ADVENTURE SPECIFICATION

D.J.Seal and J.G.Thackray

This document describes a pair of programs that can be used for defining and playing ADVENTURE-type games. The first program takes as input a specification of the game and produces a database from it; this database can then be used by the second program to play the game.

Chapter 1

Introduction

1.1 The database – general information

An ADVENTURE database has a name of the form `userid.name` (or `.name` if the userid is your own). It consists of two files, called `userid.name.STAT` and `userid.name.INIT`, which contain the static information and the initial values of the variable information for the game respectively. `Userid.name.STAT` is a direct access file, with format (F,2052,2052), and contains information about the geography, objects, vocabulary and messages of the game. It can be expected to be fairly large for any reasonably sized game (although considerably smaller than the original specification of the game). `Userid.name.INIT` is a sequential file, also with format (F,2052,2052); it contains information about the initial locations, states and properties of the objects and rooms, and initial values of variables. It will occupy 1 track only.

A game can be saved and resumed later. This is done by saving all the variable information in a file, in the same format as in `userid.name.INIT`. The game is then resumed by taking initial values from the saved game file instead of `userid.name.INIT`. Note that a saved game file must be sequential, not a member of a pds.

1.2 Playing the game

To play the game, type

```
CALL DJS6.L:ADVENTUR [database name]
```

If a database name is not specified, you will be prompted for it. The program then always prompts you for the name of a file to restore from (type carriage return if you don't want to restore a game). The game should then start. When it ends you will be returned to Phoenix.

All words input to the program are truncated to 5 characters, except for the purpose of substitution into messages. Words greater than 20 characters long are treated as erroneous. Upper and lower case letters are treated as being the same, again with the exception of words substituted into messages. Commands consist of 1 or 2 words, separated by spaces or the punctuation marks ',.;:?!'; the third and subsequent words on the input line are ignored. Carriage return or a sequence of more than 20 spaces and/or punctuation marks is taken to terminate the command.

The following severe errors may occur:

"Not enough region available for the game" — the program cannot get hold of enough store to do all its file handling properly. This should not happen under MVS.

"Save/initial file – static file mismatch" — the database production program puts a random security code into both halves of the database when it is compiled. At the start of a game, the playing program checks the codes against each other and only allows the game to continue if they match; if they don't, this message is produced. When this occurs, the reason is usually either 1) that the wrong database name or saved game was specified, or 2) that the game has been recompiled since the saved game was produced (usually to remove bugs). This usually changes various offsets within the initial values file, making old saved files invalid.

Under certain limited circumstances, these offsets are guaranteed not to change. In this case, it is possible to stop saved files becoming invalid when the game is recompiled. The procedure for doing this is given in section 1.4 below. It must be emphasised that this procedure must not be followed unless the conditions given in section 1.4 are satisfied; otherwise, severe and difficult to trace errors will occur.

"Severe database error. Please send details to the database writer." or "Too few save areas." — an inconsistency or error has been detected in the database. Please make a note of the circumstances in which the error occurred, and let the database writer know about it. If he fails to find the problem, he should ask one of us (DJS6 or JGT1).

There is also a self-explanatory message to indicate that an invalid saved game file or (rarely) initial values file has been encountered. This is usually caused by giving it a file to restore from that does not contain a saved game at all. The database writer should only be alerted if you did not specify a file to restore from, i.e. if the error occurred in the initial values file.

1.3 Writing a database

Several warnings should be given to anyone contemplating writing a database:

1. It will involve quite a lot of work, not only when writing it, but also later when people find bugs in it. It should be noted that the specification has to include apparently trivial information such as how to TAKE or DROP objects, as well as the more obviously necessary information such as how to implement magic words.
2. You will need quite a lot of filespace, especially for the input to the database production program.
3. The input to the database production program should not be left around for people to read, as the set of messages associated with a game can give away a lot about the game (messages are held in a scrambled form in the database, and so cannot be read there). Three solutions to this problem are:-
   (a) Hold the input on tape and only bring it down to disc for editing and compiling purposes;
   (b) Hold the input on cards, which also solves the filespace problem in part (but not the carrying problem!); and
   (c) Keep the source in files with `UACC NONE` (but note that your `RELATION`s can still read it).

To produce a database, first produce the input file, which should be a pds containing members `STAT` and `INIT`. (Other members may exist: they will be ignored.) The formats of these are defined in chapters 3 and 4 below. Then type:

```
C DJS6.C:DB IN=input file name DB=database name
            [VER=dataset name] [SYMB=4 digit integer]
            [STORE=integer] [TIME=integer]
```

Defaults are: `VER=* SYMB=0050 STORE=100 TIME=5`. No defaults exist for `IN` and `DB`: if they are not present, the command sequence prompts you for them when used online and stops if it used offline.

`VER` specifies the verification dataset, which receives all messages from the program.

`SYMB` specifies the size of the symbol table in kilobytes: the default size should be sufficient for about 3000 labels to be defined. Note that it must consist of precisely 4 digits.

Chapter 7 contains a list of the error numbers produced by the database production program, together with their meanings. It is a good idea to get a listing (with line numbers) of the database source at the same time as compiling it, so that the error messages can be understood. It should also be noted that the program is a two-pass compiler, with the second pass proceeding only if the first pass was error-free, and that some errors can only be detected on the second pass: it is therefore possible for errors only to come to light on the second or later compilations.

It is permissible when compiling a database to give a name of the form `&XYZ`; this will produce a database held in temporary files and should be of use when developing a database. However, the name must not be more than 4 characters long (including the &), as invalid temporary filenames will result otherwise.

1.4 Preventing saved games from becoming invalid

When you recompile the database of the game, there is no need to make saved games invalid, provided the following two conditions are satisfied:

1. All initial values must be the same as for the previous version, i.e. the member `INIT` of the input file must not have been altered.

2. No rooms, objects or variables (including the text variable) may have been added or removed. Note that it is not sufficient that the number of these is unchanged – the individual rooms, objects and variables must be the same ones as before, and their order must not have changed.

Under these circumstances, it is possible to use ZED to prevent saved games becoming invalid when the database is recompiled. The sequence of operations is as follows:

1. First type:

```
 ZED databasename.STAT TO $/FF/L=2052/B=2052 V-; X+; MXLL 2052;
20\#; 4>; DFA//; ?; STOP
```

   A security code of 8 hexadecimal digits will be output by ZED. This should be noted down and used in step (3).

2. Type:

```
 C DJS6.C:DB IN=inputfilename DB=\&A \ldots
```

   to compile the database into temporary files.

3. Type:

```
 ZED \&A.STAT TO .ZZZZZZ/FF/L=2052/B=2052/DA/TRK=5,5 V-; X+;
MXLL2052; 20>; 4\# E//securitycode/ W
```

4. Finally, type:

```
 RENAME .ZZZZZZ AS databasename.STAT
```

Chapter 2

Structure of an Adventure Database

2.1 Data structures and facilities

An ADVENTURE database contains information about the following data structures:

1. Messages
2. Objects
3. Rooms
4. Exits
5. Variables
6. Vocabulary
7. Programs

These are each discussed in detail below, with descriptions of the various facilities available for each.

2.1.1 Messages

Almost every message produced by the game comes from the database, the few exceptions to this rule being messages such as 'It is pitch dark' which are built into the playing program. These are described at the appropriate places below.

Database messages are produced in three main ways:

1. If you have moved to a new room during your last turn, and under certain other circumstances as well, the playing program automatically describes the room the player is currently in before prompting him for a command.
2. A program can contain a command to describe your current room or any particular object, either with or without any objects it holds.
3. A program can contain a command to print a specific message.

An additional feature of messages is that they can be "switched" into one of a number of different endings according to the value of an integer. The main use of this is to vary the description of a room or object according to its state: e.g. a bottle in state 0 might produce the message "There is an empty bottle nearby.", while in state 1 it would produce the message "There is a bottle of water here.". When objects or rooms are being described (cases (1) and (2) above), the program automatically assumes you want to switch its description by its state. When a specific message is being printed (case (3) above), it is possible to specify how it is to be switched.

Finally, it is possible for variable values and/or the words of the player's command to be substituted into a message.

