Alan Standard Library v2.0

User's Manual

May 16, 2016

USER'S MANUAL FOR ALAN STANDARD LIBRARY v2.0

Introduction

This is the manual for the ALAN Standard Library v2.0. This manual describes how to use the library together with the ALAN Interactive Fiction Language system v3.0 and subsequent versions, to create works of interactive fiction, or text adventures. The library defines responses for verbs and commands commonly used in gameplay, such as *examine*, *take*, *drop* and *attack*, together with numerous other ones. The library also defines various global attributes (for example *takeable*, *NOT openable*), as well as object and actor attributes and classes (for example CLOTHING, WEAPON, PERSON), together with illegal parameter messages (for example *That's not something you can examine*).

The first official version (v1.0) of the ALAN Standard Library was published in 2010. Before that, versions 0.x, written by Steve Griffiths, were available for use. The writer of the current version is Anssi Räisänen.

The ALAN Standard Library v2.0 consists of the following five primary library files:

lib_classes.i
lib_definitions.i
lib_locations.i
lib_messages.i
lib_verbs.i

In addition, the following files, none of which are necessary for using the library, can be found in the library distribution package:

changelog.txt	A text file listing the changes made to the library after version v2.0 (= none so far)
copying.txt	A text file clarifying some copyright issues
library.i	A file that imports all library files. Just use the line IMPORT 'library.i.'. to import the standard library to your game
mygame_import.i	An auxiliary, not obligatory, definition file for the library. If you need to edit a great number of default library messages (for verb outcomes etc.), you can edit this file and import it to your game project
newgame.a3c	A compiled game of the source code included in 'newgame.alan' (below)
newgame.alan	A barebones game source file defining some necessary coding when starting to write a new game. You can use this as a starting point for a new project
newgame.ifid	An IFID identification number of the compiled 'newgame.a3c' file
quickref.text	A quick summary of the library features and how to use them
quickstart.pdf	A quick summary for starting to use the library.
testgame.a3c	The compiled test game, ready to run and play, to test the features of the library
testgame.alan	The source code for a test game showcasing the features of the library
testgame.ifid	An IFID identification number of the compiled 'testgame.a3c' file

Thank you to Steve Griffiths for the score notification code snippet and for the early versions of the library, and to Alan Bampton for the code used for clothing objects (layered clothing).

And naturally many thanks to Thomas Nilsson for the ALAN Interactive Fiction Language.

What to read if you're a complete newbie

It would be advisable to start with the ALAN Manual (available on the ALAN website at www.alanif.se) to get an idea of how the language works in general. After that, read this manual thoroughly, with the following exceptions:

- The chapter *What is different in v2.0* is not that necessary to read if you haven't used the previous version of the library, but it might be useful if you want to have a quick preview of some features the library is capable of.

- Read only the early part (pp. 27-28) of the chapter *The my_game instance and its attributes* to get an idea of how that meta-instance is to be used in the game source. The later passages in that chapter that list all possible illegal parameter message attributes and verb check attributes of the *my_game* meta-instance are meant to serve as a searchable index sooner than to be read systematically.

What to read if you're familiar with ALAN but haven't used the standard library v1.0

Read through the whole of this library manual carefully, with the exception of the chapter *The my_game instance and its attributes*, of which you should read only the early part (pp. 27-28) to understand the significance of using the *my_game* meta-instance in the game source. The latter part of that chapter, where the illegal parameter messages and other messages are listed, is meant to be read only cursorily and to be used as a searchable index when needed.

What to read if you have been using the v1.0 of the standard library

V2.0 works in many places quite differently from v1.0, so read first the chapter *What is different in v2.0* to get an idea of what has changed since the previous version. Read cursorily through the chapters on *Locations*, *Actors* and *Objects* to learn about new features such as certain attributes. Read carefully the first part (pp. 27-28) of chapter on *The my_game instance and its attributes* to understand the significance of using that meta-instance in the game source. The latter part of that chapter, where the illegal parameter messages and other messages are listed, is meant to be read only cursorily, to be used as a searchable index when needed.

What is different in v2.0?

- the author doesn't necessarily need to access the library files any longer when writing a game. It is possible to write a game just importing the library files and define everything in your own game source file(s). It is still possible of course to edit the library directly if this is preferred. There are also some cases when the library needs to be accessed, for example when changing standard runtime messages.

- there are five library files in v2.0 (instead of four in v1.0): 'lib_classes.i,' 'lib_definitions.i' 'lib_locations.i', 'lib_messages.i' and 'lib_verbs.i'.

- the hero instance is left out of the library. It can be now defined from scratch by the game author. (There are still checks for the hero within various verbs, and these work whether the author defines the hero instance or not. There are also numerous default verb responses and other messages that take the hero into account, just like in the previous version of the library, but these can be easily overridden.)

- an obligatory meta instance, *my_game*, needs to be coded by the author to each new game. Its shortest possible formulation is

```
THE my_game ISA DEFINITION_BLOCK
END THE.
```

Without it, the game won't compile correctly. Inside this instance, it is possible for example to override defauilt messages provided by the library, like this:

```
THE my_game ISA DEFINITION_BLOCK

    VERB examine
         DOES ONLY "Nothing special."  -- <- your own message, instead
    END VERB.                          -- of the default provided by the
                                       -- library

END THE.
```

- check messages and illegal parameter messages can be edited much more smoothly. For example, you don't have edit the same check for every verb separately (or cut and paste); you can now change the wording in one place and it will affect all the places where that same check is found, throughout the library.

- the 'listable_container' class of version 1.0 has been renamed 'listed_container' which sounds slightly better.

- there are some new attributes:

    - *'allowed'* is needed for container objects to indicate which objects they can take:

```
THE drawer ISA LISTED_CONTAINER IN nightstand
     HAS allowed {diary, keys}.
  …
END THE drawer.
```

This will effectively prohibit unwanted successful outcomes for player commands such as `>put coffee cup in drawer` or `>put suitcase in drawer`. Besides *put_in*, this attribute also applies to the verbs *empty_in*, *pour_in* and *throw_in*.

