The ALAN Adventure Language

Reference Manual

Preliminary!

Version 3.0alpha8

This version of the manual was printed on September 07, 2010

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 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 requiring only limited programming skills.

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

The author and a very good friend designed the first version of the Alan language some fifteen years ago. During several years of incremental improvement, it has now reached its third major version. This means that the language has a sound foundation, based on practical use. Therefore, features have been added as experience from use and understanding of the most prioritised needs have grown.

In this version modern and novel object orientation features has been incorporated into the language that allows definition of classes, instantiation and inheritance of attributes and other features. Do not worry if you find these terms incomprehensible at this point, Alan is still

an easy to use language and by reading this manual, you will understand how these new features may aid you in your quest for adventures.

1.1 A note on the pre-beta version

In this version of the Alan manual, you will find most information on version 3 of the language and the system. However, some parts of this manual are still version 2. Sections, which have not been changed, edited or at least reviewed for relevance has been given a grey background. In this way, you will know what to read with caution. In future pre-beta versions, these sections will gradually get fewer.

1.2 Programmer's Pitch

Alan is an application-oriented language. It features constructs that are natural to an author of Interactive Fiction. Alan is a strictly typed, compiled, object-oriented language with single inheritance. Classes inherit properties from their super-classes. The class system allows polymorphism so that instances of subclasses are valid wherever a super-class is specified. There are no explicit type declarations; instead, types are automatically inferred from expressions.

1.3 To the reader

There are probably four major types of readers of this document:

1. Readers completely new to interactive fiction – read the whole document from the beginning.

2. Readers familiar with writing IF but new to Alan – you should read from section 2.4 onwards.

3. Alan v2 users wanting to upgrade – read appendix A and then section 2.4 and onwards, with frequent use of chapter 4 as a reference.

4. Alan v3 users looking for detailed answers – use the index (and mail the author if you could not find an answer this way), look up the relevant sections in chapter 4 but also glance through chapter 2 from section 2.4. Use chapter 3 as a collection of examples.

Happy reading!

2 CONCEPTS

2.1 What Is An Adventure?

As long as man has been around there have been stories, fairy tales and fantasies. In the early days, storytellers told their stories 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.

Crowther & Woods started this form of games in the late sixties when they designed the famous *Colossal Cave Adventure,* which became

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 re-implementation 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 highly appreciated today.

Other companies have followed Infocom's example and a handful of them seem to make a living out of creating adventure games. However, today most of the works are created by devoted people that do it for the fun of it, releasing their games 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 (sometimes 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.

Currently, a community of addicted authors and players are still out there. Visit the vaults of interactive fiction on the Internet, and you will be surprised!

The Alan Adventure Language has been designed to aid construction primarily of pure text adventures or, in the words of Infocom, interactive fiction. Some sound and graphics functions are still available to spice up your game if you so desire.

The main feature of adventures is the interaction between the player and the game through commands input through the keyboard and descriptions printed on the screen. In appendix 8.58.5you can find a sample interaction.

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, we can see some common features. There is always the 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 to just two or three, or even a single location.

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. You can easily guess how to use a key, but what about the velvet pillow? Red herring objects are also common in adventuring.

The player must be able to express his wishes. Complete understanding of natural language commands from the player is probably overkill, 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

Alan is all about adventure games, or interactive fiction. In this manual, we will use both terms interchangeably since they convey two slightly different views on the purpose. But the technical platform, the Alan language and its support system, is the same, works the same and looks the same, regardless if you are designing a treasure hunt featuring an elaborate combat and hit point system or if you are competing with sir William Shakespeare himself.

2.3.1 What Is A Language?

A computer language is usually described as a set of rules for textual instructions for a computer. The idea is that a computer can follow those rules and perform the necessary and/or intended actions.

The Alan Adventure Language is a high-level computer language designed to make it easy to create text Adventures. This means that the rules for the language have been designed so that the textual instructions are relatively easy to read and write if you understand about the mechanisms that adventures are made from. In addition, the rules call only for minimal additional instructions to make those mechanisms work.

Compared to programming in a typical programming language, the Alan system handles most of the tiresome tasks and supplies reasonable defaults so that you can concentrate on the plot, the puzzles, the objects and the map. This makes Alan a true high-level computer language.

The Alan *system* consists of two computer programs, one of which analyses an input following (or at least intended to follow) the Alan *language*. This program is called the *compiler* and the analysis ensures that the input (the game description, in fact) makes sense. The compiler also, at the same time, converts the input into something more compact, the game file. This game file can be transferred and used without the compiler. Instead, to run an adventure the *interpreter* is needed. The interpreter is another program that reads the information in the game file, communicates with the player of the game (or reader of the work, if you like) and interprets all the complex mechanisms in your game logic so that it gives the player the illusion of the activities and events that you have designed.

To create works of interactive fiction using Alan, you also need a program with which to construct your Alan source code, a standard text editor, like Notepad or similar programs. However, you cannot use a word processor, like Microsoft Word, since the files created with those usually contains formatting information that the Alan compiler don't understand.

There are also special editors, or additions to standard editors, available, which supports Alan and helps with formatting and even compiling and running your game.

You might wonder why the game is not a single executable program. The answer is simple, compare the game with a Word-document. In order for the document to be visible, you need the Word-program that reads the Word-file and displays the content on the screen. As you probably know, the same program does not run on all computers. For example, you cannot install Word for Windows on a Macintosh.

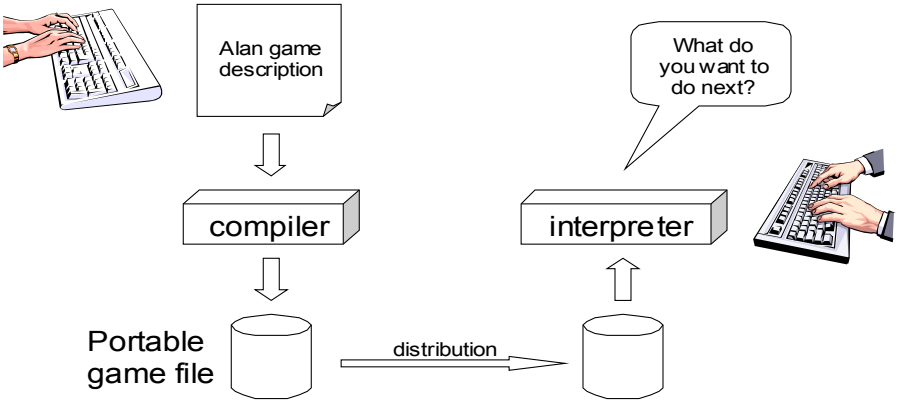

Figure 1 : The principles for and relations between a game description, a compiler, a game file and the interpreter, or, in other words, authoring and playing.

In view of this, it might be considered a nice thing that there are programs for Macintosh that read, display and print Word-documents. This makes the document files portable. Once you have a reading program on your computer, you can use all similar files on it. This is also one reason behind the compiler-interpreter design of Alan.

2·3·2 The Alan Idea

2The Alan language does not focus on variables, subroutines or other traditional programming constructs, because Alan is not primarily a *programming* language. Instead, Alan takes a descriptive view of the concepts of adventure authoring. The Alan language contains constructs that make it possible for you, 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 ordinary text shown to make the player experience the story that you have designed by reading them.

You will still need to understand how to vary your output depending on various conditions or information, let the player input control which events will happen, how to connect one location to another and how to store information for later need. In a way this is programming, but in an unusual sense.

In order to understand the rules of the Alan language, which this manual is all about, it is necessary to first establish some common ground. As an author you will need to have the same view as the Alan language has on some fundaments of what a work of interactive fiction is all about.

2·3·3 What's Happening?

The first fundament is that the execution of an adventure is primarily driven by the input of player commands. A command is analysed according to the player command syntax allowed by the author and, if understood, it is transformed into execution of verbs or movements, which in turn may trigger other parts in the game as described in the Alan source. After a player turn, other, scripted non-player characters or actors, can move, controlled by the computer, again according to the definitions in the source. Scheduled events are then run, and then the

player takes another turn. This is described in more detailed in section 6.1, A Turn of Events on page 158.

The following sections describe a number of the fundamental concepts that are present in an adventure game and what the Alan view of them is.

2·3·4 The Map

The scene for the game is a map of a number of connected locations. A location has a description that is presented to the player when that location is entered. A location may also have a number of exits stating in which direction there are exits and to which locations they lead. Alan places no restrictions on the layout of the map, any topology is allowed. Note however, that, in Alan, exits are one-way, and an explicit declaration of a backward path (if such is desired) must be made.

2·3·5 The Things

Most objects in an adventure 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 edible and movable, which may be changed during the execution of an Alan program. Most objects would e.g. probably not be edible so there is also a mechanism for declaring how these properties should be set by default, as well as mechanisms to override them, both for a particular object and for groups of objects.

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

2·3·6 Other People and Monsters

An extra thrill and dimension are additional characters in the game. In Alan, these are called actors and may 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 or a dragon guarding the entrance to its lair.

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.

2·3·7 Acting

The player commands action by typing imperative statements. These statements are analysed and results in execution ("calls") to verbs. The effects of these commands must be declared in verbs by the game author, either in an object (describing the effects of the verb when applied to an object) or as a general (global) verb that only applies without object.

2·3·8 The Input

To make it possible for the player to input more complex commands a means to specify the syntax for a verb is also available. A particular syntax is connected to a verb and describes how the player must phrase his input in order to command the triggering of a particular verb. Using this

mechanism, 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
```

2.4 Introduction to the Language

Alan is an Adventure language, i.e. a language designed to make it easy to write Adventures. This means that constructs in the Alan language reflects the various concepts encountered when creating an Adventure plot.

A common 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, we use **`Locations.`**

The next step is to introduce tools, weapons and other objects possible to manipulate. These are the **`Objects.`**

Then the player will need words to command action. The Alan language construct to supply these with is the **`Verb`**. Using the **`Syntax`** construct, you can also define more complex player input.

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

2.4.1 Notation

In this document, there are some typographical clues. Example Alan code is typeset in separate sections with a mono-spaced font:

```
This is an example of an example.
```

Later in the manual, there are semi-formal definitions, grammar rules, for how various constructs may be constructed. These sections are typeset against a coloured background:

```
The rules for the rules are available in appendix
LANGUAGE GRAMMAR on page 242.
```

In running text, words that are keywords or signify an Alan construct is written in a mono-spaced, bold, font. This helps distinguish the word 'the' from the Alan keyword '**The**'.

As shown in the last example, Alan keywords are written with the first letter capitalized. This is simply a convention and has no effect other than the visual. A keyword can be written **Keyword**, **KEYWORD**, **keyword**, or even **KeYwOrD** (if you are keen to show how good you are with a keyboard…). This manual tries to be consistent with using the first version (except in grammar rules).

2·4·2 The Locations

The scene for your Adventure is a series of "rooms" or, rather, locations. Locations are connected by exits, leading out of one location into 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 must have an identifier. This is so that you, the designer, may refer to that location easily, instead of having to remember a magic number for it.

Unless you plan to 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 necessary in a location, so lets look at an example (you would probably like to try this out, referring to appendix 8.5, SYSTEM DETAILS, on page 240 for instructions for your particular system).

```
The kitchen Isa location
   Exit east To hallway.
End The Kitchen.

The hallway Isa location
   Exit west To kitchen.
End The 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 a "**Start At**" sentence at the end, indicating in which location to put the hero when the game starts.

Type the above text into a text file, e.g. using a notepad program. Run this little Alan source through the Alan Compiler (see appendix 8.5, HOW TO USE THE SYSTEM, on page 237 and appendix 8.5, SYSTEM DETAILS, on page 240) 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" and press the return/enter key. The word "Hallway" and the prompt will appear. Typing "west" will take you back to "Kitchen" again.

The identifier for a location is automatically used as a description, a heading, shown when that room is entered. The words listed in the **Exit**-parts are also translated into directional commands the player can use in his input.

You should remember that exits are strictly one-way. An **Exit** from one 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.

However, just the name of the location is not much of a description. So in order to provide the "purple prose" descriptions often found in many Adventures there is an optional **Description**-clause that you can use. Let us describe the Hallway.

```
The hallway Isa location
   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 The 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, even spanning multiple lines with extra white-space included.

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.
```

2.4.3 The Objects

Another essential feature in Alan is the objects. Like the location, the object is a means to describe the "physical" world where your Adventure is taking place. Many objects are probably used to provide puzzles, such as 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 make a game so much more enjoyable.

Objects, like locations, have identifiers and descriptions, so you might guess the general structure of an object:

```
The door Isa object 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 The door.
```

An object may initially be located at a particular location. This is indicated by the **At**-clause, in this case telling us that the door is initially located in the Hallway. Objects do not have to start at a particular place in which case they are not present in the game until located, by executing some code, at some place where the player may lay his hands on them.

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 door should have the attribute `closed`, which initially is set to **TRUE** (implying that 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 142 and If on page 135).

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

It must of course be possible to change the value of attributes of an object. You can do this using 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 stating

```
Make door Not closed.
```

To close it (i.e. setting the closed attribute to TRUE again) you write

```
Make door closed.
```

The **Make** statement changes Boolean (or True/False) attributes. The **Set** statement changes numeric or string attributes, for example

```
Set level Of bottle To 4.
```

Note: These statements only change attributes. The implications of such a change must be implemented by writing Alan code that test these attributes and provides differing text output to the player. This is what gives the player *the illusion* of a door being open or closed for example.

Note: Alan do not understand, or enforce, any semantic in the identifiers for attributes, they are only identifiers. The illusion of the effects of differences in the value must be implemented by varying the output. In addition, Alan does not understand that an attribute 'closed', for a human would be the opposite of an attribute 'open'. You should choose one and stick to it.

Of course, attributes are not only available on objects, but on locations and other types of entities also.

Another manipulation 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 limbo to the current location with:

```
Locate treasure Here.
```

You could also relocate it to a particular place using the statement:

```
Locate vase At hallway.
```

2.4.4 The Actors

Actors can be used to populate the adventure with creatures, beings and other people. They might be pirates or monsters, but the thing they have in common is that they move around or at least 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.

```
The charlie_chaplin Isa actor Name charlie chaplin
   Script going_out
      Step
         Locate Actor At outside_house.
      Step
         Locate Actor At hallway.
         Use Script going_out.
End The charlie_chaplin.
```

2.4.5 Inheritance and Object Orientation

Object orientation is a term that is often used when talking about programming. The concept is modelled after a natural phenomenon first described by the Swedish botanist Carl Linnaeus (or Carl von Linné). He devised a naming system for flowers and plants that was based on features common between various species and families. The idea is that a general concept such as a mammal is defined by listing some features which all

mammals share. Specialisations such as sub-species in turn have other, more specialised, features in common.

In nature, we talk about species and individuals. In object oriented programming we talk about classes and instances, which are similar. Classes are abstract definitions of what the common features are and instances are individuals (data objects) having those features.

Inheritance and Instances

Inheritance means that a more general class can be restricted or specialised into new sub-classes. We say that the specialised class inherits from the more general. Most object oriented programming languages allows creating instances from any class, which does not happen in nature, there are no individuals that are mammals, they are individuals of some specific species of horse for example.

In programming, we can use this concept to make some things easier for ourselves. By collecting features that are common to many types of data objects into classes and sub-classes we can inherit those features. In this way, we can avoid explicitly, and repeatedly, stating those for every data object. One small drawback is that we have an implicit declaration of features, which can make reading a bit more obscure. We need to look up the parent class (or classes) for complete information about the object.

Polymorphism

By using inheritance, we can also guarantee the properties of similar, or related, objects. We can use this knowledge to use objects of various kinds, or classes, when we only require their common features. Both a horse and fish might for example be used if we only require properties common to all vertebrates. This flexibility, know as polymorphism, is possible in programming only by object orientation.

Every and The

The Alan language supports object orientation and inheritance with two constructs:

```
Every mammal Isa vertebrate …
The house_pet Isa cat …
```

The **Every**-construct defines a class and its properties and the **The**-construct declares an instance, which in this case inherits from the class

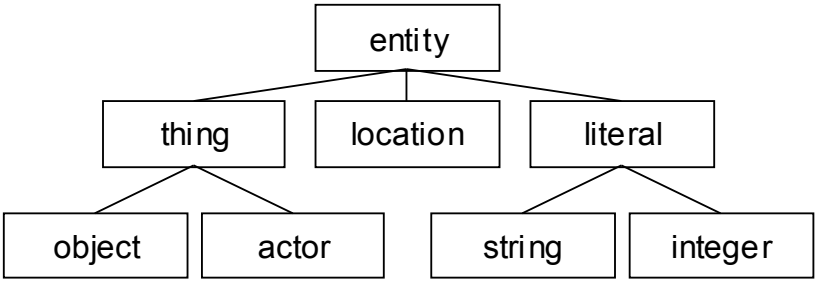

Figure 3 : Relationships between the pre-defined classes.

'cat'. The **Isa**-construct defines from which class another class or an instance inherits.

The Pre-defined Classes

To make it easy to get started there are eight classes pre-defined in the Alan language. They are **entity**, **thing**, **location**, **actor**, **object**, **literal**, **string** and **integer** and have the relationship, inheritance tree, shown in Figure 2 above.

The semantics of these pre-defined classes are in short:

- Only locations (instances inheriting from **location**) can be visited by the hero (the players alter ego)
- Only actors may have scripts that they perform
- Only things will be described automatically when encountered
- Literals and its sub-classes cannot be sub-classed. They are used to handle integers and strings in player input

See the subsections of Instances on page 62 for more detailed descriptions.

Creating Classes and Instances

In the sections above about locations, objects and actors the examples show how to create an instance of a class. Those examples show how to do it from the pre-defined classes. However, it is the identical if you have defined the class yourself. In general the format is

```
The <instance identifier> Isa <class identifier> …
```

To define a class you do much what you would expect:

```
Every <class identifier> …
```

After this, declarations of all the properties for that class follow. This could include inheriting from another class, e.g.

```
Every door Isa object
End Every.

Every openable_door Isa door
   Is open.
End Every.

The kitchen_door Isa openable_door
End The kitchen_door.
```

In this example, the **kitchen_door** has the attribute **open** although it does not specifically show in the declaration. It is initially set to true as specified in the class declaration.

Specialising and Overriding

Sub-classing, or specialisation, is usually used to add properties and thus make the instances of the sub-class more restricted, or specialised. In the example above, 'openable_door's are specialisations of 'door's since they have an attribute that the more general class does not have.

However, a sub-class can also redefine a feature. In the example above a class named '**closed_openable_door**' could be defined:

```
Every closed_openable_door Isa openable_door
   Is Not open.
End Every.
```

This makes all instances of the new class have the same attribute but it is set to false instead. The important thing is that the feature of having the attribute is common to all '**openable_door**'s. This is called overriding a feature.

This concludes this short description of object orientation and how the Alan language supports it. In the following descriptions, you just need to remember that most features can be inherited along the inheritance tree and be overridden, both during that inheritance and explicitly in the instance declaration itself.

2.4.6 The Verb Construct

The **Verb** is the construct that implements the effects of an action requested by the player. Verbs are associated with a class or an instance. 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 41 for an introduction, and Syntax Definitions on page 95 for a more thorough discussion). In this case, the attribute **closed** must also be available for all objects (by making it an attribute to the class **object**, see Additions on page 95 on how to add a attribute to a predefined class).