2.1.2 Objects

Up to 255 objects may exist in a game. Of these, the first one is assumed to be the player himself and so has a special role in the game. An object may hold other objects, which may in return hold yet other objects, etc., subject only to the condition that no loops may be formed – e.g. you must not have object A holding object B, which holds object C, which in turn holds object A.

It should be emphasised that no other limitations are imposed on this structure directly by the playing program – e.g. the playing program would find nothing wrong with strange structures such as having the player held by some food. It is the game writer's responsibility to ensure that the game's programs only allow the sorts of structure that he wants to be created.

Each object has a location, which is either a room or a null value. In this last case, the object is said to be destroyed and is not in any room. If an object is held by another, it must have the same location as the object holding it.

An object has three descriptions associated with it, which describe it in the three cases of it being held by nothing, by the "player" object (mainly of use in the INVENTORY command) and by some other object.

Other information held for an object is its state, which is an integer in the range 0-255 and is usually used to switch between different descriptions of the same object, and its properties. These are 16 bits, numbered 0-15, which can be used to indicate such things as whether the object is immovable, whether it is a treasure, etc. Most of these are completely under the database writer's control, but three of them have special significance to the playing program; these are:

If property 0 is set, the program assumes that the object is a light source. Any attempt to describe an object or room will produce the message "It is pitch dark." unless there is a non-hidden, visible light source in the current room or the room is lit (see the corresponding property of rooms).

If property 1 is set, the program assumes that the object is invisible and hides all objects it holds. This means that neither the object itself nor any of its held objects may act as light sources, that the object itself will not be described under any circumstances, and that any held objects will only be described when this is specifically requested (not, for instance, when the room holding the object is described).

If property 2 is set, the program assumes that the object hides any object it holds. This has the same effect as property 1, except that the object itself can act as a light source and can be described.

2.1.3 Rooms

Rooms are in many ways similar to objects: each room has a state in the range 0-255 and 16 properties. The first three of these again have special meanings, the details of which are given below. A room only has two descriptions, a long one which is given if it has not been visited before and a short one which is given if it has. A room may hold objects, but may not itself be held either by other rooms or by objects. There are at most 1023 objects and rooms combined.

The other pieces of information held for a room are a list of exits from it (described in the next section) and a list of the objects contained in it (which includes the "player" object in the current room).

The special meanings of properties 0-2 are:

If property 0 is set, the room is assumed to be lit. See the corresponding property for objects for more details.

If property 1 is set, the program assumes that the room has been visited and so gives the short description when it describes the room. Otherwise, it gives the long description. Also, at the end of each turn, the program sets this property for the current room.

If property 2 is set, the player is assumed to be "disoriented" when he is in the room. The main effects of this are that commands of the type GO BACK will not work, and that he cannot move to an adjacent room by referring to it by name.

2.1.4 Exits

There is a list of these associated with each room. Each exit has a specified direction (which is an integer in the range 1-255) and a destination room. Note that there is no concept of the "opposite" exit to a given exit.

Optionally, an exit can have a program attached to it. This program is obeyed whenever the player goes through the exit, except when he is carried through it by another object. It is not obeyed when objects other than the player pass through the exit. This facility has many uses, the main ones being the production of special messages when the player passes through the exit,

the implementation of random destinations for the exit and the provision of other features such as exits which do not allow a particular object to pass.

2.1.5 Variables

There may be up to 256 integer variables in a game, as well as a single text variable. The text variable can be used to remember strings of up to 126 characters which can be substituted into messages.

The integer variables are halfword integers which may be used to hold any integer in the range -32768 to 32767. A number of instructions are provided for setting, testing and doing arithmetic with them. The first four integer variables have a special role; they can be used to switch messages and their values can be substituted into messages. As it is likely that these facilities will be used frequently in any given game, we recommend that these four variables should not be used for storing any particular values, but that they are used as a workspace in which to do calculations, etc.

2.1.6 Vocabulary

This contains a list of words known to the program. Each word can be given meanings as an object or room, as a direction and as a special word. The last type of meaning is an integer in the range 1-255 which can be tested by a program: this is mainly useful when picking out particular words or classes of word while doing command decoding (e.g. the word TAKE may be followed either by the name of an object or by the word INVENTORY – this is best implemented by giving the word INVENTORY a special meaning).

Each word also has stored for it a meaning as a first word, the most common ones being "Not a valid first word" and "Obey program XXX", and a set of requirements for a second word to follow it. These are general requirements of the form "there must be a second word and it must refer to an object", "there may not be any second word", etc. More complicated requirements (e.g. those for TAKE in the example above) are checked by programs.

Finally, the database contains information on how each word may be abbreviated.

2.1.7 Programs

These are where all the special features of the game and many of the standard ones are implemented. They are lists of instructions which may be invoked in the following six ways:

1. The first word of the command given by the player can have the first word meaning "Obey program XXX", in which case that program will be obeyed at the appropriate point during the turn.

2. If the player passes through an exit that has a program attached to it (and is not being carried by another object), that program will be obeyed.

3. A particular program can be designated as the "welcoming" program. This will be obeyed once at the beginning of the game, before the first command is requested from the player.

4. A particular program can be designated as the "pre- command" program. This is obeyed each turn, between checking the command for validity and actually obeying the command.

5. A particular program can be designated as the "post- command" program. This is also obeyed once each turn, between obeying the command and prompting for the next command.

6. Finally, a program can be called as a subroutine by another program.

The instructions themselves include ones for testing various conditions, for jumping to other instructions, for setting and doing arithmetic with variables and states of objects or rooms, for setting and unsetting properties of objects or rooms, for printing messages and describing objects and rooms, for asking questions of the player, for returning from the program and a few other miscellaneous instructions.

A particular feature to note is that the return instruction in the program can request various things to be done by the playing program. These include requesting that the current room be described before the next command, regardless of whether the player has moved; requesting that a move be to a changed destination or be aborted completely; requesting that processing of the current command be aborted and requesting that the command be reinterpreted and obeyed again after changing the first word meaning of the first word of the command (for this turn only). This last is particularly useful - e.g. it can be used to change the first word meaning to "Not a valid first word" when command decoding shows that the command given makes no sense at all. This will then induce the message "I don't understand that!".

2.2 How the playing program processes the database

In this section, we describe the rough order in which the playing program works. Many details have been omitted for the sake of brevity.

1. The playing program logs its use.
2. It welcomes the player, asks him for names of the database files, allocates these files and opens the database. It then reads the variable information into core and some of the information in the static part of the database.
3. It obeys the welcoming program, if it exists.
4. If the player has moved since his last turn, or a description has been specifically requested, or he's at the beginning of the game, the current room of the player is described.
5. The "visited" property of the current room of the player is set.
6. The program gets a command from the player.
7. It checks the command against the vocabulary for validity. If the command is found to be invalid, it types the message "I don't understand that!" and returns to step 6.
8. It obeys the pre-command program, if one exists.
9. It obeys the command – this may involve stopping or saving the game, starting a new game by stopping this one, then returning to step 2, obeying a program, printing a message, etc.
10. It obeys the post-command program, if one exists.
11. It returns to step 4.

If at any stage in this process there is a request to reinterpret the first word meaning of the first word of the command, the command is re-checked for validity (with the same action as above if it is invalid) and then control is returned to step 9.

If there is a request for processing of the command to be aborted, control is immediately returned to step 4.

2.3 Playing program limits

The following is a fairly comprehensive list of the limits imposed by the playing program and the database structure.

1. <= 255 objects, of which the first must be the "player".
2. <= 1023 objects and rooms combined.
3. 16 properties per object or room.
4. State of object or room in range 0-255.
5. <= 255 directions.
6. <= 255 special meanings of words.
7. <= 256 integer variables, each in the range -32768 to 32767.
8. <= 1 text variable.
9. Overall length of the static section of the database to be <= 256 2052-byte blocks, i.e. about 43 tracks.

If any of these 9 limits are exceeded, this will be detected by the database compilation program and an error message produced.

