The ALAN Adventure Language Reference Manual

Revision Beta7

August 27, 2021

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Text adventures or, using a more appropriate term, interactive fiction, is a form of computer game which has many things in common with fiction in book form, role-playing games and puzzle-solving. To create a high quality interactive fiction game, you need to be more of an author than a 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, it should be very easy to do common things, but it should also be possible to do more complicated things by constructs that are more complex. This means that wherever a construct is optional, the system supplies some sensible default.

The author and a very good friend designed the first crude version of the Alan language in 1985. During many years of incremental improvement and use, 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 has grown, from actual use and understanding of the most prioritised needs.

In this version, modern and novel object-orientation features have been incorporated into the language that allow 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 language to use and by reading this manual, you will understand how these new features may aid you in your quest for adventures.

1.1. 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, except for instances of classes; instead, types are automatically inferred from expressions such as integers, strings or instances of a particular class.

1.2. 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 interactive fiction but new to Alan — read from Section 2.4, “Introduction to the Language” onwards.
3. Alan v2 users wanting to upgrade — you should read the separate document on conversion, then Section 2.4, “Introduction to the Language” and onwards, with frequent use of Chapter 3, *Language Reference* as a reference while doing your conversion.
4. Alan v3 users looking for detailed answers — use the *Index* (PDF version only), look up the relevant sections in Chapter 3, *Language Reference* but also glance through Chapter 2, *Concepts* from Section 2.4, “Introduction to the Language”. Visit www.alanif.se for a collection of examples.

All readers are encouraged to give feedback on the documentation, particularly if you could not find the answer to what you were looking for by using the index, the table of content or skimming through what you thought might be relevant parts of the documentation. You can find the authors through the web page www.alanif.se, where you also can enrol in the Alan mailing list, a place for new and seasoned Alan users alike!

Chapter 2. Concepts

This chapter introduces the concepts used in the Alan language. You might already have a good idea about these things, especially if you are a seasoned adventure player, and perhaps even author.

But I would suggest that you read through it anyway since it introduces some important concepts that are specific to how Alan treats them.

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 time, 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.

These games are played interacting with the computer. The program describes a scene or situation (usually in text, but pictures may also be 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). Although the original authors are long scattered, the Infocom games are still highly appreciated even 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 early examples are the Sierra games (notably the *Leisure Suit Larry* and the *Kings Quest* series) which had mouse oriented moves but also allowed single line text commands, games from ICOM Simulations (*DejaVu* and *The Uninvited*) which were 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 of text-based adventure games are still out there. Visit the vaults of interactive fiction on the Internet, and you will be surprised by the abundance of modern, high quality interactive fiction available.

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 also 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 via the keyboard and descriptions printed on the screen. In Appendix B, *A Sample Interaction* you can find a sample of this type of 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 language have been designed so that the textual instructions are relatively easy to read and write if you understand the mechanisms that adventures are made from. In addition, it requires 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.

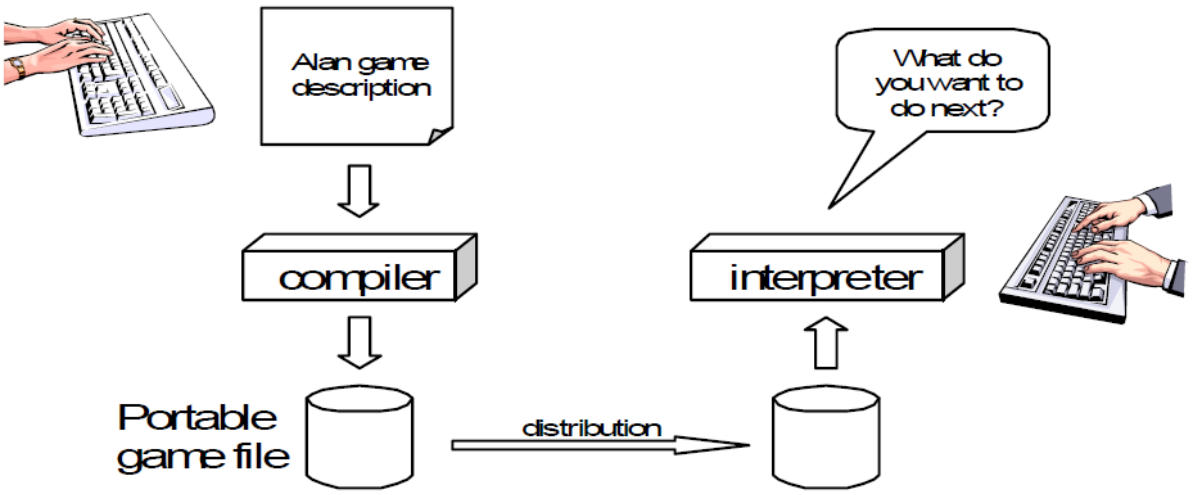

Figure 2.1. The principles for and relations between a game description, a compiler, a game file and the interpreter, or, in other words, authoring and playing.

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 doesn't understand.

There are also special editors, or additions to standard editors, available, which supports Alan coding 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.

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

The 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, how the player input controls which events will happen, how to connect one location to another and how to store information for later use. In a way this is programming, but in an unusual sense.

In order to understand the rules of the Alan language we will have to establish some common ground. As an author you'll have to learn to view works of interactive fiction in the same way as the Alan language does. It will not be hard as most of the concept are familiar already.

2.3.3. What's Happening?

The execution of an adventure is primarily driven by the input of player commands. A command is analysed by the interpreter according to the command syntax prescriped by the author and, if understood, it is transformed into execution of verbs or movements.

After the player has had his turn, any special conditions, or rules, that might have been programmed by the author are evaluated.

Then, other, scripted non-player characters, or actors, will move. Their movements and actions are 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 detail in Section 5.1, "A Turn of Events".

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.

> In Alan, exits are always one-way, and an explicit declaration of a backward path (if such is desired) must be made. Although, normally you would probably want them to be two-way, if they where automatically two-way, it would be very hard to handle the rare, but important, cases when you don't want them to be so.

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, describe 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 result in verbs executions (“calls”). 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) 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 reflect 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 `Location`s.

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

Then the player will need words to command action. The Alan language construct to supply these 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 source code is typeset in separate sections with a mono-spaced font:

```
This is an example of some source code.
```

You will also encounter sample game-play which will be formatted using a surrounding border (like paper...) thus:

> **Grandma's House**
> You are outside your grandma's house.

Later in the manual, you will find 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 D, Language Grammar.
```

In running text, words that are keywords or signify an Alan construct are written in a mono-spaced font. This helps distinguish the English 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).

> And this is a note!

2.4.2. The Locations

The scene for your adventure is a series of "rooms" or, rather, locations. `Location`s are connected by `Exit`s, 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 `Exit`s and to which `location`s they lead.

In fact, this is all that is necessary in a `location`, so lets look at an example.

```
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 can 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 and try the adventure (see Appendix A, *How to Use the System* on how to do this). After starting the adventure, two lines will be shown on your screen.

```
Kitchen

>
```

The first line contains "Kitchen", the name of the initial location, and the second a ">", which is the default 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. (Use **Ctrl**+**C** to exit the game if you are running it in a console window.)

The identifier for a `location` is automatically used as a description, a heading, shown when that room is entered. And the words listed in the `Exit` parts are 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, you 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 are 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 `object`s too. A large number of objects that can be examined and manipulated make a game so much more enjoyable.

`Object`s, like `location`s, have an identifier and a `Description`, 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 necessarily 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"`).

The example also introduced 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 (Section 3.21, “Expressions” and the section called “IF Statement”).

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

It must of course be possible to change the attributes value of an object. You can do this using the `Make` or the `Set` statements. 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.
```

These statements only change attributes. The implications of such a change must be implemented by writing Alan code that tests 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.

Alan does not understand, nor 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 `object`s and `location`s do. An actor performs his movements by following `Script`s, each having a number of `Step`s. 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, instances. If every mammal is a vertebrae, we know that all properties of vertebrates also applies to mammals. We can use this knowledge to handle commonalities without knowing anything about the more specialized kinds, or classes. One example of this might be lockable things like doors and drawers. If they inherit from a common ancestor `lockable_things`, then we do not need to know if it was a `door` or a `drawer`, if we are only interested in the `locked` property. This flexibility, know as polymorphism, is possible in programming only through object-orientation and inheritance.

EVERY and THE

The Alan language supports object-orientation and inheritance via two constructs:

```
Every mammal IsA vertebrate ...

The house_pet IsA cat ...
```

The `Every` construct defines a class and its properties, including inheriting from another, even more general class. The `The` construct declares an instance, which in this example inherits from the class `cat`. The `IsA` construct defines from which class properties are inherited.

The Predefined Classes

To make it easy to get started there are eight classes predefined in the Alan language.

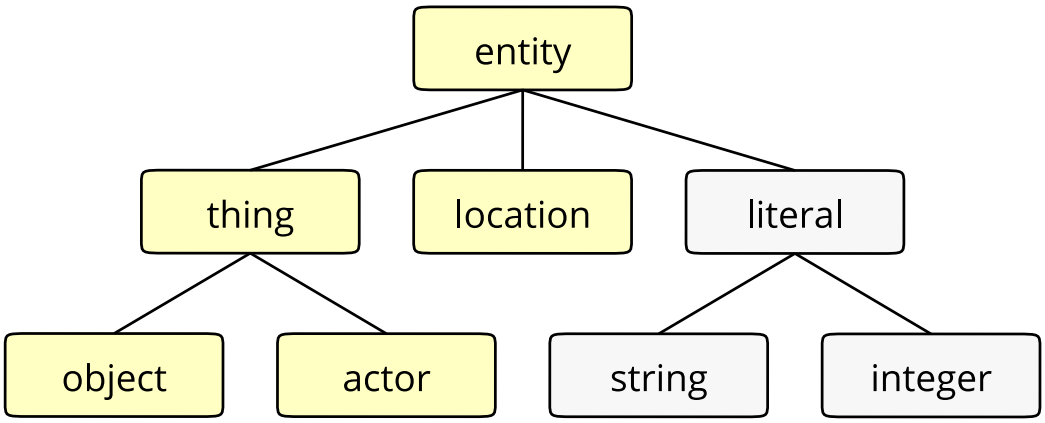

Figure 2.2. Relationships between the predefined classes (only classes shown with yellow background are inheritable and can be explicitly instantiated).

They are `entity`, `thing`, `location`, `actor`, `object`, `literal`, `string` and `integer` and have the relationship, inheritance tree, shown in the figure above.

The semantics of these predefined classes are in short:

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

See the subsections of Instances 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 predefined classes. However, it is the identical if you have defined the class yourself. In general the format is:

```
The <instance identifier> IsA <class identifier> ...
```

Defining a class is pretty much as you would expect it to be:

```
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 declaration of the class `openable_door`.

Specialising and Overriding

Sub-classing, or specialisation, is usually employed 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 as:

```
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 property.

This concludes our brief presentation of object-orientation and how the Alan language supports it. The most important thing to remember is that you can extend the inheritance tree with new classes and that most features can be inherited along that tree and be overridden, both by intermediate classes in the inheritance chain and explicitly in the instance itself.

2.4.6. Containment, Classes and Transitivity

One basic property of instances is that they may contain other instances. Although conceptually simple there are twists that you should know about.

Containers Containing Containers

Containers might contain other instances that are in turn containers and so on, of course. If you want to consider everything inside a container recursively, you might actually get types of instances you did not expect.

As an example, let's consider a container named `cont`, that takes a subclass of `object`, named `subobject`. Assume there is also an instance of that class, named `inst`, which is also a container, but takes `object` instead.

```
Every subobject IsA object
End Every subobject.

The cont IsA object
  Container Taking subobject.
End The cont.

The inst IsA subobject
  Container Taking object.
End The inst.
```

When you search, recursively, for instances in the container `cont`, you might then get instances that are both of class `subobject` and `object`, e.g. if the `inst` is inside the `cont` and in turn contains something, which would then be of class `object`.

This might lead to, completely correct, but surprising, error messages from the compiler indicating that an attribute or other property that you thought existed is not available. Especially surprising is perhaps the case where the classes are not even decendants of each other. In this case the contained instances can only be guaranteed to be their common ancestor, which might be a quite general class like `thing` or even `entity`.

Transitivity

For many scenarios the above works well, and as expected. But for other cases the notion of transitivity is introduced. Transitivity describes how e.g. containment should be interpreted.

The scenario above may be described as “transitive containment” meaning that something is in a container if it is in that container or in any container it contains, recursively.

There are two other types, namely direct and indirect transitivity. Direct transitivity actually means no transitivity. If you investigate a container with direct transitivity you will only get its direct contents, not the contents of the containers within it.

Finally, indirect transitivity means instances indirectly contained by a container. In a way it is the opposite of direct transitivity, all instances recursively contained *except* the directly contained. Here’s a rule to remember:

> Transitive = Direct + Indirect

What this means is that if you use transitive containment you get the same instances that direct *and* indirect will give you. And it is usually the indirect ones that you should look out for.

2.4.7. The VERB Construct

The `Verb` is the construct that implements the effects of an action requested by the player. Verbs are usually 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 open.
```

```
End Verb open.
```

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 they are 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.

```
The kitchen_door IsA object
  Verb open
    Does
      Make kitchen_door open.
  End Verb open.
End The kitchen_door.
```

This defines the effects of applying the `open` verb to that specific door. The implementation makes direct references to the `kitchen_door`, so to make the verb more general it should be possible to apply to all `door`s. So let’s move the definition from the `kitchen_door` instance to the `door` class:

```
Every door IsA object
  Verb open
    Does
      Make This open.
  End Verb open.
End Every door.
```

With this definition it is possible to apply the verb to all doors.

A reference to the exact object the player mentioned in his command is propagated in the parameter (see Section 2.4.8, “The SYNTAX” and Section 3.10, “SYNTAX Definitions” for a more thorough discussion). Since this verb is now applicable to all `door`s, the attribute `closed` must also be available for all instances of the `door` classs. We can do that by ensuring that the attribute exists in the class using a normal attribute declaration. (See Section 3.9, “Additions” on how to add an attribute to a predefined class such as `object`).

Of course, there are often also conditions that need to be checked before we can execute this code (perhaps to see if it was possible to open the object!). Therefore, `Verb`s may have `Check`s, as we will see next.

Checking Things

In order to ascertain 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 o Is openable
    Else "You can't open the $o."
  Does
    Make o open.
End Verb open.
```

This definition of the `open` verb is more realistic than the previous one. It specifies that before the statements after `Does` are executed, the condition after `Check` must be checked (which, in this case, checks that it's really possible to open the object indicated by the player). If that condition is TRUE then the requirements are fulfilled and the body of the `Verb` (following the `Does`) can be executed. If this is not the case the `Else` part is executed instead (normally showing some message).

A `Check` may have multiple conditions, as the following code shows:

```
Verb take
  Check o takeable
    Else "You can't take that."
  And o Not In hero
    Else "You already have it."
  Does
    Locate o In hero.
End Verb take.
```

Here we also encounter a variation on the `Locate` statement — the capability to place an object inside a container (the inventory).

> You can never destroy an instance or remove it from the game. Instead, you can define a limbo location, i.e. a location that is not accessible to the player and may thus be used as a storage for "destroyed" objects and other things the player is not supposed to see.

2.4.8. The SYNTAX

Normally a verb acts on an object or actor, henceforth called a parameter, referenced by the player in a command. This means that the format of player input is usually 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 and should be interpreted as 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, referred to as `cont`), 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 `object` class (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 code. 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 Section 5.4, "Run-Time Contexts" for a detailed discussion).

The `Syntax` construct allows for more than one parameter, in order to make it possible to define more complex player commands. Therefore, the verb execution order previously described regarding 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).

2.4.9. 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. Long lines will be automatically split without breaking in the middle of words.

Secondly, a few special cases are also handled automatically:

- After a full stop (period, the character "."), an 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.

- 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, a 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 ensures 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. "."
  ```

2.5. Strict and Safe

The Alan language is strict and type safe. This means that the compiler will attempt to prevent any constructs that might generate a problem for the player, such as assigning values of one type to attributes of another type, accessing properties that are not guaranteed to exist on the instance, and so on.

