ALAN

Adventure Language

Reference Manual

version 2.8

This version of the manual was printed on October 17, 2000

TABLE OF CONTENTS

1. INTRODUCTION

Text adventures or, using a more appropriate term, interactive fiction is a form of computer game which has many things in common with fiction in book form, role-playing games and puzzle-solving. To create a high quality interactive fiction game, you need to be more of an author than a games programmer.

Alan is a special purpose computer language specifically designed to make it very easy to create such adventure games with little programming skills.

The main principle of the design of the language is simplicity. That is, to make it very easy to do normal things, but also allow more complex things using more complex language constructs. This means that wherever a construct is optional, the system supplies some sensible default instead.

The Alan language has been designed by the author and a very good friend during several years of incremental improvement. This means that the language has a sound foundation, based on practical use, a concept forgotten in many software projects today. Tools develop and are made sharper and more powerful as usage is intensified, the problem domain more and more understood and the requirements increased to tackle new aspects of the problem.

This, we believe, is how software tools must be developed to give the support intended. Therefore Alan and its support system will also develop further. This version is, however, a complete and powerful tool as it stands.

2. TUTORIAL

2.1. WHAT IS AN ADVENTURE?

As long as man has been around there have been stories, fairy tales and fantasies. In early days the stories were told by story tellers to silent and astonished audiences. After Gutenberg, the stories were printed and the readers partook in the fantasies of the author. In our days, passive viewers are fed from the silver screen or through the tube.

In our century, at last, there has evolved a way for the "audience" to take part in the story themselves. It started in the forties and fifties and continued to develop into the games today known as Dungeon and Dragons, Tunnels and Trolls, etc. Games where a game leader designs the story, but the players decide (and perform) the actions of the characters in the story.

These games, of course, have a computerized counterpart.

The games are played interacting with the computer. The program describes a scene or situation (usually in text, but pictures are more and more frequently used), the player decides on some action and gives orders to the computer to carry out his wishes. Usually there are objects to manipulate, traps to negotiate and puzzles to solve, the object being to find the hidden treasures or save the world.

This form of games was started by Crowther & Woods in the late sixties when they designed the famous *Colossal Cave Adventure* now available on many mainframe computer systems. Inspired by this, Lebling et.al. (then at MIT) took a giant step forward in adventuring by creating the Great Underground Empire and making it available for venturing Adventurers in the game Dungeon. This game contained a much more developed story and could handle much more complex commands.

Later, Dave Lebling & Co started Infocom, a company where they continued to develop their technique, first with *Zork I, II* and *III* (the first a reimplementation of Dungeon, the others equally successful sequels). Since then a host of games has been released (*Starcross*, *Witness*, *Enchanter* are some of the names that come to mind). Infocom today only exists as a label with Mediagenic, the original authors scattered, but the Infocom games are still among the best available today.

Other companies have followed in Infocom's footsteps and a handful of them seem to make a living out of creating adventure games. However, today most of the works are performed by devoted people that produce games for the fun of it, as shareware or completely free.

There have been many attempts to use computer graphics to display the surroundings and objects in adventure games. Some of the more successful are the Sierra OnLine games (notably the *Leisure Suit Larry* and the *Kings Quest* series) which have mouse oriented moves but also allows single line text commands, games from ICOM Simulations (*DejaVu* and *The Uninvited*) which are purely graphics games with mouse and icon interfaces. Other manufacturers have tried to use (some

times optional) pictures to accompany the text, for example Magnetic Scrolls games (e.g. *the Pawn*), which shift the picture automatically as you move around using the normal directional commands.

A large number of so called Arcade Adventures are also available but they are always more Arcade than Adventure and do not belong in this discussion.

The Alan Adventure Language is designed to aid construction of pure text adventures or, in the words of Infocom, interactive fiction. Possibly, in future versions sound and graphics functions may be available.

2.2. ELEMENTS OF ADVENTURES

The success of all Infocom games can probably be attributed to three distinctive features. First, they all have a 'believable' and consistent plot, which is flavoured with humour and wittiness. Second, the descriptions are extensive and give a lot of atmosphere to the game. Third, the command handler recognizes and understands a large vocabulary and complex input. Add to this the worlds best graphics device (the human brain) and you are unbeatable!

Looking at Adventures in more detail, one can see some common features. There is, of course, a world or universe (called the map) where the Adventure is taking place. Although you can move around quite freely there are usually some problems getting into certain parts of the world (e.g. locked doors, no air to breathe or even finding the entrance).

The size of the map ranges from hundreds of locations (like in some of the Infocom games) to just two or three (some of the Scott Adams games are very compact and very difficult).

Then, there are the objects in the game. These range from your tools, like lamps and shovels, to immaterial things like a hole in the ground, in short, anything you can manipulate. Ideally everything that is mentioned in a description should be an object, but this is normally impossible because of storage limits (and perhaps the stamina of the games designer!).

Most objects have uses. A key is easy to guess how to use, but what about the velvet pillow? Red herring objects are also common in adventuring.

The player must be able to express his wishes. Natural language commands, advocated in many advertisements, are probably overkill since most players do not wish to wear their fingers out typing full sentences, but single verb-object input is not sufficient for a good game either. The player must be able to say things like

```
> take all except the blue vase
```

or

```
> put the ring and the bag in the box
```

2.3. ALAN FUNDAMENTALS

The Alan Adventure Language is a high level computer language designed to make it easy to create text Adventures. The Alan system handles all the tiresome tasks and supplies reasonable defaults so you can concentrate on the plot, the puzzles, the objects and the map.

There are no variables, subroutines or other traditional programming constructs in Alan because Alan is not a *programming* language. Instead Alan takes a descriptive view of the concepts of adventure authoring. The Alan language contains constructs to make it possible for the author to describe the various features of these concepts. By describing for example how the locations in the adventure are connected you have described the geography in which the story will take place. Much of what should be described is in terms of text that should be output to make the player experience the story that you have designed.

Another fundament of Alan is that the execution of an adventure is event driven. This means that the things happening in the adventure are triggered by one thing only, namely the input of a player command. This command is analysed according to the allowed syntax and transformed into execution of verbs or movements which in turn executes other part of the description in the Alan source. After a player turn other actors can move and scheduled events can be run, then the player takes another turn.

2.3.1. The Map

In Alan, the map is a number of locations connected (or not!) by any number of exits. A location has a description that is presented to the player when it is entered. A location may also have a number of exits stating in which direction the exit is and to which location it leads. Alan places no restrictions on the layout of the map, any topology is allowed. Note, however, that since exits are one-way, an explicit declaration of a backward path (if such is desired) must be made.

2.3.2. The Objects

The other vital entity in Alan is the object. Most Alan objects are things that in real life would be objects too, like a knife or a key. In addition, other things that should be possible to manipulate by the player, e.g. parts of the scenery, must be declared as an object. For example if you require the player to ‘whistle the melody’, then the melody must be an Alan object.

Objects, like locations, have a description that is presented when they are encountered during the game.

Every object may also have a set of properties, like eatable and movable, which may be changed during the execution of an Alan program. Most objects would probably not be edible so there is also a mechanism for giving default properties to objects.

Some verbs have special meaning or effects when applied to a certain object. These verbs and their effects are also declared within the object.

2.3.3. The Verbs

Verbs are imperative statements input by the player to command some action. These must also be declared in an Alan adventure, either in an object (as described above) or as a general (global) verb that describes the effects of the verb when applied to any object or for verbs to which no object may be given.

To make it possible for the player to input more complex commands a means to specify the syntax for a verb is also available. Using this verbs can also be made to operate on literals (strings and integers) giving the player the possibility to input things like

```
> write "Merry Christmas, Mr. Lawrence" on the xmas card
```

2.3.4. The Actors

An extra thrill and dimension are additional characters in the game. These are called actors and have a life of their own. For each move the player makes these programmed characters also get a turn to do their thing. An actor may be a thief running around and stealing your collected treasures, a dragon guarding the entrance to its lair and so on.

Actors get their behaviour from scripts that step-by-step describes what is going to happen for each player interaction.

One of the interesting things about playing adventure games with actors is to figure out how to interact with and influence the other characters.

2.4. ALAN LANGUAGE DESCRIPTION

Alan is an Adventure language, i.e. a language designed to make it easy to write Adventures. This means that the Alan language must reflect the various entities encountered when creating an Adventure plot.

The first step after having come up with a plot for your Adventure is to draw a map of the world where the Adventure is taking place. For this purpose the `LOCATION` construct is provided.

The next step is to introduce tools, weapons and other objects possible to manipulate. Here the `OBJECT` construct is used.

Then the player will need words to command action. The Alan language construct to supply these with is the `VERB`. You may also define more complex types of player input using the `SYNTAX` construct.

Additionally, you may also want other characters and creatures in your adventure. For this the `ACTOR` construct is provided.

2.4.1. The Location Construct

The scene for your Adventure is a series of “rooms” or, rather, locations. They are connected by paths leading from one location to another. This makes it possible for the hero to travel through the world of your design, exploring it and solving the puzzles.

What is required if we want to describe a location? Every location (or LOCATION, when we are referring to the Alan language construct) must have a name. This is so that you, the designer, may refer to that LOCATION easily, instead of having to remember a magic number for each LOCATION.

Unless you provide other means for transportation from a LOCATION, you should also describe in which directions there are EXITs and to which LOCATIONs they lead.

In fact, this is all that is really necessary in a LOCATION, so lets look at an example (you would probably like to try this out, referring to appendix E, *SYSTEM DETAILS*, on page 111 for instructions for your particular system).

```
LOCATION Kitchen
   EXIT east TO hallway.
END LOCATION Kitchen.

LOCATION Hallway
   EXIT west TO kitchen.
END LOCATION Hallway.

START AT Kitchen.
```

This is a complete Alan Adventure (although very primitive). As you see, every Alan construct ends with a period (’.’) and there is also a “START AT” sentence at the end, indicating in which location to put the hero when the game starts.

Run this little Alan source through the Alan Compiler (see appendix D , *HOW TO USE THE SYSTEM*, on page 110 and appendix E , *SYSTEM DETAILS*, on page 112) and try the Adventure. After starting the Adventure, two lines will be shown on your screen. The first line will contain “Kitchen” and the second a “>”, which is the prompt for the player to input a command. Now try typing “east”. The word “Hallway” and the prompt will appear. Typing “west” will take you back to “Kitchen” again.

We see that the name of a LOCATION is automatically used as a description shown when that room is entered. We also see that the words listed in the EXIT-parts are translated by Alan into directional commands that are usable by the player.

You should remember that exits are strictly one-way. An EXIT from a LOCATION to another does not automatically imply the opposite path. Thus one must explicitly declare the path back, in the definition of the other LOCATION.

But just the name of the location is not much of a description. So in order to provide the “purple prose” descriptions often found in high-class Adventures there is an optional DESCRIPTION-clause in the LOCATION. Let us describe the Hallway.

```
LOCATION Hallway
  DESCRIPTION
    "In front of you is a long hallway. In one end
    is the front door, in the other a doorway. From
    the smell of things the doorway leads to the
    Kitchen."
  EXIT west TO Kitchen.
END Hallway.
```

We introduce another feature in this example, namely the text enclosed in double quotation marks (") which is called a STRING or, when used on its own like this, an output statement. When executed this string will be presented to the player and formatted to suit the format of his screen.

Invent a description for the Kitchen, enter it in the Alan source and run the changed adventure. You notice, of course, that the text in the output statements is reformatted during output to suit your screen, in order to make room for as much text as possible. Note also that you do not have to worry about this at all - in your source file you may format the text any way you like.

This type of output statement is just one of the statements in the Alan Language, and we will see more of them later.

It is also possible to have conditions and statements in the EXIT-clauses of a LOCATION to restrict the access to the next location or to describe what happens during this movement.

```
EXIT west To Kitchen
  CHECK    kitchen_door IS open
    ELSE "The door is closed."
  DOES
    "As you enter the kitchen the smell of
    something burning is getting stronger."
END EXIT west.
```

2.4.2. The Object Construct

Another essential feature in Alan is the OBJECT. Like the LOCATION, the OBJECT is a means to describe the “physical” world where your Adventure is taking place. Most objects are probably used to provide puzzles, like closed doors, keys and so on, but other objects should be promoted to OBJECTs too. A large number of objects that can be examined and manipulated makes a game so much more enjoyable.

OBJECTs, like LOCATIONs, have names and descriptions, so you might guess the general structure of an OBJECT:

```
OBJECT door AT Hallway
  IS closed.
  DESCRIPTION
    "The door to the kitchen is a sliding door."
    IF door IS closed THEN
      "It is closed."
    ELSE
      "It is open."
    END IF.
END OBJECT door.
```

An OBJECT may initially be located at a particular LOCATION (although objects do not have to start at a particular place in which case they are not present in the game until located at some place where the player may lay his hands on them). This is indicated by the AT-clause, in this case telling us that the door is initially located in the Hallway.

In addition, OBJECTs may have attributes indicating the state of certain properties of the object. In this example with a door, the `IS closed` part indicates that the OBJECT door should initially have the attribute closed set to TRUE (i.e. the door is initially closed). The opposite would be indicated with a NOT, (i.e. `IS NOT` closed).

Alternatively, attributes may be numeric (e.g. `HAS weight 5`) or be of string type (e.g. `HAS inscription "Kilroy was here"`).

We also introduce another Alan statement, the IF statement. The IF statement allows you to select which statements to execute according to some condition. In the example, the `closed` attribute of the door selects which description to show. There are further variations of expressions and the IF statement, but we will come back to these later (*Expressions* on page 51 and *If* on page 49).

Instead let's look at some other statements in relation to OBJECTs.

The attributes of an OBJECT must, of course, be changeable, and this is done with the MAKE statement or the SET statement. For example if the door should be opened (the player having said "open door", perhaps) this could be performed by

```
MAKE door NOT closed.
```

or closed (i.e. setting the closed attribute to TRUE again) by

```
MAKE door closed.
```

The SET statement changes numeric or string attributes, for example

```
SET level OF bottle TO 4.
```

Another OBJECT manipulating statement is the LOCATE statement. This is the statement to use when moving objects from one location to another. Opening a lid might cause a previously hidden object to fall to the floor, something that could be performed by moving the object from a limbo LOCATION to the current one with

```
LOCATE treasure HERE.
```

Or to a particular place with

```
LOCATE vase AT hallway.
```

2.4.3. The Actor Construct

The ACTOR is used to populate the adventure with other creatures. They might be pirates or monsters, but the thing they have in common is that they move around and perform various actions more or less in the same way as the player does.

An ACTOR may have a DESCRIPTION and attributes like OBJECTs and LOCATIONs. An ACTOR performs his movements by following scripts, each having a number of steps. Each step corresponds to one player move.

```
ACTOR charlie_chaplin NAME Charlie Chaplin
   SCRIPT going_out
      STEP
         LOCATE ACTOR AT outside_house.
      STEP
         LOCATE ACTOR AT hallway.
         USE SCRIPT going_out.
END ACTOR charlie_chaplin.
```

2.4.4. The Verb Construct

The VERB is the construct that implements the effects of an action requested by the player. VERBs may be global, local to a particular LOCATION or associated with an OBJECT. We will look at the implications of various combinations of these in the next few sections.

To implement a VERB you need a name for it (which is also the default word the player should input to request that action). You must also decide which effects this verb should have under various circumstances.

If we want to implement the VERB open for the door we could use the following code

```
VERB open
   DOES
      MAKE door NOT closed.
END VERB open.
```

This implementation makes direct references to the door, so to make the verb more general it would instead need to reference the object the player mentioned in his command (see *The Syntax* on page 13 for a discussion). In this case the attribute closed must also be available for all objects (by making it a default object attribute, see *Attributes And Default Attributes* on page 27).

Of course, there are also conditions that have to be checked before we could execute this code (perhaps to see if it was possible to open the object!). Therefore VERBs may have CHECKs.

CHECKING THINGS

In order to assert that the correct conditions are fulfilled before a VERB is actually executed the VERB has an optional CHECK part.

```
VERB open
   CHECK OBJECT IS openable
      ELSE "You can't open the $o."
   DOES
      MAKE OBJECT NOT closed.
END VERB open.
```

This is a more probable definition of the open VERB than the previous one. What it means is that before the statements after DOES are executed, the condition after

CHECK must be checked (that the object indicated by the player is really openable). If the condition is TRUE then the requirements are fulfilled and the body of the VERB may be executed. If this is not the case, the ELSE part is executed instead (normally some error message).

A CHECK may have multiple conditions as the following code shows:

```
VERB take
   CHECK OBJECT takeable
      ELSE "You can't take that."
   AND OBJECT NOT IN inventory
      ELSE "You already have it."
   DOES
      LOCATE OBJECT IN inventory.
END VERB take.
```

Here we encounter a variation on the LOCATE statement - the capability to place an object inside a container.

GLOBAL, LOCAL AND OBJECTIVE VERBS

VERBs may be defined on three levels.

- Globally. These are always used, no matter in what location the player currently is, or what object he is trying to manipulate.
- Locally (within a particular LOCATION). A local VERB is only considered when the player issues the VERB at a particular LOCATION.
- Within an object. When the player tries to manipulate the object within which the VERB is defined, the VERB definition in that OBJECT is executed.

A VERB may be defined on all three levels (as well as in other LOCATIONs and OBJECTs of course), and may have CHECKs in all instances. The implication is that all CHECKs must be passed before any execution and if they all do pass the verb bodies (DOES parts) are executed. The order is global/local/ object.

An example:

```
VERB throw
   CHECK OBJECT IN inventory ...
   DOES
      LOCATE OBJECT HERE.
END VERB throw.
LOCATION dark_place
   VERB throw
      CHECK "Too dark to aim."
   END VERB.
END LOCATION dark_place.
OBJECT vase
   VERB throw
      DOES
         "The vase breaks."
         LOCATE vase AT limbo.
   END VERB throw.
END OBJECT vase.
```

The CHECK without a condition in dark_place is called unconditional and is always FALSE (i.e. it will always fall out as if the CHECK had failed).

Now assume the player is carrying the vase at dark_place. He says

```
> throw vase
```

So we have a VERB globally as well as locally and in the mentioned object. The CHECKs are examined in the following order:

- OBJECT IN inventory? (in the global VERB)
- unconditional (in the LOCATION)
- None (in the OBJECT)

We fall out already in the LOCATION (player receiving the response “Too dark to aim.”) so the third (empty) CHECK is never tested. Now the player tries the same thing at bright_place where there is no restriction on throwing (no local VERB “throw”).

This time there is no local VERB so we skip that level and get the CHECKs

- OBJECT IN inventory? (in the global VERB)
- None (in the OBJECT)

Each is tried in turn and none fail, so we can go ahead and execute. This is done in the same order, i.e.

- LOCATE OBJECT HERE (in the global VERB)
- nothing (in the LOCATION)
- ”The vase breaks...” (in the OBJECT)

You can never destroy an OBJECT or remove it from the game. Instead, you will probably need a limbo location, i.e. a location that is not connected to any of the others and may thus be used as a storage for destroyed objects and other things the player is not supposed to see.

2.4.5. The Syntax Construct

Normally a verb acts on one object or actor, henceforth called a parameter. This means that the format of player input normally is something like

```
> take vase
```

This form, or syntax, is the default form when you don’t specify anything else. The default syntax might thus be described as

```
SYNTAX
   ? = ? (object)
      WHERE object ISA OBJECT.
```

where the question marks are replaced by the name of the verb.

But in order to allow different and more complex player input the SYNTAX construct is supplied.

The SYNTAX construct is a way to describe the words and parameters the player may use in order to execute a particular verb (its global and more specialised parts). Below is the syntax for `put_in`, the verb to put something inside a container.

```
SYNTAX
   put_in = 'put' (obj) 'in' (cont).
```

This syntax defines the `put_in` verb to be executed when the player has input the word `'put'` followed by a reference to an object or actor (a parameter named `obj`), followed by the word `'in'` followed by a reference to a second parameter (the container), as in

```
> put the green pearl in the black box
```

This will bind the parameter `obj` to the object that represents the green pearl and the parameter `cont` to the black box. It is also possible to restrict the types of the parameters:

```
SYNTAX
     put_in = 'put' (obj) 'in' (cont)
        WHERE obj ISA OBJECT
           ELSE "You can't put that into anything."
        AND cont ISA CONTAINER OBJECT
           ELSE "Nothing fits inside that."
```

This restricts the parameter `obj` to being an object (as opposed to an actor for example) and the parameter `cont` to a container object (an object with the container property).

The parameters are used as normal identifiers in the Alan source, provided they are defined in the current context, i.e. they can only be used in the bodies of the verb (see also *Run-time Contexts* on page 61 for a detailed discussion).

The SYNTAX construct generalises the verb execution order described previously from execution of verbs in one object, to verb bodies in all the parameters. In the example above, verb bodies in both the object or actor referenced as `obj` and `cont` (the green pearl and the black box) are executed (if present in their declarations).

Another use for the SYNTAX is to define the syntax for simple verbs such as `quit`, `score` etc. They also need a SYNTAX definition as they do not fit into the default verb/object format. An example would be

```
SYNTAX q = 'quit'.
```

But for simple verb/object forms no SYNTAX is actually necessary.

In expressions, OBJECT always refers to the first parameter. This makes it consistent with the default syntax of verb/object (and also with the definition of OBJECT in version 1).

3. THE LANGUAGE

This chapter describes the Alan language in detail. For each construct the exact syntax can be found in the form of BNF productions in appendix B, *ALAN LANGUAGE GRAMMAR*, on page 91.

3.1. AN ADVENTURE

An adventure starts with an optional options section (see *Options* below) followed by a set of units.

The units constitute the major part of the adventure. The units are rules, synonyms, syntax definitions, verbs, locations, objects, containers, actors and events can be declared in any order. Any combination and number are allowed. Default attributes (see *Attributes And Default Attributes* below) for objects, locations and actors may be declared in any number of places.

The adventure source text must end with a start section. It indicates where the hero is when the game starts and can also be used to set things up, welcome the player and so on. The start section is mandatory.

```
START AT bedroom.
   SCHEDULE alarm_clock AFTER 2.
   "Slowly you come to your senses, your numb limbs
    starting to feel the blood flowing through them..."
```

3.2. OPTIONS

Options define things concerning the overall behaviour of the generated Alan adventure. An option is for example written either as

```
LANGUAGE Swedish.
```

(for multiple-valued options) or

```
PACK.
NO PACK.
```

(for boolean options).

