A,AF,4,T
10400 DATA   A,IC,0,20
10410 DATA  T,church,*
10420 DATA   A,GO,N
10430 DATA  T,woods,*
10440 DATA   A,GO,W
10450 DATA  C,N,*,10
10460 DATA  C,W,*,18
10470 DATA L,14
10480 DATA  D,*,The road.
10490 DATA  D,*-V14,You know this road well. To the north it
      passes by the mill on its way to town.
10500 DATA  T,w,west,*F8
10510 DATA   A,PR,WHOOPS! You tripped and spilt the water.
10520 DATA   A,AF,8,F
10530 DATA  T,follow road,along road,*
10540 DATA   A,GO,N
10550 DATA  C,W,*,4
10560 DATA  C,N,*,15
10570 DATA L,15
10580 DATA  D,*,On a bridge over a stream.
10590 DATA  T,water,stream,fill,down,d,*
10600 DATA   A,PR,The stream is totally inaccessable from her
      e.
10610 DATA  C,S,*,14
10620 DATA  C,N,*,16
10630 DATA L,16
10640 DATA  D,*,The mill.
10650 DATA  T,listen,*-F7
10660 DATA   A,PR,You can hear a faint scurrying noise. You w
      onder if it is a rat.
10670 DATA  T,rafters,*-F7
10680 DATA   A,PR,Something is making noises. You wonder if i
      t is a rat.
10690 DATA  T,rat,rats,*-F7
10700 DATA   A,PR,Rat! Pah! I'm a mouse you fool.
10710 DATA  T,feed mouse,drop cheese,*C4
10720 DATA   A,PR,The mouse scurries down and nibbles the che
      ese.
10730 DATA   A,AF,7,T
10740 DATA   A,ZD,4
10750 DATA  T,catch mouse,get mouse,take mouse,pick up mouse,
      *F7.C1.-(W1/F8)
10760 DATA   A,GE
10770 DATA   A,PR,You caught the mouse in the hat.
10780 DATA  T,catch mouse,get mouse,take mouse,pick up mouse,
      *F7
10790 DATA   A,PR,He slips through your fingers.
10800 DATA  T,cheese,*F7.(06/C6)
```

```
10810 DATA   A,PR,The mouse holds onto the cheese.
10820 DATA  T,flour,sack,grain,bag,mill,*-F9
10830 DATA   A,PR,The miller growls at you!
10840 DATA  T,give,*F8.-F9
10850 DATA   A,PR,"The miller puts the sack down, takes the h
      at and drinks some water.
10860 DATA   A,PR,He says "Ta Lad." and then pours the rest o
      ver his head. As he does this you notice that the hat is
       too small for him to wear.
10870 DATA   A,IS,10
10880 DATA   A,AF,8,F
10890 DATA   A,ZI,10
10900 DATA   A,AF,9,T
10910 DATA  1,give,*F8.F9
10920 DATA   A,PR,He smiles and says "Go away creep."
10930 DATA  C,S,*,15
10940 DATA  C,W,*,17
10950 DATA L,17
10960 DATA  D,*,By the stream.
10970 DATA  D,*-V17,A loud sploshing sound comes from the wat
      er wheel. The water looks as fresh and clear as ever.
10980 DATA  1,fill,*C1.-(W1/C6)
10990 DATA   A,PR,If you insist.
11000 DATA   A,AF,8,T
11010 DATA  T,fill hat,*W1
11020 DATA   A,PR,Gargle...gargle...bubble! You are unable to
       hold your breath any longer and take your head out of t
      he stream.
11030 DATA  T,mill,*
11040 DATA   A,GO,E
11050 DATA  C,E,*,16
11060 DATA L,18
11070 DATA  D,*,In the deep dark woods.
11080 DATA  D,*-V18,Careful! You might get lost.
11090 DATA  C,E,*,13
11100 DATA  C,W,*,19
11110 DATA L,19
11120 DATA  D,*,In the deep dark woods.
11130 DATA  C,E,*,18
11140 DATA  C,S,*,20
11150 DATA  C,W,*,21
11160 DATA L,20
11170 DATA  D,*,In the deep dark woods.
11180 DATA  C,W,*,23
11190 DATA  C,N,*,19
11200 DATA  C,S,*,22
11210 DATA  C,E,*,21
11220 DATA L,21
11230 DATA  D,*,In the deep dark woods.
11240 DATA  C,W,*,20
11250 DATA  C,E,*,21
11260 DATA  C,S,*,23
11270 DATA L,22
11280 DATA  D,*,In the deep dark woods.
11290 DATA  C,N,*,20
11300 DATA  C,E,*,23
11310 DATA L,23
11320 DATA  D,*,In the deep dark woods.
11330 DATA  D,*-V23,As you arrived you thought you saw someon
      e run away.
11340 DATA  C,N,*,20
11350 DATA  C,E,*,24
11360 DATA  C,W,*,22
```

```
11370 DATA L,24
11380 DATA  D,*,In a clearing in the woods.
11390 DATA  D,*-W1,It is sheltered from the wind here and the
      sun is uncomfortably hot.
11400 DATA  D,*-V24,You hear a loud chopping sound to the sou
      th.
11410 DATA  T,s,south,*-V25.W1
11420 DATA   A,1S,10
11430 DATA   A,GO,S
11440 DATA  C,W,*,22
11450 DATA  C,S,*,25
11460 DATA L,25
11470 DATA  D,*,The wood cutter's hut.
11480 DATA  T,hut,*
11490 DATA   A,PR,The woodcutter bars your way. "I've already
       had my gold stolen. I'm not going to lose anything else
      . Keep out!"
11500 DATA  C,N,*,24
11510 DATA O,0
11520 DATA  P,1
11530 DATA O,1
11540 DATA  D,*-F8,a witch's hat.
11550 DATA  D,*F8,a witch's hat full of water.
11560 DATA  P,23
11570 DATA  N,hat,*
11580 DATA  S,EX,*,It has a label on the inside which says 'A
      CME Witch's Hat - SIZE 9'. You wonder who wears a size 9
       hat!