A **Verb** is either a simple command taking no parameters, like 'look', 'save' or 'help', or it involves one or more parameters that the player can reference. Simple verbs should be declared at the top level, globally, i.e. outside of any other declaration. Verbs taking parameters, on the other hand, must be declared within the class or instance, with which it is associated. For example, if a verb will handle objects it should be declared in the object class. The example above should probably best be placed in the door object itself.

Of course, there are also conditions that must be checked before we can execute this code (perhaps to see if it was possible to open the object!). Therefore, **Verbs** may have **Checks**, as we will see next.

Checking Things

In order to assert that the correct conditions are fulfilled before the body of a **Verb** is actually executed the verb may have 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 possible to open). 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.

Note: You can never destroy an instance, such as 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.

2.4.7 The Syntax

Normally a verb acts on one object or actor referenced by the player in a command, 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 if you don't specify anything else. The default syntax might thus be described as

```
Syntax
   ? = ? (parameter)
```

The question marks are placeholders for the name of the verb.

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 instance that represents the green pearl and the parameter **cont** will be bound 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
         Else "Nothing fits inside that."
```

This restricts the parameter **obj** to being an instance inheriting from the class object (as opposed to an actor for example) and the parameter **cont** to a container (an instance with the container property).

The parameters are used as normal identifiers in the Alan source. The parameters can only be referenced if they are defined in the current

context, i.e. they can only be used in the various bodies of the verb for which the syntax applies (see also Run-time Contexts on page 161 for a detailed discussion).

The **Syntax** construct allows more than one parameter, in order to make it possible to allow more complex player commands. Therefore, the verb execution order described previously from execution of verbs in one instance must be generalised to verb bodies in all the parameters. In the example above, verb bodies in the objects or actors referenced as **obj** and **cont** (the green pearl and the black box) are executed (if the verb is present in their definitions).

2.4.8 Text Output Formatting

Text output on the screen is caused by what you have written in the Alan source code. However, since text is coming from various places it is not easy or even possible, to anticipate the full context of a particular text.

Therefore, the Alan system takes care of some specific formatting issues. First, text will always flow neatly inside the window or screen. Lines will be broken automatically without braking in the middle of words.

Secondly, a few special cases are also handled automatically:

- After a full stop (period, the character '.'), exclamation ('!') or question mark ('?') and in the beginning of paragraphs, including location headings, the first character will be guaranteed to be upper case, automatically converted if necessary. This means for example that you don't have to consider the case when the name of an object might be printed as the first thing in a sentence. The name will automatically be capitalized. For example:

```
The postmen Isa actor At postoffice …
The postoffice Isa location
   Description
```

```
      Describe postmen.
….
```

Given the above snippet from a game source, the transcript would read:

```
Postoffice
Postmen are working behind the counters.
```

This would be the case even if the description of the postmen started with a lower case character.

<<Better example>>

- Two outputs following each other will automatically be separated by a space (a blank character). Except for the following case:

- If an output is immediately followed by another output starting with a full stop (period, the character '.'), an exclamation or question mark or a comma, and it is the only character in that output or it is followed by a space (blank character), no space will be inserted before that output. This rule will make sure that the full stop in the following source is automatically adjacent to the previous text, without the need to suppress spacing.

```
"You can't take" Say p. "."
```

3 WRITING WITH ALAN

This chapter will show you the basics of using the Alan features to write a small adventure. It is not a complete adventure. Instead, it will show you examples on how to implement commonly encountered needs.

The source examples are mostly excerpts from what is needed to get a complete game; this is indicated by the leading and trailing ellipses. Examples that are made by changing or adding from a previous example emphasize the differences by boxing them in.

More details about each construct are available in the Language Reference on page 53.

3·1 Locations

3·1·1 A Simple Location

A very simple location would still have a description. The game should probably begin at this simple location.

```
The starting_place Isa location.
  Description
    "This place looks like the place where everything
     begins."
End The starting_place.

Start At starting_place.
```

This is a complete game. Try it.

3·1·2 Interconnecting Locations

Add another location and then we can connect them with exits.

```
…
The starting_place Isa location.
  Description
    "This place looks like the place where
      everything begins."

  Exit w To other_location.
End The starting_place.
…
```

Do the same with the other location; otherwise, we cannot travel back.

3·1·3 Doors between Locations

Sometimes we want to place an obstacle in the way of the player. One, perhaps to common, is a door that hinders the free passage. To implement this we need a door object.

```
…
The door Isa object At starting_place.
  Description
    "There is a door to the west."
End The door.
…
```

If you try this you will find that the door shows, but doesn't hinder much. We must 1) make the door open and close, and 2) check that it is open before allowing passage. Let's take item one first.

```
…
The door Isa object At starting_place.
  Is closed.
  Description
    "There is a door to the west."
    If door Is closed Then "It is closed."
    Else "It is open."
    End If.
End The door.
…
```

The new lines add an attribute that stores the open/closed state of the door. However, as the player cannot see the attributes we have to show him the state by adding a conditional printout of our choosing.

Then we need to prohibit traversal when the door is closed.

```
…
The starting_place Isa location.
  Description
    "This place looks like the place where everything
     begins."

  Exit w To other_location
    Check door Is Not closed
      Else "You can not walk through a closed door."
  End Exit.
End The starting_place.
…
```

Of course, we also need the player to be able to open it. This is requires a verb for the door.

```
…
The door Isa object At starting_place.
  Is closed.
  Description
    "There is a door to the west."
    If door Is closed Then "It is closed."
    Else "It is open."
    End If.
  Verb open
    Does
      Make door Not closed.
      "You open the door."
  End Verb.
End The door.
…
```

You can implement "close" in a similar way. If you run this, you will notice that there is no door in the other room. In fact, there isn't even a problem getting back even if the door would be closed. You can find tips on how to fix this in Doors on page 168.

3.1.4 Specialized Locations

3.1.5 Dark Locations

3.1.6 Locations with common scenery

There are a couple of ways to provide scenery. By scenery, we mean instances that are available but not important to the game. Often these are common to a region, or set of locations in the game, e.g. outdoors you will always find the ground and the sky. The easiest way to do this is to create regions of locations using the nested location feature.

This feature simply allows locations to be located at other locations, in effect enclose them by an "outer" location. The effect of this is simply that instances present in an outer location are reachable from the inner. Instances in an outer location are not automatically described.

This can of course be expanded further but a simple example would look like:

```
The outdoor Isa location
End The outdoor.

Every outdoorScenery Isa object At outdoor
End Every outdoorScenery.

The sky Isa outdoorScenery
End The sky.

The ground Isa outdoorScenery
End The ground.

Every outdoorLocation Isa location At outdoor
End Every outdoorLocation.
```

3·2 Objects

3·2·1 Creating an Object

3·2·2 Specialized Objects

3·3 Actors

3·3·1 People versus Monsters

Since people usually have names, we might want to use that both in player references and in game output. An actor, Mr. Andersson, would need to have a name that the player can use:

```
The mr_andersson Isa actor
  Name mr andersson …
```

This allows the player to refer to Mr Andersson as in:

```
> ask mr andersson about neo
```

However, consider the natural implementation of the verb (details removed for clarity):

```
Verb ask_about
  Does
    Say The act.
    "does not want to tell you anything about"
    Say The subj.
    "."
End Verb.
```

This implementation will answer with

```
The mr andersson will not tell you anything about the neo.
```

Note the lower case initials of both mr Andersson and Neo. To make this right we will have to give them proper spelled names for the game to use. You can do this simply by writing

```
The mr_andersson Isa actor
   Name mr Andersson …

The neo Isa actor
   Name Neo …
```

This will instead give the output

```
> ask mr andersson about neo

The mr Andersson will not tell you anything about the Neo.
```

People are seldom talked about in definite form like that You don't say, "See, there goes the John." (At least not if you mean that John goes over there…) Instead, you leave out definite and indefinite articles. The Alan run-time system, and probably some of your own messages, will need to refer to an unknown instance in definite or indefinite forms. E.g.

```
You can not do that to a ball.
```

This works with things and actors that are referred to by their occupation or looks. However, to make people with names have natural definite and indefinite forms, you need to set the definite and indefinite articles to empty strings. You (or a standard library) could define a sub-class of actor to do this to all people, or persons with names:

```
Every namedPerson Isa actor
  Indefinite Article ""
  Definite Article ""
End Every person.

The mr_andersson Isa namedPerson
   Name mr Andersson
End The mr_andersson.

The neo Isa namedPerson
  Name 'Neo'
End The neo.
```

```
Office
Mr Andersson is here.

> talk to mr andersson

You can not talk to mr Andersson.

> ask mr andersson about neo

Mr Andersson knows nothing about Neo.
```

3.3.2 Making Actors Act

<scripts, descriptions, …>

3.3.3 Conversing with Actors

<verbs like ASK, TELL, actor status, memory …>

3.3.4 Specialized Actors

3.4 Player Interaction

3.4.1 Restricting Player Reference To Some Objects

<default local instances, class restrictions, …>

3.4.2 Easing Player Input

<synonyms, multiple syntaxes>

3.5 Common mistakes

<show how to interpret compiler messages and other problems which are caused by missing terminating string quotes, quoting including spaces, reserved words, events running where the hero isn't>

4 LANGUAGE REFERENCE

This chapter describes the Alan language in detail. Within each section, grammar rules are used to precisely define allowed formats. A description of how these rules should be interpreted can be found in appendix LANGUAGE GRAMMAR on page 242.

General Rules

The Alan language is divided into syntactic components of different kinds. Each component may be composed of text and/or other components. A component is terminated by a period or full stop ('.'). This indicates that that component is complete. Some components start with a keyword or initial phrase, such as '**Description**' or '**Exit east To kitchen**'. If it is to be followed by further components, such as statements or output strings, that keyword or phrase are normally not be followed by a period, but by its continuing components. For example:

```
Exit east to Kitchen.
```

But

```
Exit east To Kitchen
  Check kitchenDoor Is open
  …
End Exit.
```

Note that the first is terminated, but the second example is continued with a check, and not terminated until the **End Exit**.

4.1 An Adventure

An adventure starts with an optional options section (see Options on page 55) followed by a set of declarations.

```
adventure = {option} {declaration} start_section

declaration  = import
             | class
             | instance
             | addition
             | syntax
             | verb
             | rule
             | synonyms
             | event
             | messages
```

The declarations constitute the major part of the adventure. The declarations can be declared in any order and repeated freely.

```
start_section = 'START' where '.' statements
```

The adventure source text must end with a start section. It indicates where the hero is when the game starts but 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..."
```

4.2 Options

Options define things concerning the overall behaviour of the generated Alan adventure. As is implied they are optional and are only required if you need to change the value of an option from its default setting. An option follows the grammar

```
option = id '.'
       | id id '.'
       | id integer '.'
```

The following examples illustrate how an option can be written.

```
Language Swedish.
Debug.
No Pack.
```

The available options are

Option name	Values	Default value
Language	English, Swedish, German[1]	English
Width	24-255	80
Length	5-255	24
Pack	Boolean (on or off)	Off (No Pack)
Debug	Boolean (on or	Off (No Debug)

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

off)

The Language option specifies the language in which the adventure is assumed 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. It is completely possible to write in other languages, but then you must customize all the message texts. See page 210, appendix section Input Response Messages for a complete list of such messages.

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). This means that portability between platforms should be good even for text containing multi-national (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>**). These values are only used if the interpreter itself cannot get the actual values.

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 code file. As a minor drawback, it does make the execution of the adventure a bit slower (noticeable only on some very old, smaller, computers).

In order to allow debugging of the generated adventure (see Debugging on page 181), the debug option must be turned on. This may also be

performed using the debug compiler switch (see Compiler Switches, on page 237).

4.3 Types

The Alan language handles information in bits, values. Each such bit of information, or data, is of a specific type. Alan is a strictly typed language, which means that assignment, comparisons and other statements will require that rules concerning the compatibility between such values are not broken.

In the Alan language, you cannot explicitly state the type of a value. Instead, this is inferred from how values are used, e.g. the initial value of an attribute or the restrictions put on a syntax parameter.

4.3.1 Basic, Simple and Compound Types

The basic types of values available in the Alan language are:

- Integer – e.g. a simple integer constant, a reference to an integer typed attribute or a numeric expression using any of the mathematical operators.
- String – e.g. a string constant or a reference to an attribute typed as a string.
- Boolean (true or false) – comparisons yield Boolean values, Boolean attributes.

Two other simple types are available:

- Instance – a reference to an instance or an attribute typed as a reference attribute that refers to an instance.
- Event – a reference to an event or an attribute typed as a reference attribute that refers to an event.

There is one compound type in the Alan language:

- Set – an unordered list of values.

4.3.2 Instance Type

Every time a reference to an instance is made, it can be considered an expression of instance type. In these cases, the class of the instance also often matters. E.g. assigning a reference attribute can only be made if the new value refers to an instance that belongs to the same class or a subclass of the initial value of that attribute.

Some types of expressions return a value referring to an a class or instance in the Alan source. Examples include an identifier bound to a parameter allowing instances and a reference attribute.

4.3.3 Event Type

Event is a set of statements that can be scheduled to execute with a specified delay. Each reference to an identifier of an Event is of course of the Event type. Events can be referenced by attributes and any reference to such an attribute is of Event type.

Expressions of Event type can be used in **`Schedule`** and **`Cancel`** statements.

4.3.4 Set Type

A Set is a set of values collected into a single value. The order of elements is not specified. Each member can only occur once in the same set, but a member can occur in multiple sets. You could for example include one set of numbers (integers) in a set and another set in another set. It is then possible to investigate the sets and remove all members that are members in both.

The Set type is a compound type since it is not complete without a member type. You can only include members in a set if the type compatibility rules allow it. A Set may include members that are instances or integers.

If the Set includes instances, the subclass compatibility rule applies. All members in the set must inherit from the same class. See the section on type compatibility below.

Note: The fact that an instance is in a Set does not affect the instance. In fact, there is no way to find out in which Sets, if any, a particular instance is included. In particular, it does not affect the instances location.

4·3·5 Type Compatibility

Assignment and comparisons between values requires the values to be compatible. The three basic types (integer, string and Boolean) are only compatible with themselves.

Values of the Instance type can be compared without restriction, except that there is no notion of lesser or equal, so only equality can be tested. Assignment can be made if the new value is of the same class, or of a subclass, as the attribute or variable that receives the value. This class is normally inferred from the initial value of the declaration.

For example, a reference attribute (an attribute referencing an instance) is inferred to be restricted to instances of the class of its initial value. Any subsequent change of the attribute (setting it to refer to another instance) requires that the new instance be of the same class or a subclass thereof.

These rules ensure that attribute references and other properties are always retained during the execution of the whole game. Thus, it will never cause a run-time error on the player.

4.3.6 Type Requirements

Some statements require their arguments to be of a specific type. This is enforced by the compiler. The compatibility rules apply here also, given that the required type is given by the statement itself.

Examples include the conditional **If** statement that requires a Boolean value (or expression) to test and the **Use** statement, which requires references to instances that are subclasses of the predefined class 'actor'.

4.4 Import

The source text for a large adventure might become entangled and complex. A way to break up a large text is to divide it into separate files. Each such file can then be imported into the main source using the **import** statement.

```
import = 'import' quoted_identifier '.'
```

The quoted identifier is the name of the file to import, see File on page 157. The **import** may be placed anywhere in a file where a declaration can occur, and the effect will be the same as if the contents of the named file had been inserted at that position in the file. Imports may be nested, so an imported file may in turn import more files, without limits.

An imported file is searched for first in the current directory and then in any of the directories indicated using the **import** switch as described in Compiler Switches on page 237, this search is performed in the same order as the **import** switches occurred on the command line.

4.5 Classes

```
class = 'EVERY' id
           [inheritance]
           {property}
        'END' 'EVERY' [id] ['.']
```

Classes are definitions of templates of instances. That means that a class declaration only describes instances, and does not add anything to your game in itself. Instead, you have to create an instance of the class to make it available in the game (see Instances below).

The **id** is the identifier used by the author to refer to this class throughout the source code, e.g. when referring to it in the inheritance clause of other classes and instances.

The **properties** are described in Properties on page 67.

4.5.1 Inheritance

Every instance must inherit from a class (see Inheritance and Object Orientation on page 35). Furthermore, user-defined classes must also inherit from other classes. A class or an instance inheriting from a class will get all properties of that class. All properties explicitly declared in a class or instance inheriting from another class will extend, override or complement those properties as specified in the original, parent, class. This way, you can easily create new classes by extending existing ones.

You specify which class another class or an instance inherits from using a clause following the grammar:

```
inheritance = 'ISA' id ['.']
```

For example

```
The door Isa object …
Every coin Isa treasure …
```

4.6 Instances

The most important part of an Alan game source is probably the declarations of instances. Instances are the objects, locations, actors and other things that fill your game universe. The player traverses and interacts with these in his quest to negotiating your game.

```
instance = 'THE' id
             [inheritance]
             {property}
           'END' 'THE' [id] ['.']
```

Every instance must inherit from a class (see Inherit above) keeping all properties of that class. Each inherited property can be amended or overridden by specifying it in the declaration of the instance, and new attributes, exits and scripts can be added in the same way as in class declaration.

Exactly the same rules for declaring properties apply to instances. The only difference is that an instance will actually show up in the game when it is run. Remember also that properties declared in an instance are not common to any other instances (unless the declaration overrode the value of a class property).

Instances inheriting, directly or indirectly, from the predefined classes **thing**, **entity**, **object**, **location**, **actor** and **literal,** are subject to special semantics and restrictions.

4.6.1 Entities

The base class **entity** represents the lowest denominator of all instances. All other pre-defined classes inherit from **entity**. So adding a property to **entity** will add it to every instance.

Entities cannot have an initial location, nor can they be located anywhere. On the other hand, they can be considered to be available everywhere. They are not described when encountered. They can only be shown by explicitly executing a **Describe** statement.

So, if you want an instance to always be available but invisible, create an instance of **entity**. It is also possible to create subclasses of **entity**. Instances of such classes will follow the same rules.

4.6.2 Things

Thing is a pre-defined subclass of **entity** that adds the property of having a location. This means that they can have an initial location and be located to locations and into containers. They will, however not show up in descriptions or listings, but the player can refer to and interact with them. They can be described by explicitly executing a **Describe** statement.

Creating an instance of **thing** is a good choice if you want an invisible instance that should only be available at particular locations, or under specific circumstances.

Note:A thing can be put in a container, but that container will not show any visible traces of that thing. It will be rendered as empty if listed. The thing is however subject to selection by a random selection of items in the container. See Random Values on page 144 for a description of random selections of container items.

4.6.3 Objects

Objects are instances inheriting directly or indirectly from the predefined class **object**. 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). In addition to the properties inherited from **thing**, any present object will by default, be described when the player enters a location or otherwise encounters it.

4.6.4 Actors

The predefined class **actor** is intended for providing so called NPC:s, non-player characters, in your game. Like the player, they can move around but to do this they have to be scripted, i.e. programmed with some behaviour using scripts.

An instance inheriting from the **actor** class will be described when encountered. Actors can be located, as can any **thing**, but not be inside a container. In addition, they can have scripts.

Actors also exhibit special behaviour when they are described, e.g. when they are encountered. If an actor is executing a script with a description, (see Script on page 93) this description will be used instead of the one declared in the description clause.

```
The kirk Isa actor 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 The kirk.

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...
:
```

The Hero

There is one very special actor, the hero, which represents the player. This actor is always pre-declared, but if necessary, it can be re-declared in the same way as any other actor.