1. Subroutine nesting – not more than 5 deep. I.e. program A calling program B calling program C calling program D calling program E is OK, further nesting is not.
2. References (see later in this chapter). When resolving a reference, it often happens that the program resolves another reference, then modifies the result (e.g. when asked for the first object in room XYZ, it first resolves XYZ as a room, then finds the first object in that room). This recursion may not go more than 8 deep. Any normal use of references should be well within this limit.

   These two errors are detected by the playing program and produce the message "Severe database error. Please send details to the database writer." if they are encountered. It should be pointed out that this message can also be produced for other causes, notably for corrupted databases and possibly for program bugs in the playing program.
3. There is another source of recursion in the playing program, namely that moving the player through an exit can cause a program to be obeyed, but an instruction in a program can cause the player to be moved through an exit. In other words, the subroutines `MOVE` and `OBEY` of the playing program can call each other. This interaction is allowed up to depth 5, e.g. `OBEY` calling `MOVE` calling `OBEY` calling `MOVE` calling `OBEY` is OK, further nesting is not. If this limit is broken, the playing program will produce the message "Too few save areas."; these should be the only circumstances in which this message is produced.

2.4 Definitions of concepts used in the game specification

This section describes the various concepts used in chapters 3 and 4 to define the allowed input to the database production program.

2.4.1 Labels

A label is a string of any number of alphanumeric characters, the first one of which must be alphabetic. Note that national characters are not allowed. Only the first 8 characters of any label are significant, any remaining ones being ignored. Thus the labels `NOCHOICE` and `NOCHOICE1` are regarded as the same, for instance.

Each label is of one of the following eight types:

1. a direction label or dlabel – refers to a direction.
2. an instruction label or ilabel – refers to an instruction in a program.
3. a message label or mlabel – refers to a message or part of one.
4. an object label or olabel – refers to an object.
5. a property label or plabel – refers to a property.
6. a room label or rlabel – refers to a room.
7. a special meaning label or slabel – refers to a special meaning of a word.
8. a variable label or vlabel – refers to a variable.

Each label must occur in a specific context to be defined, this context depending on the type of the label – e.g. a room label must occur in a `!ROOM` directive. No label may be defined more than once, whether or not the two definitions are for labels of different types.

2.4.2 Words

A word is a string of any number of alphanumeric characters, the first one of which is alphabetic. Only its first 5 characters are used, the rest being ignored. It is not entered into the symbol table and so cannot conflict with a label.

2.4.3 Integers

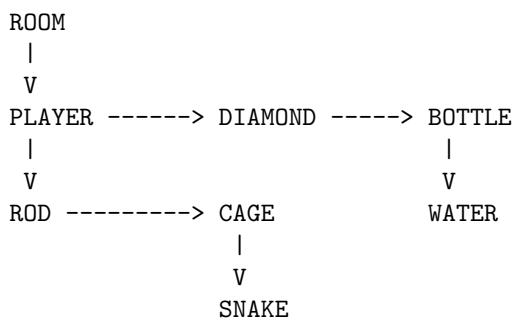
An integer is any unsigned integer in the range 0-32767. When the maximum value of an integer is to be smaller than this, it will be indicated by value in brackets after the word integer – e.g. "integer(255)" denotes an integer in the range 0-255.

2.4.4 References

A reference is a more general way of referring to objects or rooms than a label, allowing one to refer to things such as "the room holding the player", "the object referred to by the second word of the command", etc.

To understand these, it is important to know how the program stores the containments of objects in objects and rooms. For each room, the program stores a pointer down to the first object held in that room (all pointers, of course can take null values). For each object, four pointers are stored: 1) to the room holding the object; 2) to the object up from this one, i.e. the object immediately holding this one; 3) to the object next from this one, i.e. the next object in the chain of objects in the same object or room as this one; 4) to the object down from this one, i.e. the first object held by this one.

For instance, suppose the room `ROOM` contains the objects `PLAYER`, `DIAMOND` and `BOTTLE`, that the object `BOTTLE` holds the object `WATER`, that the object `PLAYER` holds the objects `ROD` and `CAGE` and that the object `CAGE` holds the object `SNAKE`. Then the structure of next and down pointers has the following appearance (it should be clear where the up and room pointers then point):



The following table then gives the possible forms of references and their meanings: it also gives the internal representation of the reference, which is useful in some complicated programs:

()O	The object specified by the second word, or the first object in the room referred to by the second word.	-2048
(olabel)O	The given object.	-2048+obj.# or olabel
(rlabel)O	The first object in the given room.	-2048+room#
(vlabel)O	Resolve the reference held in the variable, then take that object or the first object held in that room.	-1024+var.#
()U	The object up from the object referred to by ()O.	-768
(olabel)U	The object up from the given object.	-768+obj.#
()N	The object next from the object referred to by ()O.	-512
(olabel)N	The object next from the given object.	-512+obj.#
()D	The object down from the object referred to by ()O.	-256
(olabel)D	The object down from the given object.	-256+obj.#
()R	The room specified by the second word, or the room holding the object referred to by the second word.	0
(olabel)R	The room holding the given object.	obj.#
(rlabel)R	The given room.	room# or rlabel
(vlabel)R	Resolve the reference held in the variable, then take that room or the room holding that object.	1024+var.#

Note that object numbers lie in the range 1 to #objects, room numbers in the range #objects+1 to #objects+#rooms, and variable numbers in the range 0 to #variables-1.

It will be noticed that references are often unresolvable - e.g. (olabel)R cannot be resolved if the object is destroyed, ()O cannot be resolved if the command only contains one word, etc. When an unresolvable reference is found by the playing program, its usual response is to print the message "You can't do that!", stop processing the current command and ask for the next command. There are exceptions to this rule – these are detailed in chapter 3 at the appropriate places. A well-constructed game should not produce this message – the intention is merely that a bug of this sort should not stop a game, as it is fairly well understood and does not usually indicate that the database is unusable.

2.4.5 Directives and input handling

The database production program works in two modes while processing its input; these are:

1. After processing a `!MESSAGE` directive and before encountering any other directive, it is looking for message lines. All input lines are therefore left unchanged, unless they start with an exclamation mark `!` – this indicates that the line is a directive line, and the program reverts to mode (2).

2. At all other times, the program strips leading and trailing blanks from each input line, uppercases the line and removes any comments – these are any characters following a slash `/` in the line. Then the program checks for a `!` in the first column to see whether the line is a directive.

To summarise, therefore, a line is a directive if its first character is a `!` under all circumstances, or if its first non-space character is a `!` and the program is not looking for message lines. The only restriction on message lines is that they may not start with `!`.

Blank lines (or lines containing only comments) are allowed – they have no effect except when they are message lines.

Chapter 3

Input for the static part of the Database

This should consist of the following eight sections, which must appear in the following order:

1. Preliminary section
2. Objects section
3. Rooms section
4. Exits section
5. Instructions section
6. Words section
7. Messages section
8. Final section

These are described in the rest of this chapter.

3.1 The Preliminary section

This section contains directives that define the directions, special meanings, variables, properties and a few other things that are used during the game. They may appear in any order. The directives are:

```
!PRECOMMAND ilabel
!POSTCOMMAND ilabel
!WELCOME ilabel
```

These set the entry points of the pre-command, post-command and welcoming programs respectively. Each of them may only occur once.

```
!DIRECTION dlabel
!VARIABLE vlabel
!SPECIAL slabel
```

These define the directions, variables and special meanings used in the game respectively, acting as the defining occurences for these three types of label. One of these must therefore occur for each direction, variable or special meaning used in the game. The directions and special meanings are numbered in the order in which they appear here, starting at 1; the variables similarly, but starting at 0.

```
!TEXTVAR
```

This informs the program that a text variable will be used in the game. As only one text variable may exist in a game, no label is required. The directive may only appear once in the input.

```
!PROPERTY plabel integer(15)
```

This acts as the defining occurrence of a property label and so must occur once for each property used in the game. The integer specifies which property is to be used: this is done this way because the same property may be used for different purposes in objects and rooms, for instance.

3.2 The Objects section

This contains one `!OBJECT` directive for each object in the game: this directive contains the defining occurence of an object label. The directive has the following syntax:

```
!OBJECT olabel mlabel1 mlabel2 mlabel3
```

The three message labels point to the three descriptions, as follows:
mlabel1 — the description if no object is holding this one.
mlabel2 — the description if the "player" is holding it.
mlabel3 — the description if anything else is holding it.