11590 DATA  S,GE,*
11600 DATA  S,DR,*-(F8/C6)
11610 DATA  S,PO,*-(F8/C6)
11620 DATA  S,TO,*
11630 DATA O,2
11640 DATA  D,*-F3,a small wart-covered toad.
11650 DATA  D,*F3,a wet and frightened goat herd.
11660 DATA  P,8
11670 DATA  S,GE,*-F3
11680 DATA  S,DR,*
11690 DATA  N,toad,boy,herd,*
11700 DATA  S,EX,*-F3,a very human looking toad.
11710 DATA  S,EX,*F3,a very toady looking human.
11720 DATA O,3
11730 DATA  D,*,several ducks.
11740 DATA  P,7
11750 DATA  N,ducks,duck,*
11760 DATA  S,EX,*,They seem to be sitting on something.
11770 DATA O,4
11780 DATA  D,*,a small piece of cheese.
11790 DATA  P,3
11800 DATA  N,cheese,*
11810 DATA  S,GE,*
11820 DATA  S,DR,*
11830 DATA  S,EX,*,Looks a bit cheesy!
11840 DATA O,5
11850 DATA  D,*,a loaf of bread.
11860 DATA  P,3
11870 DATA  N,loaf,bread,*
11880 DATA  S,GE,*
11890 DATA  S,DR,*
11900 DATA O,6
11910 DATA  D,*-F7,something moving around in the rafters.
11920 DATA  D,*F7.-C6,a piece of cheese with a mouse attatche
      d to it.
```

```
11930 DATA  D,*F7.C6,the mouse and the cheese in the hat.
11940 DATA  P,16
11950 DATA  N,mouse,cheese,*F7
11960 DATA  S,GE,*
11970 DATA  S,DR,*
11980 DATA O,7
11990 DATA  D,*,a large brass bell.
12000 DATA  P,11
12010 DATA  N,bell,*
12020 DATA  S,EX,*,a large church bell inscribed with the let
      ters 'ring me!'.
12030 DATA O,8
12040 DATA  D,*,a lot of goats.
12050 DATA  P,5
12060 DATA  N,goat,goats,*
12070 DATA  S,EX,*,They are tethered to posts. Strange! They
      seem to have eaten all the good grass they can reach. Pe
      rhaps they have not been moved for a while?
12080 DATA O,9
12090 DATA  D,*-(F1/F2),a very small priest blessing the secr
      et crypt.
12100 DATA  D,*F1.-F2,a very agitated priest looking up at th
      e belfrey.
12110 DATA  D,*F2,a priest wearing a black hat over his head
      and shoulders!
12120 DATA  P,12
12130 DATA  N,priest,*
12140 DATA  S,EX,*,He looks very small to you.
12150 DATA O,10
12160 DATA  D,*,a sack of flour.
12170 DATA  P,-1
12180 DATA  N,sack,bag,flour,*
12190 DATA  S,GE,*
12200 DATA  S,DR,*
12210 DATA  S,EX,*,It is labelled 'MegaMill Flour Co.'
12220 DATA O,11
12230 DATA  D,*-V25,an out of breath woodcutter resting on hi
      s axe.
12240 DATA  D,*-V25.W1,Suddenly the woodcutter snatches the h
      at and trys it on. "I wonder if this will shield me from
       the sun ?" he says. "Pity... not my size." he grumbles
      and replaces the hat on your head.
12250 DATA  D,*V25,the woodcutter hard at work.
12260 DATA  P,25
12270 DATA  N,woodcutter,*
12280 DATA  S,EX,*,A rather hot sweaty woodcutter.
12290 DATA O,12
12300 DATA  D,*F7,a friendly black cat drooling around your a
      nkles.
12310 DATA  D,*-F7,a friendly black cat.
12320 DATA  P,-1
12330 DATA  N,cat,*
12340 DATA  S,EX,*,It looks friendly.
12350 DATA O,13
12360 DATA  D,*,the innkeeper.
12370 DATA  P,3
12380 DATA  N,innkeeper,*
12390 DATA  S,EX,*,He is rather large.
12400 DATA O,14
12410 DATA  D,*,some gold coins!
12420 DATA  P,-1
12430 DATA  N,coins,gold,*
12440 DATA  S,GE,*
12450 DATA  S,DR,*
```

```
12460 DATA  S,EX,*,No. They are not sliced golden egg. They m
      ust have come from somewhere else.
12470 DATA O,15
12480 DATA  D,*,The miller humping sacks about.
12490 DATA  P,16
12500 DATA  N,miller,*
12510 DATA  S,EX,*-F9,He looks hot and thirsty.
12520 DATA  S,EX,*F9,He looks wet.
12530 DATA O,16
12540 DATA  D,*-F10,The blacksmith hard at work.
12550 DATA  D,*F10,The blacksmith in the stocks.
12560 DATA  P,9
12570 DATA  N,blacksmith,smith,*
12580 DATA  S,EX,*-F10,He looks rather hot.
12590 DATA  S,EX,*F10,He looks stuck.
12600 DATA O,17
12610 DATA  D,*,some stocks.
12620 DATA  P,4
12630 DATA  N,stocks,*
12640 DATA  S,EX,*-F10,There is a brass plaque with 'Made by
      OXO' engraved on it.
12650 DATA  S,EX,*F10,There seems to be a blacksmith in them!
      !!
12660 DATA E,0
12670 DATA  A,PR," "
12680 DATA  A,PR,A crowd of villagers gather round you. The p
      riest points at the cat and says "Look he has a black ca
      t familiar! That proves he is a witch.". They drag you a
      way and test your inflammability.
12690 DATA  A,SC
12700 DATA  A,QU
12710 DATA E,1
12720 DATA  A,PR,The church bell rings. It is midday. The vil
      lagers drag you away and burn you. It was a really jolly
       occasion and people came from miles around to see you.