- *'distant/not distant'*, in addition to the existing '*reachable/NOT reachable*'. This has proved to be a handy distinction to have at hand. It is possible for the hero for example to talk with an NPC (non-player character) that is not reachable (for example if the hero is lying down on a bed, or tied up to a chair), but not with one that is distant. Similarly, you can throw something at, to or into a not reachable instance (for example a basketball into a basket), but not at, to or into a distant one. There are also some other individual cases where you can manipulate not reachable objects as opposed to distant ones. The default responses for not reachable and distant objects are a bit different: a not reachable object is described to be "out of your reach" but a distant object is "too far away". For example the library-defined ceiling object for indoor rooms is not reachable ("The ceiling is out of your reach") while the library-defined sky object is distant ("The sky is too far away").

- actors are defined to be either *compliant* or *NOT compliant*. By default, they are NOT compliant. This attribute is needed when we try to get something from an NPC. For example, the verb *take_from* doesn't work with actors by default; the only way to make an actor give you something in their possession by default is to *ask for* it. Also implicit taking doesn't work with actors, i.e. if an NPC is carrying an apple and you type `>eat apple`, the outcome will be "That seems to belong to the [NPC]."; the apple won't be automatically taken by the hero like it would if it was not carried by anyone.

- every door has an *'otherside'* attribute which can be used if the game author wants to ensure that a door will be correctly opened, closed, locked and unlocked from both sides. When the open/closed status of a door instance changes, the status of its *otherside* counterpart (in the next room) is changed accordingly by the library. (If the author declares no *otherside* attribute for a door, then this doesn't happen automatically.)

```
THE kitchen_door ISA DOOR AT kitchen
     HAS otherside livingroom_door.
     IS lockable. IS locked.
     HAS matching_key small_key.
END THE.

THE livingroom_door ISA DOOR AT livingroom
END THE.

THE small_key ISA OBJECT IN drawer
END THE.
```

Above, the livingroom_door will also be lockable, locked, have *otherside* kitchen_door and can be opened by the small_key, even if none of these attributes were explicitly declared in the livingroom_door code.

- every lockable door has a *'matching_key'* attribute (see the above example) which should be declared at the door instance if it's meant to be locked/unlocked. If the hero carries the matching key of a locked door, unlocking will be possible through just "unlock door"or even "open door" and not necessarily using the longer formulation "unlock door with key". This attribute also eases up the coding required for locked doors.

- the 'closed' and 'closeable' attributes have been changed to '*open*' and '*openable*' which is more intuitive.

- the SCENERY class has been removed. Instead, '*scenery*' is declared as an attribute.

- similarly, the BACKGROUND class has been removed. Use the (NOT) reachable/distant attributes instead where applicable.

- some object classes are made to work in a simpler way from v1.0. For example, an object in the subclass 'liquid' won't have to be declared to have a 'vessel' attribute any longer (if the liquid is carried in a vessel of any kind). Similarly, clothing objects worn by NPCs can now be implemented more smoothly.

- formatting the game title, author, year and version at the start of the game is made easier. There is an automatic formulation which can be easily included if desired.

- some default verb responses have been changed from v1.0. For example, the response for *ask_about* has been simplified.

Locations

Location classes pre-defined in the library:

ROOM
SITE
DARK_LOCATION

Location attributes pre-defined in the library:

IS lit.
IS visited 0.
IS described 0.

Using the standard library, basic locations are implemented just like advised in the ALAN Manual, for example:

```
THE bedroom ISA LOCATION
      DESCRIPTION "This is your bedroom."
      EXIT north TO bathroom.
      EXIT east TO closet
            CHECK "The closet is locked."
      END EXIT.
END THE.

THE bathroom ISA LOCATION
      DESCRIPTION "This is the bathroom."
      EXIT south TO bedroom.
END THE.
```

ROOM

If you want to implement an indoor location, you can declare it ISA ROOM:

```
THE kitchen ISA ROOM
   DESCRIPTION "…"
   …
END THE kitchen.
```

All ROOMs will automatically have walls, a floor and a ceiling.

SITE

If you want to implement an outdoor location, you can declare it ISA SITE:

```
THE meadow ISA SITE
    DESCRIPTION "…"
    …
END THE meadow.
```

All SITEs will automatically have a ground and a sky.

NOTE: it is often a good idea to modify the 'examine' verb for the wall, ceiling, floor, ground and sky objects to add more immersion to the game. Here is an example for 'wall':

```
THE my_game ISA DEFINITION_BLOCK
 ...
     VERB examine
          CHECK obj <> wall
               ELSE
                    IF hero AT kitchen
                         THEN "The walls are lined with shelves."
                    ELSIF hero AT livingroom
                         THEN "The wallpaper has a nice flower pattern."
                    ELSIF...
                    END IF.
             ...
      END VERB.
END THE my_game.
```

DARK_LOCATION and the 'lit' attribute

In dark locations, actions requiring seeing are automatically disabled by the library. All dark locations have the attribute 'NOT lit'. They need a lit lightsource object to be present to be lit. To implement a dark location, it is enough to implement it for example in the following way:

```
THE basement ISA DARK_LOCATION
     EXIT up TO hall.
END THE.
```

The description of a dark location will be by default "It is pitch black. You can't see anything at all."
This default can be changed by editing the *dark_loc_desc* attribute of the *my_game* instance (see p. 27-).

If you add a description of your own to a dark location, this description will be shown only if the location is lit up by any means:

```
THE basement ISA DARK_LOCATION
      DESCRIPTION "Only useless junk can be seen lying around."
      EXIT up TO hall.
END THE.
```

In order that a *dark_location* is lighted, a *lightsource* object (a lantern, a match, a ceiling lamp, any other kind of light object) should be present.

