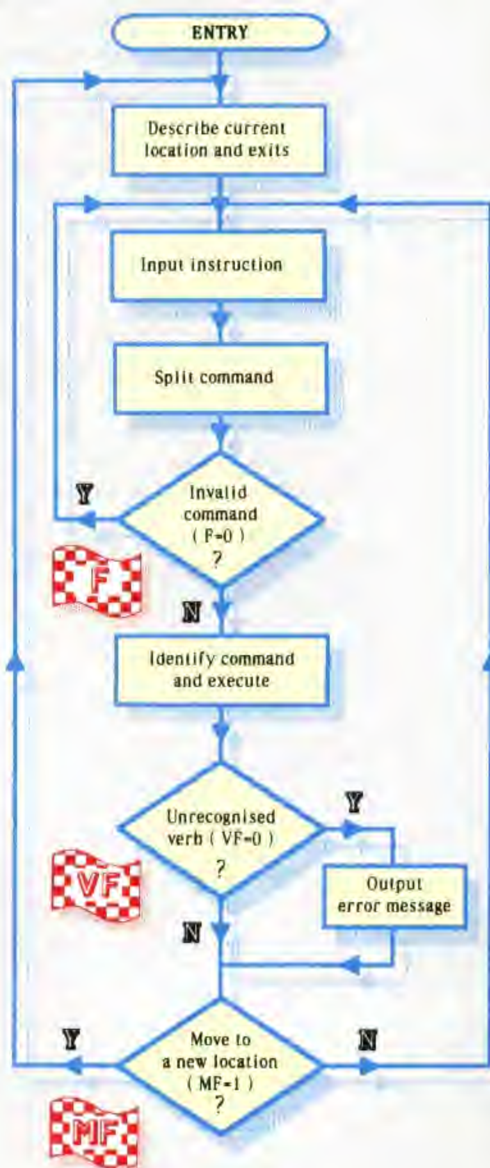
TAKING ORDERS

Up to this point in our adventure game programming project, we have discussed methods of map making, formatting output and moving around the adventure world. In this instalment, we show how the program analyses and obeys instructions given to it by the player.

Chequered Flags

Flags are used widely in programs that have a modular construction. Conditions that involve branches in program control can be tested for within a module but branching on the result of such a test can be delayed until a return to the main program section is made. By setting a variable to a pre-determined value when the test is made, the value of the variable can later be tested within the main program section. Variables used to indicate conditions in this way are termed 'flags'. The flowchart shows the main program loop, as constructed so far, for Haunted Forest. The flag F indicates whether or not a command has a valid format and is set during the 'split command' subroutine. The subroutine used to identify and execute normal commands uses two flags: VF is used to signal that the verb part of the command has been correctly recognised. If, in executing a command, the player moves to a new location, the fact that a move is to be made is signalled by MF. When MF is tested within the main program loop, a value of 1 indicates that the loop should branch back to describe the new location to which the player has moved



IAN McKINNELL

Adventures are usually constructed so that the player can move from location to location, picking up and dropping objects along the way. A set of commands allows the player to perform these simple tasks. The commands we have used are:

GO (direction)	To move between locations
TAKE (object)	To pick up an object
DROP (object)	To put down an object
LIST	To list the objects carried
LOOK	To redisplay the description of the current location
END	To end the game

Variations on these may also be available, such as MOVE instead of GO, or GET instead of TAKE. Part of the fun of playing an adventure game is to determine what words the game will accept. For example, a player might try the command SWIM when in a dry location. If the program responds by telling the player that he cannot swim *here*, then the player could reasonably assume that there are locations where swimming *is* allowed. (Alternatively, the programmer might just want the player to think that!)

The number of commands accepted by a game varies according to the complexity of the game and the amount of effort the programmer has made to cover every eventuality. The most important thing for the designer to do is to make sure the program does not crash if a player tries to enter a command that is not catered for. A failsafe routine that prints 'I don't understand' may be all that's required, bearing in mind that some flexibility should be added so that players can enter commands in different ways. For example, it would be annoying for a program that accepts the command TAKE LAMP to respond to the command TAKE THE LAMP with 'I don't understand'. Adding flexibility will be discussed at greater length later. For the moment, we need to look at the type of instructions that might be given during the game, and devise a routine that will break these down into a form that can be easily interpreted.

COMMAND SPLITTING

No matter what the instruction is, it is very likely that it will be phrased in the *imperative* — such as, GO SOUTH TOWARDS THE RIVER or KILL THE ALIEN. The advantage of this sentence structure is that it is easy to break down: the verb always comes as the first word in the sentence, the object of the verb follows this, and finally there may be some form of qualification of the action. A first stage in the analysis of a command is to separate the verb from