12730 DATA  A,SC
12740 DATA  A,QU
12750 DATA F
```

CHEXSUM

The unique CHEXSUM program validation.

WHY

When a listing, such as this is keyed in, everybody invariably makes reading and typing mistakes and then spends ages trying to sort out where and what is causing the error (errors!).

Even experienced programmers often cannot identify an error just by listing the relevant line and need to do the tedious job of going back to the book, especially with DATA statements.

Realising that this is a major cause of frustration in keying the programs, we decided to do something about it.

You should key in and save the following listings before you key in the AKS listing.

Using the Chexsum routine you will be able to find out if you have made any keying errors at all and in which lines, before you even run the program.

This means that you need not waste time looking for keying errors, you simply run the routine and look at the display to identify lines containing errors. It's that easy.

The principle behind the routines is a unique chexsum which is calculated on each individual line of the program as you have keyed it in. Compare this chexsum value with the value for that line in the list at the end of the program listing; if they are the same the line is correct, if not there is an error in that line.

WHEN

The simplest method is to enter the CHEXSUM program in now and save it to tape or disk.

You can type in the chexsum program at any time, even if you have started to type in a program. You cannot, of course LOAD in CHEXSUM from tape or disk because it will erase all you have typed so far.

The obvious solution is to MERGE the programs. The CHEXSUM program should be saved onto a separate cassette to allow easy access.

HOW CAN YOU TELL IF CHEXSUM HAS BEEN ENTERED CORRECTLY

After having keyed CHEXSUM the logical thing would be to chexsum

the program to make sure it is correct. But is it possible to to this? If you follow the instructions you will be able to check CHEXSUM.

1. Type and save CHEXSUM.
2. RUN Chexsum and it will check itself.
3. Check output against the table of values at the end of the program.
4. If the program is incorrect, edit the incorrect lines and resave the program.

Below is the listing of CHEXSUM and instructions on its use.

USING CHEXSUM

The greatest problem encountered when typing in programs from a book is errors made by the user. Most of these are picked up when the computer responds to the RUN command with the 'Syntax Error' message. The user then has only to LIST the line and compare it with the line in the book. Unfortunately, some errors are more subtle and not fatal to program operation. These types of errors will cause the program to run, but incorrectly, and the computer will not be able to detect them as such.

ChexSum is a special program which generates a unique sum for each line in a program and a grand total of all line sums. After each program listing is a table of check sums. You need only compare the numbers in the ChexSum table for each program with those generated by ChexSum. If two numbers differ, check that particular line.

1. Type in AKS. Save it to tape or disk with the statement; SAVE "AKS".
2. Reload the program if necessary, using the statement; LOAD "AKS".
3. To join ChexSum to the end of your program, enter the statement; MERGE "CHEXSUM".
4. When merged, enter RUN 60000 to activate ChexSum. The program will prompt:

   OUTPUT TO PRINTER (P) OR SCREEN (S)

Entering a P will cause output to go to the printer, and entering S will cause output to go to the screen.

5. ChexSum will now output the check-sum table for the program. To halt the program press the escape key once, and to restart the output press any key other than escape. When ChexSum has finished you may remove ChexSum from memory with the DELETE instruction. For example:

   DELETE 60000 - 62990

6. Check your grand total with that in the book. If they differ a line has been entered incorrectly. Compare line numbers until you locate the bad ones and then edit them.
7. Repeat steps 4 to 6 until the program is debugged.
9. When the program is running satisfactorily, delete the ChexSum program as described above.