The options are

Option name	Values	Default value
Language	English,Swedish[1]	English
Width	24-255	80

[1] Other non-English languages may be supported in the future depending on demand.

Length	5-255	24
Pack	Boolean (on or off)	Off (No Pack)
Debug	Boolean (on or off)	Off (No Debug)

The Language option specifies in which language the adventure is intended to be played, and selects different default message texts. Alan is primarily designed for adventures in the English language, but it is also possible to write adventures in other languages. To make this possible, the default messages output by the interpreter may be generated in different languages.

The Alan compiler and interpreter will always allow multinational 8-bit characters as input and the default messages is generated for 8-bit character sets, internally representing national characters according to the ISO multinational character set (ISO8859-1) requiring 8 bits. On output this is converted to the native character set of the machine (whenever possible) which means that portability between platforms should be good even for text containing non-ASCII characters.

Width specifies how long the lines the interpreter outputs should be (formatting is automatic!). The Length option will instruct the interpreter to how many lines to show on the screen without any player interaction (`<More>`).

In some environments the Width and Length options may be overridden by the current values of the screen or window if the operating system can supply them.

The Pack option will cause the compiler to compress the texts to occupy less space. As a bonus this also makes it impossible for the player to cheat by dumping the adventure text data file. As a drawback it does make the execution of the adventure a bit slower (quite noticeable on some smaller computers).

In order to allow debugging of the generated adventure (see *Debugging* on page 75), the debug option must be turned on. This may also be performed using the debug compiler flag (see also *Compiler Switches,* on page 110).

3.3. ATTRIBUTES AND DEFAULT ATTRIBUTES

An attribute is a definition of a property of either an object, an actor or a location (for a description of these see the appropriate sections below). An attribute can be boolean (having the value `TRUE` or `FALSE`), numeric or of string type. The type of an attribute is automatically inferred from the type of its initial value. Attributes may either be given to all objects, actors and locations (`DEFAULT ATTRIBUTES`), to all object, actors or locations respectively (`OBJECT/ACTOR/LOCATION ATTRIBUTES`) or for a single object, actor or location (local attributes, see for example *Attributes* on page 37 for attributes for locations).

A boolean attribute is declared by simply giving the attribute name, or the name preceded with the keyword `NOT` (indicating a `FALSE` initial value:

```
thirsty.
NOT human.
```

Numeric and string attributes are declared by simply typing the value after the attribute name:

```
weight 42.
message "Enter password:".
```

General attributes that every object, actor *and* location (all *entities*) have by default should be declared in a `DEFAULT ATTRIBUTES` section. To declare a boolean attribute that all objects, actors and location will have in common the following code can be used:

```
DEFAULT ATTRIBUTES
   NOT human.
```

This attribute will now be available in all entities, and if it is not set to a different value it will be false. To get another value for a particular object, actor or location you can declare it in its declaration and give it its desired value, which will be effective only for that object, actor or location.

Attributes that every object, actor or location respectively has by default should be declared in an `OBJECT/ACTOR/LOCATION ATTRIBUTES` (respectively) section. A numeric attribute that all objects must have can be declared by:

```
OBJECT ATTRIBUTES
   weight 5.
```

All *objects* will have the attribute `weight` with the default value 5.

Another example

```
ACTOR ATTRIBUTES
   NOT hungry.
   weight 50.
```

By combining these two level of defaults you can create attributes that all objects, actors and locations have but with different default values for each of these classes. For example:

```
DEFAULT ATTRIBUTES
   NOT human.

ACTOR ATTRIBUTES
   human.
```

will give all entitites the attribute `human` but the default value of the attribute will be different for objects and locations (false) and actors (true).

Note that string valued attributes are mainly intended for saving string parameters from the player input, like in

```
> scribble "Kilroy was here" on the wall
```

It is not intended for keeping long strings of descriptions, especially not as default attributes as they (in the current implementation) require much space and takes long time to initialise when starting the game.

Any number of default attributes sections are allowed. This makes it possible to group verb declarations (see below) and the declaration of the default attributes that a particular verb requires. For example:

```
OBJECT ATTRIBUTES
     NOT takeable,

VERB take
     CHECK OBJECT IS takeable
       ELSE "You can not take that."
     ...
```

This is a practical structuring aid that allows localisation of dependencies between verbs and attributes.

3.4. SYNONYMS

Synonyms declare words that, when used as player input, are interchangeable at all times.

```
SYNONYMS
   'i', 'invent' = 'inventory'.
   'q' = 'quit'.
```

The word on the right hand side of the equal sign must be a word defined elsewhere in the adventure source, such as (part of) an object or actor name (a noun or adjective) or a direction. The list of words on the left hand side are new words (*NOT* defined elsewhere) that always will be replaced by the word on the right in the player input.

When defining synonyms remember that this only defines player *words* that are interchangeable. Defining synonyms for verb names etc. will not always give you the result that you expect. For example

```
SYNONYMS
     'examine' = look_at.
   SYNTAX
     look_at = 'look' 'at' (obj).
   VERB look_at ...
```

This will result in the compiler issuing an error message indicating that the synonym word 'look_at' is not defined. This is because the SYNTAX (see below) defined the *verb* look_at to have the specified syntax (including the player words 'look' and 'at'), the player word 'look_at' is not defined, which is as well as the player would not be able to input a word with an underscore (see *Player Input* on page 60).

You can achieve the desired effect by instead giving multiple verb identifiers in the verb declarations, this will give the same verb bodies (checks and actions) to multiple verbs. See *Verbs* on page 33 for details on verb declarations.

It is also possible to define multiple names for an object or actor to achieve other effects similar to synonyms. See *Objects* on page 38 for a description of this.

3.5. MESSAGES

The Alan system has a number of standard messages built in. These messages are presented to the player in various situations, both normal and otherwise. An example is the following:

```
> go north
   You can't go that way.
```

The response "You can't go that way." is a typical example of such system messages (for details see appendix A.1, *Input Response Messages*, on page 1).

To make the user dialogue more adapted to the settings you select Alan allows you to define your own versions of these messages. An example of this is:

```
:
MESSAGE
   NOWAY: "There is no exit in that direction."
:
```

If the above is used in the source for same game as the previous example, it would instead look like:

```
> go north
There is no exit in that direction.
```

The MESSAGE constructs allows general statements following the message identifier:

```
MESSAGE:
   NOWAY:
      IF RANDOM 1 TO 2 = 1 THEN
         "There is no way in that direction."
      ELSE
         "You can't go there."
      END IF.
```

The standard message for NOWAY is replaced by the output from the statements in the definition. For a complete list of all the identifiers of messages and their use see appendix 0, *RUN-TIME MESSAGES*, on page 1.

3.6. SYNTAX DEFINITIONS

The syntax construct is used to specify the allowed structure of the input from the player. Each definition defines the syntax for one VERB. The effects triggerd by the player input are declared using the VERB construct (see *Verbs* below).

The syntax is defined as a number of *syntax elements* each being either a player word or the name of a parameter (an identifier enclosed in parenthesis).

```
SYNTAX
   quit = 'quit'.
   examine = 'examine' (obj).
```

When the player inputs a command the set of allowed syntaxes are checked for match, giving a very flexible way to extend the allowed command set (see also *Player Input* on page 60 for details on general player input).

After the player input has been matched to an allowed syntax the parameters are bound to the entities referred to by the player. The identifiers in the syntax declaration then refers to those entities and tests for attributes etc. will be done in the entity referred by the parameter.

In the example above the parameter `obj` can be used in the declaration of the verb examine and will refer to such a bound entity.

INDICATORS

Following a parameter `indicators` are allowed. These indicators can be one of

`'*'` indicating that this parameter can reference multiple objects or actors (for example by the player using `all` or concatenating a number of parameters using a conjunction like `and`, see *Player Input* on page 60).

'!' indicating that the parameter (the object or actor given in this position) need not be present at the current location. Normally the Alan interpreter requires that a referred object or actor must be present at the same location as the hero (as this is the most common case). But for the more rare cases where the player must be able to refer to objects and actors that are not present (e.g. in a verb like `talk_about`) this omnipotent indicator can be used to force the interpreter to accept references to any object or actor.

An example

```
SYNTAX
   take = 'take' (obj)*.
   drop = 'drop' (obj).
```

This shows the syntax definitions for the verbs `take` and `drop`, `take` also allowing multiple objects. This would allow inputs like

```
> take everything except the pillow
> drop the vase
```

but not

```
> drop the shovel and the bucket
```

Another example using the '!' indicator:

```
SYNTAX
   talk_about = 'talk' 'to' (act) 'about' (sub)!.
   find = 'find' (obj)!.
```

This will give the player the possibility to say

```
> talk to the policeman about the robber
> find the key
```

even though the robber or the key are not present.

For more information on player inputs refer to *Player Input* on page 60.

CLASS RESTRICTIONS

To restrict the types of entities the player may refer to in the place of a parameter its class can be defined by using explicit test in the syntax declaration.

The following example describes the syntax for a verb which only allows OBJECTs as its parameters (this is however also the default, see below).

```
SYNTAX
   take = 'take' (obj)
     WHERE obj ISA OBJECT
        ELSE "You can't take that."
```

Each parameter may be restricted to refer only to certain kinds (classes) of entities: objects, objects with the container property, actors, numeric literals, string literals or some combination of these. The statements following the ELSE will be executed if that restriction is not met, i.e. if the player made a reference to an entity not in the specified class or classes. The default is OBJECT, i.e. if no class tests are supplied for that parameter identifier the player may only refer to objects at that position in his input.

So a more elaborate example of prerequisites for conversation might look like:

```
SYNTAX
   talk_about = 'talk' 'to' (act) 'about' (sub)!
     WHERE act ISA ACTOR
        ELSE "Don't you think talking to a person might
               be better?!?!"
...
```

The classes defined for a parameter are also used by the compiler to analyse statements and expressions in which that parameter occurs to ensure that the entity referenced is guaranteed to have the properties required during run- time. A parameter identifier defined using ISA OBJECT may for example not be used in a LIST statement as this requires the entity to have the container property (ISA CONTAINER would of course restrict the entities to only those entities that are containers and would do the trick).

As both actors and objects may have the container property it is possible to restrict parameters to only objects that are containers (CONTAINER OBJECT), only actors that are containers (CONTAINER ACTOR), or that it need just have the container property (either an object or an actor). This last case will only allow access to global default attributes (see *Attributes And Default Attributes* on page 27) of the parameter, as you can not be sure if it is an actor or an object.

DEFAULT SYNTAX

If no SYNTAX is defined for a VERB at all, this is equivalent to specifying

```
SYNTAX ? = ? (object).
```

The question marks represent the name of the VERB. This means that normal verb/object type of VERBs by default have the correct syntax and may only refer to objects. It also implies that the default name for the single parameter is OBJECT (see *WHAT Specifications* on page 53 for the implications of this).

Following this default mechanism all verbs that have no corresponding syntax declaration will be assumed to require an object as parameter. This means that simple 'verb-only' VERBs must be declared using a syntax like `q = 'q'` to make it possible to input a single verb word. It also means that verbs that have no SYNTAX will only accept OBJECTs, not ACTORs for example.

3.7. VERBS

A verb declaration specifies the checks that has to be performed and the effects of something the player does (commands using a syntactically legal input).

```
VERB take, get
  ...
END VERB take.
```

A verb can be declared at three different levels, global (outside any other declaration), inside a declaration of a location or and inside an object or actor. The meaning of this is that the global declaration will always be considered, a declaration inside a location will only be considered if the hero is at this location when the verb is executed. Finally a verb declaration inside an object or actor declaration will only be considered if that object or actor is used as a parameter in the input.

A verb *need* not be declared at all of these places.

The identifiers in the list (take and get in the example) will by default be the player words that can be used to invoke the verb. But if a SYNTAX is declared for the VERB (see *Syntax Definitions* above), the identifiers in the list will not be accessible to the player, instead the sequence of words and parameters specified in the SYNTAX must be used.

If more than one identifier is used in the list, as in the example above, this can be viewed as a short hand for declaring identical checks and bodies for all the verbs in the list. This in effect will create synonymous actions for different verbs on the level where the verb declaration is. They may differ in implementation at other places, i.e. if take and get are declared in the same verb declaration on the global level, they can still have different bodies in a particular location, in fact if they must have the same implementation they must both be declared together where this is required. For example

```
VERB take, get ...
LOCATION untakable_place
  VERB take ...
END LOCATION untakable_place.
```

Suppose that the declaration of `take` in the location prohibits taking things, the global action of `get` will still function.

CHECKS

To decide if the action is possible to carry out, the CHECKs are executed. First the global checks are tried, then the checks in the verb declaration at the current

location (if the verb has a specification in the current location) and finally the checks declared for the verb in the objects or actors bound to the parameters (if any).

```
VERB take
  CHECK obj IS moveable
    ELSE "you can't take that."
  ...
END VERB take.
```

If no expression is specified for a check, the check will always fail, in effect an unconditional check. This is useful for preventing certain actions at specific locations for example, since the checks are always executed first.

```
LOCATION l
  VERB jump
    CHECK "You can't do that here."
  END VERB jump.
END LOCATION l.
```

If any check should fail, the execution of the current verb is interrupted and the statements following the failing check are executed. The user (player) is then prompted for another command.

In addition the CHECK is used when handling the user input ALL (see *Player Input* on page 60 for details on possible player input). The mechanisms for this involve examining all objects at the current location and evaluating all checks for the verb. Any objects that do not pass the checks are not considered for execution. This restricts the handling of ALL to only executing the verb bodies for objects that are reasonable, and will not fail in the CHECK.

For example assuming the above definition of the verb take and a location containing the two objects, `ball` and `box,` of which only the `ball` is `takeable` the player input

> take all

would result in `all` representing only the ball. See *Player Input* on page 60 for an explanation of the player view of this.

DOES-CLAUSE

If all checks succeed the DOES-part(s) of the VERB will be carried out. The order is normally to first execute the body of any global declaration, then the body in the verb declaration for the current location. Finally each parameter is examined to find any declarations of the VERB inside what it refers to, those verb bodies are then executed in the order in which the parameters occurred in the syntax declaration. This is the most natural order and covers most cases but in some infrequent situations another order may be necessary. By using the qualifiers, BEFORE/AFTER/ONLY, the author can decide which verb bodies will be executed and in which order (see section **Fel! Hittar inte referenskälla.** below for details).

```
VERB take
  CHECK obj NOT IN inventory
```

```
      ELSE "You already have that."
   DOES
      LOCATE obj IN inventory.
END VERB take.
```

3.7.1. Verb Alternatives

When a VERB is declared inside an OBJECT, verb alternatives are allowed. These alternatives are used in conjunction with the SYNTAX declaration defined for the verb and allows differentiating between the object and the actor occurring in different places in the input.

When a player inputs a command each parameter in the syntax (see above) is bound to an actual object or actor or receives the value of a literal, depending on the specified syntax. To find out which CHECKs to test and verb bodies to execute the parameters are examined in turn according to the algorithm described in *VERB QUALIFICATION* below. Each object may have different verb bodies executed depending on at which position it occurred (to which parameter it was bound).

For example with the syntax definition

```
SYNTAX break_with = 'break' (o) 'with' (w).
```

the VERB body for `break_with` to execute for the `delicate_vase` could differ if it occurs as the direct object (o), or if it occurs as the indirect object (w). For each such parameter in the syntax you may define different actions by supplying a verb alternative for each parameter identifier. The verb declaration could look like

```
OBJECT feather
   VERB break_with
      WHEN o DOES
         "The feather is even more flat than before."
         MAKE feather flat.
      WHEN w DOES
         "There is not much that you can break with a
           feather!"
   END VERB break_with.
END OBJECT feather.
```

3.7.2. Verb Qualification

The order in which the different verb definitions are executed is normally from the outside in, i.e. the global definition is executed first if a global definition exists, then any possible definition of this verb in the current location. Lastly, the verb bodies in the parameters (in the order they appeared in the syntax definition) on which the verb was applied (if any) is examined to find and execute their verb definitions.

In most circumstances this is the most logical order, but if another order is required the verb qualifiers AFTER, BEFORE and ONLY may be used to alter this behaviour. The qualifiers alter the order of execution and a strict definition of this is described below.

First, the verb in the last parameter (if any) is investigated and, if this definition had the BEFORE or ONLY qualifier it is executed. If the qualifier was ONLY the

execution is also aborted at this stage and no more verb definitions are examined, otherwise the other parameters are examined in the same way.

In the next step, the current location is examined and, if it contained a verb definition with a BEFORE or ONLY qualifier, that definition is now executed (and if it was ONLY, execution is aborted). As a result a BEFORE qualifier in the verb definition in an object will supersede an ONLY qualifier in the location.

At this stage, all BEFORE and ONLY qualifiers are handled appropriately since the global definition is now in turn anyway. This leaves the definitions without any qualifier or with the AFTER qualifier. The global definition is examined and if it did not have the AFTER specification, it is executed (if it had an ONLY qualifier execution is stopped after executing it). Any definition of the verb in the current location is again examined and, if it did not have the AFTER qualifier, it is executed. What remains is to execute the verb definition in the parameters if they have not been executed already, and to execute the location definition and the global definition (in that order) if they where declared with the AFTER qualifier.

So in short (with global definitions being the outermost and the definition in the entity bound to the last syntax parameter the innermost):

- From the outside in, find any BEFORE or ONLY definitions and execute them (stop if ONLY found).
- From the inside out, execute any definitions not already executed and not declared with the AFTER qualifier.
- Execute the remaining verb definitions (those with an AFTER qualifier) from the outside in.

The second item in the above list represents the normal order of execution.

The qualifiers are a powerful but confusing concept. The normal order of execution is usually appropriate and only in special cases should qualifiers be used. When they are needed, you will find that one qualifier at the correct definition will normally do the trick. The above algorithm is used to get a strict definition of the execution order. It is not expected that this complex behaviour will be needed in practice.

Note: All checks for a VERB will always be run in global-location-parameter order regardless of any BEFORE/AFTER/ONLY qualifiers.

An example of the use of qualifiers is to ensure that only the verb body within the object is executed:

```
OBJECT bomb
  VERB take
    DOES ONLY
      "Your curious fingering at the intricate
        mechanism sets it of. BOOOM!"
      QUIT.
  END VERB examine.
END OBJECT bomb.
```

3.8. LOCATIONS

A location is a declaration of a place (a “room”) in the game that (normally) can be visited by the player, have objects lying around, etc. In fact the map of your game is a set of interconnected locations.

IDENTIFIER AND NAME

The ID is the identifier used by the author throughout the source when referring to this location. By default, this will also be the name of the location written out to the player. But by using the NAME clause you can give a different name to the location when presenting it to the player (see *Objects* on page 38 and *Identifiers And Names* on page 57). For example

```
LOCATION south_of_house
   NAME 'South of House'
...
```

See *Identifiers And Names* on page 57 also for an explanation of the quoted identifiers used in this example.

ATTRIBUTES

A location can have attributes (see *Attributes And Default Attributes* on page 27). These can be local attributes available only for this location or override declared default attributes.

```
LOCATION south_of_house
   NAME 'South of House'
   IS outdoors.
...
```

DESCRIPTION

The statements in the DESCRIPTION clause should print a description of the location. These statements are executed when the hero enters the location or when executing a LOOK statement. See also *SPECIAL STATEMENTS* on page 47, concerning the VISITS statement.

```
LOCATION south_of_house
   NAME 'South of House'
   IS outdoors.
   DESCRIPTION
      "You are facing the south side of a white house.
There is no door here, and all the windows are barred."
   ...
```

DOES-CLAUSE

The optional DOES clause contains statements performed when *any* actor enters the room (is located there). An example usage of this would be if there were a weak bridge that only allows a certain total weight before it collapsed. The DOES clause of that location could contain actions for this, which would be executed whenever any actor enters that location, not only the hero.

EXITS

To build a world of locations, these must be connected. This is done using exits. An exit consists of an `id_list`, all of which are considered directional words, i.e. when input by the player they will move him to the location identified by the `ID`. It is possible to customize the exit using `CHECKs` (see *Verbs* on page 33 for a definition), that must be satisfied to allow passage through the exit, and statements that will be executed when the player passes through.

Note: If there exists an exit from one location to another, there will *NOT* automatically be an exit in the opposite direction!

Two interconnected locations might be declared like:

```
LOCATION east_end NAME 'East End of Hall'
  DESCRIPTION
    "This is the east end of a vast hall. Far away to
      the west you can see the west end."
  EXIT w TO west_end.
END LOCATION east_end.
LOCATION west_end NAME 'West End of Hall'
  DESCRIPTION
    "From this western end of the large hall it is
      almost impossible to discern the opposite end to
      the east."
  EXIT e TO east_end.
END LOCATION west_end.
```

VERBS

Local verbs may also be declared in a `LOCATION`. See *Verbs* on page 33 for a description of how to declare verbs.

3.9. OBJECTS

Objects are all the things that can be manipulated by the player. They can be picked up, examined and thrown away (if the author has allowed it). They will usually be described when the player enters a location containing objects.

As for locations, the `ID` is the name you use to refer to this object. It is also the default name for what is presented to the player and what he has to use when referring to the object.

NAME

By using the NAME clause you can give the object another name, e.g.

```
OBJECT chair3 NAME little wooden chair
```

In this example the word "chair" is a noun and "little" and "wooden" would be adjectives. When the player refers to the object with the author name `chair3`, he may use just "chair" if it is the only object with "chair" as its noun at the current location, or he may distinguish between multiple chairs by also giving one or more adjectives to pin down the chair he wanted.

Note: If the NAME clause is used the name `chair3` is *not* available to the player.

It is possible to give an object multiple names by listing a number of name clauses. Each one will define adjectives and a noun as described above. The result being that the player can use any of the names to refer to the object. For example:

```
OBJECT rod AT grate
   NAME rusty rod
   NAME dynamite
...
```

This would allow the player to refer to the object using either of 'rusty rod' or 'dynamite'.