The objects are numbered in the order in which they appear in the input, starting with 1. Remember that the first object must be the "player".

3.3 The Rooms section

This contains one `!ROOM` directive for each room in the game, this directive containing the defining occurence of a room label. It has the following syntax:

```
!ROOM rlabel mlabel1 mlabel2
```

The message label mlabel1 points to the long description of the room, while mlabel2 points to the short description.

The rooms are numbered in the order in which they appear in the input, starting at #objects+1.

3.4 The Exits section

For each room that has exits, this section should contain a block of lines, consisting of a `!EXIT` directive followed by one or more exit description lines. Nothing should appear in this section for any room without exits. The blocks may appear in any order, although it is probably easiest to have them in the same order as the `!ROOM` directives in the rooms section.

The `!EXIT` directive has the syntax:

```
!EXIT rlabel
```

This starts the block of exits for the given room. It is followed by one or more exit description lines; these have the following syntax:

```
dlabel rlabel [ilabel]
```

The label dlabel specifies the direction of the exit, while rlabel gives its destination. The optional instruction label ilabel specifies the entry point of the program associated with the exit, if it exists; this program will be obeyed before going through the exit if the player is going through the exit (while not held by any other object).

3.5 The Instructions section

The instruction section is started by the directive

```
!INSTRUCTIONS
```

This is followed by instruction lines, each with the syntax

```
[ilabel:] [instruction]
```

If ilabel is present, this is its defining occurence. It points at the instruction in its line, or if that doesn't exist, at the next instruction following that line.

The instruction, if present, can be one of a wide range of options. These are desribed in chapter 5.

3.6 The Words section

This section is started by the directive

```
!WORDS
```

The rest of the section contains word definition lines, each of which defines a single word known to the game. These must occur in alphabetical order of the word defined. No check is done to see whether words are multiply defined: e.g. if you define two different words, each starting with `W` and each allowing an abbreviation of length 1, no error will be detected by the database compilation program.

A word definition line has the following syntax:

```
     ( NONE         )
     ( IGNORE       )
     ( OBJECT       ) (           ( REC     )
     ( OBEY ilabel  ) ( MAY     ) ( ANY     )
     ( PRINT mlabel ) ( REQUEST ) ( DIR     )
word ( SAVE         ) ( MUST    ) ( OBJ     )
     ( SAVEND       ) (           ( SPECIAL )
     ( RESTART      ) (                     )
     ( FINISH       ) ( CANT                )
     ( MOVE         )
     ( RETURN       )

(continued)

                                  [ 1 ]
[reference]  [dlabel]  [slabel]   [ 2 ]
                                  [ 3 ]
                                  [ 4 ]
```

These last four items are all optional, but must occur in the correct order.

3.6.1 First word meaning

This is a single word or a pair of words, as follows:

`NONE` — this word has no valid meaning as a first word. If it is encountered, the program will reply “I don’t understand that!”.

`IGNORE` — after checking that the second word meets the requirements for this word, ignore this word, treat the current second word as being a new first word and regard the new second word as non-existent. Then recheck the command for validity. This is useful, for instance, as the first word meaning of GO in contexts such as GO EAST, etc.

`OBJECT` — Get a new first word by asking the player “What do you want to do with the ¡firstword¿?”. Ignore the current second word and treat the current first word as the new second word. Then recheck the command for validity. This is mainly useful as the first word meaning for words with object meanings.

`OBEY ilabel` — Obey the program with the given entry point.

`PRINT mlabel` — Print the given message.

`SAVE` — Save the game, then continue.

`SAVEND` — Save and stop the game.

`RESTART` — Start a new game immediately.

`FINISH` — Stop the game immediately.

`MOVE` — If the first word has a direction meaning, move the player in that direction. Otherwise, if it has a meaning as a room, move to that room, provided:

1. There is an exit to that room from the current room.
2. The current room does not have property 2 set.
3. The room has been visited.

If this doesn't work, an appropriate message is given.

`RETURN` — Move the player to the last room he occupied before this one, provided conditions (a) and (b) above hold.

3.6.2 Second word requirements

This starts with a field specifying whether the second word may exist.

`MAY` — the second word may exist (or may not).

`REQUEST` — the second word must exist. If it doesn't, the playing program requests it with the message "¡firstword¿ what?", or "¡firstword¿ where?" if the second part of the second word requirements is `DIR`.

`MUST` — the second word must exist. If it doesn't, produce the message "I don't understand that!".

`CANT` — the second word must not exist. If it does, produce the message "I don't understand that!".

Unless this part of the requirement was CANT, it is followed by one of the following:

`REC` — the second word may be any word recognised by the program (i.e. found in the database).

`ANY` — the second word may be any word at all (useful for commands such as SAY, SHOUT, SING, etc.).

`DIR` — the second word must have a meaning as a room or direction.

`OBJ` — the second word must have a meaning as an object.

`SPECIAL` — the second word must have a special meaning.

3.6.3 Other meanings

A word can have three other meanings, as follows:

`[reference]` — this gives the meaning of the word as an object or room. It may not be anything that refers to the second word – i.e. ()O, ()U, ()N, ()D, ()R, or something of the form (vlabel)O, where the given variable contains a reference to the second word, etc.

`[dlabel]` — this gives the meaning of the word as a direction.

`[slabel]` — this gives the special meaning of the word.

3.6.4 Abbreviation integers

These are the optional integers 1, 2, 3 and 4 above - the integer n indicates that the word may be abbreviated to exactly n characters if desired. As many of these as desired may be coded.

3.7 The Messages section

This contains one block for each message or part of a message. A block consists of a `!MESSAGE` directive, followed by message lines, and optionally ending with a `!SWITCH` directive to switch it into other messages or parts of messages. Remember that a message is automatically switched by the state of the object or room it's describing when used as a description; when it's used as a separate message, the `PRINT` instruction determines how it is switched.

The `!MESSAGE` directive provides the defining occurence of a message label: it has the syntax:

```
!MESSAGE mlabel
```

It is followed by the message lines, which may be up to 126 characters long. When the message is printed, these lines are printed exactly as they are given (except for substitution characters - see below), so we recommend that their length be kept down to a maximum of 72 (to avoid annoying bleeps on some terminals). The only other restriction is that message lines may not start with `!`. It is possible for a message to have no message lines at all - indeed, this is often a suitable message to switch into.

A message line may contain one or more of the following substitution characters: these are replaced by the given string before printing the line. Note that no further processing of the line takes place after this - in particular, nothing is done about lines that become too long as the result of substitutions.

The characters are given as hex codes: they are not normal EBCDIC characters and so should not interfere with anything else you may want to put into the message. To enter them into your source, either use the appropriate sequence for entering hex characters from your terminal (if any), or input the source without them, enter `ZED`, find the correct line, type X+ to enter hex mode, edit in the required character, then type X- to re-enter normal text mode.

hex 31 — the value of the text variable.

hex 32 — the first word of the command.

hex 33 — the second word of the command.

hex 34 — the value of variable number 0.

hex 35 — the value of variable number 1.

hex 36 — the value of variable number 2.

hex 37 — the value of variable number 3.

Optionally, the message lines may be followed by a `!SWITCH` directive. This has the syntax:

```
!SWITCH mlabel0 mlabel1 ... mlabeln
```

where n <= 255. The directive may continue onto more than one line – it is terminated by the next directive.

The result is that the message is switched to mlabel0 if the switching value is 0, mlabel1 if it is 1, etc., and to mlabeln if the switching value is >= n.

3.8 The Final Section

This contains a single line, which is the directive

```
!END
```

All further input after this directive is ignored completely.

Chapter 4

Input for the initial part of the Database

This part of the database is usually a lot shorter and simpler than the static part. It contains the initial values of all variable information in the game; these are specified by means of the following directives, which may occur in any order (with two exceptions, noted below):

```
!STATE ( olabel ) integer(255)
       ( rlabel )
```

This sets the state of the given object or room to the given integer. If it does not occur for an object or room, the state of that object or room is initially set to 0.

```
!PROP ( olabel ) plabel1 plabel2 \ldots plabeln
      ( rlabel )
```

where n is not limited, except by the number of property labels defined (although there is never any need for n to be more than 16). The directive may continue onto further lines; it is terminated by the appearance of the next directive.