10. Finally save the debugged version onto a clean tape or disk, with:

    SAVE "AKS".

```
 10 =  282
 20 =  1238
 30 =  1205
 40 =  271
 50 =  2273
 60 =  0
 70 =  0
 80 =  0
 90 =  0
100 =  0
110 =  0
120 =  0
130 =  0
140 =  0
150 =  0
160 =  0
170 =  3180
180 =  0
190 =  0
200 =  0
210 =  0
220 =  0
230 =  0
235 =  0
240 =  0
250 =  0
260 =  0
270 =  0
275 =  0
280 =  0
290 =  0
300 =  0
310 =  0
320 =  0
330 =  0
340 =  2720
350 =  1567
360 =  3008
370 =  0
380 =  0
390 =  470
400 =  0
410 =  0
420 =  0
430 =  3482
440 =  3371
450 =  3132
460 =  4402
470 =  4622
480 =  5675
490 =  5265
500 =  915
510 =  4794
520 =  0
530 =  0
540 =  0
550 =  2493
560 =  6226
570 =  2104
580 =  0
590 =  2694
600 =  2696
610 =  0
620 =  0
630 =  0
640 =  0
650 =  0
660 =  6224
670 =  2003
680 =  0
690 =  6444
700 =  4695
710 =  1762
720 =  0
730 =  5953
740 =  1812
750 =  0
760 =  4069
770 =  3755
780 =  5532
790 =  4741
800 =  1742
810 =  0
820 =  4578
830 =  0
840 =  4939
850 =  1265
860 =  0
870 =  932
880 =  0
890 =  0
900 =  0
910 =  138
920 =  1662
930 =  1342
940 =  1736
950 =  2673
960 =  1818
970 =  2100
980 =  2486
990 =  213
1000 =  191
1010 =  2026
1020 =  2881
1030 =  1857
1040 =  0
1050 =  0
1060 =  0
1070 =  0
1080 =  1808
1090 =  710
1100 =  5920
1110 =  1862
1120 =  1650
1130 =  808
1140 =  2955
1150 =  2284
1160 =  213
1170 =  1962
1180 =  201
1190 =  0
1200 =  0
1210 =  0
1220 =  0
1230 =  987
1240 =  5255
1250 =  3241
1260 =  1412
1270 =  201
1280 =  0
1290 =  0
1300 =  0
1310 =  0
1320 =  707
1330 =  2285
1340 =  1783
1350 =  2930
1360 =  2284
1370 =  1650
1380 =  808
1390 =  213
1400 =  1956
1410 =  201
1420 =  0
1430 =  0
1440 =  0
1450 =  0
1460 =  984
1470 =  5246
1480 =  3235
1490 =  2273
1500 =  1862
1510 =  2284
1520 =  1650
1530 =  808
1540 =  3164
1550 =  213
1560 =  5171
1570 =  1406
1580 =  201
1590 =  0
1600 =  0
1610 =  0
1620 =  0
1630 =  684
1640 =  6754
1650 =  2003
1660 =  201
1670 =  0
1680 =  0
1690 =  0
1700 =  0
1710 =  1266
1720 =  1207
1730 =  1650
1740 =  808
1750 =  3334
1760 =  2284
1770 =  213
1780 =  201
1790 =  0
1800 =  0
1810 =  0
1820 =  0
```

1830 = 691
1840 = 5526
1850 = 3375
1860 = 2524
1870 = 201
1880 = 0
1890 = 0
1900 = 0
1910 = 0
1920 = 1862
1930 = 1775
1940 = 176
1950 = 1990
1960 = 1819
1970 = 176
1980 = 1993
1990 = 2125
2000 = 176
2010 = 2000
2020 = 2236
2030 = 176
2040 = 201
2050 = 0
2060 = 0
2070 = 0
2080 = 0
2090 = 1288
2100 = 4297
2110 = 3225
2120 = 2141
2130 = 1464
2140 = 2201
2150 = 7201
2160 = 176
2170 = 201
2180 = 0
2190 = 0
2200 = 0
2210 = 0
2220 = 1793
2230 = 452
2240 = 0
2250 = 0
2260 = 1868
2270 = 2273
2280 = 1938
2290 = 1650
2300 = 808
2310 = 3565
2320 = 1556
2330 = 3553
2340 = 2284
2350 = 213
2360 = 201
2370 = 0
2380 = 0
2390 = 0
2400 = 0

2410 = 3316
2420 = 5184
2430 = 4184
2440 = 2076
2450 = 5771
2460 = 213
2470 = 1443
2480 = 1911
2490 = 5084
2500 = 3787
2510 = 4324
2520 = 3330
2530 = 213
2540 = 213
2550 = 942
2560 = 1650
2570 = 201
2580 = 0
2590 = 0
2600 = 0
2610 = 1370
2620 = 2201
2630 = 3790
2640 = 176
2650 = 201
2660 = 0
2670 = 0
2680 = 0
2690 = 0
2700 = 805
2710 = 1619
2720 = 201
2730 = 0
2740 = 0
2750 = 0
2760 = 0
2770 = 1874
2780 = 4064
2790 = 3815
2800 = 3084
2810 = 1768
2820 = 1762
2830 = 2699
2840 = 0
2850 = 0
2860 = 0
2870 = 0
2880 = 1102
2890 = 2559
2900 = 1430
2910 = 1429
2920 = 5890
2930 = 0
2940 = 1472
2950 = 2705
2960 = 2791
2970 = 4235
2980 = 213

2990 = 1507
3000 = 201
3010 = 0
3020 = 0
3030 = 0
3040 = 0
3050 = 1773
3060 = 3084
3070 = 1473
3080 = 5148
3090 = 5891
3100 = 201
3110 = 0
3120 = 0
3130 = 0
3140 = 0
3150 = 728
3160 = 3421
3170 = 2600
3180 = 1874
3190 = 4266
3200 = 11723
3210 = 213
3220 = 4611
3230 = 1535
3240 = 2963
3250 = 201
3260 = 0
3270 = 0
3280 = 0
3290 = 0
3300 = 1468
3310 = 2742
3320 = 2863
3330 = 3229
3340 = 2924
3350 = 4800
3360 = 4817
3370 = 5665
3380 = 201
3390 = 0
3400 = 0
3410 = 0
3420 = 0
3430 = 1919
3440 = 1339
3450 = 1729
3460 = 1384
3470 = 5798
3480 = 213
3490 = 201
3500 = 0
3510 = 0
3520 = 0
3530 = 0
3540 = 3720
3550 = 1583
3560 = 1732

3570 = 1919
3580 = 2921
3590 = 1729
3600 = 1583
3610 = 6033
3620 = 1994
3630 = 1422
3640 = 1383
3650 = 213
3660 = 1422
3670 = 201
3680 = 0
3690 = 0
3700 = 0
3710 = 0
3720 = 2465
3730 = 201
3740 = 0
3750 = 0
3760 = 0
3770 = 0
3780 = 1643
3790 = 1990
3800 = 3735