One situation when this is required is if you need attributes on the hero, such as "sleepy" or "hungry". A declaration like the following can then be used:

```
The hero Isa actor
  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 The hero.
```

The container property of the hero is actually the inventory container, which is also pre-declared, see The inventory 90.

4·6·5 Locations

A location is a declaration of a place (a "room") in the game that (normally) can be visited by the player, and have objects lying around, etc. In fact, the map of your game is a set of interconnected locations. A location is any instance inheriting directly or indirectly from the predefined class **`location`**. Inheriting from **`location`** implies the following semantic properties:

- only locations can be visited by the player
- only locations may have the **`Entered`**-clause
- things and locations may be located to locations
- exits can only lead to locations and only locations can have exits
- the start location must be a location
- locations can't have container properties
- verbs in locations are executed only when the hero is at that location

When a location is described (for example when entering it) it is presented with a heading (the location name), the description (in the description clause) followed by descriptions of any present objects and actors not already, explicitly, described (using a **`describe`** statement) in the description.

An interesting property of locations is that a location can be located at another, both initially and during run-time. The result of having such nested locations is that all things present at the "outer" location are also present in the inner. This can be used in multiple levels to allow access to sky, ground and other scenery items available at many locations at once. It can also be used for grouping locations into sets of similar locations and for implementing vehicles.

4.6.6 Literals

The classes **literal**, **string** and **integer** cannot be instantiated explicitly. Instead, you might say that they are implicitly instantiated when the player inputs a literal. For example

```
> turn dial to 12
```

The second parameter (see Syntax Definitions on page 96) in this player command is the integer 12. This parameter is automatically considered an instance of the pre-defined class **integer**.

It is possible to add verbs to **literal** and its sub-classes. This way it is possible to create verbs that take strings and integers as parameters.

4.7 Properties

An instance or class can be given number of different properties by declaring them in the declaration of the class or instance.

```
property = initial_location
         | name
         | pronouns
         | attributes
         | initialization
         | description
         | articles
         | mentioned
         | container_properties
         | verb
         | script
         | entered
         | exit
```

Attributes, exits, verbs and scripts can be repeated any number of times in the same declaration. You cannot use the same identifier for more than

one such property, e.g. you cannot declare two attributes with the same name.

4.7.1 Inheriting properties

A property can be inherited from the parent of the class or instance. It is not necessary to repeat the declaration in the inheriting class or instance if it should retain its inherited value. Each inherited property may be amended or overridden by specifying it also in the declaration of the inheriting class or instance according to the following table.

Inherited Property	
Initial location	Overridden
Name	Accumulated, the inherited names are appended at the end of the list of `Name` clauses
Pronoun	Overridden, each pronoun clause inhibits inheriting pronouns from the parent class.
Attribute values	Overridden, attribute declarations using the same name as an inherited can give the attribute a different value but must match the type of the inherited. Accumulated, you can add further attributes in a class or instance.
Initialize	Accumulated. Inherited initialize clauses are executed first so that the base classes may do their initialization first.
Description check	Accumulated.

Description	Overridden.
Articles & Forms	Overridden.
Mentioned	Overridden. Also overrides names.
Container	Overridden, all clauses are overridden.
Verb declarations	Accumulated. Verb bodies are accumulated for verbs with the same name as the inherited. Use qualifiers (see Verb Qualification 108) if you don't want all of them to execute.
Scripts	Overridden, for same script name.
Entered	Accumulated. Entered-clauses in nested locations are executed from the outside in. Entered-clauses in parent classes are executed first. So the first clause to be executed is the parent of an outer location.
Exits	Overridden, for same direction.

The table also show which properties are inherited separately from the parent. E.g., you can override the description but keep the description check, or even add another (since they are accumulated). You cannot override the container limits and keep the header section since the container property is overridden in its entirety.

In an inheriting class, you can also add new properties. More attributes, verbs, exits and scripts can be added to those already present through the inheritance.

The properties available for use in classes, and thus also for instances, are described in detail in the following sections. In general, all of these can be

mixed freely, however, some semantic restrictions apply as to when a particular property is or is not legal.

Initial locationWhere an instance will be located when the game starts is set using an optional **Where** clause. If no such clause is used the instance will have no location. An instance without location is not present (in the view of the player) in the game until it is moved somewhere by a **Locate** statement.

initial_location = where

Only the **At what** and **In what** forms of the **Where** construct (see WHERE Specifications on page 139) are allowed when describing an initial location of an instance.

```
The chest Isa object At tresury
…
```

An instance inheriting from **location** cannot have an initial location that is **In** something, but it can be **At** some other location, creating a nesting of locations.

4.7.2 Names

By default, the identifier ("author name") for an instance is also the name shown to the player, and by which he will be able to refer to it. Normally you would want to override this with more elaborate and alternative names. You can do that using the **Name** clause.

```
name = 'NAME' id {id} ['.']
```

The **Name** clause consists of a list of identifiers optionally followed by a full stop.

The identifiers given in the **Name** clause is used when the instance is presented to the player and which the player can use to refer to it. For example

```
The south_door Isa object At south_of_house
  Name door
  …
The south_of_house Isa location
  Name 'South of House'
  …
```

The quoted identifier used in the last example makes the name be one single text string. See Words, Identifiers and Names on page 153 for an explanation of this. This works for locations, which a player usually does not need to refer to, but for things the player should interact with, a more sophisticated mechanism is available.

```
The chair3 Isa object
   Name little wooden chair
```

In this example, the name is a sequence of words. The semantics of this declaration is that the word "chair" is a noun and "little" and "wooden" become adjectives. When the player refers to the object with the author name (identifier) **chair3**, he may use just "chair" if it is the only accessible object with "chair" as its noun, or he may distinguish between multiple chairs by also giving one or more adjectives to be more precise about which chair he meant.

Note: The **Name** clause hides the author name, so in the example, the player will not be able to use **chair3** to refer to the instance.

Note: An explicit **Mentioned** clause will override the names for presenting the instance.

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

```
The rod Isa object At grate
   Name rusty rod
```

```
    Name dynamite
...
```

This would allow the player to refer to the object using either 'rusty rod' or 'dynamite'. (Or as a side effect 'rusty dynamite'.) The first name clause is used for building a default description, if necessary (see Description on page 7979).

The character case used in any word is retained for output, but player input will always be matched without considering case. This way you can e.g. give capitalized names to people giving a correct output.

Inheriting Names

Names can of course be inherited. This is done in an additive way so that any names inherited are appended to the **Name** clauses in the declaration. This ensures that the class or instance itself can control the primary name (the first **Name** clause). In addition, this has the effect that an instance inheriting from a class defining a **Name** will be possible to refer to also using the inherited name(s). Here is an example with fruits:

```
Every fruit Isa object Name fruit …
Every apple Isa fruit Name apple …
Every pear Isa fruit Name pear …
The gravensteiner Isa pear …
The macintosh Isa apple …
```

In this example, both the pear and the apple would be possible to refer to using the word "fruit".

Displaying instances

When an instance is to be shown to the player, it must be displayed in form of text. An instance can be printed in several different ways, it can be described or only mentioned. A description of an instance is a complete and usually more elaborate description of it (see Description on

page 80). However, often an instance must be mentioned as a part of a sentence, or in a list.

Such a mentioning of an instance will involve the articles, the name and possibly the **Mentioned** clause.

The basis for this mechanism is the short form, which by default is the first of the **Names**. It will, however, be overridden by any existing **Mentioned** clause (see Mentioned on page 85).

The short form can be automatically transformed to a description (for instances that have no **Description**) by inserting the article (see Articles and Forms on page 83) and the short form in a default message. In the following example, output of the article is underlined and the short forms are emphasised, the rest is the default message templates.

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

The interpreter also uses this principle when constructing lists of instances in container content lists (as the result of the execution of an implicit or explicit **List** statement, see page 122).

4·7·3 Pronouns

In player input, it is often handy and natural to refer to items using pronouns, such as "it", "them" or "her". Alan provides a means to define with which pronouns each instance can be associated.

pronouns = 'PRONOUN' word { ',' word }

The effect of associating a pronoun with an instance is that the player can refer to that instance explicitly in one command and then in a subsequent command use that pronoun to refer to it again. Assume the player input

```
> ask the priest about the bible
```

If the priest has been associated with the pronoun "him" and the bible with the pronoun "it", the next command could be

```
> give it to him
```

Pronouns are inherited as any other property, but are overridden as soon as a pronoun clause is present.

Note: The pre-defined class **entity** defines the pronoun "it" (or equivalent for other supported languages).I39

4.7.4 Attributes

An attribute is a labelled value that instances have. The declarations of attributes are placed inside a class definition (in which case it will apply to all instances of that class or instances of any sub-class of it) or inside an instance declaration (in which case only this instance will have it, unless it overrode an already inherited attribute with new values). A declaration of an attribute follows the structure:

```
attribute_declaration = id
                      | 'NOT' id
                      | id integer
                      | id string
                      | id id
                      | id '{' values '}'
```

An attribute can be of Boolean (having truth values), numeric, string, event, instance or set type. The type of an attribute is automatically inferred from the type of its initial value.

Attributes that you want every instance of a class to have must be declared in that class. E.g. to declare a Boolean attribute that all instances of the class animal will have in common, the following code can be used:

```
Every animal …
  Is
     Not human.
…
```

The attribute **human** will now be available in all instances of the class, without further declarations, and it will be false. If you want the attribute to have another value in a particular instance, you must declare it specifically in that instance and give it its desired value, which will be effective only for that instance. You can override the value in a subclass, e.g.

```
Every person Isa animal …
  Is
     human.
…
```

Boolean Attributes

A Boolean attribute is declared by simply giving the attribute name, or the name proceeded with the keyword **Not** (indicating a **FALSE** initial value):

```
thirsty.
Not human.
```

Numeric and String Attributes

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

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

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 attributes to classes, as they (in the current implementation) require memory and takes time to initialise when starting the game.

Event Attributes

Attributes can refer to events. Such an attribute is declared by giving the identifier of an event as its initial value.

```
Event e1
 "This is e1 running."
 Set e Of l To e2.
End Event.

The l Isa location
  Has e e1.
End The l.
```

An attribute of the event type can for example be used to dynamically remember which event is scheduled, so that it can be cancelled.

Reference Attributes

Reference attributes stores references to instances. Such an attribute is of instance type; the class is determined by the class of the initial instance that the attribute is referring. You may for example store a reference to the other side of a door.

```
The east_door Isa door.
   Has otherside west_door.
…
```

You must initialize a reference attribute with a reference to an instance belonging to a class having the required properties. Any subsequent assignment to the attribute will require that the new value is a member of the same class or a subclass of it. This ensures that operations on instances referenced by that attribute will always be possible.

Inside a class declaration, reference attributes may be initialized with a class identifier instead of a reference to an instance. This makes the attribute an *abstract* attribute, since it is defined but not initialized. Any instances inheriting from this class must then initialize the attribute, either explicitly or indirectly (by initializing it in an intermediate class). E.g.

```
Every door Isa object …
  Has otherside door.
End Every door.

The east_door Isa door.
   Has otherside west_door.
…
```

Tip:If you need to set the initial value to refer to an instance of a sub-class of the actual class you want to allow, you can use an instance of the required class in the declaration and set its correct initial value in the **Start** or **Initialize** sections.

Set Type Attributes

A Set is an unordered set of integers or instance references. Initial members must be listed in the declaration of the Set. See Set Type on page 58 for details on the Set type.

The type and class of allowed members is inferred from the values actually in the initial set. If they are instance references, the common ancestor of all members is used as the class of the allowed members. An empty set is only allowed as an initial value if the attribute is an inherited attribute since in this case, the member class is known from the inheritance and need not be indicated in the declaration.

You can also initialize a set type attribute with a set consisting only of a single class identifier. This will create an empty set with instance type members restricted to that particular class.

Tip:If you require an initially empty set of another type, e.g. integer, and you cannot give the member class by inheriting it, you can initialize the set with a value of the correct type and remove that value in the **Start** or **Initialize** sections.

Inheriting Attributes

Attributes can be inherited like any other property. A declaration of an attribute with the same name as in any of the parents of the instance or class, will inherit the type of the attribute, you cannot change it in subsequent declarations. This means that any declaration of a different initial value than the inherited must follow the rules of type compatibility for assignment. (See Type Compatibility on page 59.)

This also applies to classes of instances in the reference and set types attributes. Both these types allow references to instances. The initial value given at the point where the attribute is introduced determines the required class of the set members or referenced instances. This is retained throughout the complete inheritance of that attribute even if a subsequent initial value would imply a more specialised class. An example:

```
Every door Isa object
  Has otherside someDoor.
End Every door.

Every lockable_door Isa door.
  Has otherside someLockableDoor.
End Every lockable_door.

The someDoor Isa door
  Has otherside someLockableDoor.
End The someDoor.

The someLockableDoor Isa lockable_door
  Has otherside someDoor.
End The someLockableDoor.
```

In this example, the reference attribute **otherside** is introduced in the class **door**. Its initial value is referring to the class **door**. This makes the attribute refer to doors. In the subclass **lockable_door** the attribute is used with another initial value, here it refers to a subclass of **door**. Despite this, the attribute in the two door instances will allow reference to doors, as indicated by the first declaration (in the class **door**).

As a contrast, the same example can be used with abstract reference attributes (reference attributes that are defined, but not initialized, in the class declaration).

```
Every door Isa object
  Has otherside door.
End Every door.

Every lockable_door Isa door.
  Has otherside lockable_door.
End Every lockable_door.

The someDoor Isa door
  Has otherside someLockableDoor.
End The someDoor.

The someLockableDoor Isa lockable_door
  Has otherside someDoor.
End The someLockableDoor.
```

Now the class declarations refer to classes instead of instances in their declaration of the **otherside** attribute. This changes the semantics so that the subclass indicated by **lockable_door** actually makes it illegal to use a **door** as the declaration in **someLockableDoor** does, instead a **lockable_door** is required.

Using abstract reference attribute declarations in class declarations allows you to progressively refine the class of the instances that that attribute may refer to.

4·7·5 Initialize

The attributes of an instance can be initialized using values in the attribute declaration. This is usually sufficient for many situations. For more flexibility, the **Initialize** clause can be used.

```
initialize = 'INITIALIZE' statements
```

The clause makes it possible to execute arbitrary statements before the game is started. The statements are executed before the **Start** clause is executed. This enables calculation of more complex initial attribute values to be located within the instance, or class, that requires it.

The current location is set to the start location, and the current actor is the hero during the execution of all **Initialize** clauses.

The **Initialize** clause is inherited with local declarations accumulating to the inherited. Any inherited **Initialize** clause is executed before the locally defined, this lets the base classes do their initialization before the initialization of the current, more specialized, class or instance is performed.

4.7.6 Description

The statements in the **Description** clause should print a description of the instance. These statements are executed when the hero encounters the instance. Depending on from which base class the instance inherits this can be a location description presented when the hero enters the location or when executing a **Look** statement. Other possibilities are descriptions of objects and actors. See sections 4.6.1 through 4.6.4 for descriptions of what inheriting from the predefined base classes means.

Note: The description should not change any game state since it might not always be executed depending on the settings of the **Visits**. In particular, the description of a location should not move the hero; this might lead to a recursive loop of descriptions. This might instead be managed by the **Entered** clause.

See also Special Statements on page 124, concerning the **Visits** statement.

The syntax for simple descriptions is:

```
description = 'DESCRIPTION' {statement}
```

If the **Description** clause is missing for an instance (and no description is inherited), 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 be printed, not even a default one. This can be useful for objects already described by the location description, or of objects with particular properties.

Here are some examples of simple description declarations

```
The south_of_house Isa location
   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."
   …
The door Isa object
   Description
      "In the north wall there is a large wooden
       door."
      If door Is closed Then
         "It is closed."
      End If.
   …
```

Before executing a description, you can check for various conditions to be met. A common example is the dark room. If there is no light source present, the description should not be printed. The syntax for such a description is

```
description = 'DESCRIPTION' [checks] [does]
```

You can guard the description with a check in the same form as with verb bodies (see Verb Checks on page 104 for a detailed description of checks). Of course, there are no qualifiers possible here. To be able to separate the checks statement from the actual description statements the keyword **Does** is required. This is an example of the checks for a dark location:

```
Every dark_location Isa location
   Description
      Check Sum Of light_source Here > 1
         Else "It is pitch black. You are likely
               to be eaten by a grue."
End Every dark_location.
```

Note that it does not specify any description statements. This is because the checks and the actual description are inherited separately, as described in the table on page 67. The actual descriptions are left for the instances.

If multiple description checks are available in the inheritance chain, they are all tested and must be met before any description is attempted. So the inheritance of description checks is "additive".

If any check fails, the description will not be executed. This particularly also implies that the default listings and description of present objects and actors in location instances will not occur either. Note, however, that any events and actor actions *will* be shown. See Locations below for a description of default description mechanism for locations.

If neither a check nor any description statements occur after the keyword **Description** this *is* a description, but it is empty.

Note:You should *not* put statements that changes game state in the **Description** clause. Descriptions can be executed in various

circumstances that the game author has no control over. Consider **Exit** statements and the **Entered** clause instead.

4.7.7 Articles and Forms

```
forms = indefinite | definite | negative

definite = 'DEFINITE' article_or_form

indefinite = [ 'INDEFINITE' ] article_or_form

negative = 'NEGATIVE' article_or_form

article_or_form = 'ARTICLE' {statement}
                | 'FORM' {statement}
```

The optional definite, indefinite and negative articles and forms can be used to define how an instance is printed in its indefinite, definite and negative forms. There are two cases for each form, either as an article prepended to the short display form of the instance (its names or **Mentioned** clause), or a complete form replacing the normal name printing.

Indefinite forms are used in e.g. inventory listings and when presenting instances that have no **Description** clause. Definitive forms are usually used in messages of the type:

```
The door is locked.
```

The negative forms are used in standard messages of the type:

```
I can't see any door here.
```

Articles and **Forms** can of course, be inherited.

Note: The predefined base class **entity** defines the default definite, indefinite and negative article to be “the”, "a" and “any” (if using English). You may override this by using an **Add** statement.

Articles

Printing the indefinite (or definite or negative) form of an instance having an indefinite (or definite or negative) article is simply performed by executing the article statements and then the normal printing of the instance, usually the first set of names.

For example

```
The owl Isa object
   Indefinite Article "an"
:
```

This results in output like

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

An article is not used when the instance is dispayed when acting on multiple objects, as in:

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

For instances that should not have any article at all, like ‘some money’, or ‘mr Andersson’, an **Indefinite Article** clause containing no statements must be used:

```
The money Name some money
   Article
:
```

This will lead to:

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

Instead of

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

Form

If an instance has a **Definite** (**Indefinite** or **Negative**) **Form**, either through declaration or inheritance, the printing of its definite, indefinite or negative form will be by executing the corresponding statements only; no article declaration is involved. In this way, the author gets complete control over the spelling and inflection of the instance name in definite, indefinite or negative forms. Some human languages will probably require more use of the **Form** form (like Swedish), and some less (like English). The forms are particularly useful if the natural language used, have different forms of the noun itself in definite an indefinite forms. An example is the Nordic languages, which use definite suffixes instead of articles.

The **Article** and **Form** are inherited as one property. That means that an instance may override its inherited form using either of the forms regardless of how its parent defined the form.