A simple example is

```
1. Every animal IsA object
2.   Has fur.
3. End Every animal.
4.
5. The house_cat IsA object ...
6.   ...
7.
8.   If house_cat Has fur Then ...
```

At line 8 we will get an error saying that the `house_cat` does not have the attribute `fur`.

Now, if that is true, it's a good thing that the compiler caught the error, otherwise the game might try to access that attribute — and blow up in the face of the player!

Can you spot the problem? The `house_cat` is declared as an `object`, and not as an `animal`.

Other examples include trying to use a script for an actor (or actor subclass) that does not have one, locating something inside something that is not a container and so on.

Here is a more complicated example:

```
 1. The cont IsA object
 2.   Container Taking thing.
 3. End The cont.
 4.
 5. Add To Every object
 6.   Has someAttribute.
 7. End Add.
 8.
 9. The box IsA object
10.   Container Taking object.
11. End The.
12. ...
13.   For Each f In box Do
14.     If f Has someAttribute Then
15.     ...
```

On line 14 we will get an error saying that `someAttribute` is not available since the class `f` can only be guaranteed to be `thing`.

Is it? Well, the variable `f` (in the loop) enumerates all things in the `box` and since the `box` takes `object`s, it is possible that it may contain the `cont`. And since that takes `thing`s, and `In box` is transitive (see the section called "Transitivity"), `f` may take on any `thing` that is contained in the `cont` too.

Of course, there are various ways to fix this:

- change the transitivity of the filter in the loop (`Directly In`)
- add a class-restricting filter in the loop (`IsA object`)
- rethink your class hierarchy

The Alan compiler is trying to protect you, and your players, but sometimes the error can be hard to spot.

Chapter 3. 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 D, *Language Grammar*.

3.1. 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 should 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`.

3.2. An Adventure

An adventure starts with an (optional) set of `Option`s (see Section 3.3, “Options”) followed by a set of declarations.

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

According to the rules it is actually possible to have no declarations at all (as indicated by the curly braces) but there would be no adventure without a single location, right? So, in practice you’ll need at least one declaration.

The declarations constitute the major part of the adventure. The declarations can be declared in any order and repeated freely, and are of many different possible types.

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

The adventure source text must end with a `Start` section.

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

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..."
```

You can look up the meaning of the rules "**WHERE**" and “statement” elsewhere in this chapter.

3.3. Options

`Option`s 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:

```
options = 'OPTIONS' {option}

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

The example below illustrate how options may be written, following the above rules.

```
Options
    Debug.
    Language Swedish.
    No Pack.
    Width 128.
```

The available options are:

Table 3.1. Adventure Settings via OPTION

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

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

[b] Width and Length are always overridden by the actual terminal or window size, if that can be determined, dependent on the interpreter used. If not, the value of the option is used.

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 Appendix C, *Run-Time Messages*, Section C.1, “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 are 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 multinational (non-ASCII) characters.

`Width` specifies how long the lines the interpreter outputs should be (formatting is automatic!). The `Length` option will instruct the interpreter on 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 Section 6.14, "Debugging"), the `Debug` option must be turned on. This may also be performed using the debug compiler switch (see Section A.2, "Compiler Switches").

3.4. 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.

3.4.1. 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.

3.4.2. 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.

3.4.3. 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. `Event`s 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.

3.4.4. Set Type

A Set is a collection of values that may be referenced as a single value, but also investigate, added to and removed from. An example might be a set of cards in a dealt hand, the set of spells that the hero have learned, or the set of numbers guessed so far.

The order of elements in the set 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 one set and another set of numbers 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.

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.

3.4.5. 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.

3.4.6. 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`.

3.5. 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 Section 4.5, “Filenames”). 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. `Import`s 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*, this search is performed in the same order as the `Import` switches occurred on the command line.

The `Import` statement is the way to use the Standard Library (or a library of your own design). Place the library files in a directory where the compiler will find them, either in the same directory as your other source files or somewhere else (see Section A.2, “Compiler Switches” on how to make the compiler look in more folders than just the one where the main source file resides). In your source you would refer to the main file of such a library by:

```
Import 'library.i'.
```

Another use is for dividing your own source into multiple files to make them easier to handle:

```
Import 'harbor.i'.
Import 'city.i'.
Import 'desert.i'.
Import 'actors.i'.
Start At city.
```

3.6. Classes

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).

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

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 Section 3.8, “Properties”.

3.6.1. Inheritance

Every instance must inherit from a class (see Section 2.4.5, “Inheritance and Object-Orientation”). 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 ...
```

and

```
Every coin IsA treasure ...
```

3.7. Instances

The most important part of an Alan game source is probably the declarations of instances. Instances are the `object`s, `location`s, `actor`s and other `thing`s 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 *Inheritance* 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, `Exit`s and `Script`s 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.

Here are two examples of instance declarations following the rules above:

```
The red_ball
  IsA object
  At bedroom
  Name red ball
  Is hidden.
  Description
    If This Is Not hidden Then
      "An ordinary ball is laying under the bed."
    End If.
  Verb roll
    Does
      "You roll the ball a bit. Nothing exciting happens."
  End Verb.
End The red_ball.

The mr_brown
  IsA actor
  Name Mr Brown
  Article "".
  Pronoun him.
  Is working.
  Description "Mr. Brown is here, working at his desk."
```

```
End The mr_brown.
```

In these examples the source lines between `The` and `End The` all declare various properties that we will learn more about in Section 3.8, “Properties”. The rest of the lines are fairly easy to match up to the rules of the Alan language as described by the earlier box.

All capitalized words in the examples above are keywords in the Alan language (see Table D.1, “List of Alan Keywords” for a complete list), the rest are author defined words or identifiers (with the exception of the words `object` and `actor`, which are identifiers predefined to be special classes).

3.7.1. Entities

The base class `entity` represents the lowest denominator of all instances. All other predefined 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.

3.7.2. Things

`Thing` is a predefined subclass of `entity` that adds the property of having a location. This means that they can have an initial location and be located at 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 that 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 other

> effects of being part of a container, such as the removal rules and selection by a random selection of items in the container. See *RANDOM Values* for a description of random selections of container items.

3.7.3. 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.

3.7.4. Actors

The predefined class `actor` is intended for providing so called NPCs, 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 *SCRIPTs*) 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.

The george IsA actor
  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 with some basic properties, so you don't have to declare it. But if necessary, it may 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.
  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 hero is predefined with a simple `Container` property taking `object`s with no `Limits`. It seems natural to use that as the "inventory" of the player, the storage for everything the player is picking up and carrying around. You will probably need to handle carried items in some manner, and the pre-declared container is one suggestion. You can also redeclare the `Container` property of the hero so that it suits your needs.

3.7.5. 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 at locations
- `Exit`s can only lead to locations and only locations can have `Exit`s
- the `Start` location must be a `location`
- locations can't have `Container` properties
- `Verb`s 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.

3.7.6. 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 Section 3.10, "SYNTAX Definitions") in this player command is the integer 12. This parameter is automatically considered an instance of the predefined class `integer`.

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

3.8. 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.

3.8.1. 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.

Table 3.2. Properties Inheritance

Property	Inherited as
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.

Property	Inherited as
Initialize	Accumulated. Inherited `Initialize` clauses are executed first so that the base classes may do their initialisation 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 Section 3.11.6, "Verb Qualification") 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 `Limits` of a container 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, `Verb`s, `Exit`s and `Script`s 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 legal or not.

3.8.2. Initial Location

Where 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 Section 3.19, “WHERE Specifications”) are allowed when describing the initial location of an instance.

```
The chest IsA object At treasury
...
```

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.

3.8.3. NAMEs

By default, the identifier (“author name”) of 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 are used when the instance is presented to the player, and the player can use them in his commands to refer to the instance. For example:

```
The south_door IsA object At south_of_house
  Name door
...
```

```
The south_of_house IsA location
  Name 'South of House'
...
```

The use of a quoted identifier in the last example causes the name to be a single string of text. (See Section 4.2, “Words, Identifiers and Names” for more details.) This works fine for locations, since the player usually does not need to refer to them in his commands; but a more sophisticated mechanism is available for things which the player needs to interact with (i.e. things, objects and actors).

```
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 wants to refer in a command to the object with the author name (identifier) `chair3`, he may use just “chair” if it’s the only accessible object with “chair” as its noun, or he may distinguish between multiple chairs by also providing one or more adjectives to be more precise about which chair he means.

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.

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 clause will define adjectives and a noun, as described above. As a result, the player can use any of those 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, even “rusty dynamite”.) The first `Name` clause is used for building a default description, if necessary (see Section 3.8.7, “DESCRIPTION”).

The letter case used in the original words is preserved in the adventure output, but player input will always be matched without taking into account letter case (i.e. case-insensitively). This allows you, for example, to give capitalized names to people actors, which would then be shown correctly in the 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). Furthermore, this has the effect that an instance inheriting from a class defining a `Name` can also be referred to 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 apple ...
The McIntosh IsA apple ...
```

In this example, both pear and apple can be referred to using the word “fruit”. Both the Gravensteiner and the McIntosh would be apples, not only by name, but also by all other properties of apples.

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 Section 3.8.7, “DESCRIPTION”). 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 `Name`s. It will, however, be overridden by any existing `Mentioned` clause (see the section called “MENTIONED”).

The short form can be automatically transformed to a description (for instances that have no `Description`) by inserting the article (see Section 3.8.8, “Articles and Forms”) and the short form in a default message. In the following transcript

example, output of the article is shown in bold, and the short forms in italic, 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 the section called “LIST Statement”).

3.8.4. 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 which pronouns each instance can be associated with.

```
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.

The predefined class `entity` defines the pronoun “it” (or equivalent for other supported languages).

3.8.5. Attributes

An attribute is a labelled value that instances have. Attributes declarations are placed either inside a class definition (in which case they will apply to all instances

of that class or any of its sub-classes) or inside an instance declaration (in which case only that instance will have those attributes, unless it's overriding inherited attributes with new values). An attribute declaration, or a set of declarations, is introduced using one of the keywords:

```
is = 'IS'
   | 'ARE'
   | 'HAS'
   | 'CAN'
```

And the actual 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.

Combining the keywords with well chosen attribute names can give natural reading to your attributes:

```
The rats Are hungry
The cowboy Can shoot
The chest Is heavy
The combination_lock Has numbers {1,2,4,8}
```

If you want some attributes to be present on every instance of a given class, then you must declare them in that class. E.g. to declare a Boolean attribute that will be shared by all instances of the `animal` class, 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 a different value in a particular instance, you must declare it specifically in that instance and

assign it the 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 its name preceeded by 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 type attributes are mainly intended for saving string parameters from the player input, like in:

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

They are not intended for storing long strings of descriptions, especially not as attributes to classes, as they (in the current implementation) require memory and take 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 store 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 initialise 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 either 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 initialised with a class identifier instead of a reference to an instance. This makes the attribute an *abstract* attribute, since it is defined but not initialised. Any instances inheriting from this class must then initialise the attribute, either explicitly or indirectly (by initialising 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.
  ...
```

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 either integers or instance references. Initial members must be listed in the declaration of the set. See Section 3.4.4, “Set Type” for details on the set type.

The type and class of allowed members is inferred from the actual values 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 initialise 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.

> 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 initialise the set with a single value of the correct type and remove that value in the `Start` or `Initialize` sections.

Inheriting Attributes

Attributes are inherited like any other property. An attribute declaration employing the same name of an attribute already present in any ancestor of the instance or class will inherit the type of that attribute, for you cannot change its type in subsequent declarations. This means that any declaration of a value different from the inherited one must therefore follow the rules of type compatibility for assignment. (See Section 3.4.5, “Type Compatibility”.)

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 initialised, 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.

3.8.6. INITIALIZE

The attributes of an instance can be initialised 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. Of course general statements are also allowed so any prerequisites can be catered for.

```
Initialize
  Set first_course of This To Random In first_courses Of menu.
  Set second_course of This To Random In main_courses Of menu.
  Set third_course of This To Random In desserts Of menu.
```

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

If the `Initialize` clause is inherited it will accumulate all clauses with clauses from base classes executing before the clause from the subclass. This lets the base classes do their initialisation before the initialisation of the more specialized, class or instance is performed.

3.8.7. 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 which base class the instance inherits from, 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 Section 3.7, “Instances” for descriptions of what inheriting from the predefined base classes means.

The `Description` should not change any game state since it might not always be executed depending on the settings of `Visits`. In particular, the `Description` clause 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*, 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 for 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 Section 3.11.3, "Verb CHECKs" 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 Table 3.2, “Properties Inheritance”. The actual descriptions are left for the instances.

If multiple `Description Check`s 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 the 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.

> 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.

3.8.8. Articles and Forms

The syntax for articles and forms is:

```
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` `Article`s and `Form`s can be used to define how an instance is printed in its definite, indefinite 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.

`Article`s and `Form`s can of course, be inherited.

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 these via an `Add` statement.

ARTICLE

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 displayed as the result of acting on multiple objects, e.g.:

```
> 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
  ...
```

Instead of:

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

This will produce the expected:

```
There is 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.

Both `Article` and `Form` are inherited as a single property. That means that an instance may override its inherited form using either of the two forms, regardless of how its parent defined the form.

Printing

You can use various forms of the `Say` statement (see the section called “SAY Statement”) to choose in which form the instance will be presented. In addition, the embedded parameter references allow selection of the form (the section called “String Statement”).

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 cannot 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!
```

A `Mentioned` clause declared on a class will override the names of any instance that inherits from it.

3.8.9. CONTAINER Properties

An instance can also be a container. This is declared by using 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.
```

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 its contents. You can check the state of the `opaque` attribute like any other Boolean attribute.

The predefined `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. By default the contents of the container will be listed unless it is empty or opaque. If the default description has been overridden, the author will have to explicitly decide on how to handle the container aspect and its content.

LIMITS

The `Limits` clause of the `Container` property declaration makes it possible to define limitations on *what* and *how much* can be put in any given container (as a class or a instance). These limitations are enforced by the interpreter during gameplay.

```
limits = 'LIMITS' limit {limit}

limit = limiting_attribute 'ELSE' {statement}

limiting_attribute = attribute_definition
                   | 'COUNT' integer
```

If any of these limits are exceeded when trying to put (i.e. `locate`) anything inside the container, the statements in the corresponding `Else` part will be executed and the player turn aborted. In fact, these checks are performed because of the execution of a `Locate` statement (usually, but not limited to, 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 by a `Locate` statement triggering limitations.

Each `Limits` specification requires either a limiting attribute or the special meta-attribute `Count`. Multiple limitations can be defined on the same container.

In case of a limiting attribute, it must be a numeric attribute defined on the class taken by the container (by default, `object`), and its implications are that the sum of this attribute on all objects in the container cannot exceed the specified value.

As for the special attribute `Count`, it defines a limitation on the number of instances allowed in the container.

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

The `Count` limit considers all instances directly in the container. This might differ from the number of instances listed e.g. if the container takes `Thing`s (which are not automatically shown to the player). E.g. if a container contains one instance of `Thing` and one instance of `Object` a listing will only mention the object, but the `Count` limit will still consider the count to be 2.

The calculation of a limiting attribute always sums the values of that attribute for all contained items recursively. This includes the container itself and the added item, if it happens to be a container too. In effect the calculation is transitive (see the section called “Transitivity”). As described in the section called “Containers Containing Containers”, this process might involve instances that do not have the limiting attribute, in which case these are simply disregarded from the sum.

If an item is being located into a target container which also happens to be inside a container, the limits of that “outer” container are also checked. This is repeated so that the limits of any containing containers are checked.

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

HEADER and ELSE

Syntax:

```
header = 'HEADER' {statement}

empty = 'ELSE' {statement}
```

`Header` is used when the contents of the container is listed. It is intended to produce something like:

```
"The box contains"
```

or:

```
"You are carrying"
```

It is followed by a list of instances mentioned. the section called “MENTIONED” 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 output will be equivalent to:

```
Header
  "The <container> contains"
Empty
  "The <container> is empty."
```

(<container> is replaced by the actual name of the instance.)

EXTRACT

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

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

The `Extract` clause, including optional `Check` and `Does` clauses, allows the author to prohibit the extraction of an item from the container depending on

some condition. If the `Check` is present, it works the same way as for `Verb`s (see Section 3.11.3, "Verb CHECKs"), 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 its `Else` clause, and then abort the move. The `Does` clause will be executed if the optional `Check` passes, or if there was no `Check`.

An `Extract` clause without a `Check`, but with a `Does` clause, executes the `Does` clause and then allows the extraction to take place. So, in a way, `Check`s, if triggered, prevents the extraction, and the `Does` clause amends to it, being an extensions of the normal case, much like the `Check` and `Does` clauses for `Verb`s (see Section 3.8.10, "VERBs"). The second form of the clause, with just the statements, is equivalent to an `Extract` with only a `Does` clause.

An example use of the `Extract` clause is to prohibit, put restrictions on, or modify the behaviour when the hero attempts to take things carried by another actor.