This sets the given properties at the beginning of the game for the given room or object: all properties not set in this way will be unset at the beginning of the game.

```
!VAR vlabel ( integer   )
            ( reference )
```

This sets the given variable either to the given integer or to the internal representation of the given reference at the start of the game. All variables not set in this way are initially set to 0.

```
!TEXT
```

This should be followed by a single message line, containing the initial value of the text variable. It should be <= 126 characters long. This directive should only appear once in the input.

```
!POSSESSIONS ( olabel ) olabel1 olabel2 \ldots olabeln
             ( rlabel )
```

There is no limit to the value of n, apart from that imposed by the number of objects allowed. The directive may extend onto more than one line; it is terminated by the next directive.

This directive sets up the possessions of the object given by olabel or the room given by rlabel to be the objects given by olabel1, olabel2, ..., olabeln. If this directive does not appear for a given object or room, it is assumed that that object or room initially has no possessions. If a given object does not appear among olabel1, olabel2, ..., olabeln for any `!POSSESSIONS` directive, it is initially in the destroyed state.

There is an order requirement here: this is that these directives must appear in "top-down" order. More precisely, when a `!POSSESSIONS` directive is encountered, none of olabel1, olabel2, ..., olabeln must have appeared as "olabel" in a previous `!POSSESSIONS` directive. For example, the code on the left below is incorrect, that on the right is correct:

WRONG	RIGHT
`!POSSESSIONS CAGE SNAKE`	`!POSSESSIONS ROOM CAGE`
`!POSSESSIONS ROOM CAGE`	`!POSSESSIONS CAGE SNAKE`

As another example, there follows a right way to code the containment structure given in section 2.4.4:

```
!POSSESSIONS ROOM PLAYER DIAMOND BOTTLE
!POSSESSIONS PLAYER ROD CAGE
!POSSESSIONS CAGE SNAKE
!POSSESSIONS BOTTLE WATER
```

The other order requirement is for the directive

```
!END
```

which should occur once at the end of the input. All lines following it are ignored.

Chapter 5

The Instruction Code for Programs

The instructions in a program are obeyed in sequential order, unless otherwise specified by a SKIP, GO, GOSUB, PRINTRET, DESCRET or RETURN instruction. The following instructions may occur.

5.1 Test and skip instructions

Syntax:

```
                                    ( EQ  )
                        ( R  reference1  ( LT  )  reference2
                        (                ( GT  )
                        (                ( ADJ )
                        (
                        (                ( EQ )
                        ( S reference    ( LT )  integer(255)
                        (                ( GT )
                        (
                        (                ( EQ )
                        ( V vlabel       ( LT )  integer
                        (                ( GT )
                        (
( SKIP  )               ( P plabel  reference
( SKIP1 ) ( IF     ) (
( SKIP2 ) ( UNLESS ) ( E reference
( SKIP3 )            (
( SKIP4 )            ( Q mlabel
                     (
                     ( H reference1  reference2
                     (
                     (    [ MOVED       ]
                     (    [ LIGHT       ]
                     (    [ W1RM        ]
                     (    [ W1OB        ]
                     (    [ W1DI        ]
                     (    [ W1SPX       ]
                     ( M  [ W1SP slabel ]
                          [ W2EX        ]
                          [ W2RM        ]
                          [ W2OB        ]
                          [ W2DI        ]
                          [ W2SPX       ]
                          [ W2SP slabel ]
```

Descriptions of the various fields follow:

SKIP, SKIP1, SKIP2, SKIP3, SKIP4 — these specify how many instructions are to be skipped if a skip takes place as the result of this instruction. SKIP is synonymous with SKIP1.

IF — the skip is to be done if the test succeeds; if it doesn't, the next instruction should be obeyed.

UNLESS — the skip is to be done if the test does not succeed; if it does, the next instruction is to be obeyed.

5.1.1 Comparisons of references

An R (reference) type test compares two references. These are each resolved, and the test succeeds if the given relation holds between them. The allowed relations are:

`EQ` — test succeeds if references are equal.

`LT` — test succeeds if reference1 is less than reference2, i.e. if the object or room referred to by reference1 appears before that referred to by reference2 in the object and room sections of the static part of the database.

`GT` — test succeeds if reference1 is greater than reference2.

`ADJ` — test succeeds if there is an exit from reference1 to reference2 (the two references should, of course, both refer to rooms).

If either reference is unresolvable, the test fails (and the message "You can't do that!" is not produced).

5.1.2 Testing a state of a room or object

The S (state) type test compares the state of the object of room referred to by the given reference with the given integer in the range 0-255. The same relations are used as above, except for `ADJ`. The test fails if the reference is unresolvable.

5.1.3 Testing the value of a variable

The V (variable) type test compares the value of the given variable with the given integer. The same relations are used as above, except for `ADJ`.

5.1.4 Testing a property

The P (property) type test succeeds if the given property of the given room or object is set. If the reference cannot be resolved, the test fails.

5.1.5 Testing whether a reference is resolvable

The E (existence) type test succeeds if the given reference "exists" (i.e. is resolvable) and fails otherwise.

5.1.6 Testing a yes/no answer to a question

The Q (question) type test prints out the given question, then asks the player for a yes/no answer to it. The test succeeds if the answer is "yes". The answer is taken to be "yes" if the first non-space character in the answer is "y" or "Y", "no" if it is "n" or "N", and the question is repeated otherwise.

5.1.7 An object holding another one

The H (held by) type test succeeds if the object referred to by reference1 is held by the object referred to by reference2, either indirectly or directly (e.g. in the example in section 2.4.4, `ROD`, `CAGE` and `SNAKE` are all held by `PLAYER`). The test fails if either reference is unresolvable.

Note that reference1 and reference2 must both refer to objects. To test whether a room holds an object, use the R type test, e.g.:

```
SKIP IF R (olabel)R EQ rlabel
```

5.1.7 Miscellaneous tests

The M (miscellaneous) type test contains 11 subtests, which are selected by the options that follow M. The overall test succeeds if any of the subtests selected succeeds, fails if they all fail. The subtests are:

`MOVED` — succeeds if the player is in a different room to the one he was in when the command was input (most useful in post-command programs).

`LIGHT` — succeeds if there is a non-hidden, visible light source at the player's current location, or if the room is lit.

`W1RM` — succeeds if the first word has a room meaning.

`W1OB` — succeeds if the first word has an object meaning.

`W1DI` — succeeds if the first word has a direction meaning.

`W1SPX` — succeeds if the first word has any special meaning.

`W1SP slabel` — succeeds if the first word has the given special meaning.

`W2EX` — succeeds if the second word exists.

`W2RM`, `W2OB`, `W2DI`, `W2SPX`, `W2SP slabel` — like `W1RM`, `W1OB`, `W1DI`, `W1SPX`, `W1SP slabel`, but refer to the second word, not the first.

5.2 Move instructions

Syntax:

```
                            ( TO reference2
                            ( VIAEXIT reference2
MOVE reference1 ( WITH    ) ( DESTROY
                ( WITHOUT ) ( DIR dlabel
                            ( RANDOM [plabel]
                            ( RANDADJ [plabel]
```

The object to be moved is given by reference1 - this must refer to an object. The object is moved with its possessions if `WITH` is coded, without them if `WITHOUT` is coded - in this case, the chain of objects it held is left in the place it moved away from. In all cases where the move succeeds, the object is moved to become the first object in the chain of objects held by its destination room or object. The options for the move are then:

`TO reference2` — the object is moved directly to the object or room referred to by reference2.

`VIAEXIT reference2` — the object is moved to the object or room referred to by reference2, provided the following conditions hold:

1. There is an exit in the right direction from the object's current room (i.e. to the room or the room holding the object).
2. If reference1 refers to the player, his current room must not have the disorientation property 2 set.
3. If reference1 refers to the player, the room he's moving to must have been visited.