In darkness, you are not able to manipulate things other than turn on a *lightsource* and drop items you're carrying (these checks are found in 'lib_verbs.i'). You can exit normally and use verbs that don't require seeing, such as 'smell', 'listen' and 'think'. If you are in a dark location with an NPC (= a non-player character), you are able to communicate with them by asking and telling, but not by showing and giving. If you wish to change these restrictions, see the respective verbs in 'lib_verbs.i' and modify their checks.

Note that you cannot change the name of a location mid-game. Thus, if you define a dark location called for example 'Darkness' and wish to make it lit at some point in the game, the name will still be ‘Darkness' even if the location description can be changed to describe the illuminated location. To show a change in the location name, you must locate the hero in another location when the dark location is lit. For example,

```
THE lantern ISA LIGHTSOURCE
      VERB turn_on
            DOES
                  IF hero AT mysterious_dark_room
                        THEN LOCATE hero AT treasure_chamber.
                        ...
                  END IF.
      ...
      END VERB.
END THE.
```

Alternatively, you can also use a rule, for example

```
WHEN lantern IS lit
      AND hero AT mysterious_dark_room
THEN LOCATE hero AT treasure_chamber.
```

Note that you won’t always need to define a dark location to be a member of the subclass *dark_location*. This applies in cases when you don’t wish to implement lightsource objects to make locations lit or not lit. (*All* location instances have by default the attribute ‘lit’ and they can be made ‘NOT lit’ when needed.) For example, suppose you want all dark locations in the game to become lighted simultaneously. It can be done for example like this:

```
THE main_power_switch ISA DEVICE AT lobby
      VERB switch_on
            DOES ONLY
                  FOR EACH d1 ISA LOCATION, IS NOT lit
                        DO
                        MAKE d1 lit.
                  END EACH.
      END VERB.
END THE.
```

If we had used the *dark_location* class above, all locations to be lighted should have had a lightsource object present in them, and all these lightsource objects would have needed to be changed to ‘lit’, which would have meant extra programming.

Even normal locations, when not lit, will have the description “It is pitch black. You can’t see anything at all.”, so you can use the above method with no worries. The only reason for a specific *dark_location* subclass to exist is to make it automatic for them to be lit or not lit when the hero is carrying around and/or turning on and off lightsources so that the game author won’t constantly need to remember to change the attribute of the location to ‘lit’ or ‘NOT lit’ in all imaginable cases.

Also consider the following case: suppose the hero can make a basement (a location belonging to the class *dark_location*) lighted by turning on a light switch that is at the top of the stairs leading to the basement (a different location from the basement itself). We program the light switch object so that when the hero turns it on, the basement will be ‘lit’. All ok so far. However, when the hero enters the actual basement, it will be dark. Why? Because there is no *lightsource* present in the basement; we just changed the attribute of the basement location to ‘lit’, but this is not enough. A check at entering any *dark_location* will make the location dark if no lit *lightsource* is present. You should program a lamp, a *lightsource* object, to be present in the basement, and this lamp should be made ‘lit’ at the same time when the hero turns on the switch at the top of the stairs. But again, this is more than is necessary to reach the wanted effect. Here, like above, you could just make the basement a normal location and not a dark_location (and make sure it is “NOT lit” to start with), and just change the attribute to ‘lit’ when the hero turns on the light switch.

The ‘visited’ and ‘described’ attributes

IS visited 0.

A location not visited at all has the 'visited' value 0. When the hero enters it the first time, the 'visited' value will change to 1. On the second visit the value will be 2, etc.

Now, in your source code you can define something like the following:

```
THE kitchen ISA LOCATION
      DESCRIPTION
            "You are in the kitchen."
            IF visited OF THIS = 1
                  THEN "This is your first time here."
                  ELSE "You remember you've been here before."
            END IF.
      ...
END THE.
```

Note that if you have an NPC moving around in the game, the visited value of any location will increase when the NPC enters the location, as well (ENTERED applies to all moving actors). This is most often not what is wanted, and that's why an 'if' statement (IF CURRENT ACTOR = hero) is included in the ENTERED section for all locations, in the library.

You can also check whether the hero has been in a location if needed:

```
THE king ISA ACTOR
      ...
      VERB ask
            WHEN act
                  IF topic = treasure_chamber
                        THEN
                              IF visited OF treasure_chamber = 0
                                    THEN "You are not supposed to know anything
                                          about the treasure chamber – you
                                          haven't found it yet."
                                    ELSE """Just take what you want from the
                                          chamber"", the king smiles."
                              END IF.
                  ...
                  END IF.
      END VERB.
END THE.
```

IS described 0.

Suppose you want the location description to be different after the first time the description is shown, even if you are in the location still for the first time. Then, you can use the 'described' attribute. A location not described at all has the ‘described’ value 0. When the player reads the location description for the first time, the value is 1, the next time the value will be 2, etc :

```
THE library ISA ROOM
     DESCRIPTION
          IF described OF THIS = 1
               THEN "There is an old man reading at a table in one of the
                     corners."
               ELSE "The old man keeps on reading at his table."
          END IF.
END THE.
```

or:

```
THE meadow ISA SITE
     DESCRIPTION
          "Flies and other insects buzz around you"
          IF described OF meadow > 5
               THEN ", which starts to annoy you little by little"
          END IF.
          "."
END THE.
```

Nested locations

Nesting locations is straightforward, as described in the ALAN Manual:

```
THE house ISA LOCATION
END THE house.

THE kitchen ISA LOCATION AT house
END THE kitchen.

THE bedroom ISA LOCATION AT house
END THE bedroom.

THE livingroom ISA LOCATION AT house
END THE livingroom.
```

Note: for sites and rooms (= outdoor and indoor locations, respectively) to work correctly when nested, the mother location should be of the same kind as the nested locations. For example, in the example above, if you declare the

kitchen, the bedroom and the living-room to be ROOMS, the house instance should also be declared a ROOM. Sometimes this can bring problems: say you have a driveway location, with a nested location where you are inside your car. The driveway would naturally be a SITE (outdoor location), while the inside of your car is more naturally a ROOM. The best way to solve this is to make both of these locations just LOCATIONs and implement your own floor, walls and ceiling objects for the inside of the car, and your own ground and sky objects for the driveway.