INITIAL LOCATION

It is possible to set the initial location of an object by using an optional `where` clause. If no such clause is used the object will not be present in the game until it is moved somewhere by a LOCATE statement. Only the `AT what` and `IN what` forms of the where construct (see *WHERE Specifications* on page 52) are allowed when describing an initial location of an object.

```
OBJECT chest AT tresury
...
```

CONTAINER PROPERTIES

An object can also be a container. This is declared by means of the CONTAINER property clause, which looks like an ordinary container declaration (see *Containers* on page 41).

```
OBJECT chest
   CONTAINER
      LIMITS ...
      HEADER ...
   DESCRIPTION ...
   :
END OBJECT chest.
```

ATTRIBUTES

An object can have attributes (see *Attributes And Default Attributes* on page 27). These can be local attributes or override values of declared default attributes.

```
OBJECT chest AT tresury
   IS NOT open.
...
```

ARTICLE

The optional article can be used to define the indefinite article that should be placed before the object name in e.g. inventory listings and when presenting objects that have no DESCRIPTION clause. For example

```
OBJECT owl
   ARTICLE "an"
:
```

would result in things like

```
There is an owl here.
You are carrying an owl.
```

The article is not used when mentioning the object when acting on multiple objects:

```
> take everything
(owl) Taken.
```

Note: The default article, "a" (if using english), is used for objects that have no ARTICLE declared.

For objects that should not have any article, like 'some money', an ARTICLE clause containing no statements must be used:

```
OBJECT money NAME some money
   ARTICLE
:
```

This will lead to:

```
There is some money here.
```

instead of

```
There is a some money here.
```

MENTIONED

The optional MENTIONED clause specifies a short form for this object that will be used when mentioned e.g. in listings of containers or when the ALL form is used. If no MENTIONED clause is present an appropriate default message, constructed from the object name, is supplied by the system.

```
MENTIONED
   IF mirror IS broken THEN
      "broken"
   END IF.
   "mirror"
...
> take all
(little black book) OK!
(green pearl) OK!
(broken mirror) OK!
```

The MENTIONED clause is also used when describing objects that have no DESCRIPTION, by inserting the article (see above) and the short description in a default message. In the following example output the article is underlined and the short description is emphasised, the rest is the default message templates.

```
There is a little black book, a green pearl and an owl
here.
```

The interpreter uses the same principle when constructing lists of objects in container contents lists (as the result of the execution of a LIST statement, see page 47).

DESCRIPTION

Objects can of course have descriptions, statements describing the object. This description will normally be printed when the player enters the location where the object currently is. It will also be given as a result of the DESCRIBE statement, and indirectly by executing a LOOK statement at the location where the object is. If the DESCRIPTION clause is missing the Alan system will supply a default description such as “There is a round ball here.”. If there is a DESCRIPTION clause but it contains no statements the object will be ‘invisible’, i.e. no description of it will printed. This can be useful for objects already described by the location description, or of objects with particular properties.

```
DESCRIPTION
  "On the floor there is a heavy golden chest. Its sides
    and top are completely encrusted with jewels."
...
```

VERBS

As for locations, local verbs can be declared inside an object. The verb declarations inside objects is only used when that verb is applied to the object. See *Verbs* on page 33 for details on verb declaration and usage.

3.10. CONTAINERS

A container is something that can contain objects. A container can either be an object (or an actor) in itself (in which case it is declared as an object or actor with the CONTAINER property, see *Objects* on page 38) or be a pure container. A container that is not an object or actor, a pure container, can *NOT* be manipulated directly by the player. It can, however, be manipulated indirectly, if the author has supplied some verbs to do this, such as take and drop, which usually are implemented to manipulate the inventory container. A container for worn objects, is a common example of a pure container.

A container can only contain objects, not actors or locations.

LIMITS

The LIMITS clause put limitations on what and how much can be put in the container. If any of these limits are exceeded when trying to locate anything inside the container, the statements in the corresponding THEN-part will be executed and the players turn aborted. In fact these checks are performed as a consequence of the execution of a LOCATE statement (not actually the player placing anything inside the container). This means that the execution of a sequence of statements can actually be interrupted by these limitations.

The specification of an attribute, which must be a numeric object default attribute, implies that the sum of this attribute of all objects in the container can not exceed the value specified. The special attribute COUNT is also allowed and indicates a limitation on the number of objects allowed.

```
CONTAINER inventory
   LIMITS
      weight 50 THEN "You can not lift that much."
      COUNT 2 THEN "You only have two hands!"
```

HEADER AND ELSE

HEADER is used when the contents of the container are listed. It is intended to produce something like

```
"The box contains"
```

or

```
"You are carrying"
```

The ELSE-part is used instead of the header if the container is empty.

If LIMITS or HEADER is missing the Alan system supplies the default of no limits, and the messages “The $o contains” and “The $o is empty.” respectively.

THE INVENTORY

The inventory, i.e. the container containing all objects carried by the hero is predeclared[2], so that it already exists and can be used for common purposes. It can however be re-declared if required, for example to provide limits and a different header. An equivalent of its default declaration is

```
CONTAINER inventory
   LIMITS
   HEADER
END CONTAINER inventory.
```

One possible re-declaration of the inventory can serve as a more example of a container declaration.

```
CONTAINER inventory
   LIMITS
      weight 50 THEN "You can not lift that much."
   HEADER
      "You are carrying"
   ELSE
      "You are not carrying anything."
END CONTAINER inventory.
```

2 The inventory is actually the container properties of the hero (see section 3.12 on page 43 for a discussion of actors and their container properties). Any object put into the container will be available in the hero also.

3.11. EVENTS

An event is a sequence of statements executed at a specified time (count of turns). It is also executed at some specific location. An event can e.g. be used to create an explosion where the bomb is three moves from now or to let the ceiling of the cave fall down in five moves.

```
EVENT nearby_explosion
   "Somewhere in the distance there is an explosion."
   MAKE bomb gone_off.
   SCHEDULE small_avalanche AFTER 2.
END EVENT.
```

The body of an event can be any sequence of statements. They can however not refer to any parameters, including OBJECT (since no verb is executing), or the ACTOR. See *Run-time Contexts* on page 61.

Events may be scheduled and cancelled with the SCHEDULE and CANCEL statements (see *EVENT STATEMENTS* on page 50).

3.12. ACTORS

An actor is something that seems to live its own life in the game. Another common name for actors is NPC, non-player character. The author refers to the actor by using the ID, and it is also the default name presented to the player.

NAME

By means of the NAME clause, a different name can be assigned to the actor in the same way as for an object (see *Objects* on page 38).

CONTAINER PROPERTY

The optional property (CONTAINER) clause may be used to indicate that this actor can be used as a container, i.e it may contain things, thereby implying that the actor is carrying the things contained. This is analogous to objects having the container property (see *Objects* on page 38).

ATTRIBUTES

An actor can have attributes (see *Attributes And Default Attributes* on page 27). These can be local attributes or override values of declared default attributes.

```
ACTOR kirk NAME Captain Kirk AT control_room
   HAS health 25.
   CONTAINER
      HEADER "Kirk is carrying"
      ELSE "Captain Kirk is not carrying anything."
   DESCRIPTION
      "Your superior, Captain Kirk, is in the room."
END ACTOR kirk.
```

DESCRIPTION

In the DESCRIPTION clause, a description of this actor can be given. The statements describing the actor will be executed when the player enters a location where the actor currently is. This description will also be given as a result of the DESCRIBE statement. An exception is if the actor is currently executing a script for which there is a separate description (see below).

SCRIPT

The SCRIPT is the actor's way of performing things. In a way it corresponds to what the hero is ordered to do by the player's typed-in commands.

Every script has a name (or to be compatible with previous versions, a number) to identify it. A script is selected by the USE statement. When a script is started it will continue until it reaches the end or another USE statement is executed for this actor.

The optional description allowed in the beginning of a script is used instead of the general description (in the beginning of the actor declaration) whenever the actor is executing that particular script. If it is not present the general description is used.

```
ACTOR george
   NAME George Formby
   DESCRIPTION
      "George Formby is here."
   SCRIPT cleaning.
      DESCRIPTION
         "George Formby is here cleaning windows."
      STEP ...
   SCRIPT tuning.
      DESCRIPTION
         "George Formby is tuning his ukelele."
   STEP...
:
```

STEPS

A script is divided into steps. Each step contains statements representing what the actor will do in what corresponds to one player move. A step can be defined to be executed immediately next move, to wait a number of moves before it is executed or even to wait for a special situation (condition) to arise.

For example

```
STEP WAIT UNTIL HERO HERE
   "From the shadows a waiter emerges: $p'-Bonjour,
     monsieur', he says."
```

When an actor has executed the last step of the current script, it will do nothing more until the next USE statement is executed for this actor (the actor will be "dead", but still present at the location where it was). If this is not what is wanted, it is recommended to end each script with a new USE statement.

THE HERO

There is one very special actor, the hero, which is the player. This actor is always predeclared, but if necessary it can be re-declared. One situation when this is required is if you like to have attributes on the hero, such as “sleepy” or “hungry”. Then a declaration like the following is possible:

```
ACTOR hero NAME me
   IS NOT hungry.
   CONTAINER
   VERB examine DOES
      IF hero IS hungry THEN
         "Examining yourself reveals a poor, hungry soul."
      ELSE
         "You find nothing but a poor beggar."
      END IF.
   END VERB examine.
END ACTOR hero.
```

The container property of the hero is actually the inventory container, which is also predeclared, see *The inventory* on page 42.

3.13. RULES

A rule is an arbitrary expression, which, when true, results in execution of the given statements. Rules can be used to make things happen when certain situations arise, such as starting an actor when the hero enters the cave.

```
WHEN hero AT cave AND monster NOT active =>
   USE SCRIPT 3 FOR monster.
```

The statements that are to be executed can not refer to parameters (including OBJECT), but may refer to ACTOR.

The rules are tested after each actor (including the player) has made his move and after each event that is executed. So rules must be designed to be executed multiple times for each player turn. Rules can be considered to be executed at the location where the last activity (actor move or event) was performed (see also *A Turn Of Events* on page 60). This is important to consider especially concerning use of WHERE specifications (see on page 52) in rules.

3.14. START SECTION

The start section defines where the player (the hero) will be at the start of the game. This must be a location. Optionally this may be followed by statements to be executed at the beginning of the game, such as hello-messages or short instructions as well as starting any actors and scheduling events.

```
START AT outside_house.
   SCHEDULE bird_chirp AFTER 5.
```

Only the AT what form of the where construct (see *WHERE Specifications* on page 52) is allowed in the start section. Any statements are allowed in the start section except that they can not refer to any parameters.

3.15. STATEMENTS

3.15.1. Output Statements

An output statement is in the simplest case just a string, i.e. any text, possibly stretching over multiple lines, surrounded by double quotes. Whenever it is executed, the string will be printed on the players terminal with the following exception: if an output statement is executed at a location in the game where the hero not presently is the output will not be shown. This can be used in the following way in a script for the actor `charlie_chaplin`:

```
"Charlie Chaplin leaves the house through the front
door."
LOCATE charlie_chaplin AT outside_house.
"Charlie Chaplin comes out from the nearest house."
```

If the hero is inside the house or out in the street he will now get different views of the situation. This feature ensures that the player only sees what is going on at the current location, and allows for easy adaption to various viewpoints on the events without the need for any variable tests.

There are some character combinations that have special meaning for the printout:

```
$l   The name of the current location
$v   The verb the player used (the first word)
$p   New paragraph (one empty line)
$n   New line
$i   Indent on a new line
$t   Insert a tabulation
$$   Do not insert a space
$a   The name of the actor that is executing
$o   The current object (first parameter)
$<n> The parameter with number <n> (<n> is a digit)
```

Note: The $a, $o and $<n> formats must be used with care as they are not checked at compile time, e.g. you can use "$o" in a context where no parameter is defined which would lead to a run- time error. To avoid any run-time problems use the SAY statement with the parameter name. The use of $a, $o and $<n> formats may not be forward compatible.

DESCRIBE

The DESCRIBE statement executes the description part for an actor, an object or a location. If no such description exists a default description, such as

```
"There is a $o here."
```

is used instead. If the object has the container property a `LIST` statement is also executed for that object automatically (see below).

If a `DESCRIBE` statement is used for an object in the description part of a location, the system will recognise this and make sure that the object is not described more than once during the execution of a `LOOK` statement or when the hero enters that location. This makes it possible to use objects as parts of a location and

embedding their description at the correct place in the longer description of the location.

```
"This office is dusty and probably hasn't been used for
    many years."
  DESCRIBE desk.
```

SAY

The SAY statement will output a short description of what is referred to by the what part. If it refers to an entity (a LOCATION, OBJECT or ACTOR) it will print the name of that entity or execute its MENTIONED clause if one is available. If it refers to an attribute it will print its value (integer or string). Parameter names are also allowed in the SAY statement, which, of course will result in a short description of the entity to which it is bound, or a printing of the literal (if the parameter was a STRING or INTEGER parameter).

```
IF contents OF bottle > 0 THEN
    "In the bottle there are still"
    SAY contents OF bottle.
    "litres of water left."
  ELSE
    "The bottle is empty."
  END IF.
```

LIST

The LIST statement lists all objects in a container together with the header as specified for the container. If the container is empty the statements in the empty clause of the container are executed instead.

```
"The chest is heavy."
IF chest IS open THEN
  LIST chest.
END IF.
```

3.15.2. Special Statements

QUIT

QUIT prints a question giving the player the choice of restarting the game, re-loading a previously saved game or to quit. Any scoring or other printouts have to be made explicitly before executing the QUIT statement.

LOOK

LOOK describes the current location and what it contains. The DESCRIPTION part for the location is executed, which may include describing objects or actors by explicitly executing DESCRIBE statements. Then objects and actors that have not already been described will automatically be described.

SAVE AND RESTORE

SAVE saves the game on a file for later use with RESTORE. Both save and restore asks for a file name to use for storing and restoring.

If the player should be shown the current surroundings after a RESTORE, you will have to implement a player verb like

```
VERB oops
   DOES
      RESTORE.
      LOOK.
   END VERB oops.
```

SCORE

SCORE is a way of rewarding the player by giving points for certain actions. This is done using the statement

```
SCORE points.
```

for example

```
SCORE 25.
```

The first time every such statement is executed the points given are added to the player's current score. SCORE without any arguments prints a message indicating the current accumulated score.

Note: The SCORE statements assume a simple model of scoring; a number of actions are necessary to complete the game and all those are necessary to achieve the maximum number of points. For adventures having a more complex and varied scoring system (particularly if the game can be successfully finished without performing all scoring actions or in multiple ways) manual scoring should instead be implemented using attributes (e.g. on the player) and suitable manipulation and test statements.

VISITS

The VISITS statement changes the number of times a location can be visited before the long description is presented again:

```
VISITS count.
```

The value of the argument (count) controls the number of visits to a particular location between full descriptions. The default setting (0) indicates that every time a particular location is visited its full description will be shown (which can also be expressed as: the full description will *not* be shown 0 times in between). Thus, a setting of 1 (one) would give a full description every second time the same location is visited. So

```
VISITS 0.
```

will always show long descriptions (this is also the initial setting).

Note: The familiar VERBOSE, BRIEF etc. commands can be imitated using different values in the VISITS statement.

3.15.3. Manipulation Statements

LOCATE

The LOCATE statement is a way of transferring objects and actors. When executed, the indicated object or actor will be placed at the location given. For a description on how to specify where, see *WHERE Specifications* on page 52. When an actor is located at a new location the DOES clause of that location is always executed.

One special case of the LOCATE statement is when the predefined actor HERO is located somewhere. This is analogous to what happens when the player types in a direction, i.e. the player is located at the appropriate location. Under particular circumstances, you may want to locate the player at a different location as a side effect of another action. For example:

```
EVENT explosion
   "Suddenly the door seems to bulge outwards, it bursts
     open throwing rocks and splinters everywhere. The
     impact of the explosion literally throws you back
     out in the hallway."
     LOCATE HERO AT hallway.
   END EVENT explosion.
```

In this case the new location will be described and the DOES clause of that location executed.

Another special case is when locating something inside a container. The LOCATE statement will then cause the execution of the limits of that container, and if any of the limits are exceeded the complete player turn is aborted immediately, resulting in that no more statements are executed. So if a player command should result in the location of an object inside a container, a good thing is to place the LOCATE statement as early as possible, as this enforces the limit checks in the beginning of this player turn.

EMPTY

The EMPTY statement locates all objects in the given container (or object or actor with the CONTAINER property) at a certain place. The meaning of the where part is as for LOCATE.

```
EMPTY inventory HERE.
   "You seem to have lost most of your possessions. Well,
     you can't have everything."
   LOCATE hero AT restart_point.
```

3.15.4. Event Statements

SCHEDULE

SCHEDULE means that the named event will occur at the specified location after the number of moves specified by the expression.

```
SCHEDULE ringing AT clock AFTER 60 - minutes OF clock.
```

The number of moves can be zero, i.e. AFTER 0 means that the event will occur now (during this player turn). If no location is specified, HERE is assumed, i.e. it will be executed at the current location, the location where the statement itself was executed.

The semantics of specifying the location (where) as AT id, where the identifier represents an object or an actor, is that wherever that object or actor is when the event occurs, the event will be executed at that place.

Executing a second SCHEDULE statement for the same event before it has occurred will reschedule the event to the new time. So an event can only be scheduled for one execution at a time.

CANCEL

CANCEL will remove the event referenced from the queue of scheduled events.

```
EVENT ticking
     "Tick..."
     IF timer OF bomb = 0 THEN
        SCEHDULE explosion AFTER 1.
     ELSE
        DECREASE timer OF bomb.
        SCHEDULE ticking AFTER 1.
     END IF.
  END EVENT ticking.

  VERB defuse
     DOES
        CANCEL ticking.
        CANCEL explosion.
        "Phuuui! That was close."
  END VERB defuse.

  START AT office.
     "The bomb is ticking..."
     SCHEDULE ticking AFTER 1.
```

3.15.5. Assignment Statements

There are a number of statements for changing values of attributes.

MAKE

The MAKE statement is used to set or reset boolean attributes.

```
MAKE door open.
```

INCREASE AND DECREASE

The INCREASE and DECREASE statements modifies the values of numeric attributes by increasing or decreasing them by the value of the expression given in the optional BY clause. If no BY clause is specified the attributes are changed by 1 (one).

```
INCREASE level OF bottle BY contents OF mug.
DECREASE lives OF HERO.
```

SET

The SET statement is used when assigning values to numeric or string valued attributes.

```
SET mood OF king_tut TO 3.
SET hour OF clock TO hour OF clock + 1.
```

3.15.6. Conditional Statements

In Alan there are two conditional statements, the common IF statement and the DEPENDING ON statement.

IF

The IF statement is essential for being able to vary the output and otherwise change the activities in the game. The expression is evaluated (see *Expressions* on page 53 for details and examples of expression) and if it evaluates to true, the statements following the THEN are executed. Otherwise the expressions in any following ELSIF clauses are evaluated (in order) and the statements following the first expression that results in a true value is executed. If none of the expressions in the ELSIF clauses evaluated to true, or there are no ELSIF clauses, the statements following the ELSE are executed. The ELSE clause is optional.

```
IF minute OF clock = 59 THEN
   SET minute OF clock TO 0.
   INCREASE hour OF clock.
ELSE
   INCREASE minute OF clock.
END IF.

IF level OF bottle = 0 THEN
   "You have no water"
ELSIF level OF bottle < 5 THEN
   "You have almost no water left."
ELSE
   "You have plenty of water."
END IF.
```

DEPENDING ON

The DEPENDING ON statement is a provided to select one of a number of possible conditional cases depending on an expression. A simple example of the DEPENDING ON statement is:

```
DEPENDING ON weight OF OBJECT
  = 1 : "light as a feather"
  BETWEEN 2 AND 10 : "carryable"
  BETWEEN 10 AND 20 : "heavy"
  > 20 : "immobile"
  ELSE : "weightless"
END DEPENDING.
```

The meaning of this example is to test the `weight OF OBJECT` and select one of the cases depending on that. If it is equal to one the first case will be executed. If none of the cases catch the optional `ELSE` case will be executed (in this case it will only be executed for weights of zero or less).

The cases are tested in the order specified and at most one will be executed. In the example, weight 10 will render as "carryable".

3.15.7. Actor Statements

The `USE` statement starts execution of a given script for a given actor. It is possible to leave out the `FOR id` -part when writing code within a certain actor; in this case the actor that the code is in is assumed.

```
USE SCRIPT playing FOR george.
```

3.16. WHERE SPECIFICATIONS

Many constructs in the Alan language require a specification of where the construct should operate. The general intention of a `where` specification is to return a location. The meaning of the different constructs is as follows

- `HERE` is the location where the current activity is performed. Normally this means where the hero is, but if the expression is evaluated in an event scheduled at a particular place, that place is `HERE`, and the same applies to activities performed by other actors and for expressions within rules. Note that this is equivalent to `AT LOCATION`.

- `NEARBY` means any adjacent location, adjacent meaning that there exists an exit from the other location to `HERE` (note that the direction is from `NEARBY` to `HERE`).

- `AT what` means at the location of the entity referenced by the `what` specification (see *WHAT Specifications* on page 53).

- `IN what` must refer to a container and the expression refers to inside of that container.

Note: Not all kinds of where specifications are meaningful in all constructs requiring a where specification. An example is `NEARBY` which, of course, is not allowed in a `LOCATE` statement as this needs a definite location to locate to, and `NEARBY` is not specific. Instead, `NEARBY` is useful in `IF` statements to see if the monster is somewhere near.

3.17. WHAT SPECIFICATIONS

Constructs in the grammar for the Alan language often refer to some entity defined in the Alan source. This is generally called a `what` specification, as it specifies what the construct refers to. The `what` specification may have the following forms

- `OBJECT` refers to the first parameter, i.e. the first object or actor referred to by the player in the input as described by the syntax. Normally this is intended for use with verbs relying on the default syntax handling; for verbs where a `SYNTAX` construct is specified the identifiers for the parameters should be used instead (the use of syntax declarations is strongly advised).

Note: If `OBJECT` is used in an expression no compile time checks can be made on class restrictions which might lead to run-time errors when referring the first parameter. The use of `OBJECT` in expressions might not be forward compatible.

- `ACTOR` is always set to the actor currently active and this also applies to expressions and statements within rules as these are run once for each actor.

- `LOCATION` is the current location, i.e. the location where the current activity is performed. This is normally the location where the hero is, but may also be where an event is executed or where the actor currently executing (other than the hero) is.

- An identifier, `id`, refers to the entity with that name, or a syntax parameter with that name. A syntax parameter may have the same name as an entity declared elsewhere in the source in which case the parameter overrides the entity.

Note: Not all kinds of what specifications are meaningful in all contexts. For example it is not possible to use `LOCATION` (nor an identifier referring to a location) as the `what`-part of a LOCATE statement.

3.18. EXPRESSIONS

The grammar for Alan also refers to expression. This is a generic name for a number of constructs yielding a value.