Printing

You can use various forms of the **Say** statement (see Say on page 121) to choose in which form the instance will be presented. In addition, the embedded parameter references allow selection of the form (String Statement on page 118).

4.7.8 Mentioned

The optional **Mentioned** clause overrides the name for displaying an instance in a short form that will be used when the instance is mentioned e.g. in listings of containers or when the **all** form of player input is used. A typical use of the **Mentioned** clause is to let some internal state of the instance be reflected in the short form, e.g. if you want the short form of a box to show if it is open or closed you can not rely on the

Names since they are static. Instead, the **Mentioned** clause can print a different short name depending on an attribute.

```
mentioned = 'MENTIONED' {statement}
```

For example:

```
Mentioned
   If mirror Is broken Then
      "broken"
   End If.
   "mirror"
...
> take all
(little black book) OK!
(green pearl) OK!
(broken mirror) OK!
```

Note:A mention clause declared on a class will override the names of any instance that inherits from it.

4.7.9 Container Properties

An instance can also be a container. This is declared by means of the **Container** property clause. The grammar is

```
container_properties = ['WITH'] ['OPAQUE'] 'CONTAINER'
                           ['TAKING' id]
                           [limits]
                           [header]
                           [empty]
                           [extract]
```

For example

```
The chest Isa object
   With Container
      Limits ...
```

```
      Header ...
    Description ...
    :
End The chest.
```

A container is something that can contain instances. By default, the instances it can contain must be inheriting from the base class **object**, but by using the **Taking** clause, you can allow any instances.

Instances with the container property, "inherits" a special, predefined, Boolean attribute, **opaque**. This attribute can be manipulated in the same way as any other attribute. Its current value indicates if the instances inside the container are visible and accessible or not.

By default, containers expose their content, but by placing the keyword **Opaque** in the container declaration, you indicate that this container declaration will initially prohibit access to the contained instances. A typical use of this is to prohibit access to contents of closed cases, drawers and boxes. Once open such containers usually reveal the content, which then can be accessed. You can implement such behaviour by modifying the built in **opaque** attribute. For example:

```
The drawer Isa object
  With Opaque Container
    Header "The drawer contains"
  Verb open
    Does
      Make drawer Not opaque.
      List drawer.
  End Verb.
End The drawer.
```

Note: If you want to hide the content of a container, you have to take care so that a **List** statement is not executed while the container is opaque since this will reveal the content. You can check the state of the **opaque** attribute like any other Boolean attribute.

Note: The **opaque** attribute is only available in instances and classes having the container property.

When an instance with the container property is encountered during game play, it will be described as usual. If the instance has a default description the content of the container will be listed if it is not empty and not opaque.

Limits

The **Limits** clause of the container property declaration put limitations on what and how much can be put in the container.

```
limits = 'LIMITS' {limit}

limit = limiting_attribute 'ELSE' {statement}

limiting_attribute = attribute_definition
                   | 'COUNT' integer
```

If any of these limits are exceeded when trying to locate anything inside the container, the statements in the corresponding **Else**-part will be executed and the players turn aborted. In fact, these checks are performed because of the execution of a **Locate** statement (usually as a result of the player issuing a command with the intent of placing something in a container). This means that the execution of a sequence of statements can actually be interrupted in the middle by these limitations.

The specification of an attribute, which must be a numeric attribute on the class the container takes (by default object), implies that the sum of this attribute of all objects in the container cannot exceed the value specified. The special attribute **Count** can be also be used and indicates a limitation on the number of instances allowed.

```
Container
   Limits
      weight 50 Else "You can not lift that much."
      Count 2 Else "You only have two hands!"
```

Container properties are inherited in its entirety. Locations can't have container properties.

Header and Else

```
header = 'HEADER' {statement}

empty = 'ELSE' {statement}Header is used when the
contents of the container are listed. It is intended
to produce something like
```

```
"The box contains"
"You are carrying"
```

It is followed by a list of instances mentioned. Section Mentioned on page 85 describes this listing.

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 <container> contains" and "The <container> is empty." respectively.

Extract

The **Extract** clause defines what happens when anything is extracted from the container. Any **Locate** statement that moves an instance out of a container is considered an extraction. The movement will be subject to the restrictions enforced by the **Extract** clause.

```
extract = 'EXTRACT' {statement}
        | 'EXTRACT' [check] [does]
```

The first form will just execute the statements when the extraction takes place, in effect making extraction impossible.

The second form, including optional **Check** and **Does** clauses, allows prohibiting the extraction of the item from the container. If the **Check** is present, it works the same way as for **Verb**s (see Verb Checks on page 105). I.e. a **Check** without a guard expression will unconditionally prohibit extractions; a **Check** with an expression will evaluate that expression and, if false, execute the **Else** clause. The **Does** clause will be executed if, and only if the optional **Check** passed. An **Extract** clause without a **Check**, but with a **Does** is equivalent to the first form.

An example use of this is to prohibit, or put restrictions on, the hero taking things carried by other actors.

The inventory

The inventory, i.e. the container containing all objects carried by the hero is pre-declared[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.
```

2 The inventory is actually the container properties of the hero (see section Actors on page 64 for a discussion of actors and their container properties). Any object put into the container will be available in the hero also.

One possible re-declaration of the inventory can serve as one 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.
```

4.7.10 Verbs

Verbs declared inside an class or instance are inherited in the same way as other properties.

The semantic rules for a verb in a class or instance is that it will only be a candidate for execution if the instance bound to a parameter is of the corresponding class, or is the instance. See also Verb Execution on page 109.

4.7.11 Entered

```
entered = 'ENTERED' {statement}
```

The **Entered** clause is only allowed in instances inheriting from the predefined class **location**. This clause will be executed whenever any actor enters the location. Game state changes can be made without restriction. However, it is primarily intended for setting up the location in a correct way, not for describing events, actions and states changes. For this the **Description**-clause is recommended.

If all, or some, of the statements should only apply to a particular actor, it is possible to test for the **Current Actor** with a simple **If** statement.

Note: The **Entered** clause is not executed if the actor is already at the location.

Note: If it is the Hero that is moving the **Description** of the new location will be executed *after* the **Entered** clause.

Note: The **Entered** clause is inherited in an accumulating fashion so that entered-clauses from parent classes are executed before the one in the location. This also applies to nested locations (see Locations on page 66), so that **Entered**-clauses in outer locations are executed first.

4·7·12 Exits

To build a traversable world of locations, they must be connected. This is done using exits. The syntax for an exit declaration is

```
exit = 'EXIT' id {',' id} 'TO' id [exit_body] '.'

exit_body = [checks] [does] 'END' 'EXIT' [id]
```

An exit has a list of identifiers, all of which are considered directional words. I.e. when any of those words is input by the player, he will be located at the location identified as the target of the exit. It is possible to customize the exit using a **Check**, that must be satisfied to allow passage through the exit, and statements (**Does**) that will be executed when the player passes through. The checks work as described in Verb Checks on page 104.

If either of the **Check** or **Does** clauses is present, the **End Exit** is required.

Two interconnected locations might be declared like:

```
The east_end Isa location 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 The east_end.
The west_end Isa location 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 The west_end.
```

Note:If an exit is declared from one location to another, there will *NOT* automatically be an exit in the opposite direction! You have to define the reverse passage also.

Exits are only allowed in classes or instances inheriting from the pre-defined class **location**.

4.7.13 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.

```
script = 'SCRIPT' id '.' [description] {step}
```

Every script has an identifier (the **id**) to identify it. A script is selected by the **Use** statement. When an actor is started following a script, it will

continue until it reaches the end or another **Use** statement is executed for that actor.

The optional description allowed in the beginning of a script is used instead of the general description (from the instance 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.

```
step = 'STEP' {statement}
     | 'STEP' 'AFTER' expression {statement}
     | 'STEP' 'WAIT' 'UNTIL' expression {statement}
```

For example

```
Step Wait Until hero Here
   Locate waiter Here.
   "From the shadows a waiter emerges: $p
      '-Bonjour, monsieur', he says."
```

```
Step After ticksLeft Of train
   "The train driver enters the train, and after a brief
    moment the train starts to move."
```

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 not act, but still present at the location where it was). If this is not what you wanted, you can end each script with a new **Use** statement.

4.8 Additions

In certain circumstances, you need to add properties to a class after it is defined. One simple such example is to add attributes to the predefined classes. To allow this the **Add** construct is available. It follows the grammar

```
addition = 'ADD' 'TO' 'EVERY' id
             [inheritance]
             {property}
           'END' 'ADD' ['TO'] [id] '.'
```

Using this construct, you can add any property to a class without having access to its declaration. A standard library would make heavy use of this since it would be structured so that related verbs, their syntax and synonyms are packaged together. If such a package required particular attributes in classes, they could be added using the **Add** construct.

4.9 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 triggered by the player input are declared using the **Verb** construct (see Verbs on page 103).

```
syntaxes = 'SYNTAX' {syntax}

syntax = id '=' {element} syntax_end

element = id
        | '(' id ')' [indicator]

syntax_end = parameter_restrictions
           | '.'
```

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

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

When the player types a command, it is compared to the set of declared syntaxes. This provides a very flexible way to extend the allowed command set (see also Player Input on page 159 for details on general player input).

After the player input has been matched to an allowed syntax, the parameters are bound to the instances referred to by the player. The parameter identifiers in the syntax declaration then refer to those entities. Reference to attributes etc. will be done in the instance referred by the parameter.

```
Syntax open = open (obj).
```

```
:
   If obj Is open Then …
:
```

In the example above, the parameter, **obj**, can be used in the declaration of the **open** verb and will, at execution time, refer to such a bound instance.

It is allowed to define multiple syntaxes for the same identifier (verb). See section Syntax Synonyms on page 101.

4.9.1 Indicators

Following a parameter, indicators are allowed in syntax declarations.

```
indicator = '*' | '!'
```

The indicators have the following interpretations:

'*' This parameter can reference multiple instances (for example by the player using **all** or concatenating a number of parameters using a conjunction like **and**, see Player Input on page 159).

'!' The parameter (the instance the player refers to in this position in the syntax) need not be present at the current location. The default case is that the Alan interpreter requires that a referenced instance must be present at the same location as the hero (if the parameter inherits from **thing**. Note that **entities** are always accessible). For cases when the player must 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 allows multiple objects. This would make the following inputs possible

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

Refer to *Player Input* on page 159 for details on the input of references to multiple parameters (such as objects). The above declarations would force the interpreter to reject player input like

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

This is because the syntax for the verb **drop** does not allow multiple references by not including the multiple-indicator. Another example using the '!' indicator:

```
Syntax
   talk_about = 'talk' 'to' (act) 'about' (subj)!.
   find = 'find' (obj)!.
```

Even if the robber or the key is not present, it will allow the player to say

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

For more information on player inputs, refer to Player Input on page 159.

Indicators given in one syntax declaration can affect other syntaxes if they have identical beginnings, like

```
> put everything on
```

and

```
> put everything on the table
```

Even if only one of the syntax declarations indicate that the first parameter should allow multiple instances, both syntaxes will actually allow this because they have the same syntax part before the parameter, in this case the verb "put".

4.9.2 Parameter 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.

```
parameter_restrictions = 'WHERE' restriction
                              {'AND' restriction}

restriction = id 'ISA' restriction_class
              'ELSE' {statement}

restriction_class = id
                  | 'CONTAINER'
```

Note: Any predefined or user defined class can be used. Particularly note that integer and string are pre-defined classes (see The Pre-defined Classes on page 37).

The following example describes the syntax for a verb that 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 instances of particular classes or instances with the container property, or numeric or string literals. The statements following the **Else** will be executed if that restriction is not met, i.e. if the player refers to an instance not in the specified class or classes. The default restriction is **Object**, i.e. if no

class restriction is supplied for that parameter identifier the player may only refer to objects at that position in his input.

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?!?!"
      And sub Isa subject
         Else
            Say act. "does not know anything about
                        that."
...
```

You can combine multiple restrictions, even for the same parameter. If they refer to the same parameter, they must be successively more restricted.

For example:

```
Where obj Isa object Else …
   And obj Isa openable_object Else …
   And obj Isa door Else …
```

References to attributes in the source are only allowed of it can be guaranteed that they exist during run-time. The class restrictions placed on a parameter are used by the compiler to make this guarantee for code executed by player input (verb bodies). The same applies for other semantic restrictions; you can only use a parameter in a **`List`** statement if it has been restricted to a having the container property.

You can use **`Isa Container`** to restrict instances to only those entities that are containers (have the container property).

If there is no restriction for a parameter, it is restricted to the class **`object`**.

4.9.3 Syntax Synonyms

It is possible to create multiple syntax declarations for the same verb. The semantics of this is that any of the input formats will be accepted and trigger the same verb action. This is a way to define syntactical synonyms, which are useful to allow multiple forms of input for the same action, increasing chances that the player will find the correct form. For example:

```
Syntax give = give (o) to (p) …
Syntax give = give (p) (o) …
```

The syntaxes must be compatible in the sense that the parameters must be named the same. However, the order of the parameters may differ, they will automatically be mapped as appropriate.

Restrictions are only allowed in the first of such syntax declarations. These restrictions will be applied regardless of which syntax was used.

4.9.4 Default Syntax

If no **Syntax** is defined for a **Verb** at all, this is handled with one of two default syntaxes according to the two templates below:

```
Syntax <1> = <1>.
Syntax <1> = <1> (<2>).
```

The placeholders represents 1) the name of the verb, and 2) the class in which the verb is first encountered.

The first template is used for verbs that are declared globally, i.e. outside of any class or instance. Since these are only applied when no parameters are used, this will effectively work for simple 'verb-only' **Verb**s, such as **quit**, **look**, **save** etc.

Verbs, for which there is no syntax, declared in an instance or a class, by default receives a common verb/object type of **Verb** syntax. They get a syntax corresponding to the second template above, which have a

reasonable syntax and may only refer to instances of the class where the verb was declared. It also implies that the default name for the single parameter is the same as the name of that class, e.g. **object**, **actor**, **thing**, etc. (See WHAT Specifications on page 140 for the implications of this.)

Note:A verb which is declared in a number of classes, or instances of various heritage, can not be handled with the default rules, since that would imply that the parameter should be restricted to multiple classes at the same time. This case must be handled explicitly.

Note:A verb with no declared syntax, which is declared in a location, will receive a default syntax restricting the parameter to the class **location**, which probably is not what you wanted.

4.9.5 Scope

If the player inputs a command following a syntax, which requires parameters, the interpreter first determines if the referenced instance is in scope. This is performed even before the restrictions are executed.

There are a number of ways to get an instance into scope:

- Instances of **entity**, and of any user defined subclasses thereof, are always in scope.
- An instance of **thing** and its subclasses at the current location is in scope.
- An instance of any class in a container in scope is in scope, unless that container is opaque and closed. See Container Properties on page 86 for details.

- If the syntax indicated a parameter as omni-potent, any instance is in scope for that parameter position.

If the interpreter finds multiple instances matching the input (the set of given adjectives and noun), it will try to disambiguate with preference to instances present, i.e. at the location of the hero. If there still are multiple candidates after this, the interpreter will give output a message.

Only when all parameter positions in the syntax have been resolved in this way, are the restrictions executed in the order given in the source code.

4.10 Verbs

Verbs can be declared both in instances and classes, and thus be inherited, as well as on a global level.

```
verb = 'VERB' id {',' id}
          verb_body
       'END' 'VERB' [id] '.'

verb_body = simple_verb_body
          | {verb_alternative}

simple_verb_body = [check] [does]
```

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

```
Verb take, get
   ...
End Verb take.
```

Verbs can be declared at two different levels, global (outside any other declaration) or inside a declaration of a class or instance. A global

declaration will only be considered when the verb is not applied to any instance (i.e. such as the player referring to an object). In fact, a global verb cannot have a syntax declaration with parameters.

A verb declaration inside a class definition or an instance will be considered if that instance (or an instance inheriting from that class) is used as a parameter in the input.

The identifiers in the list ('take' and 'get' in the first example) will be player words that by default can be used to invoke the verb. But if a **Syntax** is declared for the **Verb** (see Syntax Definitions on page 96), 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 there is more than one identifier 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 they are declared in the same verb declaration on one level in an inheritance tree, they can still have different bodies on another level.

4·10·1 Verbs in Locations

A special case is a verb declared in, or inherited by, the location where the player currently is located. If this verb is used, any checks or body of that verb will be considered before the verbs in the parameters. An example might be a location representing walking on a high wire. Anything dropped at this location will disappear:

```
The high_wire Isa location
   Verb drop
      Does Only
         Locate o At limbo. -- Instead of here.
   End Verb.
End The.
```

4.10.2 Verb Checks

```
check = unconditional_check
      | check_list

unconditional_check = 'CHECK' {statement}

check_list = 'CHECK' expression 'ELSE' {statement}
             {'AND' expression 'ELSE' {statement}
```

To determine if the action is possible to carry out, the **Checks** are executed. Which checks to run, is determined by the class of the instances bound by the parameters to the verb. All checks in the inheritance tree are tried by starting at the base class. In this way, the most general checks are tried first, then more specific.

A typical use of a check is to verify if the parameter has a particular property:

```
Verb take
   Check obj Is moveable
      Else "You can't take that."
   ...
End Verb take.
```

If no expression is specified for a check, that 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.