3810 = 4869
3820 = 3268
3830 = 213
3840 = 1709
3850 = 2982
3860 = 3043
3870 = 4756
3880 = 4426
3890 = 201
3900 = 0
3910 = 0
3920 = 0
3930 = 0
3940 = 1709
3950 = 4437
3960 = 3654
3970 = 1437
3980 = 1900
3990 = 0
4000 = 1709
4010 = 4439
4020 = 3117
4030 = 3120
4040 = 0
4050 = 3506
4060 = 1422
4070 = 201
4080 = 0
4090 = 0
4100 = 0
4110 = 0
4120 = 2457
4130 = 2522
4140 = 5589

4150 = 201
4160 = 0
4170 = 0
4180 = 0
4190 = 0
4200 = 2523
4210 = 4754
4220 = 2457
4230 = 201
4240 = 0
4250 = 0
4260 = 0
4270 = 0
4280 = 0
4290 = 0
4300 = 1748
4310 = 191
4320 = 1812
4330 = 201
4340 = 0
4350 = 0
4360 = 0
4370 = 0
4380 = 0
4390 = 1478
4400 = 4082
4410 = 4633
4420 = 6125
4430 = 201
4440 = 0
4450 = 0
4460 = 0
4470 = 0
4480 = 2273
4490 = 3164
4500 = 1650
4510 = 808
4520 = 3468
4530 = 2284
4540 = 213
4550 = 201
4560 = 0
4570 = 0
4580 = 0
4590 = 0
4600 = 1565
4610 = 1945
4620 = 7415
4630 = 2678
4640 = 796
4650 = 5363
4660 = 7413
4670 = 213
4680 = 1365
4690 = 4639
4700 = 3942
4710 = 213
4720 = 6051
4730 = 1285
4740 = 201
4750 = 0
4760 = 0
4770 = 0
4780 = 0
4790 = 2000
4800 = 3936
4810 = 2533
4820 = 3423
4830 = 2236
4840 = 3685
4850 = 1363
4860 = 176
4870 = 201
4880 = 0
4890 = 0
4900 = 0
4910 = 0
4920 = 0
4930 = 0
4940 = 5291
4950 = 1478
4960 = 3172
4970 = 2114
4980 = 1837
4990 = 1406
5000 = 213
5010 = 3461
5020 = 1407
5030 = 1858
5040 = 4999
5050 = 201
5060 = 0
5070 = 0
5080 = 0
5090 = 0
5100 = 2273
5110 = 3088
5120 = 1650
5130 = 808
5140 = 3462
5150 = 2284
5160 = 213
5170 = 201
5180 = 0
5190 = 0
5200 = 0
5210 = 0
5220 = 2114
5230 = 1370
5240 = 2273
5250 = 2980
5260 = 1650
5270 = 808
5280 = 3159
5290 = 2284
5300 = 213
5310 = 201
5320 = 0
5330 = 0
5340 = 0
5350 = 0
5360 = 811
5370 = 3735
5380 = 201
5390 = 0
5400 = 0
5410 = 0
5420 = 0
5430 = 5628
5440 = 2585
5450 = 2334
5460 = 3227
5470 = 1120
5480 = 3110
5490 = 5357
5500 = 670
5510 = 2553
5520 = 3080
5530 = 2425
5540 = 2878
5550 = 3128
5560 = 2472
5570 = 2277
5580 = 2753
5590 = 2378
5600 = 2330
5610 = 2548
5620 = 2802
5630 = 2883
5640 = 3162
5650 = 3230
5660 = 2677
5670 = 2637
5680 = 2444
5690 = 2337
5700 = 213
5710 = 201
5720 = 0
5730 = 0
5740 = 0
5750 = 0
5760 = 2502
5770 = 201
5780 = 0
5790 = 0
5800 = 0
5810 = 0
5820 = 2372
5830 = 5302
5840 = 1905
5850 = 2201
5860 = 4423
5870 = 446
5880 = 3703
5890 = 2821
5900 = 176
5910 = 201
5920 = 0
5930 = 0
5940 = 0
5950 = 0
5960 = 1587
5970 = 201
5980 = 0
5990 = 0
6000 = 0
6010 = 0
6020 = 680
6030 = 2294
6040 = 201
6050 = 0
6060 = 0
6070 = 0
6080 = 0
6090 = 1715
6100 = 6383
6110 = 201
6120 = 0
6130 = 0
6140 = 0
6150 = 0
6160 = 887
6170 = 1793
6180 = 1736
6190 = 191
6200 = 201
6210 = 0
6220 = 0
6230 = 0
6240 = 0
6250 = 1182
6260 = 2003
6270 = 2029
6280 = 201
6290 = 0
6300 = 0
6310 = 0
6320 = 677
6330 = 2530
6340 = 1565
6350 = 2273
6360 = 3251
6370 = 1650
6380 = 808
6390 = 2287
6400 = 2284
6410 = 213
6420 = 6079
6430 = 201
6440 = 0
6450 = 0
6460 = 0

6470 = 0
6480 = 980
6490 = 2750
6500 = 1490
6510 = 1556
6520 = 2023
6530 = 201
6540 = 0
6550 = 0
6560 = 0
6570 = 0
6580 = 2934
6590 = 2249
6600 = 7869
6610 = 201
6620 = 0
6630 = 0
6640 = 0
6650 = 0
6660 = 2944
6670 = 2249
6680 = 9642
6690 = 201
6700 = 0
6710 = 0
6720 = 0
6730 = 0
6740 = 2953
6750 = 2249
6760 = 6857
6770 = 201
6780 = 0
6790 = 0
6800 = 0
6810 = 0
6820 = 2957
6830 = 2249
6840 = 7230
6850 = 201
6860 = 0
6870 = 0
6880 = 0
6890 = 0
6900 = 2951
6910 = 2249
6920 = 12249
6930 = 2114
6940 = 201
6950 = 0
6960 = 0
6970 = 0
6980 = 0
6990 = 1406
7000 = 2147
7010 = 2491
7020 = 201
7030 = 0
7040 = 0
7050 = 0
7060 = 0
7070 = 801
7080 = 2566
7090 = 201
7100 = 0
7110 = 0
7120 = 0
7130 = 0
7140 = 681
7150 = 5691
7160 = 2029
7170 = 201
7180 = 0
7190 = 0
7200 = 0
7210 = 0
7220 = 681
7230 = 4966
7240 = 2029
7250 = 201
7260 = 0
7270 = 0
7280 = 0
7290 = 0
7300 = 1549
7310 = 501
7320 = 761
7330 = 5049
7340 = 3977
7350 = 4812
7360 = 3496
7370 = 5159
7380 = 136
7390 = 201
7400 = 0
7410 = 0
7420 = 0
7430 = 0
7440 = 1549
7450 = 762
7460 = 5103
7470 = 4031
7480 = 4866
7490 = 3550
7500 = 5213
7510 = 137
7520 = 201
7530 = 0
7540 = 0
7550 = 0
7560 = 0
7570 = 4448
7580 = 5130
7590 = 854
7600 = 4618
7610 = 7566
7620 = 455
7630 = 201
7640 = 0
7650 = 0
7660 = 0
7670 = 0
7680 = 1996
7690 = 340
7700 = 1217
7710 = 553
7720 = 587
7730 = 1095
7740 = 553
7750 = 1082