```
The waiter IsA actor
  At bar.
  Is Not annoyed.
  Description
    "A slow-moving, traditionally dressed waiter is here."
    List waiter.
    If waiter Is annoyed Then
      "He is rather annoyed."
    End If.
  Container
    Header "The waiter is carrying"
    Else "The waiter is empty-handed."
    Extract Does "The waiter is annoyed by your presupposition."
      Make waiter annoyed.
End The waiter.
```

Both limits and the `Extract` clause will, if triggered, interrupt the execution of the triggering `Locate` statement, and does not continue the execution. Currently there is no way to programmatically check if this will happen. The author needs to be aware of this effect and if necessary, take precautions.

3.8.10. VERBs

`Verb`s declared inside an class or instance are inherited in the same way as other properties. See Section 3.8.10, "VERBs" for a description on how to declare verbs.

The verbs in a class or instance will only be a candidate for execution if the instance bound to a parameter is of the corresponding class, or is the instance. See Section 3.11.7, “Verb Execution” for a detailed explanation.

3.8.11. ENTERED

Syntax:

```
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, the `Entered` clause 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.

The `Entered` clause can also be used to restrict the movements of actors other than the `Hero`. (The hero’s travels are controlled by `Exit` checks as described in *EXITs*).

If 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.

The actor is located at the location before the clause is executed so `Current Location` will be the location having the clause.

`Entered` clauses are inherited and locations can be nested (see *Locations*). The order of execution is explained by the following table:

Table 3.3. Order of Execution of ENTERED in Nested Locations

	Outer Region	...	Current Location
Base class	Outermost	↓	↓
:	↓	↓	↓
Leaf class	↓	↓	↓
Instance	↓	↓	↓

This means that the first `Entered` clause to be executed is the clause in the base class of the outermost location, if any, then moving down the inheritance of the outermost. After that, any parent classes for any intermediate locations are

considered in the same way. Finally, running any `Entered` clauses in the parents of the new location, ending with the clause in the location itself.

The `Entered` clause is only executed when the actor is entering the location. This goes for *all* actors, not only the player/hero. The actor will be at the location when the clause starts to execute.

If it is the hero that is moving, the `Description`, including the normal header containing the location name, of the new location will be executed *directly after* the `Entered` clause.

3.8.12. EXITs

To build a traversable world of locations, they must be connected. This is done using `Exit`s. 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*.

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

Two interconnected locations might be declared like this:

```
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.
```

If an exit is declared from one location to another, and you want there to be an exit in the opposite direction, you have to define the reverse passage. It is not created automatically.

Exits are only allowed in classes or instances inheriting from the predefined class `location`.

3.8.13. SCRIPTs

The `Script` is the way actors perform 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 starts following a script, it will continue with one step after the other, with all the other actors, including the hero, taking turns.

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 ...
...
```

An actor continues executing its script until:

- it reaches the end
- another `Use` statement is executed for that actor
- the actor is stopped using the `Stop` statement

- something fails

There are a few things that might fail when an actor executes. One example is an `Extract`, which means that something is removed from a container. As containers may define `Extract Check`s that action might be prevented. This means of course that that step is aborted, but also that the actor is automatically stopped, so no further steps from the script will be run. The author is responsible for handling this, e.g. by using rules to ensure that the condition is detected and handled correctly.

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, at the next move, to wait a number of moves before it's 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.

3.9. Additions

In certain circumstances, you need to add properties to a class after it is defined. A simple example of this is adding attributes to the predefined classes. For this purpose, the `Add` construct is available. It follows the grammar:

```
addition = 'ADD' 'TO' 'EVERY' id
               {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.

3.10. SYNTAX Definitions

The `Syntax` construct is used to specify the allowed structure of the player's input. Each definition defines the syntax for one `Verb`. The effects triggered by the player input are declared using the `Verb` construct (see Section 3.8.10, "VERBs").

```
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). Parameters may be in any position, including the first. Bare in mind that a syntax definition containing only parameters might be tricky for the interpreter to match as you intended, for the complete set of allowed input can easily become ambiguous.

```
Syntax
  quit = 'quit'.
  examine = 'examine' (obj).
  command_north = (act) 'north'.
  unlock_with = 'unlock' (l) 'with' (k).
```

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 Section 5.2, "Player Input" for general details on 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, and the parameter identifiers in the `Syntax` declaration will now refer to those entities. Any references to attributes, etc., will be done in the instance referred by the parameter.

```
Syntax open = open (obj).
  ...
  If obj Is open Then ...
  ...
```

In the above example, the parameter `obj` can be used in the declaration of the `open` verb and, at execution time, it will refer to the actual instance it was bound to. Consider the `unlock_with` syntax declaration of the previous example, the following table illustrates which instances the parameter identifiers `l` and `k` will be referring to in the actual game:

Player input	`l`	`k`
> unlock the door with the key	door	key
> unlock the bottom drawer with the rusty knife	bottom drawer	rusty knife
> unlock the skeleton with the tiny blue chair	skeleton	tiny blue chair

This, of course, provided that there is an instance that will match the player input, given the adjectives and nouns in the input and those in the instance declaration.

It is allowed to define multiple syntaxes for the same identifier (verb). See *Syntax Synonyms*.

3.10.1. Indicators

Following a parameter, indicators are allowed in `Syntax` declarations.

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

There are two indicators available (*multiple* and *omnipotent*):

- `*` — 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 Section 5.2, “Player Input”).

- `!` — The parameter (the instance the player refers to in this position in the syntax) need not be present at the current location. The default behavior of 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 be able to refer to objects and actors that are not present (e.g. in a verb like `talk_about`) this *omnipotent indicator* can be used to force the interpreter to accept references to any object or actor.

An example:

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

This shows the syntax definitions for the verbs `take` and `drop`; where `take` also allows multiple objects. This would make the following inputs possible:

```
> take everything except the pillow

> drop the vase
```

Refer to Section 5.2, “Player Input” for details on multiple parameters references in player’s input (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, as it doesn’t include the *multiple indicator*. Another example, this time using the `!` indicator:

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

Even if the robber or the key are 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 Section 5.2, “Player Input”.

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”.

3.10.2. 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'
```

Any predefined or user defined class can be used.

Don’t forget that also `literal`, `integer` and `string` are predefined classes (see the section called “The Predefined Classes”).

The following example describes the syntax for a verb that only allows `object`s 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 belonging to 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 provide multiple restrictions, even for the same parameter; but if they refer to the same parameter they must be presented in increasingly restrictive order. 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 if 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, e.g. you can only use a parameter in a `List` statement if it has been restricted to 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`.

3.10.3. 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.

3.10.4. 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 represent 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.

When a `Verb` declared in an instance or a class has no `Syntax` counterpart, it automatically receives a default syntax of the common verb/object type corresponding to the second template above. This is a reasonable syntax for many cases and restricts the parameters 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 Section 3.20, "WHAT Specifications" for the implications of this.)

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.

When a `Verb` declared in a location has no `Syntax` counterpart, it will receive a default syntax restricting the parameter to the `location` class, which probably is not what you wanted.

3.10.5. 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.
- Instances of `thing` and its subclasses at the current location, including any nested locations, are in scope.
- Instances of any class inside a container that is in scope are in scope too, unless that container is opaque and closed. (See *Container Properties* for details.)
- If the syntax indicated a parameter as omni-potent, any instance is in scope for that parameter position. (See *Indicators* for details.)

If the interpreter finds multiple instances matching the input (the set of given adjectives and noun), it will try to disambiguate giving preference to present instances — i.e. at the location of the hero. If there are still multiple candidates left after this, the interpreter will print a message and abort execution of the current command.

When all parameter positions in the syntax have been resolved this way, the restrictions are executed.

3.11. VERBs

A `Verb` declaration specifies what to check and the effects of something the player does (i.e. commands using a syntactically legal input).

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

verb_body = simple_verb_body
          | {verb_alternative}

simple_verb_body = [check] [does]
```

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

Verbs can be declared at two different levels:

- global (outside any other declaration), or
- inside a class or instance declaration, including inside an `Add` construct.

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 include any parameters in their syntax declaration.

A verb declaration inside a class or instance definition 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 example above) will become the default player words that can be used to invoke the verb. But if a `Syntax` is declared for the `Verb` (see Section 3.10, “SYNTAX Definitions”), 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 shorthand for declaring identical checks and bodies for all the verbs in the list. This will create synonymous actions for different verbs at the level of the verb declaration. 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.

3.11.1. META VERBs

Any action from the player usually takes one "tick" in the default simulated game time. Sometimes you want a player command to *not* consume a "tick", for example administrative commands like "help", "score" etc.

You can achieve this by attaching `Meta` in front of the verb definition:

```
Meta Verb 'score'
  Does
    Score.
End Verb.
```

If your `Verb` has multiple definitions (e.g. for various classes) applying `Meta` to any one of them will make the verb a meta verb, meaning that if the player uses that verb in any context and on any instance, it won't consume a tick, even if that particular definition did not have the `Meta` property explicitly expressed. A library might decide that `'score'` was a meta verb and there is nothing you, as an author, can do to override that short of editing the library source.

Furthermore, a `Meta` verb does not trigger evaluation of *rules* and `Event`s either, so they are genuinely "outside" the game and should only be used with verbs that are not considered part of the players progression inside the game.

The `Meta Verb` feature only applies to the built-in timing mechanism known as "ticks", where every player command counts as 1 tick. It is possible to implement your own timing mechanism, in which case the `Meta` does not help.

3.11.2. VERBs in Locations

A special case is a `Verb` declared in, or inherited by, the `location` where the player is currently 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 the following location will disappear:

```
The high_wire IsA location
```

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

3.11.3. Verb CHECKs

Syntax:

```
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 in `Check` are executed. Which checks to run, is determined by the class of the instances bound to the verb by the parameters. All checks in the inheritance tree are tried by starting at the base class. This way, the more general checks are tried first, followed by increasingly more specific checks.

A typical use of `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 becoming an unconditional `Check`. This is useful for preventing certain actions, for example at specific locations, 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. So in the above example, the verb “jump” will always result in “You can’t do that here.” at the location `jumpless`.

> `Check`s are intended to take care of any *exceptions* for executing the normal case. The normal, or positive/affirmative, case should be handled by the `Does` clause.

With this in mind, `Check`s are also used when handling the user input **ALL** (see Section 5.2, “Player Input” 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 limits the handling of **ALL** to executing only the verb bodies of objects that are reasonable, i.e. whose `Check`s will not fail.

For example assuming the above definition of the verb `take` and a location containing the objects `ball` and `box`, where only the ball is `moveable`, the player input:

```
> take all
```

would result in **ALL** representing only the ball. See Section 5.2, “Player Input” for an explanation of the player view of this.

3.11.4. DOES Clause

Syntax:

```
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 default order is to execute first 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 unfrequent 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.
```

3.11.5. Verb Alternatives

Syntax:

```
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 allow differentiating between the instance occurring at different parameter positions 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 which checks to test for, and which verb bodies to execute, the parameters are examined in turn according to the algorithm described in the section *Verb Qualification* below. Each instance may have different verb bodies executed depending on which position it occurred at (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`, the resulting actions could differ depending on whether it occurs as the direct object (`o`) or as the indirect object (`w`). To implement this, the `Verb` body for `break_with` should differentiate between the two cases. 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 verb body will be considered for all positions in the syntax. The compiler will warn for this when it detects a syntax that allows the class or instance to occur at every parameter positions.

3.11.6. Verb Qualification

Syntax:

```
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 the current location) any definition of the verb in the current location (including inherited verb bodies) are considered first. Then, the verb bodies in the parameters on which the verb was applied are examined (in their order of appearance in the syntax definition) to find and execute their verb definitions. For each parameter, its most general definition is executed first, then verb bodies down the inheritance tree are executed 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. These qualifiers alter the order of execution, as described in full detail below.

3.11.7. Verb Execution

First all parameters are evaluated according to the `Syntax` restrictions (see *Parameter Restrictions*). Then, if they passed, the `Check`s of all verb declarations are evaluated (see Section 3.11.3, “Verb CHECKs”). Finally the `Verb` bodies are executed in the normal order, as explained in the following table.

Table 3.4. Order of Execution of Verbs

	Outer Region	...	Current Location	First parameter	...	Last parameter
Base class (entity)	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.

> 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 column 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 body to be executed. This can be done using the `Only` qualifier (see Section 3.11.6, "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` then 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 executed now (and if the qualifier was `Only`, execution is also 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 then no furhter executions will be taken into consideration). 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 inside out, find any `Before` or `Only` definitions and execute them (stop if `Only` found).
- From the outside in, 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 inside out.

The second item in the above list is equivalent to the normal order of execution, without any qualifiers.

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.

All `Check`s 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 examine
    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.

3.12. 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. however, they can not refer to any parameters, since no verb is executing, nor to the `Current Actor`. See *Run-Time Contexts*.

Events may be scheduled and cancelled with the `Schedule` and `Cancel` statements (see *EVENT Statements*).

3.13. Rules

Syntax:

```
rule = 'WHEN' expression ('THEN' | '=>')
           {statement}
       ['END' 'WHEN' '.' ]
```

A *rule* is an arbitrary expression, which, when true, results in the execution of some given statements. Rules can only be declared on the global level (not inside classes or instances). The main intended use of rules is to detect particular situations and then trigger some action. Typically they can be used to make things happen when certain situations arise, such as starting an actor when the hero enters the cave.

Here is an example that investigates if the hero is in the cave and if so, activates the monster:

```
When hero At cave And monster Not active Then
  Use Script hunting For monster.
End When.
```

The expression that is tested may of course have any level of complexity:

```
When hero At cave
  And (monster Is hungry Or monster Is angry)
  And sword Not In hero
=>
  Use Script eat_hero For monster.
End When.
```

Each actor action and event execution is considered atomic (it can't be divided into smaller parts). All rule conditionals are evaluated after each actor (including the player) has acted (script step and player command respectively) and after each event has executed. In effect this will mean that a change in state will be detected almost immediately, if there is a rule for detecting that change.

The statements within the rule are triggered when the condition *becomes* true. In the first example, this means that if the monster is not active, the statements will be executed when the hero enters the cave (`hero At cave` becomes true). A rule body can never be executed twice in succession unless the conditional has been evaluated to false in between. In the example above, the triggering of the `hunting` script for the monster will not happen again unless either the hero has left the cave and entered it again, or the monster has been `active` and then become `NOT active` again.

The use of parameters, `Current Actor`, `Current Location`, `Here` and `Nearby` is not allowed in rules conditionals or bodies.

Rules are executed at no location. Therefore, within *rules* it's not possible to communicate directly with the player via output statements (since the hero

cannot be where the rule is executing, see Section 3.18.1, “Output Statements”). Triggering an `Event` that handles the output intended for the player is the recommended solution to this.

The following is a complete game using a rule:

```
The kitchen IsA location
  Exit x To kitchen.
End The kitchen.

When Count IsA actor, At kitchen = 1
  Then Schedule whee At actor After 0.
End When.

Event whee
  "Whee!"
End Event.

Start At kitchen.
```

In this example the *rule* conditional (i.e. `Count IsA actor, At kitchen = 1`) is using an aggregation (`count`, see Section 3.21.13, “Aggregates”) over two filters (see Section 3.22, “Filters”) that will count the number of `actor`s at the `kitchen`, and when that number becomes one, the rule will trigger and execute the statements, in this case scheduling an event that handles the presentation of the output to the player.

Again, remember that rules are checked after each actor has moved. What happens if there are more actors in play and they move in and out of the kitchen, is left as an exercise to the reader.