3.18.1. Types Of Expressions

Expressions are needed e.g. in `IF` and `SET` statements. The `IF` statement requires a boolean expression, i.e. an expression yielding a true or false value, while the `SET` statement needs a numeric or a string value. Some types of expressions return a value referring to an entity (an object, an actor or a location) in the Alan source as is, for example, the case with an identifier bound to a parameter allowing actors or objects. So, the possible types of expressions in Alan are

- integers
- strings
- boolean
- entities

3.18.2. Literal Values

A single integer (e.g. 42) is of course a numeric expression.

The expression RANDOM integer TO integer is also a numeric value that is randomly selected between and including the two integers.

```
SET eyes OF first_die TO RANDOM 1 TO 6.
```

A string can be used in expressions and then represents a string value, e.g.

```
SET password OF terminal TO "xyzzy".
```

3.18.3. Logical Expressions

The AND and OR operators are standard binary boolean operators. AND has higher priority, but parenthesis may be used to change the order of evaluation.

```
IF kalif HERE AND mood OF sultan IS 0 THEN ...
```

3.18.4. Binary Operators

All binary operators (plus, minus, multiplication, division) may be used on integer expressions. The result is another integer expression. The exact set of available operators are

```
+, -, *, /
```

3.18.5. Relational Operators

Relational operators (=, <, >, <=, >=, meaning: equals, less than, greater than, less than or equal, greater than or equal respectively) are used to compare expressions. The result is TRUE or FALSE and may be negated by using an optional NOT.

```
IF temperature OF oven NOT > 100 THEN...
   IF weather OF world NOT < protection OF hero THEN...
```

Comparing two string expressions using the binary operator '=' will make a case insensitive comparison, i.e. it will give a true value if the strings are the same without considering the case of the characters. The special identity operator, '==', only works on strings and compares the strings for an exact match (i.e. considering character case).

There is also a string containment operator, CONTAINS, which can be used to test if a string contains another string. The test ignores any differences in character case. An expression which would give a TRUE value is

```
"A string" CONTAINS "a S"
```

An optional NOT (before CONTAINS) can be used to reverse the test.

Two identifiers referring to entities may be compared with the '=' and '<>' operators, and may be used to test if a parameter refers to a particular entity or the same as another parameter. For example

```
SYNTAX put_in = 'put' (o) 'in' (c)
   WHERE c ISA CONTAINER
      ELSE "You can't put anything in the $2"
VERB put_in
   CHECK o <> c
      ELSE "That would be a good trick if you could
           do it!!"
DOES ...
```

Relational operations are not allowed on entities or strings, nor is it possible to compare values of different types.

A special relational operator is the BETWEEN operator which makes it possible to test if a numeric expression is within a range of values. For example

```
IF level OF water BETWEEN 2 AND capacity OF bottle THEN
...
```

3.18.6. The Value Of Attributes

Expressions following the pattern

```
primary IS something
```

are used to test the setting of boolean attributes of the entity referred to by something (which is a what specification). For example

```
IF bottle IS empty THEN ...
```

The test can be reversed by adding a NOT:

```
IF hero IS NOT hungry THEN...
```

To get the value of a numeric or string attribute expressions following the pattern

```
ID 'OF' what
```

are used.

```
IF s = password OF terminal THEN ...
"You have" SAY capacity OF bottle. "sips left."
```

3.18.7. The Whereabouts Of An Entity

The expression

```
primary optional_not where
```

is used to test if a particular entity, as specified by the `what`, is (or is `not`), at the place indicated by the `where`, as in

```
IF bottle IN inventory THEN ...
```

or

```
IF HERO NEARBY THEN ...
```

3.18.8. Aggregates

Aggregates are functions to calculate values from sets of other values.

`COUNT` counts the number of objects at the specified place, e.g.

```
"You are carrying"
SAY COUNT IN inventory.
"things."
```

The `SUM` and `MAX` aggregates return the sum and the maximum value respectively of an attribute of all objects at the specified location. This implies that the attribute must be a default object attribute in order to ensure that the attribute is available for all objects. For example

```
IF SUM OF weight AT bridge > 500 THEN ...
IF MAX OF size IN inventory > size OF small_door THEN
...
```

The last example could be adopted to make various restrictions in the possible travels of the hero.

4. LEXICAL DEFINITIONS

4.1. COMMENTS

Comments may be placed anywhere in the Alan source. A comment is opened by double hyphens ('--') and extends to the end of the line.

```
-- This is a comment
```

4.2. IDENTIFIERS AND NAMES

Words used as identifiers in an Alan source may only be composed of letters, digits and underscores. The first character must be a letter.

```
identifier = letter ( letter | digit | underscore )*
```

In order to be able to use reserved words as identifiers (e.g. for verbs) there is also a second kind of identifier, namely the quoted identifier.

```
quoted_identifier = single_quote any_character+
single_quote
```

A quoted identifier starts and ends with single quotes and may contain any character except quotes (including spaces). It may be used to make an identifier out of a reserved word such as LOOK. This may be useful in the definition of the verb LOOK that then would look like:

```
VERB 'look'
  DOES
    LOOK.
END VERB 'look'.
```

Note that normal identifiers are always translated to lower case before making any comparisons so it does not matter how you (or the player) write them (although it is easier to read if the same kind of editing is used for the same kind of words). Quoted identifiers are not changed at all, so they must always be written identically. They may also contain spaces, which make them useful as long names for locations as in

```
LOCATION pluto NAME 'At the Rim of Pluto Crater'
  DESCRIPTION
    ...
```

One single quoted identifier is used as the whole name of the location so as to preserve editing and avoiding clashes with the reserved words AT and OF.

Note: Do *NOT* use a single quoted identifier as the name for anything other than locations, as the words in objects and actor names are analysed separately and assumed to be adjectives (except for the last, which is a noun). Only quote separate words to avoid clashes with reserved words.

Be careful when using quoted identifiers, especially if the player is supposed to use the word. A player can not input words containing upper case characters, underscores, spaces or other special characters or separators.

Note: To get a single quote within a quoted identifier repeat it ('Tom''s Diner').

Some of the identifiers in an Alan description are by default used as player words. This is for example the case with verb names (unless a SYNTAX statement has been declared for the VERB) and object names (unless a NAME clause has been used). If these contain special characters the player can not enter them.

4.3. NUMBERS

Numbers in Alan are only integers and thus may consist only of digits.

```
number = digit+
```

4.4. STRINGS

The string is the main lexical component in an Alan source. This is how you describe the surroundings and events to the player. Strings, therefore, are easy to enter and consist simply of a pair of double quotes surrounding any number of characters. The text may include newline characters and thus may cover multiple lines in the source.

```
string = '"' any_character+ '"'
```

When processed by the Alan compiler, any multiple spaces, newlines and tabs will be compressed to one single space as the formatting to fit the screen is done automatically during execution of the game (except for embedded formatting information, as specified in *OUTPUT STATEMENTS* on page 46). You may therefore write your strings any way you like; they will always be neatly formatted on the player's screen.

Note: As strings may contain any character a missing double quote may lead to many seemingly strange error messages. If the compiler points to the first word after a double quote and indicates that it has deleted a lot if IDs (identifiers), this is probably due to a missing end quote in the previous string.

Note: To get a double quote within strings repeat it ("The sailor said ""Hello!""").

4.5. FILES

It is possible to write one adventure using many source files, having different parts in different files, and thus giving an opportunity for some rudimentary kind of modularisation. The method for this is the $include construct.

```
include = '$INCLUDE' quoted_identifier
```

where the quoted identifier is the name of the file to include. The `$include` may be placed anywhere in a file and the effect will be the same as if the contents of the named file had been inserted at that position in the file. Includes may be nested.

An included file is searched for first in the current directory and then in any of the directories indicated using the **include** switch as described in *Compiler Switches* on page 110, this search is performed in the same order as the **include** switches occurred on the command line.

5. EXECUTION OF AN ADVENTURE

5.1. A TURN OF EVENTS

The player in a way controls the execution of an Alan adventure. Each of his inputs are taken care of and acted upon by the run-time system. The execution of an Alan adventure starts by executing the start section. Then the player is prompted for a command.

The player input is analysed according to the explicit and implicit syntax rules and converted to an execution of verb bodies (global and in possible parameters) or exits (in case of directional commands).

After the players command has been taken care of all rules are evaluated and possibly executed. Then each of the other actors executes one step (if active) and for each actor the rules are evaluated again. Finally any events that are scheduled are fired before prompting the player again.

So to summarise:

```
get and execute a player command
evaluate all rules
for each actor
   execute one step (if active)
   evaluate all rules as above
end
check for and execute any pending events
```

Then the user is prompted for another command and everything is repeated.

A player command may be either a verb or a direction. A verb is executed by checking the syntax of the input, performing any preconditions (checks) and then executing the verb bodies (as described in *Verbs* on page 33). A directional command is executed by finding any exit in that direction, evaluating the checks and the body (if any) of that exit and locating the *hero* at the new location.

5.2. PLAYER INPUT

The syntax defined in the Alan source is the basis for what the player is allowed to input. Commands with these formats form the basic statements available to the player. In addition the following combinations and variations are possible:

- concatenating of statements using AND or THEN, like
  > open the door then enter

- the use of IT to refer to the last object mentioned in the previous command, e.g.
  > take the book and read it

- references to multiple objects using AND, this allows
  > take the blue vase and the pillow

- reference to multiple objects using ALL or EVERYTHING
  > drop all

- excluding objects using BUT or EXCEPT, like:
  > wear everything except the bowler hat

- the use of THEM to refer to the multiple objects referenced in the last command, e.g.
  > remove the hat and the scarf then drop them

The reference to multiple objects (or actors) in a position is, of course, only allowed if the adventure author has allowed it by using a multiple indicator in the syntax definition (see *Syntax Definitions* on page 30). The variations above are built in and handled automatically by the run-time system.

The interpreter also automatically restricts parameter references to objects and actors at the current location. I.e. the player can only refer to objects and actors that are present in his input. The one single exception is if the syntax for the command uses the omnipotent '!' indicator, see *Syntax Definitions* on page 30 for details. For hints on other ways to allow references to objects and actors that are not at the current location, refer to *Distant & Imaginary Objects* on page 73.

The use of ALL results in the execution of the appropriate verb for all objects at the current location, *except* the ones that does not pass all checks for the verb (see *Verbs* on page 33 for further details on this).

Another restriction placed on the player input by the interpreter is that the words the player is allowed to use can only contain alphabetic characters. This must be kept in mind when naming verbs that use the default syntax (an explicit SYNTAX statement can always specify other player words to trigger the verb).

5.3. RUN-TIME CONTEXTS

When the player enters a command the Alan run-time system evaluates the various constructs from the adventure description (source) as described above. Depending on the player's command evaluation of different parts of the adventure may be triggered. These parts all have different conditions under which they are evaluated and also have different contexts. Four different execution contexts can be identified:

- execution of a verb, during the execution of a verb (the syntax and verb checks and the verb bodies), which is the result of the player entering a command that was not a directional command, parameters are defined and may be referenced in the statements and expressions. Also the ACTOR is set to the hero and LOCATION to the location where the hero is (HERE refers to the location of the hero).

- execution of descriptions, these are started as the response to a directional command, a LOOK or DESCRIBE statement, or a LOCATE statement operating on the hero. During this no parameters are defined, ACTOR is set as above, and LOCATION of course to the current (or new) location. The description clauses for objects and locations as well as the DOES part of locations are evaluated in this context. DOES-parts are executed for all actors entering a location with ACTOR set to the current actor.

- execution of actors and rules, each actor performs his step and after each actor all rules are executed. In these contexts no parameters are defined but ACTOR is set to the actor that is executing or was executing immediately preceding the rules. So you could say that rules are run for each actor, and LOCATION is set to that of the executing actor (HERE refers to where the executing actor is).

- execution of events, no parameters and no actor is defined. The location is set to where the event was scheduled to be executed (see also *EVENT STATEMENTS* on page 50).

So the execution of various parts of the adventure source can also be said to have a number of different focuses, meaning where the action is considered to take place:

- the hero - the actions of the player are always focused on the hero and the actions performed are always related to where the hero is

- an actor - steps executed by an actor are always focused where the actor is

- an event - code executed in events are focused where the event was specified to take place (see *EVENT STATEMENTS* on page 50).

- a rule - rules are executed once after each actor (including the hero) with the focus set to where that actor is

5.4. MOVING ACTORS

The main way to move actors is the exits (see *Locations* on page 37). They, of course, only apply to the hero, but are executed if the player inputs a directional command, i.e. a word defined as the name for an exit in any location. First the current location is investigated for an exit in the indicated direction, if there is none an error message is output. Otherwise that exit is examined for CHECKs which are run according to normal rules (see *Verbs* on page 33). If no CHECK was present or if the check passed the statements in the body (the DOES-part) is executed. The hero is then located at the location indicated in the exit header, which will result in the description of the location (by executing the DESCRIPTION-clause of the location) and any objects or actors present (by executing their DESCRIPTIONs).

When any actor (including the hero) is located at a location, the DOES-clause of that location is executed as if the actor had moved into that LOCATION. The actor that was moved will be the ACTOR even though the movement was not

caused by himself (but the result of an event, for example). So this is also the last step in the sequence of events caused by locating the hero somewhere.

6. HINTS AND TIPS

This chapter will give you some ideas about how the various features of Alan may be used to implement common features in an Adventure game. These are only suggestions and you are, of course, welcome to invent your own, but these are probably some ideas that can get you started.

6.1. USE OF ATTRIBUTES

Attributes are primarily used for holding status information about the object, actor or location to which it belongs. This allows, for example, the water bottle to contain three levels of water.

```
OBJECT bottle
  HAS level 3.
  VERB drink
    DOES
      IF level OF bottle > 0 THEN
        DECREASE level OF bottle.
      ELSE
        "There is no more water in the bottle."
      END IF.
  END VERB drink.
END OBJECT bottle.
```

Another example is the broken mirror.

```
OBJECT mirror
  IS NOT broken.
  VERB break
    DOES
      MAKE mirror broken.
  END VERB break.
END OBJECT mirror.
```

The appropriate verbs defined in the objects may then modify the attributes and thus update the status information.

But attributes defined for all objects also allow a kind of classification of the objects (or locations or actors as appropriate). If the following declaration is made

```
OBJECT ATTRIBUTES
  NOT takeable.
```

then all objects receive the attribute “takeable” and if the attribute is not specifically redeclared for an object it will not be takeable. Note however that the semantic meaning of “takeable” must be implemented e.g. in the verb “take”:

```
VERB take
  CHECK OBJECT IS takeable
    ELSE "You can’t take the $o."
  DOES
    LOCATE OBJECT IN inventory.
END VERB take.
```

In the same way restrictions concerning what is possible to eat, drink, open etc. may be implemented. This use of attributes to classify objects is “action- oriented”, i.e. they imply that a particular action (verb) is applicable to the object.

An alternate approach is to classify objects after their characteristics. Consider:

```
VERB take
   CHECK OBJECT IS NOT heavy
      ELSE "That is much too heavy."
   AND OBJECT IS NOT animal
      ELSE "The $o moves quickly away, just far enough
         for you not to reach it."
   DOES
      LOCATE OBJECT IN inventory.
END VERB take.
```

This approach is more “class-oriented” as the objects are classified and a verb is possible to apply to certain classes of objects and not to others. This approach is more elegant but is harder to keep track of as you introduce new objects (which class or even classes does a new object belong to?).

6.2. DESCRIPTIONS

The attributes are also used when presenting information about status to the player. The attributes are tested in `IF`-statements to modify the `DESCRIPTIONS` and possibly even the short description in the `MENTIONED` sections. For example:

```
OBJECT mirror
   IS NOT broken.
   DESCRIPTION
      "On the wall there is a beautiful mirror with an
      elaborate golden frame."
      IF mirror IS broken THEN
         "Some moron has broken the glass in it."
      END IF.
   VERB break
      DOES
         MAKE mirror broken.
   END VERB break.
END OBJECT mirror.
```

To use this feature with the short descriptions makes the adventure feel a bit more consistent.

```
OBJECT bottle
   HAS level 3.
   ARTICLE ""
   MENTIONED
      IF level OF bottle > 0 THEN
         "a bottle of water"
      ELSE
         "an empty bottle"
      END IF.
END OBJECT bottle.

> inventory
   You are carrying
      an empty bottle
```

6.3. COMMON VERBS

As your library of adventures grow you will find that some verbs are always needed, and always function the same way. Examples are “take”, “drop”, “invent”, “look”, “quit” and so on. It is advised to use an include file (see section 4.5 on page 59) containing these verbs as well as their syntax definitions and any synonyms. Attributes needed for these particular verbs could also be placed in a default attribute declaration in this file.

All your adventures may then include this file, making these features immediately accessible when you start a new adventure. All that this takes is some thought as to what names to use for the attributes as discussed in *Use of Attributes* on page 63.

6.4. DOORS

Another common feature is the closed door. Here’s how to implement it.

```
OBJECT treasury_door AT hallway
  VERB open
    DOES
      MAKE treasury_door open.
      MAKE hallway_door open.
  END VERB open.
END OBJECT treasury_door.

LOCATION hallway
  EXIT east TO treasury
    CHECK treasury_door IS open
      ELSE "The door to the treasury is closed."
  END EXIT.
END LOCATION hallway.

OBJECT hallway_door AT treasury
  VERB open
    DOES
      MAKE treasury_door open.
      MAKE hallway_door open.
  END VERB open.
END OBJECT treasury_door.

LOCATION treasury
  EXIT west TO hallway
    CHECK hallway_door IS open
      ELSE "The door to the hallway is closed."
  END EXIT.
END LOCATION treasury.
```

Note that we need two doors, one at each location, but they are synchronised by always making them both open or closed at the same time. The check in the EXITs makes sure that the hero can not pass through a closed door.

6.5. CONTAINERS AND THEIR CONTENTS

Containers are either pure containers or objects or actors with the container property. A pure container is always considered to be where the hero is. This means that the inventory (what the hero is carrying), his clothes etc. are suitable to be pure containers.

For a container to be directly manipulable by the player it must be an object (or actor). This means that it always is located at a particular location in the same way as other objects. A container (in the following the term container is used to refer to objects with the container property) is always open, i.e the objects it contain are always accessible.

To be able to "close" a container, i.e. to make it impossible for the hero to take or see things inside a container, the following technique may be used (other techniques may be possible and even better!). Create an extra object with the container property, this container is used as a temporary storage for objects in the first container (the one the player is seeing). Place this at a location not accessible to the player (the limbo location Nowhere always comes in handy!).

The verbs "open" and "close" then get the following definition within the object:

```
OBJECT chest AT treasury
   CONTAINER
   IS NOT open.

   VERB close
      DOES
         MAKE chest NOT open.
         EMPTY chest IN chest_contents.
   END VERB close.

   VERB open
      DOES
         MAKE chest open.
         EMPTY chest_contents IN chest.
         "Opening the chest reveals its contents."
         LIST chest.
   END VERB open.
END OBJECT chest.
```

The trick used here is to make all the things in the container disappear when it is closed. To do this, the extra container `chest_contents` is used as a temporary holding place for the things inside the chest. Note that we need to make `chest_contents` an actual object since pure containers are always accessible (they are where the hero is!). When the chest is opened again we simply empty the contents of the `chest_contents` container into the chest, and *Voila*!

6.6. ACTORS

Actors are a vital component to make a story dynamic. They move around and act according to their scripts. To make the player aware of the other actor's actions they need to be described. This must be done so that the player always get the correct perspective on the actions of the actors.

A way to ensure this is to rely on the fact that output statements are not shown unless the hero is at the location where the output is taking place. This means that for every actor action, especially movement, you need to first describe the actions, then let the actor perform them and, finally, possibly describe the effects.

An example is the movement of an actor from one location to another. In this case the step could look something like

```
"Charlie Chaplin goes down the stairs to the hallway."
LOCATE charlie_chaplin AT hallway.
"Charlie Chaplin comes down the stairs and
   leaves the house through the front door."
LOCATE charlie_chaplin AT outside_house.
"Charlie Chaplin comes out from the nearest house."
```

An actor is described, for example, when a location is entered or as the result of a LOOK, in the same way as objects are. This means that a good idea is to include the description of an actor's activities in the description of him. One way to do this would be to use attributes to keep track of the actors state and test these in the description clause.

```
ACTOR george NAME George Formby
   IS
      NOT cleaning_windows.
      NOT tuning.
   DESCRIPTION
      IF george IS cleaning_windows THEN
         "George Formby is here cleaning windows."
      ELSIF george IS tuning THEN
         "George Formby is tuning his ukelele."
      ELSE
         "George Formby is here."
      END IF.
...
```

Although quite feasible, this is a bit tedious. As, at least a part of, the state is indicated by the script the actor is executing, this could be used to avoid the potentially large IF-chain. The optional descriptions tied to each script will be executed instead of the main description when the actor is following that script. So this would allow

```
ACTOR george NAME George Formby
   DESCRIPTION
      "George Formby is here."
   SCRIPT cleaning.
      DESCRIPTION
         "George Formby is here cleaning windows."
      STEP
         ...
   SCRIPT tuning.
      DESCRIPTION
         "George Formby is tuning his ukelele."
      STEP
         ...
...
```

This makes it easier to keep track of what an actor is doing. Another hint here is to describe the change in an actor's activities at the same time as executing the USE statement, like

```
EVENT start_cleaning
   USE SCRIPT cleaning FOR george.
   "All of a sudden, George starts to clean the windows."
END EVENT.
```

This makes the descriptions of changes to be shown when it takes place and the description of the actor is always consistent. You can, of course, still have attributes describing the actor's state to customize the description of the actor on an even more detailed level, but it generally suffices to describe an actor in terms of what script he is executing.