7760 = 544
7770 = 587
7780 = 960
7790 = 544
7800 = 1230
7810 = 558
7820 = 587
7830 = 1108
7840 = 558
7850 = 1122
7860 = 562
7870 = 587
7880 = 1000
7890 = 562
7900 = 898
7910 = 560
7920 = 587
7930 = 776
7940 = 560
7950 = 1092
7960 = 543
7970 = 587
7980 = 970
7990 = 543
8000 = 1459
8010 = 440
8020 = 1852
8030 = 444
8040 = 2458
8050 = 421
8060 = 2601
8070 = 431
8080 = 2591
8090 = 438
8100 = 1918
8110 = 432
8120 = 926
8130 = 431
8140 = 802
8150 = 436
8160 = 575
8170 = 817
8180 = 429
8190 = 2170
8200 = 1706
8210 = 547
8220 = 1472
8230 = 1869
8240 = 700
8250 = 1403
8260 = 3911
8270 = 5409
8280 = 578
8290 = 626
8300 = 548
8310 = 593
8320 = 512
8330 = 1859
8340 = 4417
8350 = 2426
8360 = 1949
8370 = 837
8380 = 431
8390 = 447
8400 = 341
8410 = 3538
8420 = 18611
8430 = 880
8440 = 544
8450 = 1453
8460 = 4893
8470 = 1542
8480 = 544
8490 = 686
8500 = 703
8510 = 1308
8520 = 553
8530 = 686
8540 = 703
8550 = 1213
8560 = 562
8570 = 686
8580 = 703
8590 = 1321
8600 = 558
8610 = 686
8620 = 703
8630 = 543
8640 = 550
8650 = 532
8660 = 553
8670 = 342
8680 = 2940
8690 = 6313
8700 = 13386
8710 = 3209
8720 = 5666
8730 = 641
8740 = 543
8750 = 1869
8760 = 2923
8770 = 537
8780 = 627

8790 = 544
8800 = 1055
8810 = 537
8820 = 627
8830 = 421
8840 = 1573
8850 = 10061
8860 = 1528
8870 = 6579
8880 = 10713
8890 = 578
8900 = 562
8910 = 537
8920 = 627
8930 = 642
8940 = 1299
8950 = 562
8960 = 533
8970 = 549
8980 = 343
8990 = 2695
9000 = 9243
9010 = 1576
9020 = 421
9030 = 1138
9040 = 6022
9050 = 1155
9060 = 11540
9070 = 626
9080 = 591
9090 = 1490
9100 = 5710
9110 = 1652
9120 = 562
9130 = 550
9140 = 344
9150 = 3811
9160 = 18838
9170 = 3063
9180 = 1179
9190 = 20966
9200 = 579
9210 = 431
9220 = 447
9230 = 808
9240 = 544
9250 = 545
9260 = 583
9270 = 345
9280 = 2510
9290 = 1574
9300 = 4147
9310 = 546
9320 = 1926
9330 = 4036
9340 = 547
9350 = 531
9360 = 346
9370 = 4144
9380 = 7726
9390 = 1023
9400 = 562
9410 = 1023
9420 = 558
9430 = 556
9440 = 540
9450 = 593
9460 = 347
9470 = 3222
9480 = 4001
9490 = 4754
9500 = 431
9510 = 15077
9520 = 639
9530 = 534
9540 = 2383
9550 = 7991
9560 = 545
9570 = 547
9580 = 589
9590 = 534
9600 = 1877
9610 = 3873
9620 = 552
9630 = 348
9640 = 1763
9650 = 12337
9660 = 546
9670 = 557
9680 = 536
9690 = 349
9700 = 2757
9710 = 12728
9720 = 5162
9730 = 9686
9740 = 6724
9750 = 579
9760 = 685
9770 = 680
9780 = 3135
9790 = 4195
9800 = 538
9810 = 389
9820 = 2079
9830 = 14996
9840 = 1005
9850 = 929
9860 = 537
9870 = 623
9880 = 551
9890 = 421
9900 = 2576
9910 = 948
9920 = 543
9930 = 1030
9940 = 560
9950 = 545
9960 = 596
9970 = 580
9980 = 596
9990 = 390
10000 = 1628
10010 = 4043
10020 = 1677
10030 = 4007
10040 = 679
10050 = 5962
10060 = 637
10070 = 1265
10080 = 1500
10090 = 2410
10100 = 6265
10110 = 578
10120 = 638
10130 = 543
10140 = 2455
10150 = 3621
10160 = 671
10170 = 1381
10180 = 543
10190 = 578
10200 = 391
10210 = 2224
10220 = 10161
10230 = 2801
10240 = 6534
10250 = 2009
10260 = 2870
10270 = 1836
10280 = 7329
10290 = 911
10300 = 560
10310 = 595
10320 = 392
10330 = 2053
10340 = 2748
10350 = 4801
10360 = 1943
10370 = 6383
10380 = 587
10390 = 640
10400 = 655
10410 = 1023
10420 = 553
10430 = 942
10440 = 562
10450 = 588
10460 = 605
10470 = 393
10480 = 1159
10490 = 7794
10500 = 1126
10510 = 4046
10520 = 630
10530 = 2526
10540 = 553
10550 = 552
10560 = 593
10570 = 394
10580 = 2676
10590 = 2724
10600 = 4764
10610 = 597
10620 = 594
10630 = 395
10640 = 1167
10650 = 1211
10660 = 6245
10670 = 1315
10680 = 5393
10690 = 1369
10700 = 3019
10710 = 2628
10720 = 4903
10730 = 643
10740 = 546
10750 = 5566
10760 = 421
10770 = 3405
10780 = 4923
10790 = 3371
10800 = 1560
10810 = 3463
10820 = 2961
10830 = 2814
10840 = 1157
10850 = 6683
10860 = 11588
10870 = 578
10880 = 630
10890 = 585
10900 = 645
10910 = 1112
10920 = 3517
10930 = 598
10940 = 604
10950 = 396
10960 = 1640
10970 = 9258
10980 = 1401
10990 = 1787
11000 = 644
11010 = 1294
11020 = 10327
11030 = 816
11040 = 544
11050 = 585
11060 = 397
11070 = 2436
11080 = 3143
11090 = 582
11100 = 606

11110 = 398
11120 = 2436
11130 = 587
11140 = 594
11150 = 599
11160 = 390
11170 = 2436
11180 = 601
11190 = 597
11200 = 596
11210 = 581
11220 = 391
11230 = 2436
11240 = 598
11250 = 581
11260 = 597
11270 = 392
11280 = 2436
11290 = 589
11300 = 583
11310 = 393
11320 = 2436
11330 = 5547