3.14. SYNONYMS

Syntax:

```
synonyms = 'SYNONYMS' {synonym_declaration}

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

A `Synonyms` declaration declare words that, when used in player input, are always interchangeable. 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 3.9, “Additions”) 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 Section 5.2, “Player Input”).

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 the section on *VERBs* for details on verb declarations.

It is also possible to define multiple names for an instance to achieve other effects similar to synonyms. See Section 3.8.3, “NAMEs” for a description of this.

3.15. 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 Section C.1, “Input Response Messages”).

To better adapt the user dialogue 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 would be:

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

If the above where used in the source for the 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 C, *Run-Time Messages*.

3.15.1. MESSAGE Parameters

`Message` sections must be declared at the global level, but to make it possible to create high-quality messages, parameters are available in message sections. Which parameters are available vary depending on the message, the details for each message is available in Section C.1, “Input Response Messages”.

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 predefined `entity` class. Any attribute available for this class will be available in message sections with instance parameters.

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.

3.16. PROMPT Section

The `Prompt` section allows you to customize the way players are prompted for their input.

```
prompt = 'PROMPT' {statement}
```

The default prompt for player input, which will be used if no `Prompt` section is declared, looks like this:

```
>
```

Using the following `Prompt` section it can be set to something else:

```
Prompt "What now?"
```

Then the player will of course see:

```
What now?
```

In fact, the `Prompt` section allows any statements, not just strings. So you can have the prompt change during the game.

```
Prompt
  "Hello" Say hero. "!$n"
  "Where do you want to go from"
  Say Current Location. "?"
```

This will give the following output:

```
Pirates Bay Harbor
You can see the town of Pirates Bay to the north, and your ship is at the docks, to the south.
```

```
Hello Jack Sparrow!
Where do you want to go from Pirates Bay Harbor?
```

The prompt section is global and applies to the whole game. There is currently no way to dynamically customize the prompt except by using the statements inside the `Prompt` section itself.

3.17. 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 Section 3.19, “WHERE Specifications”) 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.

3.18. Statements

3.18.1. Output Statements

There are various 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 and you as an author would need to cater for this. Instead 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, 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.

Whenever an output statement is executed, the result will be printed on the players terminal with the following important exception: if an output statement is executed at a game location other than the hero's location, the output won't be shown. This important feature 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. But see *Distant Events* for a solution in case the hero need to be informed about things happening in other locations.

String Statement

Syntax:

```
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. See also Section 4.4, "Strings" for some detailed descriptions on the definition of strings.

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
$$     Escape from automatic space insertion and capitalization
$a     The name of the actor that is executing
$l     The name of the current location
$v     The verb the player used (the first word)
$_     Print this as a '$' if in conflict with other symbols
```

You might want to output "$400" but "$4" will be interpreted as the indefinite form of the fourth parameter, as described below. So you need to use the "_" to make that happen (`$_400`).

The following can be used to refer to parameters while executing a verb, but the `Say` statement (see below) is safer and preferred whenever possible:

```
$<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>
$o        The current object (first parameter)
```

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. See *SAY Statement* below.

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.

STYLE Statement

Syntax:

```
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.

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

DESCRIBE Statement

Syntax:

```
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 instances 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

Syntax:

```
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 its name 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 in a literal printout if the parameter was a `string` or `integer`.

```
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, then the definite form will be employed, which is usually the short form preceded by a definite article. Correspondingly, the use of `An` indicates an indefinite form. A third form, using `No`, is also available. It indicates that the negative form (as defined by the negative article or form) should be output. Refer to Section 3.8.8, “Articles and Forms” for a description of the definite/indefinite articles and forms. Finally, the `It` form will print the pronoun associated with the instance.

LIST Statement

Syntax:

```
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.

3.18.2. Multimedia 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 multimedia statements, but a particular interpreter might not do so. Most GLK-based interpreters will support them, but other kind of interpreters might too. The game will still play fine, but the multimedia resources will be silently 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 in 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

Syntax:

```
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 *Filenames*) is usually required.

Alan currently supports the JPEG and PNG formats only.

PLAY Statement

Syntax:

```
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 *Filenames*) is usually required.

Alan currently supports the MOD and AIFF formats only.

3.18.3. Manipulation Statements

LOCATE Statement

The `Locate` statement is a way of transferring instances to new locations. When executed, the indicated instance will be placed at the specified location.

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

For a description on how to specify **WHERE**, see Section 3.19, “WHERE Specifications”. 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 no more statements being executed. So, if a player command should result in locating an object inside a container, it's better to place the `Locate` statement as early as possible, as this enforces the `Limits` checks at the beginning of this player turn.

A very special third case is locating a location at another location. Locations can be nested this way, 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

The `Empty` statement locates all instances currently located inside the given container (instance with the `Container` property) at a certain location.

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

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

The `Strip` statement is used to manipulate the contents of strings.

```
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
```

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.
```

3.18.4. 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.

An important case is when a `Schedule` statement without a **WHERE** clause is executed inside a *rule*. Since rules are executed at nowhere so will the event. This means that any printout will occur nowhere, and will thus be invisible to the player.

The **WHERE** expression can be any expression yeilding an instance type, such as the identifier of a location or actor, an attribute referring to an instance. The event will occur wherever that instance is when the event occurs. The event will "follow" the instance.

There is an interesting difference between the following two statements:

```
Schedule explosion At hero After 1.
Schedule explosion At location Of hero After 1.
```

The first statement will schedule the event at the hero and will execute where the hero is after one turn. The second will schedule the event at the location where the hero was when the statement was executed, even if the hero has moved after that.

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.

The event can be specified by giving an event identifier or referring to an attribute of `Event` type.

CANCEL Statement

`Cancel` will remove the referenced event from the queue of scheduled events.

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

Canceling an `Event` which is not currently scheduled is not considered an error.

```
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.

3.18.5. Assignment Statements

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

MAKE Statement

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

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

something = ['NOT'] id
```

Examples:

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

INCREASE and DECREASE Statements

Syntax:

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

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

```
by = 'BY' expression
```

The `Increase` and `Decrease` statements modify the value of numeric attributes by increasing or decreasing it by the value of the expression given in the optional `By` clause. If no `By` clause is specified the attribute is changed by one.

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

SET Statement

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

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

Examples:

```
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 follows the type and subclass compatibility rules. For example, you can only assign:

- **integer type expressions** to an integer attribute:

  ```
  Set intAttr To 4. -- Correct if intAttr is of Integer type
  Set intAttr To "hi". -- Incorrect
  ```

- **instance type expressions** to a reference attribute whose class is a parent of the instance in the expression:

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

- **set type expressions** to a Set type attribute if all members in the expression are instances of a subclass of the member class of the target attribute. Here are some examples, given the natural types of the instances:

  ```
  Has friends {monica, ross, chandler, rachel, phoebe, joey}. -- 'person's
  Set friends Of mine To {book}. -- Incorrect, probably not a 'person'
  Set friends Of mine To {}. -- Correct, an empty set is always OK
  ```

```
Set friends Of mine To {suspect Of detective}. -- Maybe correct
```

INCLUDE Statement

The `Include` statement is used to include a new member in a Set.

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

Typically, this is used to add an instance or value to a collection of similarly typed members. See Section 3.4.4, “Set Type” for an explanation of the Set type. Attempts to include a member already present in the Set will be silently ignored without generating 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

The `Exclude` statement is the reverse of the `Include` statement. It removes a member from a Set.

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

Attempts to remove something not included in the Set will be silently ignored; therefore, it’s guaranteed that after executing this statement the instance or value of its expression will not be in the Set.

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

3.18.6. Conditional Statements

In Alan there are two conditional statements: the common `If` statement and the `Depending On` statement.

IF Statement

The `If` statement is essential for varying output and otherwise change the activities in the game.

```
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 expression is evaluated (see Section 3.21, “Expressions” for details and examples of expression) and if it evaluates to true, the statements following the `Then` keyword 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 if there are no `ElsIf` clauses, the statements following the `Else` keyword 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.
```

`If` statements which have an `IsA` expression (see Section 3.21.6, “Class Expressions”) are particularly important. As an `IsA` expression test for the class of an instance, an `If` statement like:

```
If i IsA actor Then ...
```

will quarantee that for any statement inside the `Then` part of that statement, the `i` will be of the class `actor`. This means that references to attributes, container

properties, actor scripts, etc. as if `i` belongs to the class `actor`, even if that was not known outside of the `If` statement.

A typical example where this is helpful is inside verbs where parameters can be restricted to more general classes by the `Syntax` and the `Verb` body can still perform specific actions only allowed on more specialized classes.

Another example would be looping over a set of unknown instances:

```
For Each e IsA entity, Here Do
  If e IsA actor Then ...
  If e IsA object Then ...
  If e IsA container_object Then ...
End For.
```

DEPENDING ON Statement

The `Depending On` statement enables us to select one of a number of possible conditional cases depending on an expression.

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

case = right_hand_side 'THEN' statements
     | 'ELSE' statements
```

The **expression** part can be any expression. The **right_hand_side** part is the right hand side of any valid expression. When combined with **expression** (as the left hand side of the expression) they will form a complete expression, which is evaluated.

A simple example of the `Depending On` statement:

```
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 purpose of this example is to test the `weight Of obj` and select one of the cases depending on its value. If it's equal to one, then 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 specified order. At most, one case will be executed. In the example, a weight of ten will render as "carryable".

The above tests are thus 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.

3.18.7. Actor Statements

Actor statements are statements that are used to control actors.

USE Statement

The `Use` statement starts execution of a given `Script` for a given actor.

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

The `For actor` clause is optional when writing code within a certain actor — in this case, the statement applies to the actor (instance or class) containing the code.

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

For the **actor** clause you can use an expression such as a simple identifier, a parameter reference or a reference attribute.

STOP Statement

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.

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

```
actor = expression
```

The most common case is the direct reference to an actor using its identifier. More complex expressions resulting in an actor type value, such as a parameter reference or a reference attribute, can be used as the **actor** clause.

3.18.8. 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 its semantics are similar to a `Syntax` parameter. It will be dynamically bound to instances, one for each repetition. Inside the loop body (i.e. 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 being of the integer type, and a loop range encompassing the values of the two expressions (included). Otherwise, the loop variable will be of instance type and will consecutively assume the value of each instance fulfilling the filters. See Section 3.22, “Filters” for an explanation of filters.

Within the repetition, any references to the loop variable will refer to the bound instance, 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.
```

3.18.9. Special Statements

QUIT Statement

`Quit` prints a question giving the player the choice to restart the game, restore a previously saved game or to quit the game. Any scoring or other printouts have to be made explicitly before executing the `Quit` statement.

LOOK Statement

`Look` describes to the player the current location and what it contains (which location is going to be the current location depends on which context `Look` is being executed in). First, the location name is printed out, then the `Description` part for the location is executed. If you do not want the name of the location to be included, you can use a `Describe` statement instead.

Then, all object and actors at the location will be automatically executed by means of an implicit `Describe` for each of them. Any objects or actors that were explicitly described using a `Describe` statements, will be excluded from this automatic `Describe`.

The equivalent of a `Look` is automatically performed when the hero enters a new location.

> As the player will only see output generated at the same location as the hero, a `Look` executed by another actor at some other location will not be seen by the player. See Section 3.18.1, “Output Statements” for more details on this important consideration.

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'.
```

In the above example, the verb identifier `'restore'` was enclosed in single quotes (i.e. a quoted identifier) to prevent conflict with the reserved keyword `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.

The `Score` statements assume a simple scoring model: a number of specific actions are necessary to complete the game, and all of them are necessary to achieve the maximum number of points. Negative scores are not allowed and once a score has been awarded it cannot be revoked, and it can't be awarded twice either. For adventures requiring a more complex and varied scoring system (especially 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 its long description is presented again:

```
visits_statement = 'VISITS' count '.'
```

The value of the argument (**count,** which must be an integer number) 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. Therefore:

```
Visits 0.
```

will always show full descriptions (which is also the initial setting).

The classic and familiar commands **verbose**, **brief**, etc. can be imitated using different values in the `Visits` statement.

The handling of descriptions is rather conservative in that it also takes modified attributes into account. If the `Visits` calculation would indicate no full description, a modified attribute of the location will cause the full description to be shown the next time. That attribute might have caused the description of the location to change. So the `Visits` handling ensures that the player will "see" all changes to the location. If you set attributes of the location every time they are visited, the built-in `Visits` handling will not work as you expect.

Associated with this is a predefined attribute, `visits`, that exists for all instances inheriting from `location`. It will count how many times the player have visited the location. For example:

```
The secret_cave IsA location
  Description
    "This is the secret cave. You have visited it"
    Say visits Of This.
    "times before."
End The secret_cave.
```

TRANSCRIPT Statement

Using the `Transcript` statement you can turn transcripting on or off.

```
transcript_statement = 'TRANSCRIPT' ('ON' | 'OFF') '.'
```

When transcripting is turned on all player input and game output is recorded in a file (or similar) which can be studied afterwards. Example uses are:

- reading the game output as a novel (player)
- comparing the output to output from previous versions of the game (author)
- comparing output to the output of a v2 game (porting from v2)

To enable player access to the transcripting function you need to implement global verbs:

```
Syntax script_on = 'script' 'on'.
Verb script_on
  Does
    Transcript On.
    "Transcript turned on."
End Verb.

Syntax script_off = 'script' 'off'.
Verb script_off
  Does
    Transcript Off.
    "Transcript turned off."
End Verb.
```

3.19. 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 then that context is the one used. See *Run-Time Contexts* for a detailed discussion, but examples include an event scheduled at a particular location, in

which case that location is `Here`. Note that `Here` is equivalent to `At Current Location`.

- `Nearby` means at any adjacent location. An adjacent location means that there is 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 identifier or expression. In particular, instances inheriting from `location` are allowed, which can be used to see if someone at that location can use an exit to get here.
- `Near what` has a similar meaning to `Nearby` except that it refers to some other instance (the **WHAT**) instead of the current location. The results is a truth value indicating wether that other instance is at a location which is nearby (has an `Exit` to) the location of the first instance.
- `At what` means at the location of the instance referenced by the **WHAT** specification (see Section 3.20, "WHAT Specifications"). Note that an instance is always `At` itself, i.e. `x At x` is always true. This can come as a surprise, especially if you try to aggregate or loop over instances. (See *Aggregates* and *Repetition Statements*.)
- `In what` must refer to a container and the expression refers to being inside 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 in containers inside other containers too) to evaluate the expression. See *The Whereabouts of an Entity* for more information about **WHERE** expressions.

Not all kinds of **WHERE** specifications are meaningful in every constructs. For example, `Nearby` and `Near` can not be used in a `Locate` statement, as it requires a specific location to locate to.

3.20. 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.
- `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 is currently executing. See *Run-Time Contexts* for more details.
- `This` refers to the instance in which the current code (e.g. a verb body or a script) is run. This can be used, for example, to test or set attributes in inherited code, thus testing or setting attributes in an instance while the code is defined in the class from which the instance inherits. 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 3.21.3, "Attribute References", might be used depending on its type and the context of the usage of the **WHAT** expression.

> Not all kinds of **WHAT** specifications are meaningful in every 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 In` statement (since it is illogical to locate locations in containers).

3.21. Expressions

The Alan grammar often 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.

3.21.1. 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 3.4, “Types” for details on types.

3.21.2. 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 value of the Set type 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.

3.21.3. Attribute References

A reference to an attribute can be used as part of any expression provided its type matches the semantics of the context. The type of the expression is the type of the attribute.

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

There are two formats available; the first one resembles plain English:

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

The second format is more compact, which might be preferable when referring to chains of attributes referring to other attributes. See *Reference Attributes* for an explanation on how this works.

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

It might help to read the `:` as a replacement for **'s**. In this example the detective must be known to have a reference attribute, `suspect`, which can only refer to instances of a class that have an attribute named `weapon`. It would be the same as:

```
Say weapon Of suspect Of detective.
```

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 ...
```

LOCATION OF

There is a particularly useful predefined pseudo-attribute, `location`, that can be used to query an instance of which location it is currently at.