```
The jumpless Isa Location
   Verb jump
      Check "You can't do that here."
   End Verb jump.
End The jumpless.
```

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 159 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 159 for an explanation of the player view of this.

4·10·3 Does-clause

```
does = [qualifier] {statement}

qualifier = 'BEFORE'
          | 'AFTER'
          | 'ONLY'
```

If all checks succeed, the execution of the verb will be carried out. Multiple verb bodies may be involved. The order is by default to first execute the body of any verb declaration for the current location (including verb bodies inherited by it). Each parameter is then examined to find any declarations of that verb for the instance (including inherited verb bodies). These verb bodies are then executed in the order in which the parameters occurred in the syntax declaration, for each parameter starting with the body in the most basic class. By default, all of the involved verb bodies are executed. This is the most natural order and covers most cases.

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 Verb Qualification below for details).

A simple verb example:

```
Verb take
   Check obj Not In inventory
      Else "You already have that."
   Does
      Locate obj In inventory.
End Verb take.
```

4.10.4 Verb Alternatives

```
verb_alternatives = 'WHEN' id simple_verb_body
```

When a **Verb** is declared within an instance declaration, verb alternatives are allowed. These alternatives are used in conjunction with the **Syntax** declaration defined for the verb and allows differentiating between the instances 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 instance or receives the value of a literal, depending on the specified syntax. To determine the 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, assume the following syntax definition

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

If used with the **delicate_vase** actions could differ if it occurs as the direct object (o), or if it occurs as the indirect object (w). To

implement this the **Verb** body for **break_with** should also differ. For each parameter in the syntax, you may define different actions by supplying a verb alternative for each parameter identifier. The verb declaration could look like

```
The feather Isa object
   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 The feather.
```

If no alternative is explicitly specified the first parameter is assumed. I.e. that verb body will only be considered if the instance occurred as the first parameter.

4·10·5 Verb Qualification

```
qualifier = 'BEFORE'
          | 'AFTER'
          | 'ONLY'
```

The order in which the different verb bodies are executed is normally from the most general to the most specific. But, to allow for local differences, i.e. special handling of the verb at this location, a any possible definition of this verb in the current location (included inherited verb bodies) are considered first. Then, the verb bodies in the parameters (in the order they appeared in the syntax definition) on which the verb was applied are examined to find and execute their verb definitions. For each parameter, its most general definition is executed first, verb bodies down the inheritance tree next, ending with any verb body declared in the specific instance bound to that parameter.

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.

4·10·6 Verb Execution

First all parameters are evaluated according to the syntax restrictions (see Parameter Restrictions on page 99). Then, if they passed, the checks of all verb declarations are evaluated (see Verb Checks on page 105). Finally the verb bodies are executed.

Normal order of execution

	Outer Region	…	Current Location	First parameter	…	Last parameter
Base class	Outermost	↓	↓	↓	↓	↓
:	↓	↓	↓	↓	↓	↓
Leaf class	↓	↓	↓	↓	↓	↓
Instance	↓	↓	↓	↓	↓	Innermost

The table above illustrates the normal order of execution of verb bodies and checks. Starting with any base classes to the outermost region (containing location), continuing to the actual instance of that location, as illustrated by the first column. It then continues with any inner regions (second column) and the current location itself (third column). The execution then proceeds to the parameters of the syntax in order (columns four through six), traversing the inheritance tree from the base class to the instance.

Note:If you add a verb to the class **entity**, it will be inherited by all instances, including locations and objects. This will result in the execution of that verb body multiple times, since it will be in every cell in the table above.

Controlling Execution with Qualifiers

There are cases where you don't want all the bodies to be executed, or there is a special need to execute them in a different order. The most common case is to prohibit other bodies to be executed, e.g. a verb body in a location might want to stop the player from throwing any object. This verb body must then ensure that it is the only verb bodies to be executed. This can be done using the **Only** qualifier (see *4.10.5 Verb Qualification*).

Qualifiers control the order of execution of verb bodies. How does this work?

First, starting at the "innermost" according to the table above, the verb in the last parameter (if any) is investigated and, if any of its (inherited) verb bodies have 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 contains (or inherits) a verb definition with a **Before** or **Only** qualifier, that definition is now executed (and if the qualifier was **Only**, execution is aborted). Since locations can be nested, the surrounding locations are then examined in the same way.

As a result of this behaviour, a **Before** qualifier in the verb definition in an object parameter will supersede an **Only** qualifier in the location.

At this stage, all **Before** and **Only** qualifiers are handled appropriately. This only leaves the definitions without any qualifier or with the **After** qualifier. The outermost verb body (as indicated in the table above) 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 if they where declared with the **After** qualifier.

So in short (with base class definitions of the outermost location being the outermost and the instance 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 is equivalent to 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 all this complex behaviour will be needed in practice.

Note:All checks for a **Verb** will always be run in the normal 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:

```
The bomb Isa object
   Verb take
      Does Only
         "Your curious fingering at the intricate
            mechanism sets it of. BOOOM!"
         Quit.
   End Verb examine.
End The bomb.
```

This also illustrates the fact that the most commonly used qualifier is the **Only** qualifier since it is used whenever all other behaviour is replaced by some special behaviour.

4.11 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, since no verb is executing, or the **Current Actor**. See Run-time Contexts on page 161.

Events may be scheduled and cancelled with the **Schedule** and **Cancel** statements (see Event Statements on page 130).

4.12 Rules

A rule is an arbitrary expression, which, when true, results in execution of the some given statements. Rules are declared on the global level only.

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 Then
   Use Script hunting For monster.
```

The statements that are to be executed cannot refer to parameters, but may refer to **`Current Actor`**.

```
rule = 'WHEN' expression 'THEN'
       {statement}
       ['END' 'WHEN' '.' ]
```

All rules are tested after each actor (including the player) has made his move and after each event that is executed. Rules must be designed so that they can be executed multiple times for each player turn. Rules are executed at the location where the last activity (actor move or event) was performed (see also A Turn of Events on page 158). This is important to consider especially concerning use of **`Where`** expressions (see page 139) in rules.

4.13 Synonyms

A synonym declaration declare words that, when used in player input, are interchangeable at all times.

```
synonyms = 'SYNONYMS' {synonym_declaration}

synonym_declaration = word {',' word} '=' word '.'
```

For example

```
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 instance name (a noun or adjective), a direction or a verb. The list of words on the left-hand side contains new words (*not* defined elsewhere) that always will be interpreted as being replaced by the word on the right in the player input.

Synonyms are player words that can be interchanged. Defining synonyms for verb names will not always give you the result that you expect. The following example is incorrect.

```
Synonyms
   'examine' = look_at.
Syntax
   look_at = 'look' 'at' (obj).
Verb look_at ...
```

This will result in an error message indicating that the synonym word **look_at** is not defined. This is because the **Syntax** (see section 4.8) 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 159).

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 91102 for details on verb declarations.

It is also possible to define multiple names for an instance to achieve other effects similar to synonyms. See Names on page 70 for a description of this.

4.14 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 8.5, Input Response Messages).

To make the user dialogue more adapted to the settings you select, Alan allows you to define your own version of these messages. The grammar for this is

```
messages = 'MESSAGE' {message}

message = id ':' {statement}
```

An example is:

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

If the above where 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 RUN-TIME MESSAGES on page 208.

Message parameters

Message sections must be declared at the global level, but to make it possible to create high-quality messages the message sections have parameters available. Which parameters are available vary depending on the message, the details for each message is available in appendix Input Response Messages on page 210.

The parameters can be used in the same way as in verb bodies. The names of the parameters are "parameter1", "parameter2", etc. The type of the parameters will also vary.

For some messages, a parameter is an instance. In these cases, the instance is always of the pre-defined **entity** class. Any attribute available for this class will be available in message sections with instance parameters.

Note:If the message must be modified according to the case of the noun, which is the case with adjectives and negative forms in many languages, an attribute available on all instances can be used to select the correct form.

4·15 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_section = 'START' where '.' {statement}
```

An example would be

```
Start At outside_house.
   Schedule bird_chirp After 5.
```

Only the '**At what**' form of the **Where** construct (see WHERE Specifications on page 139) is allowed in the **Start** section. Any statements are allowed in the start section but they cannot refer to any parameters.

The start section must be the last declaration in an Alan source.

4·16 Statements

4·16·1 Output Statements

There are a couple of ways to present output to the player, string output, descriptions, printing expressions, listing container content and showing pictures.

The interpreter intersperses your output with spaces whenever needed. This might for example occur between two output strings:

```
"There is a door into the kitchen."
If kitchenDoor Is open Then
    "It is open."
End If.
```

If handled simple-mindedly the two texts would be adjoined. Alan realizes that a space is required between them. This space is automatically inserted by the interpreter during game play. This is also the case if the output from a **Say** statement is followed by an output string.

```
"Your wristwatch shows" Say hours Of watch.
". Time to go."
```

However, as in this example, this is not always the intended output. Particularly, if the **Say** statement terminated the previous sentence, as in the example, we want the full stop to be placed immediately after the output. So instead, the Alan interpreter will leave out the space between two outputs if the second starts with a period (full stop) followed by a space, or is the single character in the string. This special handling also applies to strings starting with a comma.

String Statement

```
output_statement = STRING
```

The simplest case of output is 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 restriction will relieve the author from the burden of constantly considering what the player will see. It can be used in the following way:

```
"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 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 adaptation to various viewpoints on the events without the need for any special tests.

Some character combinations have special meaning for the printout:

```
$p        New paragraph (usually 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
$<n>      The parameter <n> (<n> is a digit > 0, e.g. "$1")
$+<n>     Definite form of parameter <n>
$0<n>     Indefinite form of parameter <n>
$-<n>     Negative form of parameter <n>
$!<n>     Pronoun for the parameter <n>
$l        The name of the current location
$v        The verb the player used (the first word)
$o        The current object (first parameter)
```

Note: The **$<n>** formats must be used with care as they are not checked at compile time, e.g. you can use "$+1" in a context where no parameter is defined which would lead to a run-time error. To avoid the risk of any run-time problems use the **Say** statement with the parameter name wherever possible.

Note: The use of **$o** is deprecated. The **<n>** variants are better, but the recommended use is to refer to the parameters using their parameter names in a **Say** statement instead. This will ensure full reference analysis by the compiler protecting against any runtime error.

Note: Printing of instances using the **$**-forms will use the words input by the player if possible. This is in contrast to the **Say** statement, which will always use the mentioned clause or the first defined full name of the instance.

Style Statement

```
style_statement = 'STYLE' style '.'

style = 'NORMAL'
      | 'EMPHASIZED'
      | 'PREFORMATTED'
      | 'ALERT'
      | 'QUOTE'
```

The style of the text output can be controlled using the **Style** statement. With the exception of the **Emphasized** style, the styles are intended to be applied to whole paragraphs. The style indicated in the statement applies until another **Style** statement is executed.

Note: The exact visual appearance of the styles is implementation dependent. In fact, there is no guarantee that the style will actually differ.

Describe Statement

```
output_statement = 'DESCRIBE' what '.'
```

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

```
"There is a coin here."
```

is used instead. In this case, if the instance has the container property, a **List** statement is also executed for that object automatically (see below).

If a **Describe** statement is executed for another instance during the execution of the description clause, the system will recognise this and make sure that the second instance is not described more than once. This makes it possible to use instance 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.
"To the west is an open door, and to the east you can see the
   staircase."
```

Say Statement

```
output_statement = 'SAY' [form] expression '.'

form = 'THE' | 'AN' | 'IT' | 'NO'
```

The **Say** statement will output a short description of what is referred to by the expression. If it refers to an instance, it will print the name of it or execute its **Mentioned** clause if one is available. If it refers to an attribute, it will print its value, such as an integer or a string. Parameter names are also allowed in the **Say** statement, which, of course will result in a short description of the instance 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.
```

If the **what** part refers to an instance, the optional **form** may be used to control in which form the instance will be output.

If **'THE'** is used the form used will be the definite form, usually the short form preceded by a definite article. Correspondingly, the use of

'AN' indicates an indefinite form. A third form, using **'IT'**, is available. It indicates that the negative form as defined by the negative article or form should be output. Refer to Articles and Forms on page 83 for a description of the definite/indefinite articles and forms. Finally, the **'IT'** form will print the pronoun associated with the instance.

List Statement

```
output_statement = 'LIST' expression '.'
```

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.
```

Of course, the instance being listed must be an instance that has the container property, which may be inherited. This instance can be referred to by being bound to a parameter or a reference attribute for example.

4·16·2 Multi-media Statements

Alan has some multimedia provisions, although they may not be available on every platform and implementation. The **Show** statement, presents an image in the output window, and the **Play** statement plays a sound.

The Alan compiler will always support the multi-media statements, but a particular interpreter might not do so. Most GLK-based interpreters will support it but others might also. The game will still play fine, but the multi-media resources will silently be ignored. There is also no way to check for this in your source code. So, don't rely on them for your story,

particularly do not give the player necessary information only through pictures.

Image and sound files are analyzed by the compiler and copied into an Alan v3 resource file (file extension **.a3r**) that must be distributed with your game file, otherwise they will not be available during game play. The original file will be left untouched.

The format of the resource file follows the standard Interactive Fiction resource file format "blorb" and supports images of JPEG and PNG types, and sounds of MOD and AIFF formats.

If a resource file is referenced from multiple statements, it will only be copied once. The Alan compiler uses the file extension to determine the media type of the file. The following extensions are recognized: **.jpg .jpeg .png .mod .aif** and **.aiff**.

Show Statement

```
output_statement = 'SHOW' id '.'
```

The **id** should be the name of an image file. Since filenames may contain various special characters, a quoted identifier (see *File* on page 157) is usually required.

Alan currently supports the PNG and JPEG formats only.

Play Statement

```
output_statement = 'PLAY' id '.'
```

The **id** should be the name of a sound file. Since filenames may contain various special characters, a quoted identifier (see *File* on page 157) is usually required.

Alan currently supports the MOD and AIFF formats only.

Note:

4·16·3 Special Statements

Quit Statement

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

Look Statement

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 Statements

Save saves the game on a file for later use with **Restore**. Both **save** and **restore** asks the player for a file name to use for storing and restoring. This allows the player to use unlimited number of save files.

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

```
Verb restore
   Does
      Restore.
      Look.
End Verb restore.
```

Score Statement

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

```
score_statement = 'SCORE' integer '.'
```

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. Negative scores are not allowed and once a score is awarded it cannot be revoked, neither will it be awarded twice. 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 hero) and suitable manipulation and test statements.

Visits Statement

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

```
visits_statement = 'VISITS' integer '.'
```

The value of the argument (**integer**) controls the number of visits to a particular location between full descriptions. The initial setting of 0 (zero) 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 (which is also the initial setting).

Note: The familiar **verbose**, **brief** etc. commands can be imitated using different values in the **Visits** statement.

4.16.4 Manipulation Statements

Locate Statement

```
locate_statement = 'LOCATE' what where '.'
```

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 139. 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 the player typing a direction, i.e. the hero will be 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.

A third case is locating a location at another location. Locations can in this way be nested, resulting in an outer location working as a region or surrounding for the inner location. The effect of this is that any instances present in the outer location are reachable from the inner.

Empty Statement

```
empty_statement = 'EMPTY' what [where] '.'
```

The **Empty** statement locates all instances currently located inside the given container (instance with the **Container** property) at a certain location. The meaning of the **where** part, is the same as in the **Locate** statement. If it is not specified the instances will be placed at the current location.

```
Empty inventory Here.
"You seem to have lost most of your possessions. Well,
   you can't have everything."
Locate hero At restart_point.
```

Strip Statement

```
strip_statement = 'STRIP' [direction] [count] [size]
                    from_clause [into_clause] '.'

direction = 'FIRST' | 'LAST'

count = expression

size = 'WORDS' | 'CHARACTERS'

from_clause = 'FROM' expression

into_clause = 'INTO' expression
```

The **Strip** statement is used to manipulate the contents of strings content. You can use it to remove words or characters from a string, starting from the beginning or the end. The words or characters that are removed may be placed in an attribute as specified by the optional **into clause**. If the statement is used to manipulate words, blanks and separators are used to separate the words. In this case, any resulting string is also free of leading and trailing blanks.

A short example

```
The eliza Isa actor
   Has topic "".

   Verb talk_to
      Does
         Set topic Of eliza To "sailing music cooking reading".
         Strip Random 0 To 2 Words From topic Of eliza.
         Strip First Word From topic Into topic.
         "And how do you feel about" Say topic Of eliza.
   End Verb.

End The eliza.
```

4·16·5 Event Statements

Schedule Statement

Schedule will queue an event to occur at a specified location after the number of player turns specified by the expression.

```
event_statement = 'SCHEDULE' what [where]
                  'AFTER' expression '.'
```

For example

```
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, probably last, though). 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.

Specifying the location (**where**) as **At id**, where the identifier represents an instance not inheriting from **location**, means that wherever that instance is when the event occurs, the event will be executed at that place. The event will 'follow' the instance.

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

Note: The event can be specified by referring to an attribute of Event type.

Cancel Statement

```
cancel_statement = 'CANCEL' what '.'
```

Cancel will remove the event referenced from the queue of scheduled events. It is not an error to remove an Event, which is not currently scheduled.

```
Event ticking
   "Tick..."
   If timer Of bomb = 0 Then
      Schedule 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.
```

The event can be referenced using any expression of Event type, e.g. an attribute.

4·16·6 Assignment Statements

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

Make Statement

```
make_statement = 'MAKE' what something '.'

something = ['NOT'] id
```

The **Make** statement is used to set or reset Boolean attributes.

```
Make door open.
Make door Not open.
```

Increase and Decrease Statements

```
increase_statement = 'INCREASE' what [by] '.'

decrease_statement = 'DECREASE' what [by] '.'

by = 'BY' expression
```

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 one.

```
Increase level Of bottle By contents Of mug.
Decrease lives Of hero.
```

Set Statement

```
set_statement = 'SET' what 'TO' expression '.'
```

The **Set** statement is used when assigning values to numeric, string, reference of set valued attributes.

```
Set mood Of king_tut To 3.
Set hour Of clock To hour Of clock + 1.
```

Setting attributes of reference or set type requires that the expression follow the type and subclass compatibility rules. For example, you can only assign

- integer type expressions to an integer attribute

```
Set intAttr To 4. -- Correct
Set intAttr To "hi". -- Incorrect
```

- an expression that refers to an instance if the attribute being assigned to is a reference attribute which has a class of which the class of the expression is a subclass

```
Has suspect butler.
Set suspect Of detective To someLocation. -- Incorrect
```

- a set valued expression to a set type attribute if all members are instances of some subclass of the member class of the target attribute

```
Has friends {monica, ross, chandler, rachel, phoebe, joey}.
Set friends Of mine To {book}. -- Incorrect
Set friends Of mine To {}. -- Correct, empty set is OK
Set friends Of mine To {suspect Of detective}. -- Correct maybe
```

Include Statement

```
include_statement = 'INCLUDE' expression 'IN' set '.'
```

The **Include** statement is used to include a new member in a Set. Typically, this is used to an instance or value to a collection of such. See section Set TypeEvent Type on page 58 for an explanation of the Set type. A member already in the Set will silently be accepted but not generate duplicate entries.

The set may be identified using an expression involving reference attributes:

```
Include hitchhiker In friends Of driver Of car.
```

And vice versa:

```
Include driver Of car In friends Of hitchhiker.
```

Exclude Statement

```
exclude_statement = 'EXCLUDE' expression 'FROM' set
'.'
```

The **Exclude** statement is the reverse of the **Include** statement. It removes a member from a Set. An attempt to remove something not included in the Set will be silently ignored, so that after the execution of the statement it is guaranteed that the member is not in the Set.

Note: The inclusion or exclusion of an instance will not affect its location. A member may be included in multiple Sets.

4.16.7 Conditional Statements

In Alan there are two conditional statements, the common **If** statement and the **Depending On** statement.

If Statement

```
if statement =  'IF' expression 'THEN' statements
                { elsif_part }
                [ else_part ]
              'END' 'IF'

elsif_part = 'ELSIF' expression 'THEN' statements

else_part = 'ELSE' expression 'THEN' statements
```

The **If** statement is essential for varying output and otherwise change the activities in the game. The expression is evaluated (see Expressions on page 142 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 Statement

```
depend_statement = 'DEPENDING' 'ON' expression
                       {case}
                   'END' 'DEPEND' '.'

case = right_hand_side 'THEN' statements
```

The **Depending On** statement is a provided to select one of a number of possible conditional cases depending on an expression. The expression can be any expression. The `right-hand side` is the right hand side of any valid expression. When combined with `expression` (as the left hand side of the expression) they will be a complete statement, which can be evaluated.

A simple example of the **Depending On** statement is:

```
Depending On weight Of obj
   = 1 Then "light as a feather"
   Between 2 And 10 Then "carryable"
   Between 10 And 20 Then "heavy"
   > 20 Then "immobile"
   Else "weightless"
End Depend.
```

The meaning of this example is to test the **weight Of obj** and select one of the cases depending on the value of it. If it is equal to one the first case will be executed. If none of the cases match, 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. At most, one case will be executed. In the example, a weight of ten will render as "carryable".

The tests are equivalent to

```
If weight Of object = 1 Then "light as a feather"
Elsif weight Of object Between 2 And 10 Then "carryable"
Elsif weight Of object Between 10 And 20 Then "heavy"
```

```
Elsif weight Of object > 20 Then "immobile"
Else "weightless"
End If.
```

A **Depending On** statement is preferable to a chain of **If** statements when the same expression will be tested for multiple matches.

4.16.8 Actor Statements

Use Statement

```
use_statement = 'USE' script ['FOR' actor] '.'
```

The **Use** statement starts execution of a given script for a given actor. The **For actor** clause is optional when writing code within a certain actor; in this case that the statement applies to the actor that the code is in.