Actors

Pre-defined actor classes in the library:

PERSON
MALE
FEMALE

There are two points where a *person* differs from an ordinary *actor*. Firstly, a person has the ability to talk, i.e. the verbs *ask*, *ask_for*, *say_to*, *talk_to* and *tell* work with persons only. Secondly, actors and persons are described differently when their inventory is empty. Persons are described as for example "The man is empty-handed." while other actors (than persons) are described as for example "The dog is not carrying anything."

Male and *female* are subclasses of *person*, so they have the ability to talk. If you need to implement a male or female animal, do like this:

```
THE dog ISA MALE
   CAN NOT talk.
END THE dog.
```

Pre-defined actor attributes in the library

IS NOT inanimate.
IS NOT following.
IS NOT sitting.
IS NOT lying_down.
IS NOT named.
IS wearing {null_clothing}.
IS NOT compliant.

IS NOT inanimate.

Verbs *push, push_with, rub, scratch, search, touch and touch_with* won't have successful outcomes with animate objects (= actors). To ensure this, the *(NOT) inanimate* attribute is used.

IS NOT following.

By default, NPCs won't follow the hero around the game map.

To make an actor follow the hero, give it the 'following' attribute, for example:

```
THE bob ISA ACTOR
      ...
      VERB whatever
            DOES MAKE bob following.
      END VERB.
      ....

END THE bob.
```

If you wish to have an actor follow the hero right from the start of the game, you can naturally just declare

```
THE servant ISA ACTOR
      IS following.
...
END THE.
```

To stop an actor from following the hero, just make the actor NOT following.

IS NOT sitting.
IS NOT lying_down.

These two attributes exist to allow the author to make the hero, or another actor, sitting or lying down. The outcomes for the commands `>sit` and `>lie down` are not successful by default, however, and must be manually implemented by the author:

```
THE my_game ISA DEFINITION_BLOCK

  VERB sit
      DOES ONLY "You sit down on the floor."
      MAKE hero sitting.
  END VERB.

  VERB lie_down
      DOES ONLY "You lie down on the floor."
      MAKE hero lying_down.
  END VERB.

END THE.
```

Similarly, it is possible for the player to command that the hero (or an NPC) sit or lie down on a supporter object (>lie on bed, >sit on chair), but the action is not successful by default, and must be manually implemented by the author. Refer to: Objects => Supporters.

IS NOT named.

If you don't need an article in front of an actor name (for example 'Jim', as opposed to for example 'a/the man'), declare the instance as 'named':

```
THE jim ISA ACTOR AT room1
      IS named.
       ...
END THE.
```

If you have in your game an actor that starts off as unnamed (such as 'a man'), and the player learns his name later on (say, 'Jim'), you should define the actor in for example the following way to make the player able to refer to him with both 'man' and 'Jim':

```
THE jim ISA PERSON AT room1

      NAME man
      NAME Jim

      PRONOUN him
      MENTIONED
            IF jim IS NOT named
                  THEN "man"
                  ELSE "Jim"
            END IF.

      VERB ask
            WHEN act
                  IF topic = name
                        THEN """My name is Jim"", he replies."
                              MAKE jim named.
                  END IF.
      END VERB.

END THE.
```

IS wearing null_clothing.

By default, the hero character, or any other actor for that matter, isn’t described as wearing any particular clothing, If the author implements some clothing for the hero, this will show up by default in the hero’s description after *examine* (‘x me’).

To implement clothing worn by the hero, locate all such clothing items in the container *worn*:

```
THE hat ISA CLOTHING IN worn
      IS headcover 2.
   DESCRIPTION “”
END THE.
```

Refer also to: Objects => Clothing.

IS NOT compliant.

An actor only gives something to the hero if it is in a compliant mood. In practice, this happens by default only when the hero asks the actor for anything. For example, *take_from* is not successful by default with actors.

Implicit taking of objects is not successful, either, if the object happens to be held by an NPC who is not compliant, and the following happens:

```
>eat apple
That seems to belong to Mr Smith.
```

But if we declare:

```
THE mr_smith ISA ACTOR AT room1
   NAME mr smith
   IS compliant.
   …
END THE.
```

then, the outcome for implicit taking would be successful:

```
>eat apple
(taking the apple first)
You eat all of the apple.
```

To disable even the verb *ask_for*, so that the NPC won't give you something even if you ask for it, use DOES ONLY at the actor instance:

```
THE bob ISA ACTOR AT room1
  …
  VERB ask_for
     WHEN act
       DOES ONLY "He doesn't seem to be cooperative."
  END VERB.

END THE.
```

The hero

The hero instance is left out of the library altogether. If you need to add attributes or verb responses to the hero, define the hero from scratch in your own game source, for example:

```
THE hero ISA ACTOR
     HAS strength 20.
     IS NOT hungry.

     VERB examine
          DOES ONLY "You're John Smith, proud of your unusual name."
     END VERB.

END THE hero.
```

Describing NPCs

When the player types 'examine [actor]', the response will be the default "You notice nothing unusual about [the actor]." If you wish to have the actor's possessions and worn clothing listed after *examine*, you should add "LIST [actor]." manually to the appropriate verb (typically *examine*) of each actor instance:

```
THE boy ISA PERSON
   IS wearing {hat}.

   VERB examine
          DOES ONLY "A boy about twelve years old." LIST boy.
   END VERB.

END THE boy.
```

```
THE coin ISA OBJECT IN boy
END THE.

THE hat ISA CLOTHING IN boy
END THE.
```

will result in:

```
>examine boy
A boy about twelve years old. The boy is carrying a coin and a hat (being
worn).
```

Objects

Pre-defined object classes in the library

CLOTHING
DEVICE
DOOR
LIQUID
LISTED_CONTAINER
SOUND
SUPPORTER
WEAPON
WINDOW

Note: the background and scenery classes introduced in v1.0 have been removed. For backgrounds, use the ‘distant’ or ‘not reachable’ attributes. For scenery objects, use the attribute ‘IS scenery’
.