```
Make location Of magic_lantern lit.
```

This attribute is predefined on all instances and is guaranteed to return an instance of the class `location`, and it will be the innermost location of the instance (bearing in mind that locations may be nested).

3.21.4. 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 Section 3.8.9, “CONTAINER Properties” 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 the section called “Set Type Attributes”.

The optional keyword `Directly` is only allowed if the expression refers to a container. The semantics are the same as for the **WHERE** expression, see Section 3.21.12, “The Whereabouts of an Entity”.

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 author’s responsibility to ensure that this is not attempted. As a safeguard against similar runtime errors, you should always surround a random member expression with an `If` statement that ensures that the Set or container is not empty. See *Aggregates* for descriptions on how to count members in a set or container.

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.

3.21.5. Logical Expressions

The `And` and `Or` operators are standard binary Boolean operators, meaning that the result of an expression is true or false depending on the right and left expressions, which must also be boolean values or expressions.

```
expression = expression ('AND' | 'OR') expression
```

For `And` both expressions need to be true for the expression to be true. If using `Or` either of them need to true. Otherwise the expression will be evaluated to false.

`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 ...
If o IsA treasure And (size Of o > capacity Of c
  Or thief Is greedy) Then ...
```

3.21.6. 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.

```
expression = something 'ISA' class_id
```

Example:

```
If p IsA object Then ...
If opponent IsA enemy Then ...
```

There is a subtle but very important side effect when checking for an instance class in an `If` statement like the above. Doing so will ensure that the instance or parameter being checked has all the properties. This holds true for all statements in the `Then` part of the `If` statement.

This is used by the compiler to allow references to attributes, scripts, container properties, etc. that otherwise would not be allowed.

A very common use of this is to restrict parameters in a `Syntax` to a more general class, like `thing` or `entity`, and then doing "manual" restrictions using `If` statements to ensure that their usage does not conflict with the actual properties of the instance. See also the section on *IF Statement*.

3.21.7. 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
```

3.21.8. 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 ...
```

3.21.9. 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.

3.21.10. CURRENT Entities

There are two particularly interesting entities that you might want to learn about. 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 contexts can be found in *Run-Time Contexts*.

3.21.11. 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 has 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 attribute on the `door`, because this is the instance running the code.

3.21.12. The Whereabouts of an Entity

An expression following the pattern below 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**.

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

Example:

```
If bottle In inventory Then ...
```

or

```
If hero Not Nearby Then ...
```

The forms available for the **WHERE** expression are described in detail in Section 3.19, “WHERE Specifications”.

The default 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. This implicitly transitive evaluation can also be made explicit through the use of the keyword `Indirectly`. This would result in exactly the same semantics, but explicitly expressed, which can be useful.

```
transitivity =
             | 'DIRECTLY'
             | 'INDIRECTLY'
             | 'TRANSITIVELY'
```

In addition, another 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 but 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 but not inside any other container, even if that container was in the inventory. See *Containment, Classes and Transitivity* for some background information.

> If the transitivity is not `Directly`, the compiler analyses the container to see which classes of instances might be contained transitively. This includes all existing instances of the class that the container takes, and if any of those are containers, those containers and all instances of the classes they take, and so on.

3.21.13. Aggregates

Aggregates are functions to calculate values from Sets of instances.

```
aggregate_expression = aggregate filters

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

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 Section 3.22, "Filters" 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 objects."
```

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.

3.22. Filters

Filters can be used to filter out only particular instances to loop or aggregate over.

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

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

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 Section 3.21.12, “The Whereabouts of an Entity” for details on the various forms of the **WHERE** expression.

Multiple filters can be listed by using a comma to separate them. 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.

Chapter 4. Lexical Definitions

4.1. Comments

Comments may be placed anywhere in the Alan source. There are two types of comments, a line comment and a block comment.

A line comment starts with double hyphens (`--`) and extends to the end of the line.

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

A block comment can be used to comment out longer sections of Alan code. The first (ignored) line of a block comment is a line where the first four characters are forward slashes, it may then be followed by anything. The block comment ends with the first line that has four slashes in the first columns. It may be follwed by more slashes, but nothing else.

```
//// This line starts a block comment ////
These lines will be
completely ignored.
//// So will this line ////
The block comment will end after the following line
////////////////////////////////
```

4.2. 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}
```

4.2.1. Quoted Identifiers

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

```
id = identifier
   | quoted_identifier
```

```
quoted_identifier = quote {any_character} quote
```

A quoted identifier starts and ends with single quotes and may contain any character (including spaces) except single quotes — if you need to include a single quote inside a quoted identifier, you must *escape it* by doubling it, e.g.:

```
The 'Bob''s House' IsA location. -- ID will be printed as "Bob's House"
End the.
```

When the Alan compiler encounters two consecutive single quotes inside a quoted identifier, it will treat them as if they were one single quote which is part of the identifier, and not as a the single quote *delimiter* indicating the end of the identifier.

In the above example, the identifier will be printed as “Bob’s House” in the actual game — the enclosing single quote delimiters being ignored in print, and the two consecutive single quotes printed as one single quote.

Any sequence of characters enclosed within single quotes can become an identifier, except inside strings (where single quotes are treated as ordinary printable characters).

A single-word identifier can be written within or without single quotes, indifferently (unless it’s a keyword). Therefore, `'someID'` and `someID` both refer to the same identifier in the source code.

4.2.2. Keywords as Identifiers

Quoted identifiers may also be used to create an identifier out of a reserved word such as `Look`, `The`, etc. This practice is known as *stropping*, and it allows the Alan compiler to distinguish between user-created identifiers and keywords in the source code, thus avoiding clashes that would lead to a compiler error.

This is useful (indeed, required), for example, in the definition of the verb “look”:

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

Whenever an identifier contains an Alan keyword, it's mandatory to use a quoted identifier.

In the following example, the `'The Empty Room'` name needs to be enclosed within single quotes because it contains two Alan keywords (`The` and `Empty`), but there's no need to do so with the `empty_room` identifier because the word `empty` doesn't appear in isolated form (i.e. surrounded by spaces):

```
The empty_room IsA location    --> No need to stropp 'empty_room'.
  NAME 'The Empty Room'.       --> Keywords 'THE' and 'EMPTY' stropped.
End The.
```

Stropping occurs only when the quoted identifier contains at least one Alan keyword. Since in Alan it's always permissible to enclose an identifier within single quotes (where `'someID'` and `someID` both refer to the same identifier), not every quoted identifier implies stropping, but stropping *always* requires a quoted identifier.

4.2.3. Names Containing Multiple Words

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, letter casing will be ignored (i.e. comparison is case insensitive). 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.

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 to be adjectives and nouns (where it is assumed that the last is the noun). Except for this you should only quote single words to avoid clashes with reserved words.

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 refer in the typed input to words which are defined in the code as single words containing spaces or other special characters or separators (the only exception being underscores and dashes).

Also, remember that a player input word must start with a letter; so the player won't be able to refer to identifiers or names like "`1st`", "`'70s`", etc.

Escaping Single Quotes Inside Quoted Identifiers

To include a single quote as part of a quoted identifier, double it (e.g. `'Tom''s Diner'`). Inside the adventure, it will be printed as a single quote ("Tom's Diner").

This technique is known as *escaping*, for it allows the Alan compiler to distinguish between the quote character as delimiters of the quoted identifier and its occurrence within the identifier as a printable character.

Some of the identifiers in the source of an Alan game are used by default 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 won't be able to refer to them in the typed input.

4.3. Numbers

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

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

4.4. Strings

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

```
string = 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 Section 3.18.1, “Output Statements”). 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 the section called “String Statement” for a list) to indicate (but not precisely control) the formatting.

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.

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

4.5. 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 4.2, “Words, Identifiers and Names”).

Chapter 5. Running An Adventure

5.1. A Turn of Events

The player controls the execution of an Alan adventure. Each of his inputs are taken care of and acted upon by the run-time system in the interpreter. 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 is 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 checks and bodies. Global `Verb` `Check`s and bodies are used for verbs taking no parameters, otherwise the verb bodies are found in the parameter instances or their classes. In case the player typed a directional command the corresponding `Exit` check and code is executed.

After the player's command has been taken care of, all *rules* are evaluated and possibly executed. Then each of the other `actor`s executes one step in its `Script` (if active) and for each actor the rules are evaluated again. Finally, each `Event` that is scheduled for this round is fired, and then rules are evaluated yet again. Finally the player is prompted for another command.

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 for
for each pending event
   execute it
   evaluate all rules
end for
```

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

If the player enters an empty command, this is equivalent to forfeiting the turn. The empty command will result in no action from the hero, but counted as a player turn. Pending events will run and other actors will move as usual.

5.2. Player Input

The `Syntax` defined in the Alan source is the basis for what verb commands the player is allowed to input. If the syntax statement allowed referencing an instance (such as an object), then the syntax declares the basic form of that statement. In addition, other combinations and variations are possible using special characters and words.

Obviously these words are different in each language, but here we'll use generic English words, like "**AND**-word", to denote all the words that can be used in the same manner. The exact list of these words, for every language natively supported in Alan, is available in Appendix E, *Predefined Player Words*.

The following built-in syntax variations are available to the player:

- Concatenation of commands via **AND**-words, like:

```
> open the door THEN enter
> take the book AND read it
> west. north. east
```

- The use of pronouns to refer to the last object mentioned in the previous command, e.g.:

```
> take the book and read IT
> give key to guard and ask HIM to open the door with IT
```

  The pronouns have to be defined by the author in the source (see Section 3.8.4, "PRONOUNs") or by a library. The only built-in pronoun is the **IT**-word, which is automatically defined for the class `thing`.

- References to multiple objects using **AND**-word. This allows:

```
> take the blue vase AND the pillow
> examine the red key, the glass bowl AND the compass
```

- 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 Section 3.10, "SYNTAX Definitions"). All the variations above are built-in and handled automatically by the run-time system.

The interpreter also automatically restricts parameter references to things which are reachable according to the semantic rules of each built-in base class (see Section 3.7, "Instances" for the complete details). For example, objects can only be referred to if they are present at the current location, except if the `Syntax` for the command uses the omnipotent `!` indicator (see Section 3.10, "SYNTAX Definitions" for details). For some hints on ways to allow the player to refer to objects and actors that are not at the current location, refer to *Distant & Imaginary Objects*.

If the player uses **ALL** instead of a reference to an instance in his command, the verb will be applied to all appropriate instances at the current location, *except* the ones that do not pass all checks for the verb (see Section 3.8.10, "VERBs" for further details on this).

A 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).

5.3. Player Words

You use `Syntax` statements to define the structure of available player commands. The actual player input consists of "tokens" which are matched to those structures. The most prominent part of that are the words. The allowed words comes from

- the syntax statement — the verb word itself and any "filler" words, in Alan generalized to "prepositions"
- the special words mentioned above, to allow variations in reference forms
- synonyms — directly from the `Synonyms` statements
- instance names — adjectives and nouns

All those words are collected by the Alan compiler into a dictionary of known words.

Words in the player's input are separated into tokens by space or by characters not allowed inside words (see above).

Other characters are considered separate tokens. Some of those tokens are used in some forms of player input, such as comma and period, as seen above. Unrecognized characters will just result in an error.

5.3.1. Contractions

One special case is the apostrophe (" ' "). It is allowed inside words. But in many languages this is also used for contractions, or elisions, such as (English) "can't", or (Italian) "l'acqua" for the contraction of the definite article and the noun, leaving out some other characters and sounds.

In order to support the contraction of multiple words using the apostrophe, Alan does some special handling of word tokens containing an apostrophe. The complete word will first be looked up, but if that is not defined the separate parts will be looked up. E.g.

```
> prendi l'acqua
```

In this example the word "l'acqua" will first be tried as a complete word, and if found in the dictionary, the input will be interpreted as using that word (perhaps a noun). If it isn't found, the command parser will split at the apostrophe, first

trying “ l’ ” (the contracted definite article) as a separate word. Then the second part will be tried, in this case “acqua”.

This makes it possible to use natural words as nouns and create “ l’ ” as a synonym for the definite article.

5.4. 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, different parts of the adventure may be triggered. These parts all have different conditions under which they are evaluated and have different contexts. Five different execution contexts can be identified:

- **Execution of verbs** — 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 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 triggered as responses to a directional command, a `Look` or `Describe` statement, or a `Locate` statement operating on the hero. During this execution context, 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** — When an actor performs his script step there are no parameters defined but `Current Actor` is set to the actor currently executing. `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 are defined. The location is set to where the event was scheduled to execute.
- **Execution of rules** — Rules are executed without location, so neither parameters, `Current Location` nor `Current Actor` are defined. Any output statement in this context will be completely useless since the hero can never be at the same location of an executing rule.

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

- **The hero** — the actions of the player are always focused on the hero, and their execution is always relative to the hero's location.
- **An actor** — steps executed by an actor are always focused where the actor is.
- **An event** — code executed in events is focused where the event was specified to take place.
- **A rule** — rules are executed after each actor (including the hero) and after each event, with the focus set to the complete game world.

5.5. Moving Actors

The main way to move the hero around the adventure's world is through `Exit`s (see Section 3.8.12, "EXITs"). 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 `Check`s, which are run according to normal rules (see Section 3.11.3, "Verb CHECKs"). If there was no `Check`, or if the checks 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) gets located at a `location`, the `Entered` clause of that location is executed as if the actor had actually entered it by movement. The actor being located will be the `Current Actor` even if the movement was not caused by him (but was 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.

5.6. Undoing

A player might occasionally regret a typed command, perhaps realising that it was not the correct one. The Alan interpreter supports such undoing of commands. 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. The command history is saved automatically, and as many states as memory permits are saved, providing 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`.

5.7. Scripting and Commenting

Most versions of the Alan interpreter, Arun, supports both the creation of a transcript for the game in progress, as well as playing back a saved transcript as input passed to the interpreter.

These feature are very useful during the development of a game, allowing to play through the game up to a desired point and start from there, or even to 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.

Chapter 6. Hints and Tips

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

Using the `Import` mechanism of the Alan language (see Section 3.5, “IMPORT”) you can reuse snippets that you invent in multiple games or works. By building such a library you don’t have to reinvent the same thing every time.

A very easy way to get a lot of functionality, and learn about using the language, is to use the Alan Standard Library. You you can download it from the Alan website. The library implements many of the things described below, and loads of other handy things for you to use directly. For details on how to use that library, refer to its documentation.

> The following examples, hints and tips do *not* use any library, only plain vanilla Alan code.

6.1. Use of Attributes

Attributes are primarily used for holding status information about the instance to which they belong. This allows, for example, a water bottle to contain three levels of water:

```
The bottle IsA object
  Has level 3.
  Verb drink
    Does
    If level Of bottle > 0 Then
      "You take sip from the bottle."
      Decrease level Of bottle.
    Else
      "There is no more water in the bottle."
    End If.
  End Verb drink.
End The bottle.
```

Another example is a breakable mirror:

```
The mirror IsA object
```

```
  Is Not broken.
  Verb break
    Does
      Make mirror broken.
  End Verb break.
End The mirror.
```

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

Attributes defined for a whole class of instances also allow an extra dimension of classification of the instances. If the following declaration is made:

```
Add To Every object
 Is Not takeable.
```

then all objects receive the attribute `takeable` and unless the attribute is specifically redeclared for a particular instance they will not be takeable. Note however that the semantic meaning of “takeable” (i.e. what actually happens, such as preventing the “taking” action) 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.
```

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

An alternative approach is to use attributes to classify instances according to their characteristics. Consider:

```
Verb take
  Check o Is Not heavy
    Else "That is too heavy."
  And o Is Not animal
    Else "$+1 moves quickly away, just far enough
          for you not to reach it."
  Does
    Locate o In hero.
    "You take" Say The o. "."
```

```
End Verb take.
```

With this approach you need to keep track of which properties a particular verb will accept or require. This could be extended one step further, having verbs check actual dimensions, such as weight or size, instead.

And while we are talking about classification, the Alan 3 class concept can help. Often a classification can be made, clearly and succinctly, by defining a sub-class, for which every property pertaining to that type of instances can be collected. Often, the need for an attribute disappears.

Further more, you don't need to define a syntax for a single parameter verb if it only accepts instances from a particular class. Consider the following definitions:

```
Every vehicle IsA object
End Every vehicle.

Every car IsA vehicle
  Verb drive
    Does "Yoooohooooo!"
  End Verb.
End Every car.

Every bus IsA vehicle
End Every bus.

The car1 IsA car At l
End The car1.

The bus1 IsA bus At l
End The bus1.
```

Without any `Syntax` definition whatsoever, Alan will supply a default syntax for the `drive` verb which restricts its use to car instances only:

```
> look

There is a car1 and a bus1 here.

> drive bus1
You can't do that.

> drive car1
Yoooohooooo!
```

So the class mechanism not only allows for another way to classify your instances, but also makes it much easier to handle player input correctly.