```
Use Script playing For george.
```

Note: You can use an expression such as a simple identifier, a parameter reference or a reference attribute as the actor clause.

Stop Statement

```
stop_statement = 'STOP' actor '.'

actor = expression
```

The **Stop** statement stops an actor from proceeding with any script it may be executing. In effect, it will abort it and put the actor in an idle state. A simple case is the direct reference to an actor using the identifier

for it, but any expression such as a parameter reference or a reference attribute as the actor clause can be used.

4·16·9 Repetition Statements

The Alan language provides one compound statement for repetition, the **For Each** statement.

```
repetition_statement = 'FOR' 'EACH' id [filters] 'DO'
                           statements
                       'END' 'FOR' 'EACH' '.'

filters = filter { ',' filter }
        | 'BETWEEN' expression 'AND' expression
```

You can optionally leave out either **For** or **Each** but not both.

The identifier is called the loop variable and will have similar semantics as a syntax parameter. It will dynamically be bound to instances, one for each repetition. In the body of the loop, the statements, this variable can be referenced in the same way as a syntax parameter.

The optional filters can be used to restrict the values in the loop. If the **Between** form is used, the loop becomes an integer loop, resulting in the loop variable having integer type and range from the two expressions inclusive. Otherwise, the loop variable will be of instance type and will consecutively assume the value of each instance fulfilling the filters. See Filters on page 151 for an explanation of filters.

Any references to the loop variable within the repetition will refer to the instance bound, or integer value, in this repetition.

You can use any statements inside the repetition, e.g. to check for further conditions before operating on the instance. For example

```
For Each creature Isa actor Do
   If creature Here Then
      …
   End If.
End For.
```

4.17 WHERE Specifications

Many constructs in the Alan language require a specification of where the construct should operate. The general intention of such a **Where**-specification is to specify a location.

```
where = 'HERE'
      | 'NEARBY'
      | 'NEAR' what
      | 'AT' what
      | 'IN' what

The meaning of the different constructs is as follows
```

- **Here** is the location where the current activity is performed. Often this means where the hero is, but if the expression is evaluated in another run-time context this context is used. See Run-time Contexts on page 161 for a detailed discussion, but examples include an event scheduled at a particular location, in which case that location is **Here**. Other examples of contexts are activities performed by other actors and expressions within rules. Note that **Here** is equivalent to **At Current Location**.
- **Nearby** means at any adjacent location. An adjacent location means that there exists an exit from the other location to **Here** (note that the direction is from **Nearby** to **Here**). It is allowed to refer to any instance using an identifer or expression. In particular, instances inheriting from **location** are allowed, which can be used to see if a location is nearby.

- **Near what** has a similar meaning to **Nearby** except that it refers to some other instance (the what) and results in a true value if that other instance is at a location which is nearby (has an exit to) that of the location of the first instance. **AT what** means at the location of the instance referenced by the **what** specification (see WHAT Specifications 140). Note that an instance is always **At** itself, i.e. **x At x** is always true. This can come in as a surprise, especially if you try to aggregate or loop over instances. See Aggregates 150 and Repetition Statements 138.
- **IN what** must refer to a container and the expression refers to inside of that container.

These forms can be used in **Locate** statements and in some expressions for example. When used in their basic form in expressions they all look inside containers (and container in containers) to evaluate the expression. See The Whereabouts of an Entity on page 148 for more information about **Where** expressions.

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 it requires a specific location to locate to, and **Nearby** is not specific. Instead, **Nearby** could be used in an expression in an **If** statements to see if the monster is somewhere near.

4.18 WHAT Specifications

Constructs in the grammar for the Alan language often refer to some class or instance defined in the Alan source. This is generally called a **what** specification, as it specifies what the construct refers to. An example is the **Locate** statement that must refer to something that should be relocated.

```
what = 'CURRENT' 'ACTOR'
     | 'CURRENT' 'LOCATION'
     | 'THIS'
     | id
     | attribute_reference
```

The meaning of the different forms of the **what** specification are:

- **Current Actor** is always set to the actor currently active, e.g. when a non-player actor is running a script this refers to the actor instance that is running. It also applies to expressions and statements within rules as rules are run once for each actor.
- **Current Location** is the current location, i.e. the location where the current activity is performed. Normally this is the location where the hero is, but may also be where an event is executed or the location where a scripted actor currently is executing. See Run-time Contexts on page 161 for more details.**This** refers to the instance in which code, such as a verb body or a script, is run. This can for example be used to test or set attributes in inherited code, thus testing or setting attributes in the instance while the code is defined in a class that the instance inherits from. It cannot be used in events or global verbs.
- An identifier, **id**, refers to the class or instance with that name, a syntax parameter, script or loop variable with that name. A syntax parameter may have the same name as an class or instance declared elsewhere in the source in which case the parameter has precedence.
- A reference to an attribute, as described in section 4.19.3, might be used depending on its type and the context of the usage of the **what**-expression.

Note:Not all kinds of what specifications are meaningful in all contexts. For example it is not possible to use **Current Location** (nor an identifier referring to an instance inheriting from location) as the **what**-part of a **Locate** statement. (Since it is illogical to re-locate locations.)

4.19 Expressions

The grammar for Alan refers to expression. This is a generic name for a number of constructs yielding a value. The following sections describe the different kinds of expressions available in the Alan language.

4.19.1 Types of Expressions

Expressions are used 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 can handle all other types of values. See section Types on page 57 for details on types.

4.19.2 Literal Values

A single integer (e.g. 42) is a numeric expression. A string is an expression and represents a string value, e.g.

```
Set password Of terminal To "xyzzy".
```

A set type value can be constructed directly as an expression. This can be used in a **Set** statement or another expression. E.g.

```
Set suspectedWeapons Of detective To {gun, bat, axe}.
```

Each member in the set expression can be an expression of integer or reference type in itself.

4.19.3 Attribute References

```
attribute_reference   = id 'OF' expression
                      | expression ':' id
```

References to attributes can be used in any expression provided their type matches the semantics of the context. The type of the expression is the type of the attribute.

There are two formats available, of which the first resembles plain English.

```
Set password Of terminal To password Of manual.
```

The other format is more compact, which might be preferable when referring to chains of attributes of attributes. See Reference AttributesNote that string valued attributes are mainly intended for saving string parameters from the player input, like in on page 75 for an explanation on how this works.

```
Say detective:suspect:weapon.
```

You can test Boolean attributes of an instance by following the pattern

```
expression = expression 'IS' something
```

For example

```
If bottle Is empty Then …
```

The test can be reversed by adding a **Not**:

```
If hero Is Not hungry Then …
```

4·19·4 Random Values

There are three types of random expressions. The first is the traditional random integer expression.

expression = 'RANDOM' expression 'TO' expression

The random integer expression returns a numeric value that is randomly selected between and including the values of the two expressions. Arbitrary expressions yielding an integer value can be used as the boundary expressions.

```
Set eyes Of first_die To Random 1 To 6.
Decrease temp Of room By Random 0 To temp Of Room.
```

The second and third types return a random member in a set or in a container respectively.

```
expression = 'RANDOM' ['DIRECTLY'] 'IN' expression
```

If the expression refers to a container, the expression returns one of the instances currently in that container. The type of the entire expression is instances of the class accepted by the container. See Container Properties on page 86 for details on how to determine the class of instances allowed inside a container.

If the expression refers to a set, the result is one of the members in the set. The type and class of the entire expression is determined by the allowed members in the set. See Set Type Attributes on page 77.

The optional keyword **`Directly`** is only allowed if the expression refers to a container. The semantics is the same as for the **`Where`** expression, see The Whereabouts of an Entity 148.

Note:Attempting to apply a random selection from an empty set or container is one of the very few situations that could lead to a runtime error. It is the responsibility of the author to ensure that this is not attempted. You should always surround a random member expression with an **If** statement that ensures that the set or container is not empty to guard against such runtime errors. See Aggregates on page 150 for descriptions on how to count members in a set or container.

Note:A **thing** or **entity** inside a container, which normally do not exhibit themselves, will be candidates for being selected by a **Random In** statement, as any other instance.

4·19·5 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 …
```

4·19·6 Class Expressions

It is possible to check if an instance belongs to, or inherits from, a particular class. The resulting value is a Boolean type value.

```
If p Isa object Then …
If opponent Isa enemy Then …
```

4·19·7 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 is

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

For example

```
age Of golden_child + 4
```

The plus operator (+) may also be used on strings for concatenation. The meaning of such an expression is that the two strings are concatenated into a resulting string. For example

```
string1 + " " + anotherString
```

4·19·8 Relational and Equality Operators

Equality ('**=**', meaning equals) and relational operators ('**<**', '**>**', '**<=**', '**>=**', meaning: 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 equality 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).

Two values of instance type may be compared with the '**=**' and '**<>**' operators, and may e.g. be used to test if a parameter refers to a particular instance or is 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" Say c.
```

```
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. The range is inclusive, i.e. the values are included in the accepted range. For example

```
If level Of water Between 2 And capacity Of bottle Then …
```

4.19.9 String Containment

There is 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 example of an expression that is true is

```
"A string" Contains "a S"
```

An optional **Not** (before **Contains**) can be used to reverse the test.

```
"A string" Not Contains "a S"
```

The expression yields a Boolean value.

4.19.10 Current Entities

There are two particularly interesting entities that you might want to know something about or which they are. They are

- **Current Actor**
- **Current Location**

These two expressions can be used wherever a reference to an instance can be used. They will refer to the currently executing actor and the current

location respectively. Details about execution environments can be found in Running An Adventure on page 158.

4·19·11 This Instance

You can also refer to the instance that is actually executing the code containing the expression. This is particularly useful when using inheritance since the class defining the code have no way of knowing which instance will actually execute it. This expression is **`This`**.

An example is the code for objects that can be opened:

```
Every openable Isa object
  Is Not open.
  Verb open
    Check …
    Does
      Make This open.
  End Verb.
End Every openable.

The door Isa openable
End The door.
```

```
> open the door
```

Given these two declarations and some syntax declarations the door will inherit the **`open`** attribute. When the verb body, also inherited from **`openable`**, is executed, it will set *the doors* attribute, because this instance is running the code.

4·19·12 The Whereabouts of an Entity

The expression

```
expression = what ['NOT'] ['DIRECTLY'] where
```

can be used to test if a particular instance, 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 Not Nearby Then …
```

The forms available for the **Where** expression are described in detail in WHERE Specifications on page 139.

The normal behaviour of a **Where** expression is to evaluate recursively through containers, e.g. if the bottle was inside a bag which was in the inventory, the first expression above would still be true.

In addition, a qualifying keyword, **Directly**, can be used to indicate that the expression should *not* evaluate recursively into containers. To test if an instance is at a particular location and not in a container at that location you can use:

```
If key Directly At treasury Then …
```

The qualifying keyword **Directly** works in the same way with all **Where** expressions. Adding a **Directly** qualifier to the first example above would change the expression to only be true if the bottle was in the container and not inside any other container even if that container was in the inventory.

4·19·13 Aggregates

```
aggregate_expression = aggregate filters

aggregate = 'COUNT' | 'SUM' | 'MAX' | 'MIN'
```

Aggregates are functions to calculate values from sets of instances. There are four aggregates available, **Count**, **Sum**, **Min** and **Max**. Aggregates work by inspecting all instances available, applying the filters, which may remove some, or even all, from the set of instances, and then calculate the value from the remaining instances.

You can use filters to filter out instances belonging to a particular class, at a particular location or having a particular Boolean attribute. See Filters on page 151 for an explanation of filters.

Count counts the number of instances in the set, e.g.

```
"You are carrying"
Say Count Isa object, In inventory, Is big.
"big things."
```

In this example there are three filters applied, "Isa object", "In inventory" and "Is big". All of these filters must pass before an instance is counted. The result of that count is an integer, which is then printed using the **Say** statement.

The **Sum**, **Min** and **Max** aggregates return the sum, minimum and maximum value respectively, of an attribute of all instances in the filtered set.

Any attribute referred to either in the aggregation itself or in the filters, must be an attribute of some class in order to ensure that the attribute is available for all instances. You must ensure this by filtering out only instances of the relevant class, e.g. objects, using a class filter.

Some examples:

```
If Sum Of weight At bridge > 500 Then …
If Max Of size In inventory > size Of small_door Then …
If Count Isa lightsource, Is lit, Here > 0 Then
   "Let there be light…"
End If.
```

These examples could be used to create various restrictions in the possible travels of the hero.

4.20 Filters

```
filters = filter { ',' filter }

filter = 'ISA' class
       | is attribute
       | where
```

Filters can be used to filter out only particular instances to loop or aggregate over. If one of the filters is a **`Isa <class>`**, only instances of that class will be bound to the loop variable or considered in the aggregation. In particular this is required if any of the other filters refer to attributes, which is only allowed if the class is known and that class is guaranteed to have that attribute. Other ways to restrict the filtered instances is to use a **`Where`** filter which implicitly restricts to instances available at or in that location, container or set. See The Whereabouts of an Entity on page 148 for details on the various forms of the **`Where`** expression.

Multiple filters can be listed separated with a comma. Each filter must enumerate the set of values to a compatible set, e.g. using two **`'Isa'`** filters for actors and locations respectively is not allowed since those two sets can never be compatible.

5 LEXICAL DEFINITIONS

5·1 Comments

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

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

5·2 Words, Identifiers and Names

An identifier is a word in the Alan source, which is used as a reference to a construct, such as an instance. Identifiers may only be composed of letters, digits and underscores. The first character must be a letter.

```
identifier = letter {letter | digit | underscore}
```

There is also a second kind of identifier, namely the quoted identifier.

```
quoted_identifier = quote {any_character} quote

id = identifier
   | quoted_identifier
```

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

```
Verb 'look'
   Does
      Look.
End Verb 'look'.
```

Quoted identifiers retain their exact content. They may contain spaces and other special characters, which make them useful as long names for locations as in

```
The pluto Isa location Name 'At the Rim of Pluto Crater'
   Description
      ...
```

One single-quoted identifier is used as the whole name of the location to preserve editing and avoiding clashes with the reserved words **At** and **Of**. (This could also have been avoided by quoting just those words.)

Identifiers and words retain their capitalization. An example is

```
The eiffel_tower Name Eiffel tower …
```

The first word in the name will always be printed with a capital 'E'. However, when comparing the word to player input and other occurrences of the same word in the source, case will be ignored. This means that you cannot have two words or identifiers that differ only in case, they will be the same and stored in the game data as one of the occurrences, which one is implementation dependent.

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

Note:Any one of the occurrences of a word might define its capitalization, which one is unspecified. This might affect the output if you use capitalization for names of locations, such as "Name Shore of Great Sea". Such names can inadvertently make the game use "Great" for all "great" things in your game. You can avoid this by using a quoted identifier for the complete name of the location.

Be careful when using quoted identifiers, especially if the player is supposed to use the word. A player cannot input words containing spaces or other special characters or separators. The only exception being underscores and dashes. A player input word must start with a letter.

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

Some of the identifiers in the source for an Alan game 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 cannot enter them.

5·3 Numbers

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

```
number = digit {digit}
```

5.4 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 = double_quote {any_character} double_quote
```

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 117). You may therefore write your strings any way you like; they will always be neatly formatted on the player's screen. You can use special codes (see String Statement on page 118 for a list) to indicate (but not precisely control) the formatting.

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!""."").

5·5 Filenames

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 **import** statement.

```
import = 'import' quoted_identifier '.'
```

The import statement requires a filename, which must be given as a quoted identifier (see section 5.2).

6 RUNNING AN ADVENTURE

6.1 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. The player is then placed in the location indicated in the start section, the location described, and 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 and Scope on page 102). 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.

If the player inputs an empty command, this is equivalent to forfeiting his turn. The empty command will simply be ignored. The events and other actors, including turn counting, then proceeds as if the player had input a proper input, before returning to the player prompt.

6.2 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, there are various combinations and variations are possible using special characters and words. The words are of course different for different languages, but in the following generic English words, like "**AND**-word", will be used to denote all words that can be used in the same manner. The exact list of these words for every supported language is available in <section on words>.

- The following built in syntax variations are available to the player:Concatenating of statements using **AND**-words like
  > open the door then enter
- The use of pronouns to refer to the last object mentioned in the previous command, e.g.
  > take the book and read it

> give the key to the guard and ask him to open the door

The pronouns has to be defined by the author in his source (see Pronouns on page 73) or by a library. The only built in pronoun is the **IT**-word, which is automatically defined on the class **thing**.

- References to multiple objects using **AND**-word, this allows
  > take the blue vase and the pillow
- Reference to multiple objects using **ALL**-word
  > drop all
- Excluding objects using a **BUT**-word, like:
  > wear everything except the bowler hat
- The use of a **THEM**–word to refer to the multiple objects referenced in the previous command, e.g.
  > remove the hat and the scarf then drop them

The reference to multiple objects (or actors) in a position is only possible if the adventure author has allowed it by using a multiple indicator in the syntax definition (see Syntax Definitions on page 95). All the variations above are built in and handled automatically by the run-time system.

The interpreter also automatically restricts parameter references to things at the current location. I.e. the player can only refer to things present, in his input. The one single exception is if the syntax for the command uses the omnipotent '!' indicator, see Syntax Definitions on page 95 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 177.

The use of ALL results in the execution of the appropriate verb for all objects at the current location, *except* the ones that do not pass all checks for the verb (see Verbs on page 103 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 alphanumeric characters, underscores and dash. 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).

6.3 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 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. In addition, the **Current Actor** is set to the hero and **Current 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, **Current Actor** is set to the hero, and **Current Location** of course to the location being described. The description clauses for objects and locations, as well as the **Entered** clause of locations, are evaluated in this context. **Entered** clauses are executed for all actors entering a location with **Current Actor** set to the moving 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 **Current Actor** is set to the actor currently executing or that was executing immediately preceding the rules. Therefore, you could say that rules are run for each actor, and **Current**

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 execute.

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

6.4 Moving Actors

The main way to move the hero is through the exits (see Exits on page 92). They 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 printed. Otherwise, that exit is examined for **Checks**, which are run according to normal rules (see Verb Checks on page 105). If there was no **Check** 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 **Description**s, explicit or implicit).

When any actor (including the hero) is located at a location, the **Entered** clause of that location is executed as if the actor had moved into that **Location**. The actor that was moved will be the **Current Actor** even if the movement was not caused by him (but the result of an event, for example). Therefore, this is also the last step in the sequence of events caused by locating the hero somewhere.

6·5 Undoing

A player might occasionally regret a command that he gave, perhaps realising that it was not the correct one. The Alan interpreter supports such undoing of commands. This means that the player can backup commands that (s)he later regretted. The interpreter stores each game state as soon as it has changed and an **undo** command resets the game state to the last saved one. This works completely automatically and as many states as memory permits is saved, giving almost unlimited **undo** capability.

The player command to restore a previous game state is handled directly by the interpreter. It must consist of the single word **undo**.

6·6 Scripting and commenting

Most versions of the Alan interpreter, Arun, supports both taking a transcript of a game in progress and playing it back as input to the interpreter.

This is very convenient during development of a game where you can play through the game up to a point and start from there, or even automatically test your game.

To make Arun read input from a script file you can use the special command character '@', which should be followed by the name of the text file in which your commands are listed.