CLOTHING is a piece of clothing the hero or an NPC wears. Clothes are prevented from being worn in an illogical order, for example you cannot put on a shirt if you are already wearing a jacket, and so forth.

Thanks to Alan Bampton from whose ‘xwear’ extension the code for this class has been adopted.

(The following paragraphs are taken from Alan Bampton's original 'xwear' documentation, with minor alterations.)

The basic idea is that clothing is worn in 'layers' and it is rather silly to allow players to, say, take off or put on a shirt if they are wearing a jacket. To simulate this in ALAN there is a numeric based layering system, and the body is divided into five zones of coverage. The zones are 'head', 'hands', 'feet', 'top' (for top half of torso) and 'bot' (for bottom half of torso). All objects have these five zones defined as default attributes (set to 0), for the head zone the attribute is headcover for the hands zone handscover and so on. Every clothing object will thus need one or more of its 'zonecover' attributes set to reflect the zone(s) it covers and its relative position in the layers of clothing worn. A

simple example would be a shirt, this covers only the 'top' zone and so needs its topcover attribute to be set (to 8, just why it's 8 will become clear shortly).

The principle used is that the closer to the skin an item is normally worn, the lower its 'cover' attribute is. The library operates on the assumption that items with higher value cover attributes for a particular zone are worn over items with lower value attributes. When a player attempts to put on an article of clothing, each zone it would affect is checked and compared to the related zonal total of any clothes already worn. If the value of the new clothing is not greater that the total(s) of clothing already worn (on a zone by zone basis) then the library will not allow the wearing of that item. There are a few notable exceptions to this rule, but those will be dealt with a little later.

How it works in practice

This part of the library might sound complex, but it is actually very simple to use, here's an example: assume our hero starts the game wearing just a vest and shorts and the player issues the command 'put on shirt'. A quick check of the chart below should reveal that a shirt has only its topcover attribute set to non-zero, (all the other zones are zero, which means they are irrelevant for this item). The library totals the topcover attributes of all the clothes currently worn, like so: Starting with a total of 0, and checking the vest, this has a topcover attribute of 2, so total topcover is 0 + 2 = 2. Next the library checks the shorts, these have a topcover attribute of 0, so total topcover is 2 + 0 = 2. As there are no other clothes to consider, the library now compares the topcover attribute of the item we are attempting to put on, (a shirt with a topcover attribute of 8 in this case) to the total value of items already worn (2). Because the topcover attribute of the shirt (8) is greater than the calculated total (2), this is evaluated as being a 'legal' instruction and the library allows the shirt to be put on. Now consider the situation had the player started the game wearing vest, shorts and a jacket, this is what happens should he try to 'put on the shirt.' Starting with a total of 0, and checking the vest, this has a topcover attribute of 2, so total topcover is 0 + 2 = 2. Next the library checks the shorts, these have a topcover attribute of 0, so total topcover is 2 + 0 = 2. Finally the library checks the jacket, this has a topcover attribute of 32, so total topcover is 2 + 32 = 34. Because the topcover attribute of the shirt (8) is now not greater than the calculated total (34), this is evaluated as being an 'illegal' instruction and the library won't allow the shirt to be put on.

That demonstrates the basic principle of the library, removing clothes uses a variation of the 'compare to total' equation to allow / disallow removal of clothing, an example would be that our vest, shorts and jacket wearing player would NOT be allowed to remove the vest while he still had the jacket on.

Exceptions to the rule...

The numbers in the chart are not born of some sort of weird fixation with multiples, there is a very good reason why the numbers are set as they are, computer/math types will recognize the sequence and realize it is all 'binary' based and know it makes it possible to calculate exactly what the player is wearing in terms of layers.

Some female clothing breaks the rules defined above and is not so easy to deal with. An example: although pantyhose is worn under a skirt, dress or coat it can actually be put on or removed with the garment worn over it still on. The library recognizes this capability and deals with it properly by assigning the dress/skirt and coat items particular properties in that they don't affect the ability to wear or remove lower layer clothing that covers the bottom of the torso only. Although it's physically possible to put on/remove trousers while wearing a skirt or dress, this (and a few other neat dressing/undressing tricks) is considered illegal here.

How to create pieces of CLOTHING

Here is a quick overview for using the class 'clothing'.

A piece of clothing in your game code should look something similar to the following four examples:

```
THE jacket ISA CLOTHING AT lobby
      IS topcover 32.
END THE.
```

Use IN to refer to containers:

```
THE jeans ISA CLOTHING IN wardrobe
      IS botcover 16.
END THE.
```

IN worn = worn by the player character (hero):

```
THE hat ISA CLOTHING IN worn
      IS headcover 2.
END THE.
```

Worn by an NPC called Joe:

```
THE sweater ISA CLOTHING IN joe
      IS NOT takeable.
      -- Don't declare numeral clothing attributes for NPCs (unless the hero
      -- is meant to take and wear the NPC's clothing).
      -- NPCs cannot wear clothing in layers!
END THE.

THE joe ISA ACTOR AT room1
      IS wearing {sweater}.
END THE joe.
```

Note above that you must list the clothing worn by an NPC in a set named ‘IS wearing’, at the NPC instance. If you don’t do this, the NPC will be described as just carrying the piece of clothing in his hands.

Note that if the piece of clothing worn by an NPC is not meant to be takeable by the player character, you should declare the piece of clothing to be ‘NOT takeable’.