6.2. Descriptions

Attributes come in handy when presenting information about instances to the player. The attributes can be tested in `If` statements to modify the `Description`s and possibly even the short description in the `Mentioned` sections.

For example:

```
The mirror IsA object
  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 The mirror.
```

If you also use this feature with the short descriptions will make the adventure feel a bit more consistent.

```
The bottle IsA object
  Has level 3.
  Article ""
  Mentioned
    If level Of bottle > 0 Then
      "a bottle of water"
    Else
      "an empty bottle"
    End IF.
End The bottle.
```

If the bottle had `level` 0 and was in the hero container, this would result in:

```
> inventory
You are carrying an empty bottle.
```

6.3. Common Verbs

As your adventures library grows, you will find that some verbs are needed often, and always function the same way. Examples are “take”, “drop”, “inventory”,

“look”, “quit” and so on. It is advisable to put them in a file which may then be imported into your games. See Section 3.5, “IMPORT” about the `Import` mechanism. The files may then contain these common 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 import this file (or files), making these features immediately accessible when you start a new adventure. All it takes is some thought on what names to use for the attributes, as discussed in Section 6.1, “Use of Attributes”.

And of course there is already an extensive library available from the Alan website, www.alanif.se. It also includes a lot of other features common to most adventure games.

6.4. Distant Events

An effect of the feature that output is not visible unless the hero is present is that the 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 happened. The solution is to create an `Event` that goes off where the hero is.

```
Event distant_explosion
  "Somewhere far away you can hear an explosion."
End Event.
...
If Hero Nearby Then
  Schedule distant_explosion At Hero After 0.
...
```

6.5. Doors

A common feature in adventure games is the closed door. Here's one way 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 `Exit`s makes sure that the hero cannot pass through a closed door.

6.6. 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.

```
The hero IsA actor
  Is Not quitting.
End The 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 just
        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.
```

6.7. Actors

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

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** command, in the same way as objects are. This means that a good idea is to include the description of an actor's activities in its `Description`. One way to do this would be to use attributes to keep track of the actor's state and test these in the `Description` clause.

```
The george IsA actor
  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. Since the actor's state is partlyt 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 us to simplify to:

```
The george IsA actor
  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 they take place, and the description of the actor is always consistent. You can, of course, still have attributes describing the actor's state to customize the description of the actor on an even more detailed level, but it generally suffices to describe an actor in terms of what script he is executing.

6.8. Vehicles

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

Let's start with the geography:

```
The garage IsA location
End The garage.

The parking_lot IsA location Name 'Large Parking Lot'
End The parking_lot.
```

Then we implement the actual car:

```
The car IsA object Name little red sporty ferrari Name car
```

```
  At garage
  Is Not running.
  Has position 0.

  Verb enter
    Does
      Locate hero At inside_car.
  End Verb enter.

End The car.
```

We also need a description for the inside of the car. We will use another location for this:

```
The inside_car IsA location Name 'Inside the Ferrari'
  Description
    "This sporty little red vehicle can really take you
     places..."
  Exit out To inside_car
    Check car Is Not running
      Else "I think you should stop the car before getting
            out..."
    Does
      Depending On position Of car
        = 0 Then Locate hero At garage.
        = 1 Then Locate hero At parking_lot.
        --- Etc.
    End Depend.
  End Exit.

  Verb drive
    Check car Is Not running
      Else "You are already driving it!"
    Does
      Make car running.
      If car At garage Then Schedule drive_to_parking After 0.
      Else Schedule drive_to_garage After 0.
      End If.
  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 drive_to_parking. Cancel drive_to_garage.
  End Verb park.

End The inside_car.
```

We must ensure that the player can say just “drive” and “park” by defining the `Syntax` for those single word commands:

```
Syntax drive = drive.
Syntax park = park.
```

You can also see from the above code that there are (at least) two events that need to be defined too. They handle the movement of the car from one place to another:

```
Event drive_to_parking
  "You drive out from your garage and approach
   a large parking lot."
  Set position Of car To 1.
  Locate car At parking_lot.
  Schedule drive_to_garage After 1.
End Event drive_to_parking.

Event drive_to_garage
  "You drive out from the parking lot and approach
   your own garage."
  Set position Of car To 0.
  Locate car At garage.
  Schedule drive_to_parking After 1.
End Event drive_to_garage.
```

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

There are a multitude of different solutions to this problem. One possibility is to exchange the car object for an actor and the events for script steps. However, in this solution the car object is not where the hero is (`inside_car`) so the output from the scripts for the car will not automatically be shown to the player. 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!

As Alan allows nesting locations (locating a location at another as if it was an object or actor), yet another solution would be to actually move the car location between the garage and the parking lot.

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

6.9. 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, and we want to avoid creating duplicate objects, so in a way we make them “float” along with the hero, or some other actor, instead.

6.9.1. Body Parts

One example of floating objects is the various parts of the hero’s body.

To create floating objects you can use a particular feature of entities, namely the fact that they are always located where the hero is. Such an entity can of course have the container property to allow it to contain a number of other instances.

So to have the hero’s body parts available wherever the hero goes you can use:

```
The body_parts IsA entity
  Container
End The body_parts.

The right_arm IsA object Name right arm In body_parts ...
The head IsA object Name head In body_parts ...
```

Using entity containers is also a simple way to create other compartments on the hero, such as a belt:

```
The belt IsA entity
  Container
    Header
      If Count In hero > 0 Then "and"
      Else "but" End If.
        "in your belt you have"
      Else
        ""
End The belt.
```

You can combine that with the following definitions of the hero and the **inventory** verb:

```
The hero IsA actor
  Container
    Header "You are carrying"
    Else "You are empty-handed"
```

```
      If Count In belt = 0 Then "." End If.
End The hero.

Verb inventory
  Does
    List hero.
    List belt.
End Verb inventory.
```

And the following output could result:

```
> inventory
You are empty-handed but in your belt you have a knife.
```

The example uses the `Count` aggregate to see if the other container is empty or not, and select the appropriate output accordingly.

6.9.2. Outdoors and Indoors

Another example of floating objects are semi-abstract objects like air, ground and walls. Some of these add some extra complexity for they should be available only under certain conditions.

Of course, you would not want outdoor things to be available when you are indoors. To solve this, simply create yet another container object where we can store the outdoor things when they should not be accessible and place it where the hero can never be. Now we only need to make sure that the objects are transferred between the two storages:

```
The outdoor_things IsA entity
  Container
End The outdoor_things.

The outdoor_things_storage IsA object At limbo
  Container
End The outdoor_things_storage.

The air IsA object In outdoor_things_storage ...
The sky IsA object In outdoor_things_storage ...

When location Of hero Is outdoors =>
  Empty outdoor_things_storage In outdoor_things.
When location Of hero Is Not outdoors =>
  Empty outdoor_things In outdoor_things_storage.
```

You need to add the boolean attribute `outdoors` to every location to the make the rules work, of course.

And, *voilà*, 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!

An alternative to the location attribute is to use classes. Define an `outdoor_location` class and an `indoor_location` class. Then inherit as appropriate, and the rules could instead look like:

```
When location Of hero IsA outdoors_location => ...
When location Of hero IsA indoors_location => ...
```

6.9.3. Nested Locations as a Solution

Yet another option would be to make use of the nested locations feature. Put all your outdoor locations in a outdoor location where the `outdoor_storage` entity is also present (this is just a hint):

```
The outdoor_region IsA location
End The outdoor_region.

The park IsA location At outdoor_region
End The park.
```

Then the outdoor items can stay at this “region” location, no need for rules or extra containers.

6.10. 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.

Darkness and light sources can be implemented in Alan in different ways. Again, we basically have the choice between attributes and classes. The solutions are both general and rather similar, so we will have a look at the solution using

attributes and leave the other solution to the reader. (A good exercise to really understand the Alan class concept, so please take a stab at it. If you want to have a look at one solution, you can study the Alan Standard Library, which uses classes to implement light sources.)

First we need an attribute owned by all objects. We know we only need to consider objects because light sources need to be transported by the player, so they can not just be anywhere, like entities.

```
Add To Every object
  Is Not lightsource.
End Add To.
```

This ensures that all objects have the boolean (true/false valued) attribute `lightsource` with the default not being a light source. Any objects that provide light need to explicitly state that they are instead. For some instances this attribute might change value dynamically, e.g. when the lamp is lit and extinguished.

Locations then must declare themselves as lit or not:

```
Add To Every location
  Is lit.
End Add To.
```

Here we assume most locations are lit, dark locations need to declare themselves `Not lit`.

We can now count the number of instances at the current location having the attribute `lightsource` set, and if there are one or more there is some light provided. So, the **look** verb could be reworked to:

```
Verb 'look'
  Check Current Location Is lit
    Or Count IsA object, Is lightsource, Here > 0
  Else
    "You cannot see anything without any light."
  Does
    Look.
End Verb.
```

The `Check` of the `'look'` verb now checks the current locations need for light and then counts instances of object which are both light sources and present, to see if there is light.

Of course, we must also modify the dark locations so that they don't display their descriptions upon entrance. This is easy to do using another addition to every location, a description check, similar to the check in the `'look'` verb:

```
Add To Every location
  Description
    Check Current Location Is lit
      Or Count IsA object, Is lightsource, Here > 0
    Else
      "You cannot see anything without any light."
End Add To.
```

6.11. Distant and Imaginary Objects

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. One way of handling this is to use entities, since they are "everywhere". But sometimes you need better control over when they are available and when not.

6.11.1. A Mountain

For objects that need to 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:

```
The hills IsA location
  :
End The hills.

The mountain IsA object At hills
  :
End The mountain.

The scenic_vista IsA location Name Scenic Vista
End The scenic_vista.

The shadow_mountain IsA object AT scenic_vista
  Name distant mountain
  Description
    "Far in the distance you can see the Pebbly
     Mountain raising towards the sky."
```

```
End The shadow_mountain.
```

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*
> ...

If the mountain must be visible and possible to manipulate from a number of locations, you might implement one shadow object for each location, but this might become a bit tedious if they are many. If they are identical you can use a simple rule like the following:

```
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` will surely follow. However, as the rules are executed *after* the hero has already moved, a better strategy might be to make the `shadow_mountain` "silent", i.e. to have no description. Instead, its description 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 `Exit`s, like:

```
The scenic_vista IsA location Name Scenic Vista
  Exit east To path
    Does
      Locate shadow_mountain At path.
  End Exit east.
End The scenic_vista.
```

Regardless of which of these strategies you chose, you need to take care that the shadow object is not present when the real object is. In this particular case, it should not be moved to the `hills`.

6.11.2. The Melody

To allow the player to "whistle the melody" for example, there are two different tactics that can be employed. One choice is to make the melody an `entity` (or some subclass thereof that you have defined), because, as we have seen, those can be manipulated from everywhere:

```
The melody IsA entity ...
Syntax whistle = whistle (m) ...
```

The other route would be to make it an actual `object`. In this case the `Syntax` for the `whistle` verb would need to indicate omnipotence — i.e. that the player can refer in the parameter to instances which are far away, including istances inheriting from `object`. (See Section 3.10.1, “Indicators” for more details on the *omnipotent indicator*.)

```
The melody IsA object ...
Syntax whistle = whistle (m)! ...
```

The melody then does not have to be reachable, near or even be at any location at all, for the player to be able to refer to it.

In both cases you would most likely need to restrict the parameters for the syntax so that the player can’t “whistle the chair”. Which of the two strategies you would chose depends mainly on things like:

- are there many things that this applies to (many “melodies”, perhaps)?
- should the player be able to manipulate this instance in other ways?
- do you need many different entities for various purposes?

6.12. Using Events as Functions

TBD.

6.13. Structure

A good thing to do when designing an interactive fiction story is to separate the geography from the story. In Alan, you can use the `Import` facility to structure your Alan source. One approach could be to place the description of each location in a separate file along with any objects that could be considered part of the scenery and other related items. These files can then all be included in a “map” file, which in turn is included by the top-level file.

The story line can be divided into files too, one for each “scene” — a scene being comments describing the important things that are suppose to happen, any

prerequisites and objects, events, rules, etc. which are specific for this part of the story.

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

6.14. Debugging

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 debugging features. 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       Enter the debugger when the game starts
```

The `-t` and `-d` switches cannot be used unless the adventure has been compiled with the `Debug` option set (see Section 3.3, "Options").

6.14.1. Command Logs and Game Transcripts

For various purposes, such as debugging, a log of the game play can be handy.

There are two such options available. One is a command input log and it is created when the option `-c` is given to the interpreter when starting a game. Such a file is also sometimes called a "solution file" since it can be used to show the solution to a game.

A command input log can sometimes be used as input to the interpreter, and thus automate the execution of the exact player experience.

The other options is to create a complete transcript of the game play and is created when the option `-l` is used. Such a log file contains both player input and the game output.

Both kinds of 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. They will have the extension **.a3i**, for "Alan v3 Input", or **.a3t**, for "Alan v3 Transcript", respectively.

6.14.2. Interpreter and Instruction Trace

Trace mode can also be helpful when debugging. Level 1 of tracing will show each section, verb, exit, description etc., the interpreter is invoked on, making it easier to see which parts of the code are executed.

Trace level 2, instruction trace, will in addition also trace the execution of each operation in the generated code. Level 3 shows the execution of all steps, also those only pushing to or popping from stack. Level 4 dumps the content of the stack between each instruction. So higher numbers gives more information but is probably also less and less useful for normal debugging.

6.14.3. 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:

```
adbg>
```

6.14.4. Using the Debugger

Abug may also be entered during the execution of an adventure. To do this you simply give the following player command (type it at the game prompt):

```
> debug
```

The game must have been compiled with the debug option or the command will be sent to the game which probably does not recognize it.

Typing a question mark or **help** in response to the debug prompt will give a brief listing of the commands available in Abug:

```
break [[file]:[n]]   -- set breakpoint at source line [n] in [file]
delete [[file]:[n]]  -- delete breakpoint at source line [n] in [file]
files                -- list source files
events               -- show events
classes              -- show class hierarchy
instances [n]        -- show instance(s)
```

```
objects [n]          -- show instance(s) that are objects
actors [n]           -- show instance(s) that are actors
locations [n]        -- show instances that are locations
trace ('source'|'section'|'instruction'|'push'|'stack')
                     -- toggle various traces
next                 -- execute to next source line
go                   -- go another player turn
exit                 -- exit debug mode and return to game, enter again using
 'debug' as input
x                    -- d:o
quit                 -- quit game
```

Any command may be abbreviated as long as it is unambiguous. Typing **b** for **break** will work, for example.

The display commands, **actors**, **locations**, **objects** and **events**, 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 using the **next** 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 **break** 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 set a breakpoint in another source file by preceding the line number with the file name delimited by a colon.

Breakpoints can be deleted. The **delete** 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 (and optionally the file name).

The debugger knows on which source lines it is possible to place a breakpoint. If you attempt to put a breakpoint at some line where it is not possible, it will attempt to place one at a line which is numerically higher but as close a possible. This will sometimes cause a breakpoint to be placed in a context that will not be what you expected.

The **trace** command and its options correspond to the types of traces described in the section on *Command Logs and Game Transcripts* above.

Wherever different output styles are available, e.g. in GLK based interpreters like WinArun, the Alan debugger tries to use them to distinguish the debugger output from the output of your game by using the pre-formatted style (see the section on styles in Section 3.18.1, “Output Statements”).