You can add comments to each line in a script file. The interpreter will not read beyond a semicolon, ';', so anything after it can be seen as a comment. Note that this also works for direct player input.

7 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.

7.1 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.

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?).

7.2 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
```

7.3 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 5.5 157) 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.

7.4 Doors

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

```
The treasury_door Isa object At hallway

   Name treasury door
   Is Not open.

   Verb open
      Does
         Make treasury_door open.
         Make hallway_door open.
   End Verb open.
End The treasury_door.


The hallway Isa location
   Exit east To treasury
      Check treasury_door Is open
         Else "The door to the treasury is closed."
   End Exit.
End The hallway.


The hallway_door Isa object At treasury
```

```
   Name hallway door
   Is Not open.

   Verb open
      Does
         Make treasury_door open.
         Make hallway_door open.
   End Verb open.
End The hallway_door.


The treasury Isa location
   Exit west TO hallway
      Check hallway_door Is open
         Else "The door to the hallway is closed."
   End Exit.
End The treasury.
```

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

7.5 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.

7.6 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.
...
```

7.7 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.

7.8 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.

7.9 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. However, 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. This does not apply to objects and actors that have the container property of course. See Container Properties on page 86 for a discussion.

So to have the hero's body parts, 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 body_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...

7.10 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 (zero) of the attribute **lightsource**. Any object that provides 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.
```

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.

7.11 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 95, 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
...
```

This would otherwise be impossible. If the mountain should be visible and possible to manipulate 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 located 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.
```

7.12 Using Events as Functions

<to be supplied>

7.13 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 import 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.

7.14 Debugging

Occasionally your Alan code is flawed and you really can't understand what is actually happening. To aid in discovering which part of your code is run when, the interpreter Arun incorporates some features for debugging. There are a few debugging switches available when starting the interpreter from the command line:

```
-c       Log the commands input by the player
-l       Log a complete transcript of the game
-t<n> Enable trace mode (<n> = level 1,2,3 or 4)
-d       Enable debug mode
```

Note:None of the above switches is effective unless the adventure was compiled with the debug option set (see Options on page 55).

Command Logs and Game Transcripts

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

A command log can on some systems be used as input to the interpreter, and thus automate the execution of the exact player experience.

You can only activate one of the logs in a single session.

Interpreter and Instruction Trace

Trace mode can also act as an aid in debugging. Level 1 will print information about every invocation of the instruction interpreter, making it easier to see which parts of the code are being executed.

Trace level 2, single instruction trace, will also trace every single Acode instruction. The Acode is based on a stack machine but single instruction trace will not show all stack operations. Level 3 shows the execution of these also. Level 4 dumps the content of the stack for every instruction.

Debug mode

Finally, and usually most useful, there is the debug mode. If the interpreter is started with this option, it will execute the start up sequence and then prompt for a debug command with

```
abug>
```

Using the Debugger

Abug may also be entered during the execution of an Adventure. To do this you issue the single player command (type it at the game prompt)

```
> debug
```

The game must have been compiled with the debug option.

Typing a question mark or an 'h' in response to the debug prompt will give a brief listing of the commands available in Abug:

```
a     Display a list of all instances that are actors.
l     Display a list of all instances that are locations.
o     Display a list of all instances that are objects.
c     Show the class hierarchy.
e     Display a list of all events and their status.
b     Set a breakpoint.
d     Delete a breakpoint.
n     Execute to the next source line.
g     Go. I.e. proceed by executing the current player
      turn. Abug will stop and prompt for a new command
      again before the player is next in turn.
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 display commands, **A**, **L**, **O**, **C** and **E**, 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.

You can run the adventure to the next source line by entering the **N** command. If the source file is available, the interpreter will also show the source line.

Breakpoints can be set on a source line. Enter the **B** command followed by the number of the source line. Alan allows the source to be separated into multiple files, so the interpreter always indicate which file the source line is in, e.g. when hitting a breakpoint or stepping to the next source line. When setting a breakpoint the current file is always assumed. You can currently not request a breakpoint to be set in another source file. However, usually this can be worked around by stepping until that source file is entered, and then set the breakpoint.

Breakpoints can be deleted. The **D** command without a line number will remove any breakpoint at the current line. You can specify which breakpoint to delete by giving the line number after the **D**.

Note: The interpreter does not currently know on which source lines it is possible to place a breakpoint. It will attempt to place one at the line specified. This will sometimes cause a breakpoint to fail and not being hit.

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

```
<Arun, Adventure Interpreter version 3.0.30 development (2004-
08-27 22:44:50)>
<Version of 'saviour' is 3.0(30)d>
<Hmm, this is a little-endian machine, fixing byte ordering....
OK.>
<Hi! This is Alan interactive fiction interpreter Arun,
development version
3.0.30!>

abug> n

abug> Stepping to line 1345 in 'saviour.alan':
```

```
abug>     "$pWelcome to the game of SAVIOUR!$pIn this game your
abug> n

[game output deleted for brevity]

abug> Stepping to line 1353 in 'saviour.alan':
abug>     "$n$nhttp://welcome.to/alan-if"
abug> b 1355
Breakpoint set at line 1355 in 'saviour.alan'.

abug> g

[game output deleted for brevity]

abug> Breakpoint hit at line 1355 in 'saviour.alan':
abug>     Visits 2.
abug> n

abug> Stepping to line 318 in 'saviour.alan':
abug>         "To the north is a tall ancient building with a
large
abug> n
To the north is a tall ancient building with a large entrance.
On the top there is a clock tower. Most of the windows in the
building are broken, and a sign with three oval objects are
hanging lose from the wall.

> north
abug> Stepping to line 325 in 'saviour.alan':
abug>             Score 5.
abug> t
Trace on.
abug> n

<EXIT north(1) from Outside The Tall Building(3), Moving:>

Hall

abug> Stepping to line 332 in 'saviour.alan':
abug>         "Inside the entrance is a hallway full of dust
and
abug> i
Instances:
      1: pseudowords (container)
      2: nowhere
      3: Outside The Tall Building
      4: Hall
      5: door
      6: Stairs
      7: cellar
      8: rats
      9: store
     10: spool of computer tape
     11: First Floor
     12: old book
```

```
     13: office
abug> i 12
The old book (12) Isa object
    Location: First Floor (11)
    Attributes:
        Takeable(2) = 1
        Readable(3) = 1
        openable(4) = 0
        startable(5) = 0
        examinable(6) = 1
abug> g
Inside the entrance is a hallway full of dust and pieces of the
ceiling has fallen to the floor. At the west end is a
staircase, and to the south is the exit. To the east is a
folding door. It is closed.

> west

<EXIT west(3) from Hall(4), Moving:>

Stairs
You are at the landing of an old staircase. It seem steady
enough to walk in, but be careful if you are going to use it.
There is a passage leading up, and another leading down into a
dark cellar. To the east is the hallway. A strange smell
emerges from below.

> up

<EXIT up(5) from Stairs(6), Moving:>

First Floor
The landing on the first floor is as dirty as all the others.
Meters and meters of old cables are laying around, leading into
a room to the east. The stairs leads up and down. They still
seem alright. Through the dirty windows the barren field
outside the building can be seen. Almost completely covered by
dust, there is an old book laying on the floor here.

> take book and read it

<VERB 21, in parameter #1, inherited from class object(4),
CHECK:>

<VERB 21, in parameter #1, inherited from class object(4),
DOES:>
Taken.

<VERB 5, in parameter #1, inherited from class object(4),
CHECK:>

<VERB 5, in parameter #1, DOES:>
As you carefully try to open the book it falls apart into dust
and falls to the floor through your fingers.
```

```
> debug
abug> i 12
The old book (12) Isa object
    Location: nowhere (2)
    Attributes:
        Takeable(2) = 1
        Readable(3) = 1
        openable(4) = 0
        startable(5) = 0
        examinable(6) = 1
abug> s
Step on.
abug> n

> north

++++++++++++++++++++++++++++++++++++++++++++++++++++
abug> Stepping to line 325 in 'saviour.alan':
abug>             Score 5.
abug> g

 97e: SCORE         1           =5
 97f: RETURN
----------------------------------------------------



++++++++++++++++++++++++++++++++++++++++++++++++++++
 99a: PRINT      1396,     4            "Hall"

 99b: RETURN
----------------------------------------------------


++++++++++++++++++++++++++++++++++++++++++++++++++++
 98e: PRINT      1311,   158            "Inside the entrance is
a hallway full of dust and pieces of the ceiling has fallen to
the floor. At the west end is a staircase, and to the south is
the exit."

 993: DESCRIBE      5
++++++++++++++++++++++++++++++++++++++++++++++++++++
 a2e: PRINT      1648,    30            " To the east is a
folding door."

 a37: ATTRIBUTE     5,     9    =1
 a38: IF         TRUE
 a3e: PRINT      1666,    13            " It is closed."
 a3f: ELSE
    :
 a47: RETURN
----------------------------------------------------

 994: RETURN
----------------------------------------------------
```

```
> q
```

In the instruction trace, 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). Lines of dashes, indicates the return from one such level of interpretation.

8 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).

8.1 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). Therefore, the first step *must* be to get a good idea. This may be hard or easy but with time, you, like any 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.

8.2 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 why the object is needed (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.

8.3 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 might ask your friend to participate in the development by giving him a few files to work with.

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.

8.4 Polishing the Adventure

There, now you have a working adventure, it's still 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).

8.5 Beta Testing

Now you might think that you can start distributing your game. But, wait! As any complex computer program, the game may 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 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 181) to produce a list of the commands a player have used. This can greatly aid in spotting troublesome areas in your game. One common such is where the player becomes stuck and reverts to "guess-the-verb". The log will give you the output of the exact game played.

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

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

A CONVERSION FROM V2

This appendix tries to give help and advice for those that attempt to port version 2 games to version 3. Although not exactly trivial, it can be rather straightforward and is in not so complicated that it should not be attempted. On the contrary, the author believes that for most games it is simple, but perhaps a bit tedious. The use of some (mis-) features of version 2 can give rise to trouble and these features are listed here together with possible ways to minimize work or work around the problem or even possible well structured rewrites.

A complete package of the Alan Development System v3 should include a converter program. The source code is available.

A.1 Porting Strategy

Real life has learned this author one lesson: do it bitwise, little by little, one step at a time, even a long journey starts with a small step, …

Start by running your game in a V2 environment. Log the output to a file. While doing this, input any command you can think of. Abuse your game. Ask your evil brother to type some commands. Play it to the end.

Now you have a good starting point:

- A sequence of commands
- A log of the exact output of your game given that input

Copy your game to a new location. Start by running the converter program. It will do the bulk of the straightforward conversions for you. Compile the converted source. Get chocked over the number of errors. Note the number. Pick a section below or an error message in the source,

and try to address that by changing the corresponding code. Compile it again. Note the number of errors. Any progress? Repeat.

Once you have a running game, run it in a V3 environment. Log the game output. Input the exact same commands as above. Compare the output. Different? If not, you are done already! Else, you have some debugging to do. Consult the sections below again. Found anything? Fix it and run again. Progress! If not, look for appropriate sections in this manual, possibly compare it with a v2.x manual. When finally the output of your game is the same as it was in the V2 log, you are done! Well done!

I repeat, don't think you can do it by hacking along. Don't be tempted to change that little spelling mistake, or add that button to the television set. IT WILL RUIN YOUR BASELINE! I.e. you will have nothing to compare it with anymore. Instead, write that idea down on a piece of paper for memory and later action.

Take it easy. Save a new backup copy every time you have made progress, and that's every time. (File versioning software is a blessing.) Eventually you will get there.

By the way, this strategy applies not only to porting Alan games. It applies to all development, this is how Alan v3 was possible; in fact, it applies to the whole life of this author now…

A.2 The Little Words

There are a couple of new, reserved, words in Alan version 3. These that might cause compiler errors if they where used in old source. This usually occurs because they are used in location names without quotes. The most common problematic words are `THE` and `EVERY`.

The simple and usually correct change is to quote them. E.g.

```
Location l Name Outside the house
```

Needs to be converted to

```
The l Isa location Name Outside 'the' house.
```

Or even

```
The l Isa location Name 'Outside the House'.
```

Do you remember that a location name can be a single quoted identifier?

A·3 Names and Capitalization

Alan V3 has rethought the handling of names, identifiers etc. V2 converted everything to lower case and used that in the game. This caused many problems with actor names and in other cases where you wanted your output to be edited in a special way.

In V3 you can use any case in a name or identifier and that will be preserved. Comparisons are done in a case insensitive way, in both the compiler and the interpreter. This means that you can give the actor a real name with leading capitals and that name will be used in the game output. The player will still be able to input lower case and it will match.

This does away with many occasions where quoted identifiers had to be used together with multiple names to achieve the desired output while keeping the possibility for the player to refer to the items.

When converting a V2 game, especially look out for location and actor names, since these where automatically capitalized. Also, note that the first occurrence of a word will define its capitalization, which includes names of locations. (You might have used capitalization on common words that should not be capitalized, solve this by quoting the complete location name.)

A.4 Objects, Actors and Locations

Alan v2 had three built-in entities, or classes. These where built in also into the language. V3 has generalized this into the class structure and the pre-defined classes documented in this manual.

The conversion is tedious but straight-forward (Emacsophiles can probably hack out an elisp-macro):

```
OBJECT o …
   :::
END OBJECT o.
```

Should be translated into

```
The o Isa object …
   :::
End The o.
```

The conversion is analogous for actors and locations.

A converter program is also available that will do this automatically for you.

A.5 Articles

Alan v2 allowed the indefinite article to be modified by the author. Indefinite articles where used in listing of containers and default listings of a location for example.

In V3, the article mechanism has been extended to also include definite articles and, to support languages where also the form of the noun change, the **`Indefinite/Definite Form`** has been introduced. This, in conjunction with the new forms of the **`Say`** statement, to indicate definite or indefinite form (**`Say The x. Say An x.`**), allows more control over the presentation of instances. E.g. it is possible to always use the **`Say The`** form in the printout and let the instance or class handle how to print this instance in definite form. An example

would be actors with proper names who usually do not want a definite article in front of their name.

You should also look for embedded parameter references in strings (`"You xxx the $o."` etc.). These should be changed to use the new embedded definite and indefinite form printout, `"You xxx $+1."`, or even better, to the form

```
"You xxx" Say The p. "."
```

See Output Statements on page 117.

A.6 Verbs in Locations

Verbs in locations was, in Alan v2, executed simply when the hero was in them. In v3, this is also true. However, given the fact that location inherits from the same base class, **`entity`**, as objects and actors do, it may be beneficial to study Verbs in Locations 104 and Verb Execution 109.

A.7 Syntax and Syntax Restrictions

In version 2 it was possible to refer to the (single) parameter of a syntax construct using the predefined **`OBJECT`**, which was meant for use with default syntaxes. In v3, this is no longer possible. On the other hand, it is allowed to define a syntax that has "object" as the name of the parameter.

Version 2 allowed restrictions to list multiple classes in the same restriction clause:

```
Syntax v = v (o)
   Where o Isa Actor Or Object …
```

This is no longer possible; each clause must specify a single class:

```
Syntax v = v (o)
   Where o Isa actor Else …
   And o Isa object Else …
```

The contorted example above illustrates the point. Usually what needs to be done is a rephrasing so that the restriction actually refers to the common parent class to the two in the original:

```
Syntax v = v (o)
   Where o Isa Thing …
```

This will allow both actors and objects at that position. If you want to still keep the multiple restrictions, note that successive restrictions should be progressively more restrictive, i.e. the class should be more specialised.

The analysis of the ELSE-part of a restriction clause was incomplete in version 2 and allowed the use of parameters in ways that was not actually safe. This is improved in v3, possibly giving rise to errors in these clauses in games converted from v2.

A.8 Syntax Synonyms

A feature often requested is action or verb synonyms. In v3 this is possible. By declaring multiple but different syntaxes for the same verb, they will in fact work as such:

```
Syntax give = give (o) to (a) Where …
Syntax give = give (a) (o).
```

In v2 this was an error. A usual remedy was often to use separate verbs (with each its own syntax) and list both (or all) in the verb declaration:

```
VERB give, give_to …
```

This was troublesome, and incomplete, since in the above example the parameters are not in the same order. Any such attempt should be replaced by the new feature, and multiple verbs in verb declarations avoided.

A.9 Default Attributes

Version 2 allowed attributes to be given to all objects, all actors, all locations or all of these. This was done using the `DEFAULT ATTRIBUTES`, `DEFAULT OBJECT ATTRIBUTES`, `DEFAULT ACTOR ATTRIBUTES` and `DEFAULT LOCATION ATTRIBUTES` respectively.

The new classing mechanism of Alan v3 solves this in a much more flexible way by allowing adding attributes on *any* level in the class hierarchy. You can still add properties, including the special case of attributes, to a class after its definition (to aid in definition of libraries and other general extensions).

```
DEFAULT ATTRIBUTES human. NOT plural.
:
DEFAULT OBJECT ATTRIBUTES moveable.
:
DEFAULT ACTOR ATTRIBUTES real_name.
:
DEFAULT OBJECT ATTRIBUTES size 0.
```

This sequence illustrates the common practice in v2 to add attributes to the various pseudo-classes, and do it in a piece-meal fashion, usually in connection with the definition of some verbs. The above can easily (simple-mindedly) be translated into v3:

```
Add To Every thing Is Human. Not plural. End Add To.

Add To Every object Is moveable. End Add To.

Add To Every actor Has real_name. End Add To.

Add To Every object Has size 0. End Add To.
```

Note that, in V3, you can add all properties, not only attributes, to a class after its definition.

A.10 Containers

Version 2 and earlier allowed "pure" containers, i.e. entities that where only containers. In early versions, this was the only kind of container and

you had to connect that container to the object or actor by statements. In v2.8, you could add the container property to both actors and locations.

Version 3 does not support "pure" containers. One common use was the inventory of the hero. This old style inventory handling will have to be converted to use of the built in container property of the hero. E.g.

```
CONTAINER inventory
   HEADER "You are carrying"
   EMPTY "You are empty-handed."
END CONTAINER inventory.

…

VERB take
   DOES
      LOCATE o IN inventory.
END VERB take.
```

Will have to be converted to

```
The hero Isa actor
  Container       -- the inventory
     Header "You are carrying"
     Empty "You are empty-handed."
End The hero.
…

Verb take
   Does
      Locate o In hero.
End Verb take.
```

Pure containers were also sometimes used to collect items that should always be where the hero was. Since pure containers are no longer supported this has to be implemented in another way.

An instance of the predefined class **`entity`** does not have the properties of objects and actors amongst which are the properties of having a location and being described. Therefore, you could declare an instance like

```
The air Isa entity      -- note: inheriting directly from entity
  Container
End The air.
```

This container will be available at all locations but it will never be described so it can be used in the same way as pure containers in version 2.

An alternative is to make the things that should be always available inherit from **entity** themselves. This will make all of them available at all times.

A·11 Scripts

Version 3 only allows named script as opposed to the numbered scripts also allowed in v2.x. A simple solution is to put a single character in front of the number. A better solution is to invent descriptive names for the scripts instead.

A·12 LOCATION, OBJECT & ACTOR

Version 2 had the three expressions **LOCATION**, **OBJECT** and **ACTOR**. They referred to: the location where the execution was taking place, the first parameter in the current input, and the currently executing actor respectively.