In defining a piece of clothing, you should

1) define it ISA CLOTHING (and not ISA OBJECT)

2) give it one of five attributes 'headcover', 'topcover', botcover', 'footcover' or 'handcover'; sometimes two of these are needed. Which attribute(s) to use depends on the type of clothing; see the clothing table below.

3) A number 2, 4, 8, 16, 32 or 64 needs to be added after the above attribute. You cannot decide the number yourself; look it up from the clothing table below. If the value of an attribute for a piece of clothing is 0 in the table, don't mention this attribute in connection with your clothing object.

-- The above is enough; the rest is then handled automatically by the library.

The clothing table

Here is the chart showing a selection of fairly typical clothing items and the values to set to obtain appropriate behaviour. Should you wish to create an article of clothing not listed, usually a bit of lateral thought as to what it is most like and where it fits into the scheme of things will suggest a workable set of values, but be aware that you MUST use values in this chart, simply adding things with intermediate values is probably going to create nasty bugs:

Clothing	Headcover	Topcover	Botcover	Foot-cover	Handcover
hat	2	0	0	0	0
vest/bra	0	2	0	0	0
undies/panties	0	0	2	0	0
teddy	0	4	4	0	0
blouse/shirt/T-shirt	0	8	0	0	0
dress/coveralls	0	8	32	0	0
skirt	0	0	32	0	0
trousers/shorts	0	0	16	0	0
sweater/pullover	0	16	0	0	0
jacket	0	32	0	0	0
coat	0	64	64	0	0
socks/stockings	0	0	0	2	0
tights/pantiehose	0	0	8	2	0
shoes/boots	0	0	0	4	0
gloves	0	0	0	0	2

The library, as it stands, also prevents wearing of duplicate clothes, or things that are ogically mutually exclusive - for example the player can wear a dress or a skirt, but not both.

DEVICE is a machine or an electronic device, for example a TV. It can be turned (=switched) on and off if it is not broken. Default attributes: 'NOT 'on', NOT broken. A device is by default described as being either on or off when examined.

For example:

```
THE thingummyjig ISA DEVICE AT lab
END THE.
```

DOOR can be opened, closed, locked and unlocked. It is by default closed (= 'NOT open') and NOT locked. Attributes: openable, NOT open, NOT lockable, NOT locked, HAS otherside door. A door is described by default as being either open or closed when examined.

Locked doors and keys

To unlock a locked door, it has to have a *matching _key* object attributed to it. Only this object can unlock the door.

```
THE wooden_door ISA DOOR AT cellar
      NAME wooden door
      IS lockable. IS locked.
      HAS matching_key iron_key.
END THE locked_door.

THE iron_key ISA OBJECT IN drawer
END THE.
```

By default, it will possible to unlock the door both with >*unlock door* and >*open door* (if the player character is carrying the correct key at the time) as well as with the longer formulations >*unlock door with key* and >*open door with key*.

However, it is not possible to make this automatic by using compass directions only. For example, if the door was to the east of the hero, the command >*e* cannot recognize on the library level whether the hero is carrying the key or not. The author must implement this manually.

Every door between two rooms needs an *otherside* attribute in order for the other side of the door to behave correctly when the door is opened, closed, unlocked and locked.

The *otherside* of a door need not have its other side defined any longer, as the library makes the deduction that if a door has an *otherside*, this other side will have the original door as its *otherside* in turn. Also, the lockable/locked/not locked/openable/open/not open attributes of a door instance will be automatically assumed to be the same for its *otherside* counterpart at the start of a game. The same applies also to the *matching_key* attribute. That's why it is much shorter to implement the otherside of a door:

```
THE wooden_door1 ISA DOOR AT room1
      NAME wooden door
      IS lockable. IS locked.
      HAS matching_key iron_key.
      HAS otherside wooden_door2.
END THE locked_door.

THE wooden_door2 ISA DOOR AT room2
      NAME wooden door
END THE.
```

Above, the wooden_door2 is also lockable and locked at the start of the game, has *wooden_door1* as its otherside and can be opened with *iron_key*. (It wouldn't hurt even if you did declare all of these attributes under *wooden_door2*, to be sure, but it is not necessary.)

See also chapter *Short examples*, example 5.

LIQUID can only be taken if it is in a container. You can fill something with it, and you can pour it somewhere. A liquid is by default NOT drinkable.

If you have some liquid in a container in your game, you should declare the liquid instance this way:

```
THE juice ISA LIQUID
     IN bottle
END THE juice.
```

Then, taking and pouring liquids work smoothly.

The verb 'pour', as defined in this library, also works for the container of a liquid; i.e. if there is some juice in a bottle, 'pour bottle' and 'pour juice' will work equally well. Note, however, that the verb 'empty' is not a synonym for 'pour'; 'empty' only works for container objects. Consequently, `>empty bottle` will work but `>empty juice` won't.

LIGHTSOURCE is natural or NOT natural (a natural lightsource is for example a match or a torch). It can be turned on and off, lighted and extinguished (= put out) if it is not broken. A natural lightsource cannot be turned on or off, it can only be lighted and extinguished (= put out). When examined, a lightsource is by default supplied with a description of whether it is providing light or not. The default attributes for a lightsource object are: natural, NOT lit.

```
THE torch ISA LIGHTSOURCE AT cave
     IS lit.
END THE.
```

```
THE lamp ISA LIGHTSOURCE AT bedroom
      IS NOT natural.
END THE.
```

LISTED_CONTAINER is an object which has the container property. The contents of a *listed_container* will be listed both after 'look' (= in the room description), 'look in' and 'examine', if it is open. (The contents of a normal container object are not automatically listed after 'examine' but only after 'look' (=room description) and 'look in').