Also, if reference1 refers to the player and the exit has a program attached, that program will be obeyed.

`DESTROY` — the object is moved out of its current location and into the destroyed state.

`DIR dlabel` — the object is moved in the given direction if possible. If reference1 refers to the player and the exit has a program attached, the program is obeyed.

`RANDOM [plabel]` — the object is moved to a room at random. If plabel is given, the exit is aborted is the given property is set for the chosen destination room.

`RANDADJ [plabel]` — the object is moved through a random exit of its current room (i.e. to a random adjacent room). If plabel is given, the exit is aborted if the given property is set for the chosen destination room. If reference1 refers to the player and the chosen exit has a program attached, that program is obeyed.

5.3 Variable and state arithmetic instructions

Syntax:

```
( LOAD )                  ( V vlabel2    )
( ADD  ) ( V vlabel1    ) ( S reference2 )
( SUB  ) ( S reference1 ) ( I integer    )
( MULT )                  ( R integer    )
```

If the second field above is of the form "S reference1", the integers in "I integer" and "R integer" must lie in the range 0-255. These instructions do arithmetic with variables and states of objects and rooms. The operations are:

`LOAD` — load the first operand from the second.

`ADD` — add the second operand to the first.

`SUB` — subtract the second operand from the first.

`MULT` — multiply the first operand by the second.

In all cases, therefore, the first operand is changed, the second one is not. The allowed first operands are:

`V vlabel1` — the given variable.

`S reference1` — the state of the given object or room.

The allowed second operands are:

`V vlabel2` — the given variable.

`S reference2` — the state of the given object or room.

`I integer` — the given integer.

`R integer` — a random integer in the range from 0 to the given integer, including both ends (so there are "integer"+1 possible values).

5.4 Text variable setting instruction

Syntax:

```
TEXT mlabel ( WITH    )
            ( WITHOUT )
```

The text variable is set to the first line of the given message (normally, this would be a one line message). If `WITH` is specified, this is done with substitutions done; if `WITHOUT` is specified, substitutions are not done and the substitution characters are left in the text variable's value until this is substituted into a message (when these substitution characters are also resolved).

5.5 Printing and describing instructions

Syntax:

```
( PRINT    ) mlabel [vlabel]
( PRINTRET )

( DESCRIBE ) ( WITH    ) [reference]
( DESCRET  ) ( WITHOUT )
```

The variable given by vlabel, if vlabel exists, must be one of the first four variables, i.e. the variables with numbers 0, 1, 2 and 3. The reference must refer to an object if it exists.

`PRINT` and `PRINTRET` print the given message, using the given variable as a switching value, or the state of the player if no variable is given. After doing this, `PRINTRET` also returns from the program.

`DESCRIBE` and `DESCRET` describe the given object, or the current room of the player if no reference is given. If `WITH` is specified, all objects held directly or indirectly by the given object or room are also described (except hidden or invisible ones). If `WITHOUT` is specified, only the given object or room is described. Again, `DESCRET` also returns from the program after outputting the description.

5.6 Property setting instructions

Syntax:

```
( SET   )
( UNSET ) plabel reference
( COMP  )
```

These instructions set, unset and complement (i.e. change) the given property of the given object or room.

5.7 Branching instructions

Syntax:

```
( GO    ) ilabel
( GOSUB )
```

These transfer control to the given instruction. GO transfers it directly, GOSUB goes to a subroutine (so that RETURN then branches back to the next instruction following this one).

5.8 Question asking instructions

Syntax:

```
( ASK    ) mlabel
( ASKANY )
```

These instructions are intended for asking the player questions with answers ore general than "yes" and "no" (see testing instructions above for such questions). The given message is printed and then the user is prompted for a reply. The first word of his reply then replaces the second word of the command. The command is not rechecked for validity.

`ASK` expects the answer to the question to lie in the vocabulary, and will reply "I don't understand that!" if it isn't, then request another answer. `ASKANY` will accept any word as a valid reply.

5.9 Reference resolving instruction

Syntax:

```
RESOLVE vlabel reference
```

This instruction resolves the given reference, then loads the given variable with:

0	if the reference is unresolvable.
Object#-2048	if the reference resolves to an object.
Room#	if the reference resolves to a room.

In effect, provided the reference is resolvable, this puts another reference into the given variable which is guaranteed always to resolve to the same object or room as this one resolves to now. There are two main types of use for this instruction:

1. It can be used to "remember" a room or object. E.g. suppose you want to provide a magic transport which transports the player back to where he said a particular magic word. Then you set up a variable `TRANSPLACE` and include the instruction

   ```
   RESOLVE TRANSPLACE (PLAYER)R
   ```

   in the program for that magic word. The player can then be moved to that place by means of the instruction

   ```
   MOVE PLAYER WITH TO (TRANSPLACE)R
   ```

2. It can be used to produce more complicated references than those provided by the system. For instance, the system does not provide a reference referring to "the first object in the room holding the player". This can be produced, however, by means of the instruction

   ```
   RESOLVE VAR (PLAYER)R
   ```

   and then referring to `(VAR)O`.

A complicated use of this type may involve modifying the value held in the variable. E.g. if you resolve an object reference into a variable, then add 1280 to that variable, the variable then contains a reference to the object up from the object originally referred to. Similarly, adding 1536 will produce the object next from it, and adding 1792 will produce the object down from it. To give a particular case, if one wants to refer to "the second object in the room holding the `LYRE`", one should use the instructions:

```
RESOLVE VAR (LYRE)R       /Room holding the lyre.
RESOLVE VAR (VAR)O        /First object in that room.
ADD V VAR I 1536          /Modify reference.
RESOLVE VAR (VAR)O        /And find second object in the room.
```

`VAR` will then contain a reference that will always point to the current second object in the room, or 0 if the room only holds the `LYRE`. Similar techniques can be used to produce loops that scan through all objects in a room, etc.

One warning - in a complicated sequence of this type, it is usually necessary to check that the reference was resolvable after each `RESOLVE` instruction. The examples above are only OK, for instance, if we know that the `PLAYER` and the `LYRE` are not in the destroyed state.

5.10 Return instructions

Syntax:

```
RETURN

                  ( NONE    )
                  ( IGNORE  ) (            ( REC     )
                  ( OBJECT  ) ( MAY      ) ( ANY     )
                  ( SAVE    ) ( REQUEST  ) ( DIR     )
       ( RETRY    ( SAVEND  ) ( MUST     ) ( OBJ     )
       (          ( RESTART ) (            ( SPECIAL )
       (          ( FINISH  ) (                      )
       (          ( MOVE    ) ( CANT                 )
       (          ( RETURN  )
       (
RETURN ( DEST     ( rlabel )
       (          ( olabel )
       ( PASS
       ( ABORT
       ( LOOK
       ( NEXTCOMM
       ( LEAVE
```

(`RETURN RETRY OBEY ...` and `RETURN RETRY PRINT ...` are also available, but not useful, as there is no facility to change the associated instruction or message label.)

(`RETURN` on its own provides a normal return (as do (`PRINTRET` and (`DESCRET`). This returns to the calling program of this subroutine, or to the playing program if we're not in a subroutine. The remaining options of (`RETURN` each only have any effect when the program has been called in certain ways. When it has been called in ways for which the (`RETURN` option is not valid, the (`RETURN` instruction provides a normal return. We abbreviate the calling methods ((1)-(6) in section 2.1.7) as "word", "exit", "welc", "pre", "post" and "subr".

RETRY — (word, pre, post and welc): change the first word meaning and second word requirements to those that follow. Then recheck the command for validity and obey it again. (exit): Abort the exit and set a request for the word to be reinterpreted as above. This request will be obeyed immediately if the exit was invoked directly from the main playing program (via the MOVE first word meaning), and passed on to the program that contained the MOVE instruction if the exit was produced by a MOVE instruction. It can then be passed back to the main playing program via a RETURN PASS instruction.

DEST (exit only) — change the destination of the move to be the given room or object (the latter should be rare).

PASS (all types) — do not set the return options from this instruction, but pass on the last request made by a MOVE instruction in this program. If the program was called by an exit, this return option will be passed on by the exit to the main playing program or the program that contained the MOVE instruction, etc.