11340 = 589
11350 = 584
11360 = 600
11370 = 394
11380 = 2785
11390 = 6895
11400 = 4691
11410 = 1524
11420 = 578
11430 = 558
11440 = 600
11450 = 599
11460 = 395
11470 = 2396
11480 = 723
11490 = 10410
11500 = 593
11510 = 343
11520 = 345
11530 = 344
11540 = 1762
11550 = 3008
11560 = 397
11570 = 697
11580 = 9129
11590 = 525
11600 = 955
11610 = 964
11620 = 548
11630 = 345
11640 = 2971
11650 = 3339
11660 = 352
11670 = 691
11680 = 535
11690 = 1641
11700 = 3193
11710 = 3269
11720 = 346
11730 = 1740
11740 = 351
11750 = 1385
11760 = 4051
11770 = 347
11780 = 2530
11790 = 347
11800 = 1001
11810 = 525
11820 = 535
11830 = 2292
11840 = 348
11850 = 1750
11860 = 347
11870 = 1352
11880 = 525
11890 = 535
11900 = 349
11910 = 4320
11920 = 4978
11930 = 3908
11940 = 399
11950 = 1723
11960 = 525
11970 = 535
11980 = 350
11990 = 2086
12000 = 394
12010 = 795
12020 = 5828
12030 = 351
12040 = 1699
12050 = 349
12060 = 1393
12070 = 13407
12080 = 352
12090 = 5187
12100 = 5314
12110 = 5737
12120 = 395
12130 = 1043
12140 = 3084
12150 = 392
12160 = 1792
12170 = 390
12180 = 1736
12190 = 525
12200 = 535
12210 = 3574
12220 = 393
12230 = 5031
12240 = 17192
12250 = 3237
12260 = 399
12270 = 1484
12280 = 3575
12290 = 394
12300 = 5156
12310 = 2461
12320 = 390
12330 = 692
12340 = 2298
12350 = 395
12360 = 1730
12370 = 347
12380 = 1341
12390 = 2290
12400 = 396
12410 = 1865
12420 = 390
12430 = 1386
12440 = 525
12450 = 535
12460 = 7499
12470 = 397
12480 = 3310
12490 = 399
12500 = 1025
12510 = 3086
12520 = 1884
12530 = 398
12540 = 3182
12550 = 3257
12560 = 353
12570 = 2031
12580 = 2642
12590 = 2142
12600 = 399
12610 = 1547
12620 = 348
12630 = 1043
12640 = 5852
12650 = 4347
12660 = 333
12670 = 587
12680 = 17573
12690 = 431
12700 = 447
12710 = 334
12720 = 14619
12730 = 431
12740 = 447
12750 = 242

TOTAL = 2175197

```
60000 PRINT #8,CHR$(27);"G"; : CLS : LOCATE 11, 1 : PRINT "CH
      EXSUM PROGRAM"
60010 LOCATE 1, 5 : PRINT "OUTPUT TO PRINTER (P) OR SCREEN (S
      ) ?";
60020 X$ = INKEY$ : IF X$ = "" THEN 60020
60030 IF X$ = "P" OR X$ = "p" THEN STREAM = 8 : PRINT "P"
60040 IF X$ = "S" OR X$ = "s" THEN STREAM = 0 : PRINT "S"
60050 LOCATE 1, 10 : INPUT "STARTING LINE NUMBER"; X$ : IF VA
      L( X$ ) > 0 THEN LSTART = VAL( X$ ) ELSE 60050
60060 TOTAL = 0 : LLIMIT = 62990 : MEM = 368 : NEXTMEM = MEM
60070 :
60080 NEXTMEM = PEEK( MEM ) + 256 * PEEK( MEM + 1 ) + NEXTMEM

60090 LN = PEEK( MEM + 2) + 256 * PEEK( MEM + 3 ) : IF LN >=
      LLIMIT THEN 62000 ELSE IF LN < LSTART THEN MEM = NEXTMEM
       : GOTO 60080
60100 MEM = MEM + 4 : CHXSUM = 0 : QUOTE = 0
60110 IF PEEK( MEM ) = 32 THEN MEM = MEM + 1 : GOTO 60110
60120 IF PEEK( MEM ) = 1 OR PEEK( MEM ) = 197 THEN MEM = NEXT
      MEM : GOTO 61000
60130 WHILE MEM < NEXTMEM
60140 TOKEN = PEEK( MEM ) : IF TOKEN = 34 THEN QUOTE = QUOTE
      XOR 1
60150 IF QUOTE = 1 OR TOKEN <> 32 THEN 60170
60160 LASTOK = PEEK( MEM - 1 ) : NEXTOK = PEEK( MEM + 1 ) : G
      OSUB 61500 : IF IGNORE = 1 THEN 60180
60170 CHXSUM = CHXSUM + TOKEN
60180 MEM = MEM + 1 : WEND
61000 PRINT #STREAM, USING "######"; LN; : PRINT #STREAM, "
      "; CHXSUM
61010 TOTAL = TOTAL + CHXSUM : GOTO 60080
61490 :
61500 IGNORE = 0
61510 IF LASTOK = 44 OR LASTOK = 32 OR LASTOK = 40 OR LASTOK
      = 1  OR ( LASTOK > 237 AND LASTOK < 250 ) THEN IGNORE =
      1 : RETURN
61520 IF NEXTOK = 41 OR NEXTOK = 1  OR ( NEXTOK > 237 AND NEX
      TOK < 250 ) THEN IGNORE = 1 : RETURN
61530 RETURN
61990 :
62000 PRINT #STREAM : PRINT #STREAM, "TOTAL =";  TOTAL
62990 :
```

```
60000 =  2809 | 60170 =  2585
60010 =  3059 | 60180 =  1578
60020 =  2061 | 61000 =  3897
60030 =  3313 | 61010 =  2604
60040 =  3297 | 61490 =  0
60050 =  5713 | 61500 =  1022
60060 =  4283 | 61510 =  9964
60070 =  0    | 61520 =  7455
60080 =  4367 | 61530 =  201
60090 =  9211 | 61990 =  0
60100 =  3256 | 62000 =  2964
60110 =  3432 |
60120 =  4964 | TOTAL =  97746
60130 =  1687 |
60140 =  5092 |
60150 =  2819 |
60160 =  6113 |
```

APPENDIX B
BIBLIOGRAPHY

The following is a short list of some useful and interesting books and articles which will provide more information about areas we have covered in this book. If you have found some of the more advanced techniques we have mentioned intriguing, then these sources will be well worth looking at.