To implement a *listed_container* do for example like this:

```
THE box ISA LISTED_CONTAINER AT room1
 ...
END THE.
```

The contents of a *listed_container* are also listed when it is opened. This doesn't happen with normal containers.
For the command 'inventory' to list the contents of a container object the hero is carrying, redefine the verb 'inventory' under the *my_game* instance in your source file for example this way:

```
VERB inventory
      DOES
            IF bag IN hero
                  THEN LIST bag.
            END IF.

            IF box IN hero
                  THEN LIST box.
            END IF.
    ...
END VERB.
```

If you don't do this, the bag and the box will be listed after the command 'inventory' in the following way:

```
"You are carrying a bag and a box."
```

only. But with the above addition, the outcome is for example

```
"You are carrying a bag and a box. The bag contains a loaf of bread. The box
is empty."
```

To declare a *listed_container* the contents of which should not be listed after 'look' or 'examine', declare it an 'opaque container' in the following way:

```
THE box ISA LISTED_CONTAINER
   OPAQUE CONTAINER
 ...
END THE.
```

Objects in an opaque container cannot be seen or manipulated. To change this, declare for example

```
MAKE box NOT OPAQUE.
```

(This is handled automatically by the library when a container is opened or closed.)

Putting things in containers

It is only possible to put something into a container if this something is included in the '*allowed*' set of the container object.

```
THE drawer ISA LISTED_CONTAINER IN nightstand
      HAS allowed {diary, keys}.
           …
END THE drawer.
```

In the example above, it wouldn't be possible to put anything else in the drawer, for example a chair or a suitcase. The response would be for example "The chair doesn't belong in the drawer.", etc.

This applies not only to the verb *put_in* but also to *empty_in*, *pour_in* and *throw_in.*

Everything programmed to be in a container by the author at the start of the game will be automatically included in the 'allowed' set of the container. Thus, for example if the author implements an apple in a bowl and the hero character takes it, it will be possible for the hero to put the apple back into the bowl, without the author having to implement any separate 'allowed' attributes for this to happen. But note if you have for example a ticket dispenser in your game and the hero takes a ticket from it, it would be possible to put the ticket back into the dispenser, the way things work by default. This is not what is wanted in this case. That's why in that case you should do either:

```
THE ticket_dispenser ISA LISTED_CONTAINER AT lobby
…
      VERB put_in
           WHEN cont
                DOES "That's not possible."
      END VERB.

END THE.
```

or, alternatively:

```
THE ticket ISA OBJECT IN ticket_dispenser

  INITIALIZE
      EXCLUDE THIS FROM allowed OF ticket_dispenser.

END THE ticket.
```

SOUND can be listened to but not examined, searched, smelled or manipulated. It cannot initially be turned on or off, this has to be implemented manually by giving the sound the 'switchable' attribute.

SUPPORTER You can put things on a supporter and you can stand, sit down or lie on it. Standing, sitting or lying down on a supporter is not allowed by default, however, but must be manually implemented by the author:

```
THE bed ISA SUPPORTER AT bedroom
   …
   VERB lie_on
      DOES ONLY
           “You lie down on the bed.”
           MAKE hero lying_down.
   END VERB.
END THE.
```

Remember that it is not possible to locate an actor inside an object, for example in a bed container. Using the sitting or lying_down attributes should be enough to account for these situations and to create the impression that the hero is located on a supporter object. When the hero is made sitting or lying_down, certain actions are disabled by the library (for example *attack*, *jump* etc.). It is the author’s responsibility to make certain objects in the location NOT reachable as needed, while the hero is lying down or sitting, and also to prohibit movement or at least implement a clarifying message of the hero standing up, before going in any direction.

A supporter is declared a container, so that you can take things from it. When there's something on a supporter, a listing of it will appear in the room description and after 'examine', by default.

To place objects on a supporter:

Define the supporter first; for example

```
THE tray ISA SUPPORTER
...
END THE.
```

Then, implement the objects on the supporter like this:

```
THE apple ISA OBJECT
      IN tray
      ...
END THE.
```

Note the IN above, even if the apple will be described as being *on* the tray. Similarly, to implement a book to be found on a table (the table being a supporter object) :

```
THE book ISA OBJECT
   IN table
    ...
END THE.
```

Note that the 'examine' command will list what is on the surface of a supporter, not what, if anything, is inside the supporter. For example, if you have a supporter called 'table' in your game with two drawers in it,

DON'T do this:

```
THE drawer1 ISA OBJECT
     NAME bottom drawer
     CONTAINER
       IN table.
END THE.
```

or this:

```
THE drawer2 ISA LISTED_CONTAINER
     NAME top drawer
     IN table.
END THE.
```

This would result in something like "There's a table here. On the table you see a book, a bottom drawer and a top drawer."

Instead, do the following:

```
THE table ISA SUPPORTER
     AT bedroom
     HAS components {drawer1, drawer2}.
       ...
     VERB examine
           DOES
                FOR EACH c IN components OF THIS DO
                     SAY "The table has" SAY AN c. "."
                          IF c IS open
                               THEN LIST c.
                               ELSE SAY THE c. "is closed."
                          END IF.
                END FOR.
       END VERB.
      ...
END THE.

THE drawer1 ISA LISTED_CONTAINER
    OPAQUE CONTAINER
    NAME bottom drawer
    AT bedroom
      IS NOT open.
END THE.

THE drawer2 ISA LISTED_CONTAINER
      NAME top drawer
      AT bedroom
      IS open.
END THE.

THE book ISA OBJECT IN table
 ...
END THE book.

THE diary ISA OBJECT IN drawer2
 ...
END THE diary.
```

In other words, declare the drawers components of the table, in the manner described above. The result will then be for example something like this:

"You see a table here. There is a book on the table. The table has a bottom drawer. The bottom drawer is closed. The table has a top drawer. The top drawer contains a diary."