6.7. DISTANT EVENTS

A slight problem with the feature that output is not visible unless the hero is present, is that a description of an event might not always be presented to the player.

```
EVENT explosion
   "A gigantic explosion fills the whole room with smoke
      and dust. Your ears ring from the loud noise. After
      a while cracks start to show in the ceiling,
      widening fast, stones and debris falling in
      increasing size and numbers until finally the
      complete roof falls down from the heavy explosion."
   MAKE LOCATION destroyed.
END EVENT.
```

If the hero isn't at the location where the event is executed, he will never know anything about what has happened. The solution is to create an event that goes of where the hero is.

```
EVENT distant_explosion
   "Somewhere far away you can hear an explosion."
END EVENT.
...
IF HERO NEARBY THEN
   SCHEDULE distant_explosion AT HERO AFTER 0.
...
```

6.8. VEHICLES

The current version of Alan does not support actors being inside containers or inside other actors, which could be a straight forward way to implement vehicles. However, as the reader/player does not need to know how the output is generated we can use a location and a row of events to substitute for the vehicle. Try the following complete example:

```
SYNONYMS
   car = ferrari.

SYNTAX
   drive = drive.
   park = park.

SYNTAX l = l.

VERB l
   DOES
      LOOK.
END VERB.

LOCATION garage
END LOCATION.

LOCATION parking_lot NAME 'Large Parking Lot'
END LOCATION.

OBJECT car NAME little red sporty ferrari
   AT garage
   IS
      NOT running.
   HAS
```

```
    position 0.

  VERB enter
    DOES
      LOCATE hero AT inside_car.
  END VERB enter.

END OBJECT car.

LOCATION inside_car NAME 'Inside the Ferrari'
  DESCRIPTION
    "This sporty little red vehicle can really take you
    places..."

  EXIT out TO inside_car -- just a dummy, since we are
                         -- going to change it below
    CHECK car IS NOT running
    ELSE "I think you should stop the car before
getting
      out..."
    DOES
      IF position OF car = 0 THEN
        LOCATE hero AT garage.
      ELSIF position OF car = 1 THEN
        LOCATE hero AT parking_lot.
        --- Etc.
      END IF.
  END EXIT.

  VERB drive
    CHECK car IS NOT running
    ELSE "You are already driving it!"
    DOES
      "You start the car and drive off."
      MAKE car running.
      SCHEDULE drive1 AFTER 1.
  END VERB drive.

  VERB park
    CHECK car IS running
    ELSE "You are not driving it!"
    DOES
      "You slow to a stop and turn the engine off."
      MAKE car NOT running.
      CANCEL drive1. CANCEL drive2. --- Etc.
  END VERB park.
END LOCATION inside_car.

EVENT drive1
  "You drive out from your garage and approach a large
parking lot."
  SET position OF car TO 1.
  LOCATE car AT parking_lot.
  SCHEDULE drive2 after 1.
END EVENT drive1.

EVENT drive2
  "You drive out from the parking lot and approach your
  own garage."
  SET position OF car TO 0.
  LOCATE car AT garage.
  SCHEDULE drive1 after 1.
END EVENT drive2.

START AT garage.
```

The main idea is that the player/reader is inside the car, and the events are executed at this location thus emulating movement. It is possible to exchange the events for

script steps and the car object for an actor. However as the car object is not where the hero is ('inside_car') the output from the scripts will not be shown. There are (at least) two different ways to deal with this (one involving attributes, the other involving an extra object), but the solutions are left as an exercise to the reader!

Sincere thanks go to Walt (sandsquish@aol.com) for inspiring communication that brought this example to life.

6.9. QUESTIONS AND ANSWERS

Sometimes it may be necessary to ask the player for an answer to some question. One example is if you want to confirm an action. The following example delineates one simple way to do this, which could be adopted for various circumstances.

```
ACTOR hero IS NOT quitting.
END ACTOR hero.

SYNTAX
   'quit' = 'quit'.
   yes = yes.

SYNONYMS
   y = yes.
   q = 'quit'.

VERB 'quit' DOES "Do you really want to give up?
                  Type 'yes' to quit, or to carry on
                  type your next command."
   MAKE hero quitting.
   SCHEDULE unquit AFTER 1.
END VERB 'quit'.

VERB yes CHECK hero IS quitting
         ELSE "That does not seem to answer any
question."
   DOES QUIT.
END VERB yes.

EVENT unquit MAKE hero NOT quitting.
END EVENT unquit.
```

Thanks to Tony O'Hagan (aoh@maths.nott.ac.uk) for this excellent idea.

6.10. FLOATING OBJECTS

Floating objects is a term used for objects that are available everywhere or at least at many places. Usually they are available wherever the hero is.

Examples of floating objects are the air, the ground and such semi-abstract objects. But sometimes you also need to make actual objects be floating objects such as parts of the heroes body.

To create floating objects you can use a particular feature of containers, namely the fact that they are always located where the hero is.

Note: This only applies to containers that are pure containers, for objects and actors that have the container property this does not apply of course. See *Containers* on page 41 for a discussion.

So to have the hero's body parts and the air and the sky to be available wherever the hero goes you can use:

```
CONTAINER body_parts
END CONTAINER body_parts.

CONTAINER outdoor_things
END CONTAINER outdoor_things.

OBJECT right_arm NAME right arm IN bocy_parts ...
OBJECT head NAME head IN body_parts ...
OBJECT sky IN outdoor_things ...
OBJECT air IN outdoor_things ...
```

Of course you would not want the outdoor things to be available when you are indoors, but this can be fixed in a way similar to the container contents trick shown in *Containers and Their Contents* on page 66. Simply create a container object and place it where the hero can never be:

```
OBJECT outdoor_things_storage AT limbo
  CONTAINER
END OBJECT outdoor_things_storage.

WHEN location IS outdoors =>
  EMPTY outdoor_things_storage IN outdoor_things.
WHEN location IS NOT outdoors =>
  EMPTY outdoor_things IN outdoor_things_storage.
```

And Voila', every time the hero arrives at an outdoor location he will find the air and the sky. And every time he enters a location that has the attribute `outdoors` set to false he will not find them available.

Well, perhaps he would like to have the air available indoors too, but that is left as an exercise for the reader...

6.11. DARKNESS AND LIGHT SOURCES

A very common puzzle in old time adventures (so much so that it has possibly been exploited beyond its potential) is the problem of dark locations and finding a source of light.

This puzzle can be implemented in Alan in a rather general way by using a default object attribute, a default location attribute and a few additions to the descriptions of the dark locations and the `look` verb.

```
Object Attributes
     lightsource 0.
  Location Attributes
    lit.
```

This will give all objects the value of 0 of the attribute `lightsource`. Any object that provide light should set this to something larger than zero. The attribute

might of course change value dynamically, e.g. when the lamp is lit and extinguished. We can thus sum all the values of the attribute `lightsource` at a location and if the sum is above zero there is some light provided. So the look verb could be reworked to:

```
Verb 'look'
     Does
       If Sum Of lightsource Here = 0
           And Location Is Not lit Then
         "You cannot see anything without any light."
       Else
         Look.
       End If.
   End Verb 'look'.
```

Of course we must also modify the dark locations:

```
Location indoors
   Is
      Not lit.
   Description
      If Sum Of lightsource Here > 0 Then
        "This is usually a very dark room. But in this
light
        you can see..."
      Else
        "You can not see anything in the dark."
      End If.
   Exit out To outdoors.
End Location.

Location outdoors
   Description
     "Out here in the sun you can see everything."
   Exit 'in' To indoors.
End Location.
```

So for every location which should be dark we must add the above test to the description clause.

There is however still a small problem with this solution. Objects available at the location are visible (described) as you enter the location. This must be taken care of, e.g. by moving all objects present to a limbo location (analogous to the container contents trick described in section 6.5 on page 66) in the dark part of the `IF` statement, and back in the `ELSE` clause.

Thanks goes to Thomas Ally (Thomas_Ally@freenet.richland.oh.us) for prompting this solution.

6.12. DISTANT & IMAGINARY OBJECTS

A feature introduced in v2.7 made the following section almost obsolete. The new feature is the ability to refer to distant objects and actors (see *Syntax Definitions* on page 30 for a discussion on the omnipotent '!' indicator). I.e. the previous restriction that the player could only refer to objects and actors at the same location was removed. However there are instances where it may still be required to separate the handling of an object when it is present and when it is not, therefore this section

gives a few examples of what can be done using some trickery with the mechanisms of Alan.

Sometimes you need to make it possible for the player to refer to things either far away, that are not really objects or that may be at many places at once. Examples of these are a distant mountain that may be examined through a set of binoculars, the melody in “whistle the melody”, and water or walls.

For objects that should be visible from a distance the easiest method is to introduce a ‘shadow object’. This is a second object acting on behalf of, or representing, the distant object at the locations where it should be possible to refer to it. For example:

```
LOCATION hills
   :
END LOCATION hills.

OBJECT mountain AT hills
   :
END OBJECT mountain.

LOCATION scenic_vista NAME Scenic Vista
END LOCATION scenic_vista.

OBJECT shadow_mountain
   NAME distant mountain AT scenic_vista
   DESCRIPTION
      "Far in the distance you can see the Pebbly
      Mountain raising towards the sky."
END OBJECT shadow_moutain.
```

This would allow for example at scenic_vista:

```
Scenic Vista.
   Far in the distance you can see the Pebbly Mountain
raising
   towards the sky.

> look at mountain through the binoculars
...
```

which would otherwise be impossible. If the mountain should be visible and manipulable from a number of locations, you might implement one shadow object for each location but this is a bit tedious if they are identical. One trick here is to use something like the following rule:

```
WHEN hero AT scenic_vista OR hero AT hill_road =>
   LOCATE shadow_mountain AT hero.
```

This will ensure that whenever the hero moves to any of the places from where the mountain is visible, the `shadow_mountain` is sure to follow. However, as the rules are executed *after* the hero has moved, a better strategy might be to make the `shadow_mountain` ‘silent’, i.e. to have no description. Instead the description of it should be embedded in the description of the adjacent locations. Yet another possibility would be to move the pseudo-object around using statements in the exits, like

```
LOCATION scenic_vista NAME Scenic Vista
   EXIT east TO hills
```

```
      DOES
        LOCATE shadow_mountain AT hills.
    END EXIT east.
END LOCATION scenic_vista.
```

Objects that are always present, such as the air or the parts of the hero's body, may be treated like normal objects. I.e. they are defined as the objects they represent. They are then placed in a container that is not an object, which makes the objects always accessible, since containers (that are not objects) are considered to be where the hero is (cf. the inventory). This is also a simple way to create other compartments on the hero, such as a belt.

```
CONTAINER belt
   LIMIT count 2
      ELSE "You can't fit more in your belt."
END CONTAINER belt.

VERB invent
   DOES
      LIST inventory.
      LIST belt.
END VERB invent.

CONTAINER pseudo
END CONTAINER pseudo.

OBJECT air IN pseudo
   VERB breathe
      :
   END VERB breathe.
END OBJECT air.
```

6.13. STRUCTURE

A good thing to do when designing an interactive fiction story is to separate the geography from the story. In Alan you can use the include facility to structure your Alan source. One approach could be to place the description of each location in a separate file together with any objects that could be considered part of the scenery or at least is not only a tool in a puzzle. These files can then be included in a 'map' file, which in turn is included by the top-level file.

The story line can be divided into files too, one for each 'scene'. A scene being comments describing the important things that are suppose to happen, any prerequisites and objects, events, rules etc. which are specific for this part of the story.

This strategy will both give you a better structure of your adventure as well as help you design a better story, much like the storyboarding technique used in making movies or plays.

6.14. DEBUGGING

To simplify the development of adventures written in the Alan language, the interpreter Arun incorporates some features for debugging. There are a few debugging switches available when starting the interpreter:

```
-l   Create a log of the player commands
-t   Enable trace mode
-s   Enable single instruction trace
-d   Enable debug mode
```

COMMAND LOG

For various purposes, such as debugging, an actual log of the player commands can be handy. Such a log is created if the option **-l** is given to the interpreter when starting a game. The log file is created in the directory, which was current when the interpreter was started, the name of the log file will be the same as the game with the extension **.log**.

INTERPRETER AND INSTRUCTION TRACE

Trace mode can also act as an aid in debugging. It will print information about each invocation of the instruction interpreter, making it easier to see which parts of the code are being executed.

Single instruction trace will, in addition to the trace mode information, also trace every single Acode instruction.

DEBUG MODE

Finally, debug mode will execute the start up sequence and then prompt for a debug command with

```
ABUG>
```

Note: None of the above switches are effective unless the adventure was compiled with the debug option set (see *Options* on page 26).

Abug may also be entered by typing the single command

```
> debug
```

during the execution of an Adventure that was compiled with the debug option.

A question mark or an 'h' will give a brief listing of the commands available in Abug:

```
a    Display a list of all actors.
c    Display a list of all containers.
e    Display a list of all events and their status.
g    Go on. I.e. proceed by executing the next turn.
Abug
     will stop and prompt for a new command again before
     the player is next in turn.
l    Display a list of all locations.
o    Display a list of all objects.
q    Quit the adventure (and Abug).
```

s Toggle single instruction trace.

t Toggle trace mode (off and on).

x Exit Abug, i.e. proceed without stopping.

The commands A, C, L and O may optionally be followed by a number. Abug will then display detailed information about the entity requested, such as values of attributes, its present location etc.

Currently there is no way to modify anything using Abug.

The following is a short excerpt from a debugging session (user input in bold face):

```
<Arun, Adventure Interpreter version 2.6 alpha>
<Version of 'saviour' is 2.6(0)a>
Welcome to the game of SAVIOUR!
[introductory text deleted for brevity]
ABUG> s
Step on.
ABUG> t
Trace on.
ABUG> g
> n
<EXIT 1 (n) from 22 (Outside The Tall Building),Executing:>

++++++++++++++++++++++++++++++++++++++++++++++++++
 dd9: PUSH          1
 dda: SCORE         1           (5)
 ddb: RETURN
--------------------------------------------------

<EXIT 1 (n) from 22 (Outside The Tall Building), Moving:>

++++++++++++++++++++++++++++++++++++++++++++++++++
 de4: PUSH          4
 de5: PUSH       6229
 de6: PRINT      6229,     4    "Hall"
 de7: RETURN
--------------------------------------------------
.

++++++++++++++++++++++++++++++++++++++++++++++++++
 de8: PUSH        158
 de9: PUSH       6235
 dea: PRINT      6235,   158    "Inside the entrance is a hallway full of
dust and pieces of the ceiling have fallen to the floor. At the west end
is a staircase, and to the south is the exit."
 deb: PUSH          1
 dec: DESCRIBE      1
++++++++++++++++++++++++++++++++++++++++++++++++++
 620: PUSH         30
 621: PUSH       1428
 622: PRINT      1428,    30    " To the east is a folding door."
 623: PUSH          6
 624: PUSH          1
 625: ATTRIBUTE     1,     6    (1)
 626: IF         TRUE
 627: PUSH         13
 628: PUSH       1446
 629: PRINT      1446,    13    " It is closed."
 62a: ELSE
 62f: RETURN
--------------------------------------------------

 ded: RETURN
--------------------------------------------------


ABUG> a
ACTORS:
     17: Hero

ABUG> a 17
ACTOR 17 : Hero
    Location = 23 Hall
    Script = 0
    Step = 0
    Attributes =
```

```
ABUG> o
OBJECTS:
       1: door
       2: rats
       3: spool of computer tape
       4: old book
       5: 3 metre long ladder
       6: rather heavy computer terminal
       7: small coin
       8: birds nest
       9: set of rusty keys
      10: clock
      11: drawer
      12: desk
      13: dirty manual
      14: computer
      15: vending machine
      16: old mouldy candy bar

ABUG> o 6
OBJECT 6 : rather heavy computer terminal
    Location = 30 Terminal Room
    Attributes =
      1: 1 (takeable)
      2: 1 (readable)
      3: 0 (openable)
      4: 0 (startable)
      5: 1 (examinable)
      6: 0 (connected)
      7: 0 (showing_msg1)
      8: 0 (showing_msg2)

ABUG> q
```

Lines of '+' characters indicates the start of interpretation, thus they can be present inside other single step traces (like the `DESCRIBE` in the example above). Likewise lines of '-' indicates the return from one such level of interpretation.

7. ADVENTURE CONSTRUCTION

This chapter will give a few clues on how to be a successful adventure author, because creating a *good* adventure is more like writing a book than writing a program (although Alan can be viewed as a kind of programming language).

7.1. GETTING AN IDEA

As with a book, the success or failure depends on how intriguing the story is, how hooked you can get the reader (in our case the player). So, the first step *must* be to get a good idea. This may be hard or easy but with time you, like a good author, learn to pick up ideas when you get them in ordinary every-day life, and store them for later use.

A seemingly simple idea might also be developed into a good adventure if it is placed in the correct setting and supplied with additional features, tricks and problems.

When you have a good idea, try to refrain from typing it in directly in a text editor and compile it with Alan. Instead, write the story down as if it were the story line for a book or a movie. Where appropriate, insert hints on various diversions and alternate paths that come to mind, but try to stay mainly with the main story from beginning to the preferred end. Then, let a close friend read it.

7.2. ELABORATING THE STORY

After having rewritten the story line once or twice, start creating the scenery. If your setting is small, you could draw a map of the locations needed, but a better way is probably to make a list of major locations first (those essential to the story). For each location note what important properties the location must have and which objects are necessary (just as notes, *don't* create the Alan declarations yet!). For each object, make a small note on what the object is needed for (by the player!).

This may also be done using a scene-by-scene approach. By this we mean that the story is segmented into scenes (and maybe also acts) like in a play. For each act and scene you do the above. This makes it easier to get an overview over a larger adventure.

I also suggest that you also create a story on a level above the actual game, at least in your own mind. This story should explain why the game-world exists and thus give a consistency to the text that you will present to the player. Nobody likes an adventure without a cause. This story or world of ideas need not be revealed to the player.

This also applies to the narrator, i.e. the imaginary person or creature that carries out the conversation with the player. Create an image of him or it and stick to it. Receiving comments about your (limited) progress in the game might be funny as long as they are not out of character.

7.3. IMPLEMENTING IT

At last it is time to sit down at the terminal. Divide the adventure text into files containing global verbs, the map (possibly divided further according to the scenes), the actors (perhaps one file for each actor) and a main file including the other files. This makes it easy to handle the adventure and you may also ask your friend to participate in the development by giving him a few files to work on.

First, just declare the locations and connect them with exits. Do not work on the "purple prose" descriptions yet. The Alan system supplies good defaults for descriptions and so on so use these while developing the structure of the adventure. Do not bother even with the details of making it impossible to pick up the elephant, etc.

Play the adventure continuously during the development, but do not try the things you plan to make impossible later. Just go through it according to the line you planned the story to follow. A hint here is to use a separate file for the start section. In this file you can easily set up the situation you wish to test while not having to tire yourself by playing the adventure from the start every time.

7.4. POLISHING THE ADVENTURE

So, now you have a working adventure, a bit bare bones, but still the story plays the way you planned. Now it is time to insert all the nice descriptions, the limitations and perhaps the extra things to divert and hinder the hero. Just be careful not to fall into the locked-door-syndrome. Too many adventures have been tedious to play because you need to find-key/get-key/unlock-door- with-key/open-door (anyway, why do people go around locking doors and throwing away the keys). Think big.

Start by fixing the verbs so that they prohibit the impossible. Introduce as many synonyms as you can think of, this makes the adventure so much more playable.

Create the location descriptions. Remember to use the same style in all your descriptions; breaking out of style does not look good in the eyes of the adventurous. The descriptions must give the player the correct image, the brain is still the best graphic interface available, but they should also plant ideas in the player on how to solve the problems you place before him.

Another thing to aim for is the feeling that a player gets when he somehow finds information explaining things he has encountered earlier in the game. Here, as always, it is good advice to ask a friend to read the texts and convey his or her impressions (remember you know it all because you wrote it!).

Lastly fill in the adjectives for the objects, their descriptions and short descriptions (if needed).

7.5. BETA TESTING

Now you might think that you can start distributing your game. But, wait! As any complex computer program it can have various kinds of bugs. Bugs in a work of interactive fiction range from misspellings and grammar errors in your descriptions, logic errors in your implementation of puzzles or events or omissions in the descriptions of surroundings that make the player miss or misunderstand how to act, to inconsistencies in the settings or story, plots that don't work.

So how do you find these? Your only help are the beta testers. They are the people that you now should consider first a first trial beta release of your game. They should be people who you trust do give their honest opinion and also really play it through to find any problems.

The beta testers will probably give you a long list of issues that you have to address before the next release. Some of the issues are simple; others may affect the basis of your story. You should seriously consider (and if possible discuss) such suggestions.

One aid in finding any problems in the playability of the game is to use the log file facility of the interpreter (see *Command Log* on page 76) to produce a list of the commands a player have used. This can greatly aid in spotting troublesome areas, such as where the player is stuck and reverts to "guess-the- verb". This log can be fed into the interpreter and will give you the exact game played.

After having collected all this information, considered which ones to act upon, and implemented these, you should probably do this again (sigh!).

Now, at last, your adventure game is ready to meet its audience.

A RUN-TIME MESSAGES

This appendix describes the errors that may occur during the running of the adventure, i.e. during interpretation of the generated Acode. There are two classes of errors, player input response messages and system errors.

Input response errors are not fatal but abort the execution of the current player command and discard the rest of the user input, which is a normal part of the interaction between the player and the Alan run-time system. System errors *are* fatal and abort the execution of the adventure.

A.1 Input Response Messages

Various messages are printed for the benefit of the player. Most messages probably come from the adventure itself, i.e. they where provided by the adventure author. But there is a set of messages that can be given directly by the Arun interpreter. They are presented below using the Alan STRING-format, i.e. containing the special character combinations described in *Output Statements* on page 43. These standard messages exist for all languages and the default value of the texts are selected depending upon the setting of the language option.

The contents of any message may be modified using the MESSAGE statement (see section 3.5 on page 30). The identifier on the first line of a message explanation is the identifier that should be used in the MESSAGE statement to change the contents of that message. The second line is the default English message text, and finally a short explanation is given.

All messages are available in all supported languages but below the English message texts are shown.