ABORT (exit only) — abort the exit.

LOOK — (word, pre, post and welc): Forces the playing program to describe the current room before prompting the player for his next command, whether or not he has moved since this command. (exit): Sets a request for the current room to be described as above. This request can be passed up by the RETURN PASS instruction in the same way as a RETURN RETRY ... request.

NEXTCOMM (all types) — abandon processing of thsi command and get the next one from the player immediately. If called by an exit, this involves aborting the exit.

LEAVE (subr only) — return directly to the main program or to the calling exit, i.e. leave the subroutine nest completely.

Chapter 6

EXAMPLES

6.1 Movement commands

Here we give an example of the code needed to get the following commands to work:

```
direction
GO direction
BACK
GO BACK
RETURN
```

where "direction" is one of DOWN, D, EAST, E, NORTH, N, NE, NW, SOUTH, S, SE, SW, WEST, W, UP, U.

In the preliminary section

```
!DIRECTION B    /Dummy direction for "GO BACK".
!DIRECTION D
!DIRECTION E
!DIRECTION N
!DIRECTION NE
!DIRECTION NW
!DIRECTION S
!DIRECTION SE
!DIRECTION SW
!DIRECTION W
!DIRECTION U
```

In the words section

```
BACK RETURN CANT B
DOWN MOVE CANT D 1
EAST MOVE CANT E 1
GO IGNORE REQUEST DIR
NORTH MOVE CANT N 1
NE MOVE CANT NE
NW MOVE CANT NW
RETURN RETURN CANT
SOUTH MOVE CANT S 1
SE MOVE CANT SE
SW MOVE CANT SW
WEST MOVE CANT W 1
UP MOVE CANT U 1
```

If any word is given first word meaning MOVE and a direction or room meaning, then the commands "GO word" and "word" wil also work.

The special points to note about this example are that the short forms of the directions are provided via abbreviation integers (which here do not allow such forms as DO for DOWN), and that the word GO checks whether it is followed by an appropriate second word, then is ignored. Thus the command GO WEST becomes simply WEST internally after the program has checked that WEST is a direction. Also note that BACK is given a dummy direction meaning to make GO BACK have the right effect.

Finally, the command GO will produce the response "GO where?" in accordance with the second word requirements of the word GO.

6.2 A random exit

We suppose here that the rooms ROOM1, ROOM2, and ROOM3 are set up, and that an exit going north from ROOM1 is desired, with 70% chance of going to ROOM2, 10% chance of going to ROOM3, and 20% chance of getting nowhere.

In the preliminary section

```
!VARIABLE VAR0
```

In the exits section

```
!EXIT ROOM1
 N ROOM1 RANDPROG
```

In the instructions section

```
RANDPROG:
  LOAD V VAR0 R 9
  SKIP UNLESS V VAR0 LT 7
  RETURN DEST ROOM2
  SKIP UNLESS V VAR0 EQ 7
  RETURN DEST ROOM3
  PRINTRET HOLESMESS
```

In the messages section

```
!MESSAGE HOLESMESS
You crawled around some little holes and wound up back in
the main passage.
```

6.3 The INVENTORY command

A reasonable way to set this up is as follows:
In the preliminary section

```
!PROPERTY LIGHT 0
!SPECIAL INVSPEC
```

In the objects section

```
!OBJECT PLAYER HOLDING HOLDING HOLDING   /On first line.
```

In the instructions section

```
INVPROG:
  SET LIGHT PLAYER       /Ensure place is lit.
  DESCRIBE PLAYER WITH   /Do the inventory.
  UNSET LIGHT PLAYER     /Get rid of surplus light.
  SKIP IF E (PLAYER)D    /Is he carrying anything?
  PRINT NOTHING          /No.  Print "Nothing".
  RETURN
```

In the words section

```
INVENTORY OBEY INVPROG CANT INVSPEC 3
```

In the messages section

```
!MESSAGE HOLDING
You are holding:
!MESSAGE NOTHING
Nothing.
```

The commands `INVEN` and `INV` should then produce an inventory of what the player is holding. Note the use of the `LIGHT` property of the "player" to ensure that the message "It is pitch dark." does not occur. The special meaning `INVSPEC` attached to `INVENTORY` is used in the next example.

6.4 The TAKE command - a more complicated example

The TAKE command will usually have a number of special cases built into it for objects that are difficult to take or have other special effects. In this example, we give a basic form of it, which when combined with the previous example allows the following commands to be obeyed:

```
TAKE INVENTORY
TAKE INV
TAKE objectname
TAKE          (takes the first takable object in the room)
TAKE ALL      (takes all takable objects in the room)
```

An object is assumed to be untakable if it has the property NOTAKE set. It is assumed that the variable OBJHELD contains the number of objects currently being carried by the player, and that the variable STRENGTH holds the maximum number he can carry.

No further explanation is given of this example, except for the comments in the code below.

In the preliminary section

First four !VARIABLE directives are:

```
!VARIABLE VAR0          /Workspace
!VARIABLE VAR1          /Workspace
!VARIABLE VAR2          /Workspace
!VARIABLE VAR3          /Workspace
```

Also, there appear:

```
!VARIABLE OBJHELD
!VARIABLE STRENGTH
!PROPERTY NOTAKE 3
!SPECIAL ALLSPEC
```

In the instructions section

```
/Subroutine to try to take the object referred to by the
/reference in VAR0.  OBJHELD is updated if it is taken, and
/VAR1 is set to:
/ 1 if the object was taken;
/ 2 if it was untakable;
/ 3 if it wasn't taken because the player couldn't carry it.
TAKESUB:
  SKIP IF R (VAR0)0 EQ PLAYER        /Can't take himself!
  SKIP2 UNLESS P NOTAKE (VAR0)0      /Is it untakable?
  LOAD V VAR1 I 2
  RETURN
  LOAD V VAR1 V STRENGTH             /Check OBJHELD
  SUB V VAR1 V OBJHELD               / against STRENGTH
  SKIP2 IF V VAR1 GT 0
  LOAD V VAR1 I 3
  RETURN
  MOVE (VAR0)0 WITH TO PLAYER        /Move the object.
  ADD V OBJHELD I 1                  /Update OBJHELD.
  LOAD V VAR1 I 1
  RETURN

TAKEPROG:
```

```
/Command decoding section.
  SKIP IF M W2EX                    /Is it "TAKE"?
  GO TAKEFIRST
  SKIP UNLESS M W2SP ALLSPEC        /Is it "TAKE ALL"?
  GO TAKEALL
  SKIP UNLESS M W2SP INVSPEC        /"TAKE INVENTORY"?
  GO INVPROG
  SKIP IF M W2OB                    /Must be "TAKE
  RETURN RETRY NONE CANT            / objectname"
/Now check it's OK to take the given object.
  SKIP IF R (PLAYER)R EQ ()R        /In the same room?
  PRINTRET DONTSEE
  SKIP UNLESS R PLAYER EQ ()U       /Already holding it?
  PRINTRET ALRHELD
  RESOLVE VAR0 ()O                  /Try taking it.
  GOSUB TAKESUB
  SKIP UNLESS V VAR1 EQ 2           /Untakable?
  PRINTRET CANTTAKE
  SKIP UNLESS V VAR1 EQ 3           /Hands full?
  PRINTRET HANDSFULL
  PRINTRET OKMESS

TAKEFIRST:
/Take the first takable object in the room.
  RESOLVE VAR0 (PLAYER)R            /Find first object
  RESOLVE VAR0 (VAR0)O              / in room.
TAKEF1:
  SKIP UNLESS V VAR0 EQ 0           /Got an object?
  PRINTRET NOTHNGHERE
  GOSUB TAKESUB                     /Try taking it.
  SKIP UNLESS V VAR1 EQ 1           /Successful?
  PRINTRET OKMESS
  SKIP UNLESS V VAR1 EQ 3           /Hands full?
  PRINTRET HANDSFULL
/This object wasn't takable. Try the next one in the room.
  ADD V VAR0 I 1536                 /Modify reference.
  RESOLVE VAR0 (VAR0)O              /& resolve it.
  GO TAKEF1

TAKEALL:
/Try taking all objects in the room. VAR3 keeps count of the
/number of objects taken.
  LOAD V VAR3 I 0
/Start a loop. In this loop, VAR2 will always point to the
/next object to be tried.
  RESOLVE VAR0 (PLAYER)R            /Find first object
  RESOLVE VAR0 (VAR0)O              / in room.
TAKEA1:
  LOAD V VAR2 V VAR0                /Find next object
  ADD V VAR2 I 1536                 / from this one.
  RESOLVE VAR2 (VAR2)O
  GOSUB TAKESUB                     /Try taking this one.
  SKIP2 UNLESS V VAR1 EQ 3          /Hands full?
  PRINT HANDSF2
```

```
  PRINTRET TOOKOBJ2 VAR3
  SKIP UNLESS V VAR1 EQ 1            /Taken this object?
  ADD V VAR3 I 1                     /Yes.
  LOAD V VAR0 V VAR2
  SKIP IF V VAR0 EQ 0                /Does next one exist?
  PRINTRET TOOKOBJ1 VAR3
```