Those expressions are, obviously, not available anymore. The words are no longer keywords and the identifiers actor, object and location can now be used as any other identifiers. (Remember though that the language predefines them as classes corresponding to their previous semantics.) The need to refer to the current actor (**ACTOR** expression) and the current location (**LOCATION** expression) remains and has, in V3, been replaced by two new expressions:

- **Current Actor**
- **Current Location**

See Current Entities on page 147 for details.

The reference to the first parameter in a syntax declaration, using `object,` is not available except for verbs declared in an object without an explicit syntax. (It will receive a default syntax, where the class name is the identifier for the first parameter.)

A.13 Global Verbs

In version 2, global verbs had the semantics of being a generalization of verbs inside objects. I.e. it was assumed that they would be used with parameters.

In version 3, you must add such generalized verbs to a common super-class, e.g. the pre-defined class **`thing`**. Verbs inside classes or instances having no explicit syntax will receive a default syntax similar to the one in version 2:

```
<verb> = <verb> (<class>).
```

The default identifier for the parameter is the name of the class (or class of the instance) in which the verb was declared.

Version 3 assumes that global verbs (verbs outside of any class or instance) are intended to have no parameters. Global verbs without an explicit syntax therefore receive the default syntax of:

```
<verb> = <verb>
```

Global verbs with parameters from pre-3 version source must be moved to an **`Add To`** clause for the appropriate class, usually **`thing`**, but if your verb handles literals you should move it to **`string`** or **`integer`** as appropriate.

You can use the compiler switch **`-information`** to see compiler messages indicating which verbs get which default syntaxes.

A·14 Aggregates

Aggregates are functions that calculate values from sets of items. In Alan v2, the **SUM**, **MAX** and **COUNT** aggregates worked on objects only. You could only filter the objects aggregated to be only those at a location.

Alan V3 has generalized all aggregates to allow a list of filters. The following is a V2 aggregate expression:

```
IF COUNT HERE > strength OF bridge THEN …
```

In order to be exactly equivalent in V3 it must be changed so that the aggregation is only performed over objects:

```
If Count Isa object, Here > strength Of Bridge Then …
```

Naturally, if you have made other changes to your code, this might need further analysis.

A·15 Multimedia

Alan v2 had no provisions for multimedia. It has been reported that the SYSTEM statement has been used to some extent. Alan v3 provides the **Show** statement to support graphics in a portable way. Note that this might still not be available on all platforms.

A·16 Include directive

In v3 the **$include** has been replaced by the **import** statement. They work the same, except that the new **import** statement can only be placed where declarations (classes, instances etc.) can occur. This ensures a better structure of the contents of files, and aids in developing tools for analysis of Alan source.

A·17 Messages

Most default messages given as a response to the player have changed from V2 to V3. In V3 they all use the definite and indefinite features of course, but the message sections for messages which require so, also are defined so that run-time parameters appropriate for attribute testing or printout are available. The most significant change is that many of the identifiers for the messages have changed. Read Input Response Messages on page 210 for a list of V3 message identifiers.

A·18 Pronouns

In V2, the language in itself defined a few pronouns. V3 allows an author to take control over the pronoun handling. This should however, make little difference when porting a V2 game.

A·19 New Features

Of course, there are new features not available in v2, but these rarely affect the porting of source from a v2 game, except where new keywords have been introduced in the language. These need to be quoted or replaced when porting a v2 game to V3.

B A SAMPLE INTERACTION

In order to give you an idea of what playing an adventure might be like, if you do not have done that, this appendix lists a typical, and authentic, interaction between a player and a game. Player input is preceded by a prompt ('>') and is emphasised for clarity.

In reading this through, you will find that it is possible to use more or less full imperative statements to communicate with the game. However, you may leave out non-important words and the game usually provides a set of common abbreviations, such as 'w' for 'west', which really means 'go west'.

It is also common to provide possibility to refer to multiple objects in a command, either by use of 'all' or specifically listing the objects.

Your input is in italics.

```
West of House.
Welcome to Dungeon (ALAN Demo). This version created 29-FEB-92.
You are in an open field west of a big white house with a
boarded front door. There is a small mailbox here.

> examine the mailbox
I see nothing special about the mailbox.

> open it
The small mailbox is now open. The small mailbox contains a
leaflet.

> take the leaflet and read it
Taken!
'I hope you have noticed that this isn't your usual Dungeon but
rather an ALAN implementation brought to you by
    ThoNi & GorFo Adventure Factories
Happy Adventuring!'

> inventory
You are carrying a leaflet.

> go south
South of House.
You are facing the south side of a white house. There is no
door here, and all the windows are barred.
```

```
> east
Behind House.
You are behind the white house. In one corner of the house
there is a window which is slightly ajar.

> open the window
With great effort, you open the window far enough to allow
passage.

> go in
Kitchen.
You are in the kitchen of the white house. A table seems to
have been used recently for the preparation of food. On the
table you can see an elongated brown sack, smelling of hot
peppers and a clear glass bottle. A passage leads to the west,
and a dark staircase can be seen leading upward. To the east is
a small window which is open.

> examine all
You can't refer to multiple objects with 'examine'.

> take all
(elongated brown sack, smelling of hot peppers) Taken!
(clear glass bottle) Taken!

> up
Upstairs.
You are in the attic. The only exit is stairs that lead down. A
large coil of rope is lying in the corner. There is a square
brick here which feels like clay. A nasty-looking knife is
lying here.

> take all
(rope) Taken!
(square clay brick) Taken!
(nasty knife) Taken!

> examine knife
I see nothing special about the knife.

> down
Kitchen (again)
A table seems to have been used recently for the preparation of
food. The table is empty. To the east is a small window which
is open.

> w
Living room.
You are in the living room. There is a door to the east. To the
west is a wooden door with strange gothic lettering, which
appears to be nailed shut. There is a trophy case here. In the
center of the room is a large oriental rug.
```

```
On hooks above the mantlepiece hangs an elvish sword of great
antiquity. A battery-powered brass lantern is on the trophy
case. There is an issue of US NEWS & DUNGEON REPORT dated 29-
FEB-92 here.

> move rug
With a great effort, the rug is moved to one side of the room.
With the rug moved, the dusty cover of a closed trap door
appears.

> open door
I don't know which door you mean.

> open trap door
The door reluctantly opens to reveal a rickety staircase
descending into darkness.

> take the sword, the lantern and the issue
(rug) The rug is extremely heavy and cannot be carried.
(elvish sword) Taken!
(battery brass lantern) Taken!
(issue) Taken!

> read issue
    US NEWS & DUNGEON REPORT
29-FEB-92.......................Late Dungeon Edition
    --- LATE NEWS FLASH!! ---
The Dungeon prelude is now available as an ALAN implementation.
The Dungeon game was selected as an example that everybody
interested in adventure gaming should recognize.
```

C PREDEFINED PLAYER WORDS

Alan defines a set of words for the player to use, which are required for the syntax variations described in Player Input on page 159. These words are available even without any declarations at all in the game source. Some of these might conflict with, or complement, words defined in the source. The lists below contain those player words for the currently defined languages.

English

```
ALL: all everything
AND: and then
BUT: but except
THEM: them
NOISE: go the
```

Swedish

```
ALL: alla allt
AND: och
BUT: förutom utom
THEM: dem dom
NOISE: gå
```

German

```
ALL: alles
AND: und
BUT: ausser
```

```
THEM: sie
NOISE: das der die gehen
```

D 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.

D.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. However, some messages 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 117. 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 Messages4.14 on page 115). 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 text after the colon on the first line is the default English message text. Then follows a short explanation, including possible availability of parameters and their types.

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

Note: Although the default values of the messages are static strings, it is possible to create messages that are more dynamic. The **Message** statement allows any statements, not only strings, and supplies dynamic values as parameters for many messages. See Messages on page 115 for details.

`UNKNOWN_WORD : "I don't know the word '$1'."`

A word not in the dictionary was used by the player.

parameter1 is a string containing the word used.

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

The input did not follow any syntax the Arun parser knows about. I.e. the input could not be matched to any of the defined syntaxes.

WHAT_WORD : "I don't know what you mean by '$1'."

The player input a multiple word, such as "all", "them" or a pronoun, but the Arun parser could not find any objects or actors that it could refer to.

parameter1 is a string which is the word used by the player.

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

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

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

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

`AFTER_BUT : "You must give at least one object after '$1'."`

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

parameter1 is a string containing the **BUT**-word the player used.

```
BUT_ALL : "You can only use '$1' after '$2'."
```

The **BUT**-words may only be used after an **ALL**-word.

parameter1 is a string containing the **BUT**-word used by the player.

parameter2 is a string containing the **ALL**-word used by the player.

```
NOT_MUCH : "That doesn't leave much to $v!"
```

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

```
WHICH_START : "I don't know if you mean $+1"
WHICH_COMMA : ", $+1"
WHICH_OR : "or $+1."
```

```
Multiple objects (or actors) matched the words given
by the player. More adjectives are necessary to
distinguish between them. The three messages are used
to list the possibilities. The player can repeat the
command with a more precise wording. The first
message is used for the first alternative, the last
for the last alternative and the middle for all the
middle alternatives.
```

For each message, **parameter1** is a reference to the alternative instance.

```
WHICH_PRONOUN_START : "I don't know if you by '$1'"
WHICH_PRONOUN_FIRST : "mean $+1"
```

When a pronoun given in a command matched multiple parameter in the previous command, these messages are issued to explain this and which the alternatives where. Note that the message is different from the multiple match above only for the start of the message, the list of alternatives are the same, i.e. WHICH_COMMA (repeated) and WHICH_OR (the final).

```
NO_SUCH : "I can't see any $1 here."
```

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

parameter1 is an instance referring to an instance.

Note:If there did not actually even exist an instance in the game with the combination of the adjectives and nouns that the player used, the interpreter uses any instance matching the noun. This still allows inflecting in accordance with the noun case, which is common in many languages (English being one of few exceptions).

```
NO_WAY : "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.",
```

The interpreter could match the input to some syntax, but did not find any 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.

```
SEE_START : "There is $01"
SEE_COMMA : ", $01"
SEE_AND : "and $01"
SEE_END : "here. "
```

These messages are used to construct the default text for describing **things** present at the current location that have no description clause. The message parts are used as in "*There is* <*indefinite form object1*>, <*indefinite form object2*> *and* <*indefinite form object3*> *here.*" The underlined parts are the ones in the messages and each object is printed in its indefinite form as appropriate.

```
CONTAINS : "$+1 contains"
CARRIES : "$+1 carries"
```

The messages above are used to construct the default headers for listing containers. The CARRIES message is used if the container instance is an **actor**.

```
CONTAINS_COMMA : "$01,"
CONTAINS_AND : "$01 and"
CONTAINS_END : "$01."
```

The messages above are used to construct the contents listing of a container in much the same way as for the object listing above. The messages are used according to the pattern "*<header for container> contains <indefinite form contents1>, < indefinite form contents2> and < indefinite form contents3>.*"

You can modify these messages to change the formatting of listings. e.g. to one element per line.

```
CAN_NOT_CONTAIN : "$+1 can not contain $+2."
```

If an attempt to put something in a container that does not meet the class restrictions for the container, this message will be delivered.

`IS_EMPTY : "$+1 is empty."`

The default messages for empty containers.

`EMPTY_HANDED : "$+1 is empty-handed."`

The default messages for empty containers that are **actor**s.

`HAVE_SCORED : "You have scored $1 points out of $2."`

This is the default message for presenting scores.

parameter1 is an integer containing the current score.

parameter2 is an integer containing the maximum score possible.

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.

`SAVE_WHERE : "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.

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

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

`SAVE_FAILED : "Sorry, save failed."`

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

`RESTORE_FROM : "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.

`SAVE_MISSING : "Sorry, could not open the save file."`

When executing a **Restore**, Arun could not find, or open, a save file with the name entered.

`NOT_A_SAVEFILE :"That file does not seem to be an Alan game save file."`

The save file found by the **Restore** statement was not Alan game save file.

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

The save file found by the **Restore** statement was created by a different version of the Alan interpreter or the game.

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

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

`REALLY : "Are you sure (RETURN confirms) ? "`

This is the confirmation prompt, e.g. before overwriting an already existing save file.

QUIT_ACTION : "Do you want to RESTART, RESTORE, QUIT or UNDO?"

The **Quit** statement requests an action from the player.

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

`UNDONE : "'$1' undone."`

When an action is undone, this message is presented to confirm the player action.

parameter1 is a string containing the player command that was undone. Note that since only commands that change any state in the game world are logged the command might very well not be the last command.

`NO_UNDO : "No further undo available."`

If the player tries to undo an action and no further actions where recorded (because of lack of memory, undone to beginning of session, etc.) this message is used to inform the player of that fact.

D.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 *8.5* on page 245 for how to reach us). This might also be caused by reading incomplete or corrupted game files.

`Incompatible version of ACODE program.`

The version of the interpreter you are using is different 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. It should *not* be used in descriptions of **LOCATIONs** and **OBJECTs** because 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:

Player Errors

Errors caused by incorrect arguments or file names.

`Can't open adventure code file.`
: The player attempted to run an adventure for which there were no code file available, probably a misspelling.

`Could not read all A3C code.`

`Checksum error in Acode (.A3C) file (%1 instead of %2).`
: These two messages indicate problems in the adventure file. possibly caused by transfer problems of the **.a3c** file, this must always 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. 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 181).

E COMPILER MESSAGES

E.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.

```
ZILexample.alan

23.  If barfoo Is foobared Then
====>     1
*1*   310 E : Identifier 'barfoo' not defined.

27.      Exit north To Rumble.
====>                      1
*1*   310 E : Identifier 'rumble' not defined.

28.      Exit west To Tumble.
=====>                  1
*1*   310 E : Identifier 'tumble' not defined.

46.
====>  1
*1*   101 E : 'START' 'HERE' '.' inserted.
*1*   211 E : Must start at a Location.

          5 error(s).
          No detected warnings.
          2 informational message(s).
```

The following information is available in the compiler listing, framed for visibility:

1. File name
2. Line number and source text of a line
3. Message indicator or pointer
4. Message number and text
5. Message summary (three lines)

For information on how to select which levels of messages to show and where output is directed, refer to the options and their descriptions in section Compiler Switches on page 237.

E.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 the letter '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 will continue 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 File on page 157.

```
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 (class or instance). 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 161.

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 161.

```
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.

F HOW TO USE THE SYSTEM

F.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 DOS-based PCs), if none is supplied. The option **-help** will give a brief help on other options to the compiler.

The primary output from the compiler, **alan**, is an adventure code file ***adventure*.a3c**.

An identifying file, ***adventure*.ifid**, is also produced. This file contains a unique identification of your game for bibliographical purposes. The content of it will be compiled into the adventure code file, which makes your game identifiable by electronic means. As long as this file exist the same identification (IFID) will be used. If it does not exist, it will be automatically generated.

F.2 Compiler Switches

If you run the compiler from a command line it 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 File 157 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 55)

• **-pack** encode and compress the text data (same as the pack option, see Options on page 55)

-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.

F.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.

On platforms with graphical user interfaces to which **arun** has been natively ported will allow double clicking a game file, or the interpreter application icon, in which case a dialog requesting a game will appear.

If the interpreter program is copied to a different name, it will automatically look for a game file 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 file **adventure.a3c**. The files **adventure** and **adventure.a3c** thus makes a complete game package, which is easy to start using the single command:

```
> adventure
```

F.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[**<n>**] various levels of execution trace, higher <n> gives more details
-l log all player command in a log-file in the current directory

Debugging support described in Debugging on page 181.

G 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://www.alanif.se for latest info). Below follows some very specific information for some of the platforms.

PC

In the PC environment the Alan compiler is a command shell program. 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.

For Windows environment there are two interpreters available. **`arun.exe`** is a command line program that uses the Cygwin run-time library to perform terminal I/O with status line ability. It will open a console window if activated from the Windows explorer. To actually run a game you will either have to drop it on the **`arun.exe`** icon.

The interpreter **`winarun.exe`** is a Windows GUI application using the GLK-library. This means that you will need a ***glk.dll*** accessible, most probably in C:\WINDOWS.

In a Windows environment, you can associate the extensions **.ala** and **.a3c** with the programs Alan and Arun respectively. This will enable compiling and running by double clicking on the files.

Unix

On UNIX systems command history, recall and editing is available.

G.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 **.a3c** file to another machine it should be possible to run the game 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 are encouraged to seriously consider this 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.

H LANGUAGE GRAMMAR

H·1 Description

The Alan language is in this manual defined using a BNF-form. This is a set of rules defining what constructs are legal in an Alan source.

The BNF form divides the structure of the input source into smaller parts (rules), which in turn are defined by other rules. For example, a rule might say that an **ADVENTURE** (in this case an Alan program) consists of options, declarations and a start section. This grammar rule would look like:

```
adventure = [options] {declaration} start_section
```

Each item that is an identifier ('options', 'declaration' etc.), is a construct that in turn is defined by other rules, possibly elsewhere in the manual.

The equal sign (=) may be read as "consists of" or "is defined as". Optional parts are surrounded by square brackets ('[' and ']'). Parts that may be repeated zero or more times are enclosed in curly braces ('{' and '}'). So the rule might be read as 'an adventure consists of options which are optional, zero or more declarations and a start_section'.

If the item to the left of the equal sign may be defined in multiple ways, the alternatives are divided by a vertical bar ('|'). For example

```
declaration  = messages
             | class
             | instance
             | verb
             | rule
             | synonyms
             | syntax
             | verb
             | event
             | addition
```

This definition says that a **declaration** might be messages, a class definition, an instance declaration, etc.

The basic component of the language is reserved words and symbols. These are in the rules represented by quoted strings of characters. These are not defined elsewhere, but should instead be written as indicated. Character case is not significant.

```
random_expression = 'RANDOM' 'IN' expression
```

The reserved words 'random' and 'in' can be followed by an expression to form an expression in itself.

H·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 if the rules described in Words, Identifiers and Names on page 153 are followed.

```
<<to be supplied>>
```

H·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.

H·4 The Grammar

```
<<to be provided>>
```

I 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!

The Alan Home Pages on the Internet can be found at
http://www.alanif.se

Here are some ideas of things we are thinking about:

- Definition of interaction with actors, perhaps through some kind of pattern matching sub-language using string literals.
- Background pictures
-
- ...

J 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 games of the time, 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 a 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 adventure system designers.

[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.

K 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.

L COPYING CONDITIONS

The Alan Adventure Development System is REGISTER-WARE. This means that to use 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.

A copy of these conditions must accompany any copy of the Alan System.

L·1 Distribution

The Alan System is mainly distributed by downloading from the Internet. 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.

L·2 Documentation

The documentation is copyrighted by author. Copying is allowed provided it is distributed as a whole, or quoted accompanied with appropriate references.

L·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 will honour 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 supported platforms, commercially or otherwise released games and other information supplied by other Alan users or The Factories.

L·4 Registration

Registration is free, preferably made through a simple email message, which can be done with an email on the Alan Home Pages at http://wwww.alanif.se.

L·5 Source

The source is not in the public domain but will be released for porting to new platforms and technologies and support of such ports. The source may not be used for other purposes without permission from the author.

L·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.

L.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.

L.8 Executive Summary

So, in short, the interpreter Arun and any game produced using the Alan System 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.

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 that you waive all rights to it. Examples add to the suite of test data and thus helps further improve the quality of the system.

INDEX

A

B

C

D

E

H

I

L

M

N

O

P

Q

R

S

T

U

V

<

W