Note: Although the default values of the messages are static strings, it is possible to create more dynamic messages as the MESSAGE statement allows any statements not only strings, see *Messages* on page 30 for details.

```
WHAT,
        "I don't understand.",
```

The input did not follow any syntax the Arun parser knows about.

```
WHATALL,
        "I don't know what you mean by 'all'.",
```

The player input ALL, but the Arun parser could not find any objects or actors that it could refer to.

```
WHATIT,
          "I don't know what you mean by 'it'.",
```

IT may only be used when the previous command contained a reference to one object or actor.

```
WHATTHEM,
          "I don't know what you mean by 'them'.",
```

THEM refers to the set of objects or actors mentioned in the previous command. If there were no multiple parameters in the previous player command, Arun will issue this message.

```
MULTIPLE,
        "You can't refer to multiple objects with '$v'.",
```

The syntax for the indicated verb did not allow multiple parameters.

```
WANT,
        "I can't guess what you want to $v.",
```

The verb required a parameter.

```
NOUN,
        "You must supply a noun.",
```

The player started to specify an object or actor but only supplied the adjectives.

```
AFTERBUT,
        "You must give an object after 'but'.",
```

In a command containing ALL BUT, the player must also give the object or objects excluded.

```
BUTALL,
        "You can only use 'but' after 'all'.",
```

The words BUT and EXCEPT may only be used after ALL.

```
NOTMUCH,
        "That doesn't leave much to $v!",
```

The player used an ALL BUT construct which explicitly excluded everything matched by the ALL.

```
WHICHONE,
        "I don't know which $1 you mean.",
```

There were multiple objects (or actors) that matched the description given by the player. More adjectives are necessary to distinguish between them.

```
NOSUCH,
        "I can't see any $1 here.",
```

The player referred to an object or actor that was not present.

```
NOWAY,
        "You can't go that way.",
```

A directional word was used but there is no exit in that direction.

```
CANT0,
        "You can't do that.",
```

Somehow Arun found no verb body to execute. This may be a situation overlooked by the author or the player may be trying to do something that is not possible.

```
CANT,
        "You can't $v the $1.",
```

This is a variation of the above message.

```
SEEOBJECT1,
         "There is",

SEEOBJECTCOMMA,
         "$$, ",

SEEOBJECTAND,
         "and ",

SEEOBJECTEND,
         "here.",
```

These four messages are used to construct the default text for describing objects present at the current location (unless they have a description clause, in which case they are used instead). The message parts are used as in "*There is <article> <object>, <article> <object> and <article> <object> here.*" The underlined parts are the ones in the messages and <article> and <object> are inserted as appropriate.

```
SEEACTOR,
         "is here.",
```

The default message for presenting actors present, unless they present themselves (have a description).

```
CONTAINS1,
         "The",

CONTAINS,
         "contains",

CONTAINSCOMMA,
         ", ",

CONTAINSAND,
         "and ",

CONTAINSEND,
         "$$.",
```

The four messages above are used to construct the default contents listing of a container in much the same way as for the object listing above. The messages are used according to the pattern "*The <container> contains <article> <object>, <article> <object> and <article> <object>.*"

```
EMPTY1,
          "The",

ISEMPTY,
          "is empty.",
```

The default messages for empty containers.

```
HAVESCORED,
          "You have scored",

SCOREOUTOF,
          "points out of",
```

Two parts of the default scoring message.

```
UNKNOWNWORD,
          "I don't know that word.",

MORE,
          "<More>",
```

The classic message when the screen is full. The player should press RETURN to proceed.

```
AGAIN,
          "(again)",
```

This message is presented immediately after the location name if the location has been visited before to give the player the information that he has visited this location before (a good thing in some adventures). If you wish to disable this set this message to an empty string.

```
SAVEWHERE,
          "Enter file name to save in",
```

When executing a SAVE the player can enter the name of the file to save in. The name used in the previous SAVE is used as a default.

```
SAVEOVERWRITE,
          "That file already exists, overwrite (y) ? ",
```

If the save file already existed the player must confirm the overwrite.

```
SAVEFAILED,
          "Sorry, save failed.",
```

When executing a SAVE, the file system indicated some error, usually a write protected directory or full disks.

```
SAVEVERSION,
        "Sorry, the save file was created by a different
    version.",
```

The save file found was created by a different version of the Alan interpreter.

```
SAVENAME,
        "Sorry, the save file did not contain a save for
    this adventure.",
```

The indicated save file did not contain a save of this adventure.

```
RESTOREFROM,
        "Enter file name to restore from",
```

A RESTORE statement can restore from any named file. The previously used file name is used as the default.

```
SAVEMISSING,
        "Sorry, could not open the save file.",
```

When executing a RESTORE, Arun could not find a save file with the indicated name in the current directory.

```
QUITACTION,
        "Do you want to RESTART, RESTORE or QUIT ?",
```

The QUIT statement requests an action from the player.

Note: The possible answers are currently hard-wired into the interpreter, so changing RESTART, RESTORE or QUIT will probably confuse the player!

```
DEFARTICLE,
        "a",
```

The indefinite article is needed for objects, objects which have none declared (using the ARTICLE clause) will receive the default article. This is mainly available for the ease of constructing adventures in unsupported languages.

A.2 System Errors

System errors are errors caused by internal malfunctions. Mainly these are implementation errors (aka. bugs!), but may (in some manner) also result from user errors. The system error messages also have a purple prose style to fit in with your game, e.g.:

```
As you enter the twilight zone of Adventures, you
stumble and fall to your knees. In front of you, you can
vaguely see the outlines of an Adventure that never was.

SYSTEM ERROR: Can't open adventure code file.
```

AUTHOR ERRORS

The following system errors are in some sense caused by the Adventure author (you).

`Out of memory.`

The adventure was so large that the interpreter could not allocate enough dynamic memory for it. Try to finish other running applications (does not work or is not possible on all systems), get more real memory, or complain to the Alan implementors (see appendix 0, *FUTURE DEVELOPMENTS* on page 117 for how to reach us).

`Incompatible version of ACODE program.`

The interpreter you are using have a different version than the Alan compiler used to compile the adventure. Use a different Arun or recompile the adventure with the matching compiler.

Note: the Arun switch `'-d'` will, beside entering debug mode, also print the version of both the Arun interpreter and the version of the Alan compiler used to compile the adventure.

`Recursive LOOK.`

This message is shown when a LOOK statement is executed as a result of a LOOK, i.e. a recursive LOOK! The LOOK statement should only be used in global verb bodies, *not* in descriptions of LOCATIONs and OBJECTs as there is a definite risk that it will be executed as the effect of a LOOK, either explicit or implicit (by the hero entering that location!).

`Locating something inside itself.`

This means that an attempt to locate an object (that is a container) inside itself has been made. This might happen if the adventure author has neglected to check this in a verb like

```
put_in = 'put' (o) 'in' (c)
```

`Non-existing parameter referenced.`

A parameter that wasn't available was referenced. This is probably due to using a parameter shorthand such as $2 inside a string in a context where the syntax was restricted to only one parameter. This may avoided by using the SAY statement instead of the embedded string parameter references, which would result in compile time checking avoiding the risk of having this happen to the player.

Note: Parameter references embedded in strings are *not* currently checked during compile time.

Note: Embedded string references ($1, $o, etc.) is *not* guaranteed to be forward compatible (i.e. it may be removed in future versions).

PLAYER ERRORS

Errors caused by incorrect arguments or file names.

`Can't open adventure code file.`

`Can't open adventure data file.`

The player attempted to run an adventure for which there were no code or data file available, probably a misspelling.

`Could not read all ACD code.`

`Checksum error in .ACD file (%1 instead of %2).`

These two messages indicate problems in the adventure files. Possibly caused by transfer problems of the **.acd** and **.dat** files which must be made in binary mode.

IMPLEMENTOR ERRORS

Any other text in a system error message is really a SYSTEM ERROR. Scribble down the text and contact the implementors (see appendix 0, *FUTURE DEVELOPMENTS* on page 117). If possible supply the source for your adventure, a trace of the few last player commands (if possible with single step and trace turned on, see *Debugging* on page 75).

B ALAN LANGUAGE GRAMMAR

B.1 Description

The Alan language is defined formally below using a BNF-form. This is a set of rules defining exactly what constructs are legal in an Alan source. The rules are numbered for easy reference.

The BNF form divides the structure of the input source into smaller parts (rules) which in turn are defined by other rules. For example rule 1 says that an ADVENTURE (in this case an Alan program) consists of options, units and a start section. In rule 1 we can see that an adventure is defined to contain an `optional_options` section, some `units`, and then the `start_section`. Each is in turn defined further down in the rules.

The equal sign (=) may thus be read as “consists of” or “is defined as”. The exclamation mark indicates a choice between the two different constructs, for example in rule 6 through 8 one can see that an option may either be a single identifier (`ID`), an identifier followed by another identifier or an identifier followed by an integer. The semicolon indicates the end of the complete definition of the symbol on the left hand side of the equal sign.

B.2 Reserved words

The following is a complete list of all words reserved for special use in the Alan language. Note that the reserved words can still be used as identifiers in a source file provided that the rules described in *Identifiers And Names* on page 57 are followed.

```
ACTOR
AFTER
AND
ARE
AT
ATTRIBUTES
BEFORE
BY
CANCEL
CHECK
CONTAINER
COUNT
DECREASE
DEPEND
DEPENDING
DESCRIBE
DESCRIPTION
DOES
ELSE
ELSIF
EMPTY
END
EVENT
EXIT
FOR
HAS
HEADER
HERE
IF
INCREASE
INTEGER
```

```
IS
ISA
LIMITS
LIST
LOCATE
LOCATION
LOOK
MAKE
MAX
MENTIONED
NAME
NEARBY
NOT
OBJECT
OF
ONLY
OPTIONS
OR
RANDOM
SAY
SCHEDULE
SCORE
SCRIPT
SET
START
STEP
STRING
SUM
SYNONYMS
SYNTAX
SYSTEM
THEN
TO
UNTIL
USE
VERB
VISITS
WHEN
WHERE
```

B.3 Additional Keywords

The following words are also keywords in the Alan language but may be used as identifiers without requiring the use of single quotes.

```
ARTICLE
BETWEEN
CONTAINS
DEFAULT
IN
MESSAGE
ON
QUIT
RESTART
RESTORE
SAVE
WAIT
```

B.4 The Grammar

```
 1. <adventure> = <optional_options> <units> <start>
                ;

 2. <optional_options> =
 3.                    ! 'OPTIONS' <options>
                       ;

 4. <options> = <option>
 5.           ! <options> <option>
              ;

 6. <option> = ID '.'
 7.          ! ID ID '.'
 8.          ! ID Integer '.'
             ;

 9. <units> = <unit>
10.         ! <units> <unit>
            ;
```

```
 11. <unit> = <default>
 12.        ! <object_default>
 13.        ! <location_default>
 14.        ! <actor_default>
 15.        ! <messages>
 16.        ! <rule>
 17.        ! <synonyms>
 18.        ! <syntax>
 19.        ! <verb>
 20.        ! <location>
 21.        ! <object>
 22.        ! <container>
 23.        ! <actor>
 24.        ! <event>
           ;

 25. <default> = 'DEFAULT' 'ATTRIBUTES' <attributes>
              ;

 26. <location_default> = 'LOCATION' 'ATTRIBUTES' <attributes>
                        ;

 27. <object_default> = 'OBJECT' 'ATTRIBUTES' <attributes>
                      ;

 28. <actor_default> = 'ACTOR' 'ATTRIBUTES' <attributes>
                     ;

 29. <attributes> = <attribute> '.'
 30.              ! <attributes> <attribute> '.'
                  ;

 31. <attribute> = ID
 32.             ! 'NOT' ID
 33.             ! ID <optional_minus> Integer
 34.             ! ID STRING
                 ;

 35. <synonyms> = 'SYNONYMS' <synonym_list>
                ;

 36. <synonym_list> = <synonym>
 37.                ! <synonym_list> <synonym>
                    ;

 38. <synonym> = <id_list> '=' ID '.'
               ;

 39. <messages> = 'MESSAGE' <message_list>
                ;

 40. <message_list> = <message>
 41.                ! <message_list> <message>
                    ;

 42. <message> = ID ':' <statements>
               ;

 43. <syntax> = 'SYNTAX' <syntax_list>
              ;

 44. <syntax_list> = <syntax_item>
 45.               ! <syntax_list> <syntax_item>
                   ;

 46. <syntax_item> = ID '=' <syntax_elements>
<optional_class_restrictions>
                   ;

 47. <syntax_elements> = <syntax_element>
 48.                   ! <syntax_elements> <syntax_element>
                       ;

 49. <syntax_element> = ID
 50.                  ! '(' ID ')' <optional_indicators>
;

 51. <optional_indicators> =
 52.                       ! <optional_indicators> <indicator>
                           ;

 53. <indicator> = '*'
 54.             ! '!'
                 ;
```

```
 55. <optional_class_restrictions> = '.'
 56.                               ! 'WHERE' <class_restrictions>
                                   ;

 57. <class_restrictions> = <class_restriction>
 58.                      ! <class_restrictions> 'AND'
<class_restriction>
                          ;

 59. <class_restriction> = ID 'ISA' <classes> 'ELSE' <statements>
                         ;

 60. <classes> = <class_identifier>
 61.           ! <classes> 'OR' <class_identifier>
               ;

 62. <class_identifier> = 'OBJECT'
 63.                    ! 'ACTOR'
 64.                    ! 'CONTAINER'
 65.                    ! 'INTEGER'
 66.                    ! 'STRING'
 67.                    ! 'CONTAINER' 'OBJECT'
 68.                    ! 'CONTAINER' 'ACTOR'
                        ;

 69. <optional_verbs> =
 70.                  ! <optional_verbs> <verb>
                      ;

 71. <verb> = <verb_header> <verb_body> <verb_tail>
            ;

 72. <verb_header> = 'VERB' <id_list>
                   ;

 73. <verb_body> = <simple_verb_body>
 74.             ! <verb_alternatives>
                 ;

 75. <verb_alternatives> = <verb_alternative>
 76.                     ! <verb_alternatives> <verb_alternative>
                         ;

 77. <verb_alternative> = 'WHEN' ID <simple_verb_body>
                        ;

 78. <simple_verb_body> = <optional_checks> <optional_does>
                        ;

 79. <verb_tail> = 'END' 'VERB' <optional_id> '.'
                 ;

 80. <optional_checks> =
 81.                   ! 'CHECK' <statements>
 82.                   ! 'CHECK' <check_list>
                       ;

 83. <check_list> = <check>
 84.              ! <check_list> 'AND' <check>
                  ;

 85. <check> = <expression> 'ELSE' <statements>
             ;

 86. <optional_does> =
 87.                 ! <does>
                     ;

 88. <does> = 'DOES' <optional_qual> <statements>
            ;

 89. <location> = <location_header> <location_body> <location_tail>
                ;

 90. <location_header> = 'LOCATION' ID <optional_name>
                       ;

 91. <location_body> =
 92.                 ! <location_body> <location_body_part>
                     ;

 93. <location_body_part> = <description>
 94.                      ! <does>
 95.                      ! <is> <attributes>
```

```
 96.                      ! <exit>
 97.                      ! <verb>
                          ;

 98. <location_tail> = 'END' 'LOCATION' <optional_id> '.'
                     ;

 99. <optional_exits> =
100.                   ! <optional_exits> <exit>
                       ;

101. <exit> = 'EXIT' <id_list> 'TO' ID <optional_exit_body> '.'
            ;

102. <optional_exit_body> =
103.                      ! <optional_checks> <optional_does> 'END'
'EXIT'
                            <optional_id>
                          ;

104. <object> = <object_header> <object_body> <object_tail>
              ;

105. <object_header> = 'OBJECT' ID <optional_where> <optional_names>
                       <optional_where>
                     ;

106. <object_body> =
107.                ! <object_body> <object_body_part>
                    ;

108. <object_body_part> = <properties>
109.                      ! <description>
110.                      ! <article>
111.                      ! <mentioned>
112.                      ! <is> <attributes>
113.                      ! <verb>
                          ;

114. <object_tail> = 'END' 'OBJECT' <optional_id> '.'
                   ;

115. <optional_attributes> =
116.                        ! <optional_attributes> <is> <attributes>
                            ;

117. <is> = 'IS'
118.      ! 'ARE'
119.      ! 'HAS'
          ;

120. <optional_description> =
121.                          ! <description>
                              ;

122. <description> = 'DESCRIPTION'
123.                ! 'DESCRIPTION' <statements>
                    ;

124. <article> = 'ARTICLE'
125.           ! 'ARTICLE' <statements>
               ;

126. <mentioned> = 'MENTIONED' <statements>
                 ;

127. <optional_name> =
128.                 ! <name>
                     ;

129. <optional_names> =
130.                   ! <optional_names> <name>
                       ;

131. <name> = 'NAME' <ids>
            ;

132. <properties> = 'CONTAINER' <container_body>
                  ;

133. <container> = <container_header> <container_body> <container_tail>
                 ;

134. <container_header> = 'CONTAINER' ID
                        ;
```

```
135. <container_body> = <optional_limits> <optional_header>
<optional_empty>
                      ;

136. <container_tail> = 'END' 'CONTAINER' <optional_id> '.'
                      ;

137. <optional_limits> =
138.                    ! 'LIMITS' <limits>
                        ;

139. <limits> = <limit>
140.          ! <limits> <limit>
              ;

141. <limit> = <limit_attribute> 'THEN' <statements>
             ;

142. <limit_attribute> = <attribute>
143.                    ! 'COUNT' Integer
                        ;

144. <optional_header> =
145.                    ! 'HEADER' <statements>
                        ;

146. <optional_empty> =
147.                   ! 'ELSE' <statements>
                       ;

148. <event> = <event_header> <statements> <event_tail>
             ;

149. <event_header> = 'EVENT' ID
                    ;

150. <event_tail> = 'END' 'EVENT' <optional_id> '.'
                  ;

151. <actor> = <actor_header> <actor_body> <actor_tail>
             ;

152. <actor_header> = 'ACTOR' ID <optional_where> <optional_names>
                      <optional_where>
                    ;

153. <actor_body> =
154.               ! <actor_body> <actor_body_part>
                   ;

155. <actor_body_part> = <properties>
156.                    ! <description>
157.                    ! <is> <attributes>
158.                    ! <verb>
159.                    ! <script>
                        ;

160. <actor_tail> = 'END' 'ACTOR' <optional_id> '.'
                  ;

161. <optional_actor_script> =
162.                          ! <optional_actor_script> <script>
                              ;

163. <script> = 'SCRIPT' <integer_or_id> '.' <optional_description>
                <step_list>
              ;

164. <step_list> = <step>
165.              ! <step_list> <step>
                  ;

166. <step> = 'STEP' <statements>
167.         ! 'STEP' 'AFTER' Integer <statements>
168.         ! 'STEP' 'WAIT' 'UNTIL' <expression> <statements>
             ;

169. <rule> = 'WHEN' <expression> '=>' <statements>
            ;

170. <start> = 'START' <where> '.' <optional_statements>
             ;

171. <optional_statements> =
```

```
172.                        ! <statements>
                            ;

173. <statements> = <statement>
174.              ! <statements> <statement>
                  ;

175. <statement> = <output_statement>
176.             ! <special_statement>
177.             ! <manipulation_statement>
178.             ! <event_statement>
179.             ! <assignment_statement>
180.             ! <actor_statement>
181.             ! <comparison_statement>
                 ;

182. <output_statement> = STRING
183.                     ! 'DESCRIBE' <what> '.'
184.                     ! 'SAY' <expression> '.'
185.                     ! 'LIST' <what> '.'
                        ;

186. <special_statement> = 'QUIT' '.'
187.                      ! 'LOOK' '.'
188.                      ! 'SAVE' '.'
189.                      ! 'RESTORE' '.'
190.                      ! 'RESTART' '.'
191.                      ! 'SCORE' <optional_integer> '.'
192.                      ! 'VISITS' Integer '.'
193.                      ! 'SYSTEM' STRING '.'
                         ;

194. <manipulation_statement> = 'EMPTY' <what> <optional_where> '.'
195.                           ! 'LOCATE' <what> <where> '.'
                              ;

196. <event_statement> = 'CANCEL' ID '.'
197.                   ! 'SCHEDULE' ID <optional_where> 'AFTER'
<expression>
                         '.'
                       ;

198. <assignment_statement> = 'MAKE' <what> <something> '.'
199.                          ! 'INCREASE' <attribute_reference>
                               <optional_by_clause> '.'
200.                          ! 'DECREASE' <attribute_reference>
                               <optional_by_clause> '.'
201.                          ! 'SET' <attribute_reference> 'TO'
<expression>
                               '.'
                              ;

202. <optional_by_clause> =
203.                       ! 'BY' <expression>
                           ;

204. <comparison_statement> = <if_statement>
205.                          ! <depending_statement>
                              ;

206. <if_statement> = 'IF' <expression> 'THEN' <statements>
                      <optional_elsif_list> <optional_else_part> 'END'
'IF'
                      '.'
                    ;

207. <optional_elsif_list> =
208.                       ! <elsif_list>
                           ;

209. <elsif_list> = 'ELSIF' <expression> 'THEN' <statements>
210.              ! <elsif_list> 'ELSIF' <expression> 'THEN' <statements>
                  ;

211. <optional_else_part> =
212.                      ! 'ELSE' <statements>
                          ;

213. <depending_statement> = 'DEPENDING' 'ON' <primary> <depend_cases>
'END'
                             'DEPEND' '.'
                           ;

214. <depend_cases> = <depend_case>
215.                ! <depend_cases> <depend_case>
```