In the words section

```
ALL NONE CANT ALLSPEC
TAKE OBEY TAKEPROG MAY REC
```

In the messages section

COMMENT: The character represented by "?" in the message TOOKOBJ4 is the character hex 37, not the query character.

```
!MESSAGE ALRHELD
You're already holding that!
!MESSAGE CANTTAKE
You can't take that!
!MESSAGE DONTSEE
I don't see that around here!
!MESSAGE HANDSFULL
You can't carry anything more - you'll have to drop
something before you can take that.
!MESSAGE HANDSF2
You've had to leave things, as your hands are now full.
!MESSAGE NOTHNGHERE
There's nothing here you can take!
!MESSAGE OKMESS
OK.
!MESSAGE TOOKOBJ1
!SWITCH NOTHNGHERE TOOKOBJ3 TOOKOBJ4
!MESSAGE TOOKOBJ2
!SWITCH TOOKOBJ4 TOOKOBJ3 TOOKOBJ4
!MESSAGE TOOKOBJ3
You took 1 object.
!MESSAGE TOOKOBJ4
You took ? objects.
```

Chapter 7

Error Numbers and their Meanings

The following list gives a brief description of each possible error produced by the database production program. Note that one error on a line of input can induce others - error number 6 is particularly likely to be caused in this way. Only the first error message for any particular line of input is therefore trustworthy.

1. Directive name missing (static or initial section) or unrecognised (initial section only).
2. Unrecognised directive name (static section).
3. Invalid termination to directive, label or integer.
4. Bad or duplicate label in `!DIRECTION.`
5. Bad or duplicate label in `!OBJECT.`
6. Excessive text at end of line, or bad input line.
7. More than one `!TEXTVAR` directive.
8. Bad or duplicate label in `!SPECIAL.`
10. Bad or duplicate label in `!MESSAGE.`
13. Bad message label in `!ROOM` or `!OBJECT`.
14. Bad room label in `!EXIT`.
15. Words out of order in words section.
16. Bad direction label in exit description line.
17. Bad room label in exit description line.
18. Bad instruction label in exit description line.
19. Bad or duplicate label in `!VARIABLE`.
21. Bad word in word definition line, or bad meaning as a first word.
22. Bad second word requirements in word definition line, or bad message label in `PRINT` option, or bad reference meaning.
23. Bad termination, or bad special or direction meaning in word definition line.
26. Abbreviation integer out of range in word definition line.
27. Message line too long.
28. Duplicate instruction label.
29. Bad or duplicate label in `!PROPERTY`, or property number out of range.
30. Use of `!TEXT` directive in initial section when no text variable declared.
31. Double use of `!TEXT` directive.
32. Message line following `!TEXT` directive null or too long.
33. Unrecognised or bad possessor in `!POSSESSIONS`.
34. Incorrect ordering of `!POSSESSIONS` directives (this includes the case of requesting "circular" possessions).
35. Bad room or object label in `!PROP`.

36. Property label bad or missing in `!PROP`.
37. Bad or missing variable label in `!VAR`, or bad attempted initialisation.
38. Bad or missing room or object label in `!STATE`, or integer out of range.
40. Bad or unrecognised instruction name.
41. Bad reference for object to be moved in `MOVE` instruction.
42. Bad `MOVE` type, or bad operand following the type.
43. Bad message label in `PRINT` or `PRINTRET` instruction.
44. Bad variable label in `PRINT` or `PRINTRET` instruction.
45. `WITH/WITHOUT` missing or bad in `DESCRIBE` or `DESCRET` instruction.
47. Bad option in `RETURN` instruction.
48. Bad new meaning in `RETURN RETRY` instruction.
49. `IF/UNLESS` missing or bad in `SKIP` type instruction.
50. Skip type (i.e. R/S/V/P/E/Q/H/M) missing or bad in `SKIP` type instruction.
51. Bad first operand following `SKIP(n) IF/UNLESS <type>`.
52. Bad comparison operator in `SKIP` instruction.
53. Bad second operand in `SKIP` instruction.
54. Bad variable label in `RESOLVE` instruction.
55. Bad instruction label in `GO` or `GOSUB` instruction.
56. Bad message label in `ASK` or `ASKANY` instruction.
57. Bad message label in `TEXT` instruction, or `WITH/WITHOUT` bad.
58. Bad property label in `SET/UNSET/COMP` instruction.
59. Bad type for first operand in `LOAD/ADD/SUB/MULT` instruction, or bad first operand.
60. Bad second operand type, or bad second operand in `LOAD/ADD/SUB/MULT` instruction.
70. Bad instruction label in `!PRECOMMAND`.
71. Bad instruction label in `!POSTCOMMAND`.
72. Bad instruction label in `!WELCOME`.
73. Bad or unrecognised message label in `!SWITCH`.
74. Unrecognised input line.

Chapter 8

Messages produced by the Playing Program

In this chapter, we give a list of the messages that are built into the playing program, together with a brief description of when they occur, etc.

"Welcome to Adventure!" — when a game is started, before the database and save file names are requested.

"What is the name of the database?" — at the beginning of the game, unless playing under Phoenix and a database name has been provided as a parm string.

"If you want to restore a saved game, type the name of the file it was saved in:" — at the beginning of the game. Type carriage return if you don't want to restore a game.

"Type the name of the file in which you want to save the game:" — when trying to save the game (in response to the `SAVE` or `SAVEND` first meaning of words).

"Database allocation failed.", "Save file allocation failed.", "File name too long.", "Do you want me to try again?" — in response to various problems with allocating the database or a saved game file.

"Save attempt failed.", "Game saved." — to report on whether the game has actually been saved.

"Not enough region available for the game.", "Too few save areas.", "Severe database error. Please send details to the database writer" , "Save/initial file - static file mismatch." — as described in sections 1.2, 2.3 above.

"I don't understand that!" — in response to an unrecognised first word of the command, or a second word that doesn't fit the first word's requirements, or to a bad answer to a question asked via the `ASK` instruction.

"I'm afraid I've forgotten how you got here!" — when the `RETURN` meaning of the first word cannot be implemented.

"You can't do that!" — usually in response to an unresolvable reference being encountered.

"¡firstword¿ what?", "¡firstword¿ where?", "What do you want to do with the ¡firstword¿?" — in response to the `REQUEST` second word requirement and the `OBJECT` first word meaning of the first word of the command. See section 3.6.

"You can't go in that direction!" — if the "player" tries to go in a direction in which there is no exit.

"I don't know how to get there!" — when the player tries to get to a particular room through an exit and cannot. This occurs either in response to the `MOVE` first word meaning of a word combined with a meaning as a room, or in response to the `MOVE PLAYER VIAEXIT` instruction.

"You're already there!" — when the player tries to get to his current room via an exit and cannot.

"Please answer the question:" — when the player has not produced a good reply to a question asked as the result of an `ASK` or `ASKANY` instruction.

"It is pitch dark." — in response to any attempt to describe an object or room when there is no light present.

"Please answer the question (Y or N):" — When the player has not given a yes/no answer to a question asked via the `SKIP IF Q ...` instruction.