The following is a short excerpt from a command line debugging session (user input in italics):

```
<Arun - Adventure Language Interpreter version 3.0beta8 (2021-04-05 21:19:57)>
<Version of 'saviour.a3c' is 3.0beta8!>
<Hmm, this is a little-endian machine, fixing byte ordering....OK.>
<'saviour' contains the following IFIDs:
  IFID: UUID://c065a752-a476-a252-731f-e9fe96fcdc6d//
>
adbg> n

adbg:  Stepping to saviour.alan:1346
<01346>:     "$pWelcome to the game of SAVIOUR!$pIn this game your mission
adbg> n

Welcome to the game of SAVIOUR!

<<Game output deleted for breivity>>

adbg:  Stepping to saviour.alan:1354
<01354>:     Show 'logo.png'.
adbg> n

adbg:  Stepping to saviour.alan:1355
<01355>:     "$iVisit the Alan Home Pages at:"

adbg> break 1357
Line 1357 not available, breakpoint instead set at saviour.alan:1358
<01358>:     Visits 2.

adbg> g

    Visit the Alan Home Pages at:


    http://www.alanif.se

adbg:  Breakpoint hit at saviour.alan:1358
<01358>:     Visits 2.
adbg> n

Outside The Tall Building

adbg:  Stepping to saviour.alan:318
```

```
<00318>:         "To the north is a tall ancient building with a large entrance.
adbg> 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
adbg:  Stepping to saviour.alan:325
<00325>:             Score 5.

adbg> ?
Alan 3.0beta8 -- Adventure Language System (2021-04-05 21:19)
ADBG Commands (can be abbreviated):
    help               -- this help
    ?                  -- d:o
    break [[file:]n]   -- set breakpoint at source line [n] (optionally in [file])
    delete [[file:]n]  -- delete breakpoint at source line [n] (optionally in
 [file])
    files              -- list source files
    events             -- list events
    classes            -- list class hierarchy
    instances [n]      -- list instance(s), all, wildcard, number or name
    objects [n]        -- list instance(s) that are objects
    actors [n]         -- list instance(s) that are actors
    locations [n]      -- list instances that are locations
    trace ('source'|'section'|'instruction'|'push'|'stack')
                       -- toggle various traces
    next               -- continue game and stop at next source line
    go                 -- go another player turn
    exit               -- exit to game, enter 'debug' to get back
    x                  -- d:o
    quit               -- quit game

adbg> trace section
Section trace on.
adbg> n

<EXIT north[1] from Outside The Tall Building[4], Moving:>
<ENTERED in class entity[1] is empty>
<ENTERED in class location[2] is empty>
<ENTERED in instance Hall[5] is empty>

Hall

adbg:  Stepping to saviour.alan:332
<00332>:         "Inside the entrance is a hallway full of dust and pieces of
adbg> instances
Instances:
    [1] #nowhere ("#nowhere")
    [2] pseudowords ("pseudowords") (container), at [1] #nowhere ("#nowhere")
    [3] nowhere ("nowhere")
```

```
    [4] outside ("Outside The Tall Building")
    [5] Hall ("Hall")
    [6] door ("door"), at [5] Hall ("Hall")
    [7] stairs ("Stairs")
    [8] cellar ("cellar")
    [9] rats ("rats"), at [8] cellar ("cellar")
    [10] store ("store")
    [11] tape ("spool of computer tape"), at [10] store ("store")
    [12] first_floor ("First Floor")
    [13] book ("old book"), at [12] first_floor ("First Floor")
<<list abbreviated>>

adbg> instance 13
The [13] book ("old book") Isa object[4]
    Location: at [12] first_floor ("First Floor")
    Attributes:
        Takeable[2] = 1
        Readable[3] = 1
        openable[4] = 0
        startable[5] = 0
        examinable[6] = 1

adbg> 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[5], Moving:>
<ENTERED in class entity[1] is empty>
<ENTERED in class location[2] is empty>
<ENTERED in instance Stairs[7] is empty>

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[7], Moving:>
<ENTERED in class entity[1] is empty>
<ENTERED in class location[2] is empty>
<ENTERED in instance First Floor[12] is empty>

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 str(#1)=old book[13], inherited from object[4], CHECK:>
<VERB 21, in parameter str(#1)=old book[13], inherited from object[4], DOES:>
Taken.

<VERB 5, in parameter object(#1)=old book[13], inherited from object[4], CHECK:>
<VERB 5, in parameter object(#1)=old book[13], DOES:>
As you carefully try to open the book it falls apart into dust and falls
to the floor through your fingers.

> debug

adbg> instance 13
The [13] book ("old book") Isa object[4]
    Location: at [3] nowhere ("nowhere")
    Attributes:
        Takeable[2] = 1
        Readable[3] = 1
        openable[4] = 0
        startable[5] = 0
        examinable[6] = 1

adbg> trace instruction
Instruction trace on.
adbg> n
> _north

++++++++++++++++++++++++++++++++++++++++++++++++++
1f85: PRINT        10048,     22                  "You can't go that way."
1f86: RETURN
--------------------------------------------------

> west

++++++++++++++++++++++++++++++++++++++++++++++++++
1f85: PRINT        10048,     22                  "You can't go that way."
1f86: RETURN
--------------------------------------------------

> east

<EXIT east[2] from First Floor[12], Moving:>
<ENTERED in class entity[1] is empty>
<ENTERED in class location[2] is empty>
<ENTERED in instance office[14] is empty>

++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
 fcb: LINE             0,      0
 fce: PRINT         3479,      6                  "Office"
 fcf: RETURN
-------------------------------------------------

+++++++++++++++++++++++++++++++++++++++++++++++++
 fd2: LINE             0,    598
adbg:  Stepping to saviour.alan:598
<00598>:         "In front of you is a deserted office area. Desks and chairs

adbg> g

 fd5: PRINT         3485,    404                  "In front of you is a deserted
office area. Desks and chairs are piled up in one corner. The ventilation
system has partly fallen to the floor, tearing part of the ceiling down
with it. Under the twisted tubing a couple of old coffee makers are
crushed to pieces. One shelf, having some kind of lettering, no longer
readable, is thrown to one side, and another is still standing in a
corner, full of dust."
 fd6: RETURN
-------------------------------------------------

+++++++++++++++++++++++++++++++++++++++++++++++++
100c: LINE             0,    616
100f: ATTRIBUTE       15,     17                =0
1010: NOT           FALSE                       =TRUE
1011: IF             TRUE
1014: LINE             0,    617
1017: PRINT         3711,     43                 " There is a ladder laying on the
floor here."
1018: ELSE
    :
1029: RETURN
-------------------------------------------------

> look

<VERB 19, GLOBAL, DOES:>

+++++++++++++++++++++++++++++++++++++++++++++++++
 7dc: LINE             0,    199
 7dd: LOOK
+++++++++++++++++++++++++++++++++++++++++++++++++
 fcb: LINE             0,      0
 fce: PRINT         3479,      6                  "Office"
 fcf: RETURN
-------------------------------------------------

+++++++++++++++++++++++++++++++++++++++++++++++++
 fd2: LINE             0,    598
 fd5: PRINT         3485,    404                  "In front of you is a deserted
```

```
office area. Desks and chairs are piled up in one corner. The ventilation
system has partly fallen to the floor, tearing part of the ceiling down
with it. Under the twisted tubing a couple of old coffee makers are
crushed to pieces. One shelf, having some kind of lettering, no longer
readable, is thrown to one side, and another is still standing in a
corner, full of dust."
 fd6: RETURN
------------------------------------------------

++++++++++++++++++++++++++++++++++++++++++++++++
100c: LINE             0,    616
100f: ATTRIBUTE       15,     17                =0
1010: NOT           FALSE                       =TRUE
1011: IF             TRUE
1014: LINE             0,    617
1017: PRINT         3711,     43                 " There is a ladder
laying on the floor here."
1018: ELSE
    :
1029: RETURN
------------------------------------------------
 7de: 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 `Look` in the example above). Lines of dashes, indicates the return from one such level of interpretation.

Chapter 7. Adventure Construction

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

7.1. Getting an Idea

As with a book, the success or failure depends on how intriguing the story is, how hooked you can get the reader (in our case the player). 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.

7.2. Elaborating the Story

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

7.3. Implementing it

At last, it is time to sit down at the terminal. Divide the adventure text into files containing global verbs, the map (possibly divided further according to the scenes), the actors (perhaps one file for each actor) and a main file including the other files. This makes it easy to handle the adventure and you 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.

7.4. 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).

7.5. 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 the section *Command Logs and Game Transcripts*) to produce a list of the commands a player have used. This can greatly aid in spotting troublesome areas in your game. A common example is when the player gets 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.

Appendix A. How to Use the System

How to actually set up and use the complete Alan system depends very much on which platform, OS and in which environment you are going to use it.

If you just want to run a game, there is of course the original command line version distributed from www.alanif.se. To use this, read the relevant sections below to get a feel for how that will work.

But there are also a number of other packages that include an Alan interpreter, Gargoyle and Spatterlight, are two, with Gargoyle at the point of writing, being the one most up to date, and also available on multiple platforms.

If you actually want to write interactive fiction using Alan you also need the compiler. This is distributed from www.alanif.se in what's called the Development Kit, or the SDK (for Software Development Kit), which includes the compiler, interpreter, a conversion program for v2 games and some examples.

Whatever option you choose, there should be more detailed instructions on how to install included with that package. Below follows some information pertaining to the original versions from the Alan website.

A.1. Compiling

Although there are other options, like the AlanIDE, WinAlan et al., basic use of the Alan Adventure Development System is through a traditional command line batch compiler. This means that the actual development system is a compiler that reads text files created using any standard 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 assume an extension of **".alan"** if none is supplied. The option `-help` will give a brief help on other options to the compiler.

The primary output from the compiler 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, a new one will automatically be generated.

A.2. Compiler Switches

If you run the compiler from a command line you can get information about which switches it supports using the `-help` switch:

```
$ alan -help
```

Here are some examples of other switches:

- `-version` shows the version of the compiler
- `-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`, `-infos` show warning (and/or informational) messages from the compilation process
- `-import` add a directory to the search path for imported files (see Section 4.5, "Filenames" for details on the `Import` statement). This switch can be used multiple times, each adding a new directory
- `-listing` direct compiler output (error messages etc.) to a file with the same base name as the input (source) file, but with the extension **.lis**
- `-full` will produce a complete listing of the source on the screen, or if combined with the `-listing` option, in the listing file
- `-debug` include debugging information in the produced adventure files (same as the debug option, see Section 3.3, "Options")
- `-pack` encode and compress the text data (same as the pack option, see Section 3.3, "Options")
- `-summary` produce a summary about number of actors, size of adventure files, timing information etc.
- `-dump` print the internal form (developers use mainly)

Giving an extra hyphen before the option reverses its meaning (where appropriate), e.g. `--warnings` means don't show warnings. Switches may be abbreviated as long as they are unambiguous.

A.3. Running the Adventure

To play the generated adventure the Alan interpreter, **arun**, is executed with the adventure name as a parameter. For example:

```
$ arun adventure
```

No extension on the adventure name is allowed, the **.a3c** and, if applicable, **.a3r** files are found automatically from that name.

On platforms with graphical user interfaces to which **arun** has been natively ported will allow double clicking a game file to start a game, or double clicking the interpreter application icon, in which case a dialogue 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 make a complete game package, which is easy to start using the single command:

```
$ adventure
```

A.4. Interpreter Switches

interpreter, switches The interpreter supports the following switches:

- `-h` print some help
- `-v` verbose mode
- `-l` log game transcript to a file ('.a3t')
- `-c` log player command input to a file ('.a3i')
- `-n` no Status Line
- `-d` print the version of interpreter and enter debug mode

- `-t[<n>]` various levels of execution trace, higher <n> gives more details
- `-i` ignore CRC and version errors in the adventure files (dangerous)
- `-r` make regression test easier (don't timestamp, page break, randomize...)
- `-version` print version of the interpreter, the version and IFID of the game, if given, and then exit

Debugging support is described in Section 6.14, "Debugging".

Appendix 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.

Appendix C. 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.

C.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*. These standard messages exist for all languages and the default value of the texts are selected depending upon the setting of the language option.

The contents of any message may be modified using the `Message` statement (see Section 3.15, “MESSAGEs”). 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.

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 Section 3.15, “MESSAGEs” 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 what you mean by '$1',"
WHICH_PRONOUN_FIRST : "$+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.

> 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 `thing`s 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 parts in bold 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 `container`s. 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 `container`s.

```
EMPTY_HANDED : "$+1 is empty-handed."
```

The default messages for empty `container`s that are `actor`s.

```
HAVE_SCORED : "You have scored $1 points out of $2 in $3 moves."
```

This is the default message for presenting scores, if you use the `Score` statement.

parameter1 is an integer containing the current score.

parameter2 is an integer containing the maximum score possible.

parameter3 is an integer containing the elapsed turns since the game started.

```
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 (RETURN confirms) ? "
```

If the save file already exists, 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 that 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 UNDO, RESTART, RESTORE or QUIT ? "
```

The `Quit` statement requests an action from the player.

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 : "Nothing to undo."
```

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.

```
IMPOSSIBLE_WITH : "That's impossible with $+1."
```

If a player action is impossible with a particular parameter combination, but might be possible otherwise, this message is shown to indicate that it is the action *with the parameter* that is impossible.

```
CONTAINMENT_LOOP : "Putting $+1 in itself is impossible."
```

The interpreter detected an attempt to locate an instance inside (contained) itself. This message reliefs the author from the responsiblility to check for every possible circumstance where this might happen.

```
CONTAINMENT_LOOP2 : "Putting $+1 in $+2 is impossible since $+2 already is inside $
+1."
```

Same as above but in this case the containment was transitive, i.e. it would create a containment loop with more that one instance involved.

C.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.
```

C.3. Player Errors

These errors are usually caused by incorrect arguments or file names entered by the player.

```
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.

C.4. 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. 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.

> 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.

```
Index not in container in 'containerMember()'
```

This is most likely caused by doing `Random In` on an empty `container`.

```
Recursive LOOK.
```

This message is shown when a `Look` statement is executed as a result of a `Look` ! The `Look` statement should only be used in `Verb` bodies. It should *not* be used in descriptions of instances 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 a `location` which would trigger a `Look` in itself thus starting the recursion!).

```
Locating a location that would create a recursive loop of
locations containing each other.
```

This means that an attempt to locate a `location` inside itself has been made. Probably in an attempt to dynamically manipulate the `location` structure (the map).

```
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 allow compile time checking, thus avoiding the risk of having this happen to the player.

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

```
Interpreter recursion.
```

The interpreter keeps track of its execution so that it can never enter an endless loop. There are a few situations where this can occur. One example is if the `Description` of an instance in some way, directly or indirectly, executes `Describe This`. As the interpreter is already executing a `Description` of the current instance the invocation of the second will create a loop that never terminates.

C.5. 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*).

Appendix D. Language Grammar

D.1. Description

In this manual, the Alan language is defined using a BNF-form, which you can see in most descriptions. The grammar is a set of rules defining what constructs are legal in the source for an Alan program. Below follows a brief explanation on how to interpret these rules by using some short examples. For details on the actual rules, refer to the content of Chapter 3, *Language Reference*.

The BNF form divides the rules for structure of the input source by describing it in smaller parts, which may in turn be 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 are enclosed in curly braces (`{` and `}`).

```
=  : 'is defined as'

[] : 'optional'

{} : 'zero or more times'
```

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 components of the language are reserved words and symbols. These are represented in the rules 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` must be followed by an expression (which, to make sense, must refer to a `container` instance) to form a **random_expression** (which in itself is an expression).

D.2. Keywords

The following is a complete list of all keywords in the Alan language. Note that they can still be used as identifiers in a source file if the rules described in Section 4.2, "Words, Identifiers and Names" are followed. Basically this means that if you surround them by single quotes they can be used as identifiers in your source code anyway. This might be especially important if you want to use any of these words as words the player might want to input, such as part of a name for an item.

Table D.1. List of Alan Keywords

`Add`	`After`	`An`	`And`	`Are`
`Article`	`At`	`Attributes`	`Before`	`Between`
`By`	`Can`	`Cancel`	`Character`	`Characters`
`Check`	`Container`	`Contains`	`Count`	`Current`
`Decrease`	`Definite`	`Depend`	`Depending`	`Describe`
`Description`	`Directly`	`Do`	`Does`	`Each`
`Else`	`ElsIf`	`Empty`	`End`	`Entered`
`Event`	`Every`	`Exclude`	`Exit`	`Extract`
`First`	`For`	`Form`	`From`	`Has`

`Header`	`Here`	`If`	`Import`	`In`
`Include`	`Increase`	`Indefinite`	`Indirectly`	`Initialize`
`Into`	`Is`	`IsA`	`It`	`Last`
`Limits`	`List`	`Locate`	`Look`	`Make`
`Max`	`Mentioned`	`Message`	`Meta`	`Min`
`Name`	`Near`	`Nearby`	`Negative`	`No`
`Not`	`Of`	`Off`	`On`	`Only`
`Opaque`	`Option`	`Options`	`Or`	`Play`
`Prompt`	`Pronoun`	`Quit`	`Random`	`Restart`
`Restore`	`Save`	`Say`	`Schedule`	`Score`
`Script`	`Set`	`Show`	`Start`	`Step`
`Stop`	`Strip`	`Style`	`Sum`	`Synonyms`
`Syntax`	`System`	`Taking`	`The`	`Then`
`This`	`To`	`Transcript`	`Transitively`	`Until`
`Use`	`Verb`	`Visits`	`Wait`	`When`

Although the Alan language is case-insensitive, the keywords in the above list are cased according to the styling conventions adopted in this document, for the sake of consistency and to provide an intuitive association between keywords and their role in the language.