Blanc, M. S. and Galley, S. W., 1980: How to fit a large program into a small machine. Creative Computing VOL 6 no 7, 80-87.

Daynes, R., 1982: The video disk interfacing primer. Byte, June, 48-59.

Jackson, Principles of Program Design. Academic Press.

Knuth, Donald. Fundamental Algorithms. Addison Wesley.

Lebling, P. D., Blanc, M. S. and Anderson, T. A., 1979: ZORK: A computer fantasy simulation game. IEEE Computer, April, 51-59.

Lister, A. M. Fundamentals of Operating Systems. Macmillan Publishers Ltd.

Reed, K., 1980: Adventure II — an epic game for non-disc systems. Practical Computing, August 68-75.

Tolkein, J. R. R. The Lord of the Rings. Unwin Paperbacks.

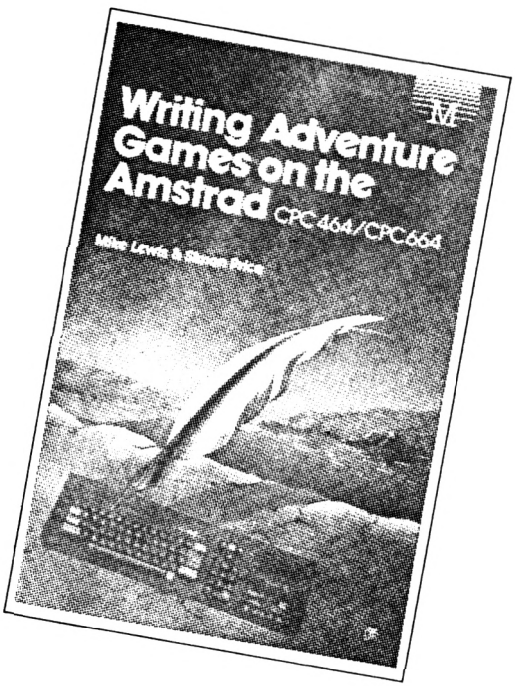
Winston, P. H. Artificial Intelligence. Adison Wesley.

AKS CASSETTE

A cassette containing the Adventure Kernel System is available on cassette from:

Bookpoint Ltd.,
39 Milton Trading Estate,
Abingdon,
OXON, OX14 4TD.

Please send 80p (post & pack) plus £ 3.95.



Writing Adventure Games On The Amstrad

Customer Registration Card

Please fill out this page (or a photocopy of it) and return it so that we may keep you informed of new books, software and special offers. Post to the appropriate address on the back.

Date . . . . . 19. . . . . .

Name . . . . . .

Street & No. . . . . .

City . . . . . Postcode . . . . . .

Model of computer owned . . . . . .

Where did you learn of this book?:

☐ FRIEND ☐ RETAIL SHOP

☐ MAGAZINE (give name) . . . . . .

☐ OTHER (specify) . . . . . .

Age? ☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ 25 and over

How would you rate this book?

QUALITY: ☐ Excellent ☐ Good ☐ Poor

VALUE: ☐ Overprice ☐ Good ☐ Underpriced

What other books and software would you like to see produced for your computer?

. . . . . .

. . . . . .

. . . . . .

EDITION 7 6 5 4 3 2 1

Melbourne House addresses

Put this Registration Card (or photocopy) in an envelope and post it to the appropriate address:

United Kingdom

Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

Australia and New Zealand

Melbourne House (Australia) Pty Ltd
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

WRITING ADVENTURE GAMES ON YOUR AMSTRAD is for every Amstrad owner interested in adventures, whether a beginner or experienced programmer.

This book describes what adventure games are, how to play them, and more importantly how you can write them. And it's all here for you, ready to type in. The book includes a simple adventure game that you will be able to play immediately.

The sample game in this book was written using the authors' own Adventure Kernel System (AKS). This is the heart of all adventure programs, and with this you will be able to write and devise any adventure game of your own.

The mysteries of creating adventures are unravelled for you and simply explained. Mike Lewis and Simon Price show you how to define the problems, break them down into structured elements as well as more advanced techniques such as text compression and much more.

WRITING ADVENTURE GAMES ON YOUR AMSTRAD will enable you to enjoy a complete adventure game immediately as well as opening up the potential of hundreds of new and exciting adventures that you and your friends will be able to write.

£6.95

Melbourne
House
Publishers

ISBN 0-86161-196-9

9 780861 611966

Writing Adventure Games on the Amstrad

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/