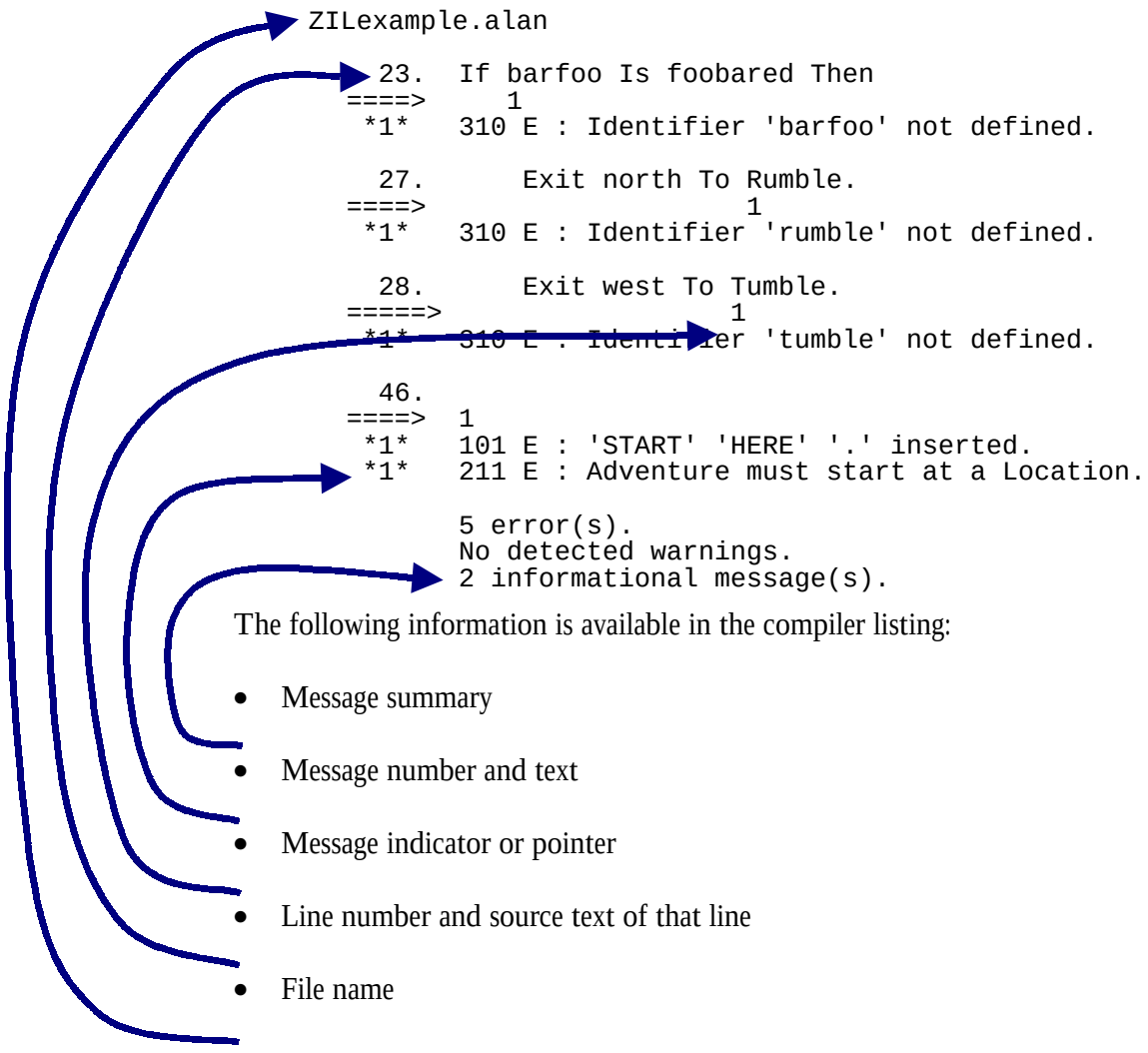
```
                  ;

216. <depend_case> = 'ELSE' <statements>
217.               ! <right_hand_side> ':' <statements>
                   ;

218. <actor_statement> = 'USE' 'SCRIPT' <integer_or_id>
<optional_for_actor>
                         '.'
                       ;

219. <optional_for_actor> =
220.                       ! 'FOR' ID
                           ;

221. <expression> = <term>
222.              ! <expression> 'OR' <term>
                  ;

223. <term> = <factor>
224.        ! <term> 'AND' <factor>
            ;

225. <factor> = <primary>
226.          ! <primary> <right_hand_side>
              ;

227. <right_hand_side> = <optional_not> <where>
228.                    ! <binop> <primary>
229.                    ! <optional_not> <relop> <primary>
230.                    ! <is> <something>
231.                    ! <optional_not> 'CONTAINS' <factor>
232.                    ! <optional_not> 'BETWEEN' <factor> 'AND' <factor>
                        ;

233. <primary> = <optional_minus> Integer
234.           ! STRING
235.           ! <what>
236.           ! 'SCORE'
237.           ! <aggregate> <where>
238.           ! '(' <expression> ')'
239.           ! <attribute_reference>
240.           ! 'RANDOM' <primary> 'TO' <primary>
               ;

241. <aggregate> = 'COUNT'
242.              ! 'SUM' 'OF' ID
243.              ! 'MAX' 'OF' ID
                  ;

244. <something> = <optional_not> ID
;

245. <what> = 'OBJECT'
246.        ! 'LOCATION'
247.        ! 'ACTOR'
248.        ! ID
            ;

249. <optional_where> =
250.                   ! <where>
                       ;

251. <where> = 'HERE'
252.         ! 'NEARBY'
253.         ! 'AT' <what>
254.         ! 'IN' <what>
             ;

255. <binop> = '+'
256.         ! '-'
257.         ! '*'
258.         ! '/'
             ;

259. <relop> = '<>'
260.         ! '='
261.         ! '=='
262.         ! '>='
263.         ! '<='
264.         ! '>'
265.         ! '<'
             ;

266. <optional_qual> =
```

```
                 ! 'BEFORE'
                 ! 'AFTER'
                 ! 'ONLY'
                 ;

<optional_not> =
                ! 'NOT'
                ;

<optional_id> =
               ! ID
               ;

<ids> = ID
      ! <ids> ID
      ;

<id_list> = ID
          ! <id_list> ',' ID
          ;

<optional_integer> =
                    ! Integer
                    ;

<optional_minus> =
                  ! '-'
                  ;

<attribute_reference> = ID 'OF' <what>
                        ;

<integer_or_id> = Integer
                ! ID
                ;

ID = IDENT
   ! 'DEFAULT'
   ! 'ARTICLE'
   ! 'MESSAGE'
   ! 'QUIT'
   ! 'SAVE'
   ! 'RESTORE'
   ! 'RESTART'
   ! 'WAIT'
   ! 'BETWEEN'
   ! 'CONTAINS'
   ! 'ON'
   ! 'IN'
   ;
```

C COMPILER ERROR MESSAGES

C.1 Format of messages

This appendix describes the error messages generated by the Alan compiler. The compiler presents the messages in the order of occurrence in the file. The offending source line is always shown together with the message. The following example illustrates a typical compiler output.



The following information is available in the compiler listing:

- Message summary
- Message number and text
- Message indicator or pointer
- Line number and source text of that line
- File name

To change what messages to show and where output is directed refer to the options and their descriptions in section *Compiler Switches* on page 110.

C.2 Message explanations

For each message, a short description of the error, possible causes etc. are given. Each message reported also indicates the severity of that error. The message is

supplemented with an indication of its severity. An informational message (indicated by an I) simply gives some information to the user, a warning message (W) indicates an error but the compilation still generates a valid output (although not always what the user intended). Error messages (E) indicate errors that have made it impossible to generate any output, but the compiler still continues to process all input. Fatal (F) and system (S) messages always terminate the compilation process immediately.

The message descriptions below may also contain the special insertion markers '%n' (where n is a digit), which indicate that text will be inserted at that point in the message during compile time, e.g. the offending identifier or a file name.

100 `Parsing resumed here.`

A severe syntax error was discovered. Some input was skipped. This error message marks the place where the parsing was restarted.

101 `%1 inserted.`

A syntax error was discovered and one or more symbols inserted in the input in an attempt to recover.

102 `%1 deleted.`

A syntax error was discovered and one or more symbols were skipped from the input in an attempt to recover.

103 `%1 replaced by %2.`

A syntax error was discovered and one or more symbols were replaced by one or more other symbols in an attempt to recover.

104 `Severe syntax error, construct ignored.`

An intricate syntax error was discovered. A complete construct was skipped in an attempt to recover.

105 `Syntax error, couldn't recover.`

106 `Parse stack overflow.`

107 `Parse table error.`

108 `Parsing terminated.`

Alan compiler implementation errors. Should not occur!

150 `Unterminated STRING.`

An opening double quote was not terminated by a closing quote before end of file. Error message points to the opening quote. Remember `STRINGs` may cover several lines.

151 `File name missing for $INCLUDE directive.`

An include directive was given but no file was indicated. The complete file name must be given according to the rules in section *Files* on page 59.

198 `Could not open output file '%1' for writing.`

The indicated output file could not be opened, probably because the directory did not exist or the file or directory was write-protected.

199 `Adventure source file (%1) not found.`

The source file given on the command line did not exist. The Alan compiler adds the **.alan** extension to the file name given, if it did not include a period.

201 `Mismatched block identifier, '%1' assumed.`

The identifier following a terminating END did not match the one given at the beginning of the construct. This indicates an illegal nesting or a missed END IF. The identifier indicates to which block the END is assumed to belong.

202 `Multiple usage of direction '%1' in this EXIT.`

203 `Multiple definition of EXIT '%1' in this location.`

The directional word indicated was used more than once, either in the same, or different exit declaration from the location. This is contradictory and not legal.

204 `Multiple definition of %1 DEFAULTS. Ignored.`

Only one declaration of default attributes per type is allowed. The second declaration is ignored.

205 `Multiple usage of '%1' in this VERB definition.`

When specifying actions for multiple verbs in the same declaration, the indicated word occurred twice.

206 `Multiple definition of SYNTAX for %1.`

More than one syntax definition for the same verb was found. This is an error. You should remove the offending one.

207 `VERB '%1' is not defined.`

A SYNTAX construct defined the syntax for a verb that was never defined.

208 `'%1' is not a VERB.`

The identifier on the left hand side of a SYNTAX definition was defined as something that was not a VERB.

209 `First element in a SYNTAX must be a player word.`

The definition of a SYNTAX construct may not start with a parameter. The first word must be a player word so as to distinguish it from other forms of input.

210 Action qualification not allowed here.

The BEFORE, AFTER and ONLY qualifiers may not be used in a DOES-clause in this context.

211 Adventure must start at a Location.

You specified a where expression in the START section that did not specify an explicit location. The start section specifies where the hero starts and must be a LOCATION.

212 Syntax parameter '%1' overrides symbol.

The SYNTAX definition valid in this context defined a symbol that is the same as an entity (object, location or actor). The syntax parameter will take precedence.

```
213  Verb alternatives not allowed here.
```

You may only specify different verb body alternatives within objects. The global verb body and the verb body in the location may not have alternatives.

```
214  Parameter not defined in syntax for '%1'.
```

The identifier given as the selector in a verb body alternative was not defined in the syntax for that verb.

```
215  Syntax not compatible with syntax for '%1'.
```

To be able to use the same body for different verbs by supplying them in a comma-separated list in the verb header they must all be compatible. This means that they have the same number of parameters and the parameters have the same names. Otherwise conflicts will arise when figuring out which parameter to use.

```
216  Parameter '%1' multiply defined in this SYNTAX.
```

The parameter was defined more than once in the same SYNTAX definition.

```
217  Only one multiple parameter allowed for each syntax.
       This one ignored.
```

To be able to use multiple parameters in a player command only one parameter may be marked as referring to multiple objects or actors using ALL or conjugations. This is a warning, the syntax will be as if the first multiple marker was the only one.

```
218  Multiple definition of attribute '%1'.
```

The indicated attribute name was defined more than once in the same context (default attribute list or within the same entity). Remove one definition.

```
220  Multiple definition of '%1'.
```

The indicated word has multiple, and possibly different, definitions.

```
221  Multiple class restriction for parameter '%1'.
```

The same parameter occurred more than once in the list of class restriction in the same SYNTAX definition.

```
222  Identifier '%1' in class definition is not a parameter.
```

Only the parameters in the syntax may be referenced in the class-restricting clause of a SYNTAX definition.

230 No syntax defined for this verb, assumed '%1 (object)'.

This message is a warning to indicate that the default syntax handling has been used.

310 Identifier '%1' not defined.

The indicated word was never defined. It must be declared either as a location, an object, a container, an actor or an event.

311 Must refer to %1.

The construct indicated does not refer to the correct kind of item, the message indicates which kind of item was expected.

312 Parameter not uniquely defined as %1, which is required.

In certain contexts it is necessary to refer to a particular type of entity, e.g. the `IN` expression must refer to a container or an object with the container property. If the reference (the WHAT part) is a parameter identifier, this parameter must be restricted to be of the required type by use of parameter restrictions (such as '`WHERE c ISA CONTAINER`').

```
315  Attribute not defined for '%1'.
```

The indicated attribute is not defined for the particular object, location or actor. It must either be a default attribute or be locally declared.

```
318  Entity '%1' is not a Container.
```

The referenced entity (object or actor) was not declared to have the container property, although the context required a container.

```
320  Word '%1' belongs to multiple word classes (%2 and %3).
```

A word was declared as to belong to different word classes such as noun, verb, adjective etc. Only multiple declarations that may lead to unexpected behaviour are reported, usually because of limitations in the current implementation. Generally it is allowed to declare a word e.g. as both an adjective and a noun.

```
321  Synonym target word '%1' not defined.
```

To define a synonym its target word (the word on the left side of the equal sign) must be defined as a proper word elsewhere in the source.

```
322  Word '%1' already defined as a synonym.
```

A word may not be declared as a synonym for different target words.

```
330  Wrong types of expression. Must be of %1 type.
```

In an expression, a value or an expression was used that had a type that was not allowed. The message indicates the correct type.

```
331  Incompatible types in %1.
```

The two values in an expression with a binary operator did not have compatible types, or the value used in a SET statement was not type compatible with the referenced attribute.

```
332  Type of local attribute must match default attribute.
```

An attribute declared locally (within an object, actor or location) that has the same name as a default attribute, has to have the same type (boolean, integer or string).

```
333  The word '%1' is defined as a synonym as well as of
       another word class.
```

Synonyms must be words *not* defined elsewhere.

400 Script not defined for Actor '%1'.

No script with the indicated identity was defined for the actor.

401 Actor reference required outside Actor specification.

Inside an actor specification it is permissible to leave out the actor reference in a USE statement in which case the surrounding actor is assumed. Outside actor specifications, the actor reference must always be supplied.

402 An Actor can't be inside a Container.

The LOCATE statement tried to locate an actor inside a container. This is not allowed.

403 Script number multiply defined for Actor '%1'.

The indicated number was used for more than one script for the same actor.

404 Attribute to %1 must be a default attribute.

To reference attributes for OBJECT, LOCATION and ACTOR the attribute used must be a default attribute, as all objects, locations or actors must have it.

405 The class of a parameter used in %1 must be uniquely defined.

In some statements the class of the identifier must be determined during compile time. This is, for example, the case in MAKE and SET statements.

406 A parameter defined as Container have no default attributes.

A parameter that was restricted to containers do not have any default attributes. Actors, objects and locations have separate sets of default attributes. In order to refer to an attribute on a parameter it must be restricted to one of these classes. If the parameter also requires the container property, use CONTAINER ACTOR or CONTAINER OBJECT.

407 Attribute in LIMITS must be a default attribute.

All objects must have the attribute that a limit is to test.

408 Attributes in %1 must be of boolean type.

The attribute referenced in the indicated context must be a boolean attribute.

409 No parameter defined in this context.

No parameter is defined in the context where a reference to OBJECT was made. Parameters are only defined within checks and bodies of verbs, so the use of OBJECT (an obsolete construct, use the parameter identifier instead) is also restricted to those contexts. See *Run-time Contexts* on page 61.

```
410  A parameter may not be used in %1.
```

In certain statements a parameter may not be used at all.

```
411  %1 ignored for Actor 'hero'.
```

It is allowed to redefine the predefined actor HERO (the player). This makes it possible to define local attributes and descriptions for the hero. However any definition of scripts or initial location is ignored (the script is supplied by the player in his input and the initial location is defined in the START section).

```
412  'ACTOR' is not allowed inside events.
```

In events no actor is active. This means that no reference to the active actor can be made. See *Run-time Contexts* on page 61.

```
413  Expression in %1 must be of integer type.
```

The context required a numeric expression.

```
414  Invalid initial location for %1.
```

The initial location specified was not valid.

```
415  Invalid Where specification in %1 statement.
```

The statement indicated does not allow the WHERE specification used.

```
416  Interval of size 1 in RANDOM expression.
```

This message informs that the interval in a RANDOM statement was just one single value, resulting in always returning the same value, not very random.

```
417  Comparing two constant entities will always yield the
same result.
```

The expression compared two identifiers none of which was a parameter. This will always give the same result. This is probably an error, but the message is still a warning as it gives a perfectly running adventure (but, perhaps not what you intended?).

```
418  Aggregate is only allowed on integer type attributes.
```

The aggregates MAX and SUM can only perform their calculation on integers.

```
419   Expression in %1 must be of integer or string type.
```

In the indicated context only integer and string type expressions may be used.

```
501   LOCATION '%1' has no EXITs.
```

In case the hero is located at the indicated location he may not be able to escape from that location. This may be intentional (as for a limbo location or a location with magic words to use as an escape) but the warning is presented as a reminder.

```
600   Multiple use of option '%1', ignored.
```

The indicated option was used more than once, this occurrence is ignored and the previous setting used.

```
601   Unknown option, '%1'.
```

A word was given in the option section that was not the name of an option.

```
602   Illegal value for option '%1'.
```

The indicated option does not allow the value used.

```
997   SYSTEM ERROR: %1
```

A severe implementation dependent error has occurred (a bug!). Please report.

```
998   Feature not implemented in %1.
```

The combination of some syntactically correct but semantically tricky constructs is not yet implemented. Please report.

```
999   No Adventure generated.
```

When an error is detected this informational message is given to indicate that no executable adventure was output.

D HOW TO USE THE SYSTEM

D.1 Compiling

This version of the Alan Adventure Development System is a traditional batch compiler. This means that the actual development system is a compiler that reads text files created using any normal text editor. To compile an adventure use the following command in a command shell:

```
alan <adventure>
```

where <adventure> is the name of the main file containing your adventure source text. The compiler will add an extension, **".alan"** (or **".ala"** on PCs), if none is supplied. The option **-help** will give a brief help on other options to the compiler.

The output from the compiler, **alan**, is two files, an adventure code file ***adventure*.acd** and an adventure data file, ***adventure*.dat**.

D.2 Compiler Switches

The compiler supports the following switches:

• **-charset** select the character set of the input files. This can be handy when you get a source file written on another platform, or for Windows where you edit in a Windows editor (ISO characters) and use the compiler in a DOS window (DOS characters). The option should be followed with one of the keywords **iso**, **mac** or **dos**

• **-verbose** print compiler version and other verbose messages

• **-warnings** show warning messages from the compilation process

• **-infos** show informational messages from the compilation process

• **-include** add a directory to the search path for included files (see *Files* on page 59 for details on the **include** directive). This switch can be used multiple times, each adding a new directory

• **-full** give a complete listing of the source on the screen

• **-height <n>** use page height <n> (lines) when producing list files

• **-width <n>** use page width <n> (columns) when producing list files

• **-debug** include debugging information in the produced adventure files (same as the debug option, see *Options* on page 26)

• **-pack** encode and compress the text data (same as the pack option, see *Options* on page 26)

-summary produce a summary about number of actors, size of adventure files, timing information etc.

• **-dump** print the internal form (developers use only)

Giving an extra hyphen before the option reverses its meaning, e.g. **--warnings** means don't show warnings. Switches may be abbreviated.

D.3 Running the Adventure

To play the generated adventure the Alan interpreter, **arun**, is executed with the adventure name as a parameter.

```
arun <adventure>
```

No extension on the adventure name is allowed.

If the interpreter program is copied to a different name it will look for code and data files with the same name. Any parameters or switches will be ignored. For example, by copying the **arun** program to **adventure** the interpreter will, when started under the new name, directly look for the files **adventure.acd** and **adventure.dat**. The three files **adventure**, **adventure.acd** and **adventure.dat** thus makes a complete game package which is easy to start using the single command:

```
> adventure
```

D.4 Interpreter Switches

The interpreter supports the following switches:

-v print the version of the interpreter

-d print the version of interpreter and enter debug mode

-i ignore CRC and version errors in the adventure files

-t trace sections executed

-s show single instruction trace

-l log all player command in a log-file in the current directory

In later versions an interactive development environment is envisioned but this is still far away. So you have to be content with the debugging support described in *Debugging* on page 75 for now.

E SYSTEM DETAILS

A complete Alan system should contain a compiler and an interpreter. They are normally called **alan** and **arun** respectively, but depending on the environment may have different names, such as **alan.exe**.

The Alan system is delivered packaged in different ways depending of the platform. On each platform the 'standard' way of packing software has been attempted. Seek local wisdom or look at the FTP-site ftp.gmd.de where the Interactive Fiction Archives are located for info.

Alan has been ported to many platforms (try the Alan Home Pages at http://welcome.to/alan-if for latest info). Below follows some very specific information for some of the platforms.

AMIGA

The Alan compiler requires more than the standard stack size (4096), a size of 20000 has been used without trouble.

The Alan interpreter **arun** supports Workbench-start-up through double- clicking on the Arun-icon. The tooltype WINDOW is supported to make it possible to select the window in which the adventure should be run. If a console handler device such as NEWCON: in 1.3 or the normal CON: in 2.x and above history and command line editing is available.

UNIX

On UNIX systems command history, recall and editing is available.

PC

In the PC environment Alan and Arun are command shell programs. This means that it needs an MS-DOS console to run. In this case it is most convenient to have the programs in your command path. Refer to your MS-DOS manuals for info on how to do this.

In a Windows environment you can associate the extensions **.ala** and **.acd** with the programs Alan and Arun respectively. This will enable compiling and running by double clicking on the files.

E.1 Portability of Games

The adventure files produced by the Alan compiler is compatible across all supported platforms. This means that by copying the binary **.acd** and **.dat** files to another machine they should be possible to interpret by an interpreter on that new machine without any changes. Note however that the files must be transferred in

binary mode (where applicable). All characters are automatically converted to the native set allowing multi-national characters to be presented correctly even on machines that do not support the IS0 8859-1 standard. This is of course restricted to characters having a representation in the current native character set.

It is a strong goal to achieve complete portability of the games to be able to provide games for all supported platforms without re-compilation. Game authors should take this into serious consideration when designing games and not use any system specific characters, character combinations or special commands that may be available on some systems.

Portability will not extend to different versions of the system. Changes in the game file format can occur between versions. Conversion tools may be available, older interpreter versions can be requested.

F VERSION DIFFERENCES

VERSION 2.8

A number of bug fixes have been performed in this release, mainly various problems with syntax declarations and named scripts.

A new statement has been introduced. The `DEPENDING ON` statement is an improved version of the common switch/case statement available in many other languages.

Two new expression types have also been introduced. The `CONTAINS` expressions performs sub-string containment tests, and the `BETWEEN` expression makes it easy to test if a value is within a specified, consecutive, range.

`SCHEDULE` and `RANDOM` now accepts general expressions instead of only literals.

VERSION 2.7

This version introduces the following radical improvements:

- objects no longer need be present to be used in a player command, the normal case is still that they are required to be present but this default behaviour can now be overridden using omnipotent indicator in the syntax clause.
- general attributes are now available, i.e. attributes that all objects, actors *and* actors have can be declared using the new `DEFAULT ATTRIBUTES` clause.
- objects and actors can now have multiple names giving the opportunity for synonyms for entities and not only for words.

Multiple minor improvements has also been made, e.g. free order of declaration of initial location and names, ',' can now be used as a conjunction in player input, a new `RESTART` statement, the `MESSAGE` clause now accepts general statements (not only strings) and named actor scripts.

VERSION 2.6

The 2.6 interpreter will run 2.5 games, but the 2.6 compiler can not generate 2.5 games. So upgrading to 2.6 will create games only playable with 2.6 interpreters, but you can keep old games and still play them.