Appendix E. 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*. 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.

E.1. English

```
ALL:   all everything
AND:   and then
BUT:   but except
THEM:  them
NOISE: go the
```

E.2. Swedish

```
ALL:   alla allt
AND:   och
BUT:   förutom utom
THEM:  dem dom
NOISE: gå
```

E.3. German

```
ALL:   alles
AND:   und
BUT:   ausser
THEM:  sie
NOISE: das der die gehen
```

Appendix F. Compiler Messages

F.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 ❶

   23.  If barfoo Is foobared Then ❷
=====>     1

  *1*   310 E : Identifier 'barfoo' not defined.

   27.      Exit north To Rumble.
=====>                      1 ❸

  *1*   310 E : Identifier 'Rumble' not defined. ❹

   28.      Exit west To Tumble.
=====>                     1

  *1*   310 E : Identifier 'Tumble' not defined.

   46.
=====>  1

  *1*   101 E : Syntax error. Inserting "start here ." before this token.
  *1*   211 E : Adventure must start at an instance inheriting from 'location'.


        5 error(s). ❺
        No detected warnings.
        1 informational message(s).
```

The following information is available in the compiler listing, framed for visibility:

❶ File name
❷ Line number and source text of a line
❸ Message indicator
❹ Message pointer, message number and text
❺ 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*.

F.2. Message Explanations

For each message, a short description of the error, its possible causes, etc., are given. Each reported message 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 Section 4.5, “Filenames”.

```
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 `Verb`s 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`.

```
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 a `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 does 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.
```

Every object must possess the attribute which the `Limits` clause needs 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 `Check`s and bodies of `Verb`s, so the use of `object` (an obsolete construct, use the parameter identifier instead) is also restricted to those contexts. See *Run-Time Contexts*.

```
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 definitions of scripts or initial location are 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 `Event`s no actor is active. This means that no reference to the active `actor` can be made. See *Run-Time Contexts*.

```
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.

Appendix G. Localization

When it comes to writing text adventures in languages other than English, the Alan IF system is a great choice. Since Alan doesn't enforce on authors a fixed world model nor a predefined set of verbs, its open-ended philosophy simplifies adapting Alan to other languages.

To create adventures in languages other than English, there are a few issues that need to be taken into account first:

- Special characters support (letters with accents, diacritics, etc.).
- Handling of grammatical number, gender person and case.
- Defining `Synonym`s for variations of the basic articles and prepositions.
- Provide some initialization mechanism on all game instances to enforce grammatical consistency via custom attributes.

Alan was designed around English as the default language, but implementing support for other languages should be a fairly easy task, thanks to Alan's flexible design and syntax. The best approach is to create a dedicated "grammar module" to supported your desired language, which can then be reused to create any adventure in that language. Alan has already been successfully localized to Italian, so you could look at the Italian grammar module for inspiration.

The sections below provide some guidelines and insight on establishing if Alan is able to support a given language, and which steps are required to implement a core grammar module to support that language. Because every language is different, it's impossible to provide a set of fixed guidelines; instead, we can provide practical examples of how specific features of other languages where (or could be) implemented in Alan.

Before embarking on localizing Alan to a specific target language, you're advised to:

- Study how other languages where implemented in Alan.
- Study how the target language was implemented in other IF systems (if any).
- Read Chapter V: Natural Language of *The Inform Designer's Manual* (*DM4*), by Graham Nelson, which provides precious insights on how IF systems can handle different languages.

Looking at how others have accomplished similar tasks is always a great source of inspiration.

G.1. Character/Glyph Availability

Many languages rely on characters (or glyphs) which are not found in the English alphabet, like ö, Ñ or æ. Alan uses a character encoding called ISO-8859-1 (aka Latin1) which covers most characters (but not all) used in European and Western countries.

To use these characters you just need to ensure that the adventure source files adopt this character encoding. Most editors allow specifying a specific encoding, either when you save a file the first time, or through a dedicated menu entry, the settings panel or by associating a file extension with a specific encoding. Some operating systems might allow you to define the default encoding.

If the Alan source files are encoded using ISO-8859-1 they will be presented correctly when the game is run in graphical interpreters like WinArun and Gargoyle. If you run the game in console mode, you must ensure that both your console program and your operating system are set (and able) to show characters using the ISO-8859-1 encoding — often just setting the correct encoding is not enough, because some console fonts might not contain all the special characters glyphs required, and will be unable to represent them. Make sure that you also install the right console font, and to set it as your shell/CMD default font.

For languages with non-Latin alphabets, currently the only viable solution is to fallback on some romanization convention. In the early years of the Internet, when Ascii was the standard charset in communications, many creative solutions were devised to allow communicating in non-Latin languages over BBSs, chats and (later on) via sms using only Ascii characters. Many of these “chat alphabets” are still popular today, and you can find many tools to convert to and from them. For example, there is the Arabish chat alphabet for Arabic, Volapuk for Russian, and Shlyokavitsa (or Maimunitsa) for Bulgarian — just to mention a few.

Beware though that in Alan you won’t be able to freely mix letters, numbers and symbols as these alphabets do, for there are some strict rules on what is considered a valid player input token. You’ll need to be creative, and find ways to adapt your needs to the restrictions of the Alan language.

G.2. Standard Messages

There are two main types of messages that are output by the interpreter to the player. They are initiated by various parts of the Alan system. The built-in messages are hardcoded into the compiler (see Section C.1, "Input Response Messages") but can be changed to messages in the available languages. If you are using any of the available standard libraries, that library also issues messages, often overriding the built-in messages.

The only way to translate messages in the library is to translate the library. This is probably what you want to do anyway, since most of the verbs would be in English.

Messages built into the compiler are generated automatically into the game file, e.g. error messages like "You can't do that." These can either be changed by the `Language` option (if the language of your choice is already supported), or translated using the `Message` statement. Appendix C, *Run-Time Messages* lists all such messages and their content.

There are a few special words that are currently not author translatable, as described in Appendix E, *Predefined Player Words*.

G.3. Player Words

When creating adventures in a language not natively supported by Alan, one of the first things you'll need to look into are the *predefined player words* (see Appendix E, *Predefined Player Words* for more details).

For example, *noise words* are useful for defining words which should be ignored by the parser. Articles are usually defined as noise since they are irrelevant in adventure games input, and it's convenient that when the player types:

```
> take the apple
```

the parser will only have to process "take apple" (the article "the" having been discarded as noise). If this wasn't the case, then authors would have to define multiple syntaxes for every verb in order to account for the presence of the article — which, in terms of understanding player commands, doesn't provide any meaningful information. Some languages have many different articles forms, one for each number and gender combination, and unless these are defined as *noise words* they would lead to numerous alternative syntaxes in the adventure source.

If you're adventure uses English as the base language, then you can define your noise words by making them synonyms of a known English noise word (e.g. "the" or "go"). For example, with the Italian language we would define all possible articles as noise this way:

```
SYNONYMS il, lo, la, 'l'', i, gli, le = 'the'.
```

This line of code ensures that all Italian definite articles will be ignored by the parser (stripped away from the player input).

In this regard, it's important that you study carefully the list of all the *predefined player words* in the supported languages, as found in Appendix E, *Predefined Player Words*, to check that none of them conflicts with a needed word of your language. Since these player words are hardcoded into Alan, you can't change them nor undefine them. This is the criteria for choosing the base language for your adventure.

Ultimately, whichever base language you choose is irrelevant, for you'll be translating all the Alan messages, so the base language is going to be hidden away in the background of your adventure.

If you can use English without conflicts, then go for it since it's the default language. To use another natively supported language (i.e. German or Swedish), just enable it via the `Language` option. E.g.:

```
Language German.
```

G.4. Word Variations

TBD.

Text about using multiple names for the same objects & synonyms, and how the interpreter chooses which word to use when.

G.5. Word Order

A language may have different word ordering than English. Usually this is one or both of the following

- verb/noun - does the verb preceed the noun in imperative sentences?

- adjective/noun - does the adjectives preceed the noun

For example, in German the command for “take the book” would have the noun preceed the verb and be:

```
> das buch nehmen
```

(literally, “the book take [it]”) where the parameter occurs in first position, requiring the following `Syntax` definition:

```
Syntax take = (o) nehmen.
```

TBD.

Description on how to handle noun/adjectives ordering.

G.6. Useful Links

Here are some links to useful resources for localizing Alan to other languages.

- Alan Italian — Alan 3 support for Italian via a core grammar module, and an Italian port of the Alan Standard Library.
- *The Inform Designer’s Manual* (4th Ed.) by Graham Nelson, 2001:
  - Chapter V: Natural Language
    - §34. Linguistics and the Inform parser
    - §35. Case and parsing noun phrases
    - §36. Parsing non-English languages
    - §37. Names and messages in non-English languages
- Wikipedia:
  - Definiteness
  - Grammatical gender
  - Inflection
  - Specificity

Appendix H. 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 (and any "**.a3r**" file, if available) 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 ISO-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 only on some systems.

Portability between different versions of the system is not guaranteed. Again, backward compatibility of compiled adventure files is a main goal, at least between different builds within the same major version. This means that the most recent v3 interpreter should run games from all previous v3 compilers.

Changes in the game file format may occur between versions, which may or may not be compatible. Conversion tools may be available, older interpreter versions can be requested.

Appendix I. Copying Conditions

The Alan System is distributed under the **Artistic License 2.0** for which the full text follows. The intent of this licensing is that the Copyright Holder retain some control over the development of the Alan System, while still keeping it available as open source and free software.

In practical terms this means that the licensing is chosen so that it should be possible to:

- freely distribute games produced with the Alan system, including for profit
- re-distribute compiled versions of the Alan system, including together with a game which is not open source or free, provided there is no charge for the Alan system
- redistribute compiled and/or source versions of the original Alan system (the Standard Version)
- acquire the source code for the Standard Version
- modify the source code for private use
- re-distribute compiled and/or source of a Modified Version provided they are done so under a compatible license with appropriate attribution **and** that the modification is described and made available, preferably by returning it to the Copyright Holder so that it can be merged into the Standard Version

I.1. Artistic License 2.0

Preamble

This license establishes the terms under which a given free software Package may be copied, modified, distributed, and/or redistributed. The intent is that the Copyright Holder maintains some artistic control over the development of that Package while still keeping the Package available as open source and free software.

You are always permitted to make arrangements wholly outside of this license directly with the Copyright Holder of a given Package. If the terms of this license do not permit the full use that you propose to make of the Package, you should contact the Copyright Holder and seek a different licensing arrangement.

Definitions

"Copyright Holder" means the individual(s) or organization(s) named in the copyright notice for the entire Package.

"Contributor" means any party that has contributed code or other material to the Package, in accordance with the Copyright Holder's procedures.

"You" and "your" means any person who would like to copy, distribute, or modify the Package.

"Package" means the collection of files distributed by the Copyright Holder, and derivatives of that collection and/or of those files. A given Package may consist of either the Standard Version, or a Modified Version.

"Distribute" means providing a copy of the Package or making it accessible to anyone else, or in the case of a company or organization, to others outside of your company or organization.

"Distributor Fee" means any fee that you charge for Distributing this Package or providing support for this Package to another party. It does not mean licensing fees.

"Standard Version" refers to the Package if it has not been modified, or has been modified only in ways explicitly requested by the Copyright Holder.

"Modified Version" means the Package, if it has been changed, and such changes were not explicitly requested by the Copyright Holder.

"Original License" means this Artistic License as Distributed with the Standard Version of the Package, in its current version or as it may be modified by The Copyright Holder in the future.

"Source" form means the source code, documentation source, and configuration files for the Package.

"Compiled" form means the compiled byte code, object code, binary, or any other form resulting from mechanical transformation or translation of the Source form.

Permission for Use and Modification Without Distribution

1. You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not Distribute the Modified Version.

Permissions for Redistribution of the Standard Version

2. You may Distribute verbatim copies of the Source form of the Standard Version of this Package in any medium without restriction, either gratis or for a Distributor Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At your discretion, such verbatim copies may or may not include a Compiled form of the Package.

3. You may apply any bug fixes, portability changes, and other modifications made available from the Copyright Holder. The resulting Package will still be considered the Standard Version, and as such will be subject to the Original License.

Distribution of Modified Versions of the Package as Source

4. You may Distribute your Modified Version as Source (either gratis or for a Distributor Fee, and with or without a Compiled form of the Modified Version) provided that you clearly document how it differs from the Standard Version, including, but not limited to, documenting any non-standard features, executables, or modules, and provided that you do at least ONE of the following:

    a. make the Modified Version available to the Copyright Holder of the Standard Version, under the Original License, so that the Copyright Holder may include your modifications in the Standard Version.

    b. ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.

    c. allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under

        i. the Original License or

        ii. a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source

5. You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution. If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

6. You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

Aggregating or Linking the Package

7. You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

8. You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or byte code versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose a direct interface to the Package.

Items That are Not Considered Part of a Modified Version

9. Works (including, but not limited to, modules and scripts) that merely extend or make use of the Package, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not subject to the terms of this license.

General Provisions

10. Any use, modification, and distribution of the Standard or Modified Versions is governed by this Artistic License. By using, modifying or distributing the Package, you accept this license. Do not use, modify, or distribute the Package, if you do not accept this license.
11. If your Modified Version has been derived from a Modified Version made by someone other than you, you are nevertheless required to ensure that your Modified Version complies with the requirements of this license.
12. This license does not grant you the right to use any trademark, service mark, trade name, or logo of the Copyright Holder.
13. This license includes the non-exclusive, worldwide, free-of-charge patent license to make, have made, use, offer to sell, sell, import and otherwise transfer the Package with respect to any patent claims licensable by the Copyright Holder that are necessarily infringed by the Package. If you institute patent litigation (including a cross-claim or counter-claim) against any party alleging that the Package constitutes direct or contributory patent infringement, then this Artistic License to you shall terminate on the date that such litigation is filed.
14. Disclaimer of Warranty: THE PACKAGE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT ARE DISCLAIMED TO THE EXTENT PERMITTED BY YOUR LOCAL LAW. UNLESS REQUIRED BY LAW, NO COPYRIGHT HOLDER OR CONTRIBUTORWILL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THE PACKAGE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

I.2. 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.

If you create a game using the Alan System, we'd very much like to see it. Send us a copy (preferably in source) and any documentation or a description of the game and its novel features. We will of course honour any copy-restrictions that you might want to place on it.

Short games or samples of Alan source are also most welcome as examples that we might use and distribute to other users. Sending an example means that you waive all rights to it. If you also supply a walkthrough to your example it will be added to the growing suite of test data and thus help us further improve the quality of the Alan system.

Glossary

Here follow some definitions of technical terms encountered in this manual, to bridge the gap between authorship as a writer and a programmer. Although Alan is designed to be usable by non-programmers, knowing the terms of this glossary might help authors to better grasp some aspects of writing adventures in Alan, by providing an outlook into the wider context of computer languages and programming, and by providing the technical jargon required to discuss technical issues and further research some topics.

stropping

In the realm of computer programming, *stropping* is a technique to enable end users to create identifiers that contain reserved words of the language, by means of a special notation that allows the compiler (or interpreter) to distinguish between a user-defined identifier and a kewyord of the language, thus avoiding clashes that would prevent compilation/execution of the code. In the Alan language, identifiers containing one or more reserved keywords can be stropped by enclosing the whole identifier within single straight quotes (`'`). Example: `The 'The Empty Room' IsA location.` (keywords `THE` and `EMPTY` stropped). For a detailed explanation and practical examples, see Section 4.2.2, “Keywords as Identifiers”. For more info on stropping, see Wikipedia.

Index

U

V

W