User definition of run-time messages is now possible.

Removed the indefinite article from the default messages. Instead introduced the `ARTICLE` slot in objects which will be used (if present) before producing the `MENTIONED` message (which may be constructed automatically). If no article is

declared a default is supplied ("a" if using english). This means that some tricks that have been used to somewhat remedy the article problem ('a' was always used!), don't work any more. Remove all 'a', 'an' etc. from the texts and names in the Alan source (usually in the MENTIONED slot and possibly in the HEADER for containers), and introduce the ARTICLE "an" declaration on objects that require it (those whos name start with a vowel sound). For objects that doesn't need an article define an empty ARTICLE clause.

It also means that there is now a new reserved word ARTICLE.

It is also now possible to define the ARTICLE, MENTIONED and DESCRIPTION on objects in any order.

VERSION 2.5

String quotes (") within strings are now allowed, if doubled ("Charlie said ""Hello!"""). The same goes for single quotes (') within quoted identifiers.
(Luis Torres <let@reef.cis.ufl.edu>)

Multiple default attribute sections simplifies using general include files as the default attributes can be distributed across the complete adventure source.

The new VISITS statement replaces the previous option with the same name, allowing setting of the visits variable during run-time.

The compiler now generates completely cross-platform compatible adventure files, including multi-national character sets, which are converted automatically to be presented correctly on any supported platform.

If the interpreter is renamed it will automatically load adventure files (**.acd** and **.dat**) with the same name.
(Jeff Harrison <harrison@mprgate.mpr.ca>)

The QUIT statement prints a restart question which may be answered affirmative, in which the game is reloaded and restarted, or negative in which case the adventure is terminated.
(Byron Montgomerie <byron@saturn.cs.mun.ca>)

SAVE and RESTORE now prompts for a filename so multiple save files can be used by the player.
(Luis Torres <let@reef.cis.ufl.edu>)

Multiplication and division can now be performed using the '*' and '/' operators respectively.
(Robert Yoke-Loong Foo <af685@freenet.carleton.ca>)

VERSION 2.4

Actors may now be containers (allows for making them carry things). The class indications in the syntax declarations have been enhanced to account for this also.

You can now restrict parameters to all entities having the container property, only actors having it, or only objects having it (see *Syntax Definitions* on page 30, and *Containers* on page 41 for details).

String comparison normally ignores the case of characters (the new operator '==' does exact matching) (see *Binary operators* on page 52).

The statements to increase or decrease values are now called `INCREASE` and `DECREASE` (instead of `INCREMENT` and `DECREMENT`).

An optional description has been introduced on actor scripts, giving a possibility to create descriptions that are directly coupled to the activities of the actor (refer to *Actors* on page 43).

The `QUIT` statement now does not print any scores. This has to be made explicitly. Also the identifier `SCORE` is now allowed in expressions, representing the current value of scores collected so far.

Containers are now listed in a more natural way; the old format of one item per line has been replaced by concatenating them into a natural sentence, like:

```
You are carrying a box, a ball and a lightbulb.
```

This might require a change to the `HEADER` declaration of containers.

VERSION 2.3

String and integer literals are introduced in the player input and in the syntax declarations. Attributes may now also be strings. No incompatibilities should occur.

VERSION 2.0

In version 2, the concept of syntax is introduced. A programmer may allow different and more complex input from the player, not just the simple verb/ object type used in version 1. However, the default mechanism is still this simpler form of input so very little needs to be changed when converting to version 2. This also follows the spirit of Alan; it means that syntax is not strictly necessary unless you want to do something extra. For player input following the simple verb/object syntax there is nothing you have to do.

Another difference is the improvement in the definition of synonyms. First, the order of definition is different, you should now supply all the synonyms first and *then* the word they are synonyms for. This will probably require some rewriting of your Alan programs, but it is the more logical way to specify synonyms. Also, synonyms are now allowed anywhere in the program, so it is now possible to group global verb definitions, syntax definitions and synonyms for the same verb together (and perhaps place them in a separate include file).

G FUTURE DEVELOPMENTS

As Alan is an application-oriented language, i.e. it is designed to fit a particular application domain perfectly (in this case adventure authoring), it is dependent on adventure authors requirements and ideas for its further evolution. So please let us know!

email: thomas.nilsson@progindus.se
gorfo@ida.liu.se

postage: Thomas Nilsson phone: Int. +46 13 651 12
Junovägen 12 Nat. 013 - 651 12
S-590 74 LJUNGSBRO
SWEDEN

Göran Forslund phone: Int. +46 13 13 39 91
Vallmogatan 22 Nat. 013 - 13 39 91
S-582 46 LINKÖPING
SWEDEN

The Alan Home Pages on the Internet can be found at
http://welcome.to/alan-if

Here are some ideas of things we are thinking about:

- Definition of common attributes, verb definitions etc. through introduction of a class structure.
- Definition of interaction with actors, perhaps through some kind of pattern matching sub-language using string literals.
- Background pictures
- Sound
- ...

H REFERENCES

[Ada80] Scott Adams : *Pirate's Adventure*; BYTE December 1980, pp 192-212
An article describing the history behind the Scott Adam's adventures, particularly the *Pirate's Adventure*. Also includes BASIC source for the adventure, consisting mostly of DATA-statements.

[Bla80] Marc S. Blank, S. W. Galley : *How to Fit a Large Program Into a Small Machine*; Creative Computing July 1980, pp 80-87
A good article on the internals of the Z-interpreter, the pseudo-machine created by Infocom for creating and running adventures. As always from the hands of the Infocom men, also very good reading.

[Bet87] David Betz : *An Adventure Authoring System*; BYTE May 1987, pp 125-135
A description of a system similar to Alan, *AdvSys*, consisting of a special purpose language, a compiler and an interpreter for it. At last the term *authoring* is used instead of *programming*. The system is available through various PD-sources such as Fred Fish, BIX etc.

[Bra84] A. J. Bradbury : *Adventure Games for the Commodore 64*; Granada Publishing 1984, ISBN 0-246- 12412-1
A good book, especially on the topic of adventure writing methodology. Carries the concept of storyboarding a bit further than [Gra83]. Also contains interspersed utilities and modules (in C64 BASIC) and a small adventure, "The Case of the Lost Adventure".

[Bri84] Tony Bridge, Richard Williams : *Sinclair QL Adventures*; Sunshine Books 1984, ISBN 0-946408-66-1
Contains a few good chapters on adventures and reviews of some games of the classical text-type, but then goes on to present the listing of a fairly uninteresting "adventure generator" for a menu-driven *Dungeon And Dragons* inspired (much fighting, strength scoring and banes and such) kind of adventures games.

[Buc87] Mary Ann Buckles : *Interactive Fiction as Literature*; BYTE May 1987, pp 135-142
A very interesting article discussing the literary heritage of adventure games and their future in that perspective.

[Fic86] Erik Fichtelius : *Nu kommer det svenska äventyrsspelet!*; Upp&Ner, nr 2 1986
A swedish article describing the famous swedish "Stuga" game, created around 1980, which at that time was available for the PC.

[Gra83] Mike Grace : *Commodore 64 Adventures*; Sunshine Books 1983, ISBN 0-946408-11-4
A fairly good book on playing and writing adventure games, written by an beginner programmer. Strictly BASIC programming but contains many good

ideas to borrow. Includes some short sections on methods and mentions the concept of storyboarding. Contains a type-in adventure ("Nightmare Planet") for the C64.

[Geu85] A.F. de Geus, J.H. Jongejan, A.M. Koelmans : *Adventure Description Language*; Sigma Press 1985, ISBN 1-85058-011-1

Describes an assembler-like Adventure Language for the BBC Micro, and uses its design as a vehicle for briefly describing a few basic computer science techniques (e.g. grammars, hashing, huffman coding and graph theory). Source (in ADL!) for "Red Button" and "Long Forgotten Arabia" adventures plus complete source for the "scanner", "interpreter" and "editor" for ADL. Note: this is *not* the better known ADL by Ross Cunniff.

[Goe93] Phil Goetz : *Interactive Fiction*; Dept. of Computer Science, SUNY, Buffalo NY 14260, USA

Interesting paper setting out to define the term interactive fiction. Also discusses history and future of IF, and various media it may use.

[Gra87] David Graves : *Second Generation Adventure Games*; Journal of Computer Game Design, Volume 1, number 2 (August 1987), pp 4-7

An article describing many of the more fundamental concepts (conceptual and implementational) of interactive fiction of today, such as object orientation, natural language, text generation and goal orientation.

[Gra88] David Graves : *Bringing Characters to Life*; Journal of Computer Game Design, Volume 2, number 2 (December 1988), pp 10-11

Describes the role and implementation of artificial personalities in interactive fiction. This feature is seldom implemented in main stream interactive fiction but would probably give greater depth to the non- player characters in the story.

[Gra91] David Graves : *Plot Automation*; Journal of Computer Game Design, Volume 5, number 1 (October 1991), pp 10-12

The interesting idea of automatically creating a plot from the personalities and goals of the actors in the story is presented and discussed.

[Het84] Tony Hetherington : *Adventure Games*; Personal Computer World, January 1984 (October 1991), pp 17-26

Introductory discussion on what makes a good adventure, text vs. graphic, then some reviews on current games, e.g. The Hobbit and Snowball.

[Has80] Greg Hassett : *How to write An Adventure*; Creative Computing July 1980, pp 88-90

A short superficial article containing nothing that can't be found elsewhere.

[Leb79] P. David Lebling, Mark S. Blank, Timothy A. Andersson : *ZORK - A Computerized Fantasy Simulation Game*; IEEE Computer, April 1979

An interesting article describing the inner workings and motivations behind ZORK by the men who (almost) started it all.

[Leb80] P. David Lebling : *ZORK and the Future of Computerized Fantasy Simulations*; BYTE December 1980, pp 172-182

Lebling again describes the Zork world and machine. This article adds discussions on various implications of continuing to development, such as intelligent actors and communication with them, how far to take the parsing of natural language and how careful you must be before adding another feature in the games universe.

[Lid80] Bob Liddil : *On the Road to Adventure*; BYTE December 1980, pp 158-170

Some tips for playing and reviews of number of not so famous adventures (by Adams, Hassett, Programmer's Guild and Mad Hatter).

[Mit86] David Mitchell : *An adventure in programming techniques*; Addison-Wesley 1986, ISBN 0-201-15030-1

An excellent book covering almost every aspect of adventure playing and writing. As the title suggests adventure writing is taken as the goal for presenting various programming techniques, but still with the problems of writing and designing adventures as the primary issue. A bible for adventurers.

[McG84] Gary McGath : *COMPUTE!'s Guide To Adventure Games*; COMPUTE! Books 1984, ISBN 0-942386- 67-1

An excellent book, its primary merit is the reviews of most of the Infocom adventures, all Scott Adam's and a bunch of various other adventure games available and popular in 1984. Also contains a field guide for adventurers and a short discussion on how to program your own games. Includes source (in various dialects of BASIC!) for "Tower Of Mystery". The concluding chapter on the future of adventure games is most intriguing and may serve as a source for inspiration when trying to push its limits.

[Owe83] Peter Owens : *Adventures in Learning*; Popular Computing, December 1983, pp. 147-150

An article discussing how computer games, adventures in particular, can be used in education and their potential effect of learning people to think.

[Sca81] Peter D. Scargill : *Adven-80, An Advanced Adventure Development System*; Dr. Dobb's Journal, Number 61 (November 1981)

An interesting predecessor, assembler like in structure with a lot of "magic numbers", but was probably a good system at the time.

EXAMPLE ADVENTURE

This section contains a small example of how an adventure can be written in Alan. The emphasis has not been on the ultimate features of the language. Instead it is intended to show how much functionality can be achieved by just a few hundred lines of code.

```
-- This is an example of an adventure written in ALAN using almost
-- nothing of the more advanced features.

-- The story is not much: You have lost your memory and stumble around
-- on a narrow path in the middle of the jungle. To the north the path
-- takes you to a river and to the south to a clearing where a tiger
-- blocks your way. The only way to get past the tiger is to eat a
-- certain kind of fungus, which works as tiger repellant (a clue about
-- this can be found in your notebook). The fungus can only be found
-- by climbing the vine hanging down over the path. When you have
-- succeded in getting past the tiger the game gets to a happy ending.

-------------
LOCATION Path
-------------
    DESCRIPTION
    "You are standing on a barely visible path in the middle of nowhere.
    The path looks like it's been walked by bare feet (or rather paws)
for
    many a year. From the small amount of light reaching the ground here
    I should say the path runs in almost straight north/south direction.
    On both sides of the path is the deepest, darkest jungle you've ever
    seen. I really wouldn't recommend going that way. The path itself
    isn't much of a place to hold on to either. You get the impression
    that the vegetation is trying hard to recapture even this tiny part
    of land. The trees on both sides seems to come closer and there are
    vines hanging down almost touching your head."

    EXIT north TO bank.
    EXIT south TO clearing.
    EXIT east, west TO jungle.
END LOCATION.

-------------
LOCATION Bank
-------------
    DESCRIPTION
    "The path ends here on the south side of a wide river. On the ground
    you can see lots of paw prints (some pretty big ones, too). The
obvious
    guess is naturally that this is a common place for the wild animals
to
    stop by for a drink or two (and maybe a bite too). The river itself
    doesn't seem to be too dangerous - it's neither too wide nor too
rapid -
    but those logs with a pair of eyes give you second thoughts."


    EXIT north, swim TO river.
    EXIT south TO path.
    EXIT east, west TO jungle.
END LOCATION.

--------------
LOCATION Trees
--------------
    DESCRIPTION
    "You have now ended up high above the ground in the middle of the
    trees and vines. The vegetation is so thick up here that it seems
    almost like a green floor."

    EXIT down TO path.
END LOCATION.

--------------
LOCATION River
--------------
    DESCRIPTION
    "Defying the obvious horrors of the river you try for the northern
    river bank. One crocodile immediately chops your left foot of, but
```

```
    you makes it almost to the middle of the river before another
merciful
    crocodile finishes you off."
    QUIT.

END LOCATION.

-----------------
LOCATION Clearing
-----------------
    DESCRIPTION
    "Here the jungle opens up a bit and the path takes you straight into
    a clearing. The path seems to continue on the south side of the
    clearing some fifty paces away."

    EXIT north TO path.
    EXIT east, west TO jungle.
    EXIT south TO camp
        CHECK hero IS repelling
        ELSE "The tiger opens its big mouth and lets out a terrifying
            growl. Apparently it won't let you pass."
        DOES
            "When you approaches the tiger it looks confused. Then it
            really takes in your smell. It suddenly bolts, turns and
            takes off into the jungle."
            LOCATE tiger AT nowhere.
    END EXIT.
END LOCATION.

---------------
LOCATION Jungle
---------------
    DESCRIPTION
    "Now you've really done it. Didn't I tell you NOT to enter the
jungle."

    EXIT north, south, east, west TO jungle DOES
        "Stumbling around in the jungle trying to make your way through
        the damp vegetation that almost seems to reach out for you,
        you suddenly stumble onto a snake, which disapprove very clearly
        of you stepping on it. One bite in the leg and you have had it."
        QUIT.
    END EXIT.
END LOCATION.

-------------
LOCATION Camp
-------------
    DESCRIPTION
        "Here is the scattered parts of what ones was the camp of your
        expedition. The sight of it makes your memory come back. When
        you were attacked last night of a herd of wild elephants everyone
        fled in panic. You yourself ran straight into a tree and must
        have lost both conciousness and memory. 'Well, hope the computer
        still works.' you think. 'I think I stick to computer adventures,
        at least for the immediate future.'"
    QUIT.
END LOCATION.

----------------
LOCATION nowhere
----------------
-- The location for disappearing objects.
END LOCATION.

--------------------------------------------------------------------------

OBJECT Tiger AT Clearing
    DESCRIPTION
        "An enormous tiger is standing here blocking your way."
END OBJECT.

OBJECT Notebook IN inventory
    DESCRIPTION
        "The book is called 'The Jungle Book: Tricks and Tips'. It
        also has your name on it."

    VERB Take DOES
        LOCATE OBJECT IN inventory.
        "Taken!"
    END VERB.

    VERB Drop DOES
        LOCATE OBJECT HERE.
        "Dropped!"
```

```
    END VERB.

    VERB Read DOES
        "You open the book and glance over the notes. It is really
        a very strange mixture. Something about a tree you shouldn't
        hide under when it rains, 'cause some kind of bugs will start
        falling of its leaves, something else about a certain kind of
        fungus, which grows up among the vines and when eaten is a
        strong tiger repellant and something about how to make a fire
        from wet moss. Here are page after page of useful hints of
        how to survive in the jungle, all in your own hand writing."
    END VERB.
END OBJECT.

OBJECT Vine AT Path
    DESCRIPTION
        "A particulary long and thick vine is hanging down just beside
        you."

    VERB climb DOES
        "The vine is quite slippery, but you still manage to climb
        well into the trees."
        LOCATE HERO AT Trees.
    END VERB.
END OBJECT.

OBJECT Fungus AT Trees
    DESCRIPTION
        "Some kind of vaguely familiar fungus is growing here on a vine."

    VERB Take DOES
        LOCATE OBJECT IN inventory.
        "Taken!"
    END VERB.

    VERB Drop DOES
        LOCATE OBJECT HERE.
        "The fungus immediately clings to a new vine."
    END VERB.

    VERB eat DOES
        "You try a bit of the fungus. It doesn't taste bad although it
        isn't that delicious either. You swallow the rest of it almost
        without chewing. After a short while a strange odour starts
        perspiring from your body."
        LOCATE fungus AT nowhere.
        MAKE hero repelling.
    END VERB.
END OBJECT.

------------------------------------------------------------------------

SYNTAX take_inventory = 'inventory'.

SYNONYMS i = 'inventory'.

VERB take_inventory DOES
    LIST inventory.
END VERB.


SYNTAX 'look' = 'look'.

SYNONYMS l = 'look'.

VERB 'look' DOES
    LOOK.
END VERB.


SYNTAX 'quit' = 'quit'.

SYNONYMS q = 'quit'.

VERB 'quit' DOES
    QUIT.
END VERB.


------------------------------------------------------------------------
-- NOTE !   It is NOT necessary to declare the actor Hero (which is the
--          player himself). But IF you want to make in possible to give
--          the Hero certain attributes, THEN you have to declare it.

ACTOR Hero
```

```
   IS NOT repelling.
END ACTOR.

-------------------------------------------------------------------------

START AT path.
    "$p'Oh, my head. It hurts. Why am I out here when I've got this kind
    of headache? And where is 'here'? And who am I?'"
```

J COPYING CONDITIONS

The Alan Adventure Development System is now REGISTER-WARE. This means that for use of the system you are only required to register. This is done preferably through a simple email, but postal mail will also do, and is free.

Copies of the documentation and executables can also be received from the ThoNi&GorFo Adventure Factories, henceforth called The Factories, through email for free on request. Requesting delivery through email will automatically register the receiver.

A copy of these conditions must accompany any copy of the Alan System.

J.1 Distribution

The Alan System is mainly distributed through electronic mail. This distribution is free. Uploads to FTP sites and BBS *are* allowed, provided that the distribution package is uploaded in its original form, and download from there is of course also free.

Physical media, such as disk or tape, may be supported depending on plat- form. A requirement is that the requester supplies appropriate media. The cost for physical media distribution may vary.

J.2 Documentation

The documentation is copyrighted by The Factories. Copying is allowed provided it is distributed as a whole, or quoted accompanied with appropriate references.

J.3 Executables

The Alan system contains two executable programs, the compiler Alan and the interpreter Arun.

Distribution of the interpreter alone or together with game data produced by the compiler is allowed without restrictions or royalty claims provided appropriate references and acknowledgment accompanies the game in documentation or program output. In addition a description of the game, its plot and major features, and/or the game itself (preferably in source) should be donated to The Factories. The Factories agree to any copying or copyright restrictions placed on such a game.

The compiler may not be used, other than for evaluation or trial purposes, without registering with The Factories.

Registered users will receive free notification of updates, new platforms sup- ported, information on commercially or otherwise released games and other information supplied by other users or The Factories.

J.4 Registration

Registration is free and preferably made through a simple email message. Requesting a distribution through email will automatically register the requestor.

Registration can be done with an email on the Alan Home Pages at http://welcome.to/alan-if.

J.5 Source

The source is not in the public domain but can be acquired for porting to new platforms and technologies and support of such ports The source will not be released for other purposes.

J.6 Examples

The Factories would appreciate any example adventures or solutions to problems to improve the documentation and user support. However Alan source marked as an example will be considered not copyrighted and may or may not be used, as a whole or in part, in the Alan documentation or distributed in other forms by decision of The Factories.

This will also add to the suite of test data and therefore improve the quality of future releases as well as allow us to find and document any incompatibilities. If you also enclose a solution we can automate the testing even further. The Factories will not redistribute your game without your written permission.

J.7 Versions, compatibility and support

The Alan System is versioned using a three level number coding schema, indicating version number, release and correction respectively. Major differences in the language or the introduction of many new features will be indicated by an increment of the version number. Minor changes to the language and introduction of features are indicated by an increment of the release number. Bug fixes will increase the correction number.

Any adventure files and interpreters having the same version and release numbers will be compatible. Adventure files are also compatible across all supported platforms. This includes character sets, the intent being to correctly present any multinational character on any system. Thus complete coverage of the supported platforms from a single development machine can be achieved.

As the Alan System is a non-profit project user support may vary. To maximise probability of handling, error reports should be sent to The Factories (preferably by email) and include source, version of compiler and interpreter as well as a detailed description of how to reproduce the error and its symptoms.

Releases and corrections will be issued on irregular intervals.

J.8 Executive Summary

So, in short, the interpreter Arun and any game produced using the Alan Sys- tem is yours. You may sell or copy it as you like, and as you need the interpreter to run the game it may be copied freely too. The Arun interpreter may also be uploaded on BBS'es or FTP-sites to allow players to download an interpreter for his platform and use that to run your game.

The documentation and examples are free to copy or place on any BBS'es or FTP-sites if their contents are not changed.

To use the Alan compiler you must register.

Distribution on disks or tape may cost depending on the media. A floppy disc distribution is free, provided you supply the disk (we'll pay the return postage).

If you create a game using the Alan System we'd like to see it. Send us a copy (preferably in source) and any documentation or a description of the game and its novel features.

Short games or samples of Alan source are most welcome as examples that we might use and distribute to other users. Sending an example means you waive all rights to it. Examples add to the suite of test data and thus helps further improve the quality of the system.

K INDEX