WEAPON is fireable (for example a cannon) or NOT fireable (for example a baseball bat), the latter being the default. The verbs *attack_with* and *kill_with* won't have successful outcomes if the second parameter in them is not a weapon. (Even when the second parameter is a weapon, the outcome of the action is not successful by default. You must implement a successful outcome manually at the instance level.)

```
THE pistol ISA weapon IN room1
   IS fireable.
END THE.
```

WINDOW can be opened, closed, looked through and out of. It will be described as being either open or closed when examined, by default. It is by default NOT open.

```
THE bedroom_window ISA WINDOW AT bedroom
    NAME bedroom window
    IS open.
END THE.
```

The 'my_game' instance and its attributes

My_game is a so called meta-instance that obligatorily has to be included by the author in the game source. Without it, the game won't compile successfully. At its shortest, the needed formulation is

```
THE my_game ISA DEFINITION_BLOCK
END THE.
```

It is called a meta-instance because everything the author defines inside it affects the whole game. The purpose of this instance is to make it less necessary for the author to access the library files to make changes to common game responses and messages needed in the game. That's why the instance is named "my_game" – the author can override library responses and replace them with default responses that better suit the particular work in progress. The things that the author can define within this instance are

a) default verb responses
   for example "There is nothing special about the key."
b) check responses
   for example "You don't have the key."
c) illegal parameter messages
   for example "That's not something you can eat."
d) the implicit taking message
   for example "(taking the key first)"

In addition, the author can let the game formulate automatically the game title, subtitle, author, year, and game version at game start. This is done through attributes of the *my_game* instance.

It is also possible to for the author to implement custom global attributes within this instance, for example:

```
THE my_game ISA DEFINITION BLOCK
   HAS tasks_left 10.
   HAS treasures_found 0.
   …
END THE my_game.
```

A typical *my_game* instance would look something like this:

```
THE my_game ISA DEFINITION BLOCK

  HAS title "The House In The Fog".
  HAS subtitle "An interactive ghost hunt".
  HAS author "Xavier Y. Zamborsky".
  HAS year 2016.
  HAS version 1.

  HAS enemies_defeated 0.

  VERB examine
     DOES ONLY “Nothing special.”
  END VERB.

  VERB eat
     CHECK hero IS hungry
           ELSE “You’re not hungry.”
  END VERB.

  HAS check_obj_not_scenery_sg “That’s just scenery.”.
  HAS check_obj_not_scenery_pl “Those are just scenery.”.

  HAS illegal_parameter_talk_sg "You can’t possibly talk to that.".
  HAS illegal_parameter_talk_pl "You can’t possibly talk to those.”.

END THE my_game.
```

In the following, all the various attributes of the *my_game* instance are listed.

1) Attributes for the start section

The following five attributes have been declared for the game start:

```
HAS title "My New Game".
HAS subtitle "".
HAS author "An ALAN Author".
HAS year 0000.
HAS version “1”.
```

If you set the version value to “0” (zero), the version line won’t be shown at all in the game banner. Note also that the version number is in quotes. This enables any kind of textual input to describe the current version, for example `version “beta0.1”,` and so on.

If the subtitle line remains an empty quote (“”), like above, it won’t show in the banner.

NOTE: In order for the banner to show up correctly, the line

```
DESCRIBE banner.
```

needs to be added after the START AT declaration:

```
START AT bedroom.
“You knew that this evening would be different from usual when you found the
mysterious note pushed under your front door.”
DESCRIBE banner.
```

See also example (3) at the end of this manual.

2) Attributes for the hero

```
HAS hero_worn_header “You are wearing”
HAS hero_worn_else “You are not wearing anything.”
```

Change these to alter the way the hero is described as far clothing is concerned. If no clothing is defined for the hero in the game, these messages won’t show at any time. By default, these messages show at *>inventory*. If the author wishes to have the clothing of the hero described after *>examine me*, the *examine* verb for the hero should be defined this way:

```
THE hero ISA ACTOR
…

  VERB examine
         DOES “Blah blah…”
         LIST worn.
  END VERB.

END THE hero.
```

3) Attributes for locations

```
HAS dark_loc_desc “It is pitch black. You can’t see anything at all.”
```

This is the default location description for dark locations. It is shown every time the hero enters a dark location or types “LOOK” while there. Edit this to change the default description of dark locations. If/when a dark location is lighted, this description won’t be shown any longer.

```
HAS light_goes_off "It is now pitch black.".
```

This message is shown when a light goes off and the location becomes dark.

4) Illegal parameter messages

In this section, all illegal parameter messages used by the library are listed. If you wish to change any of these, you can declare them again in the *my_game* instance.

NOTE: If you need to change a great number, or all, of these messages, for example if you’re writing in another language or you need to change the person or the tense of these messages to better suit your narrative, it is highly recommended that you edit the file ‘mygame_import.i’ in the library distribution package, find the list of these messages there, edit them, and import the ‘mygame_import.i’ file to your game source (together with the library). ‘mygame_import.i’ is a file that lists all the pre-defined attributes of the *my_game* instance for easy modification. It is included in the library distribution package but is not necessarily needed to run a game. It makes sense to re-declare these messages within the *my_game* instance in your own source file ONLY if you need to change a small number that you are not satisfied with. Looking through the list of these parameter messages in ‘mygame_import.i’ will give you a much better overview of them and make it easier to edit them in a uniform way to suit your purposes.

NOTE ALSO that changing illegal parameter messages is usually not the first priority of a game author and in many cases they are left as is, as defined by the library. It is much more common to modify the standard verb outcomes or add checks of your own to existing library checks, for example. If changing illegal parameter messages is not a high priority for you, you might wish to skip directly to the next section.

The illegal parameter messages, as also the verb check messages and implicit taking messages further below, use the $ parameter naming approach.

Key to the parameter symbols used in ALAN:

$v	the verb the player used
$1	the first parameter the player used (for example the noun after the first verb used), without any articles, for example "key" in the command "examine key")
$+1	the definite form of the first parameter the player used (for example "the key")
$-1	the negative form of the first parameter the player used (for example "no key") (not used in the library)
$01	the indefinite form of the first parameter the player used (for example "a key")
$2	etc. would be the second parameter the player used, (for example the word "key" in "unlock door with key")

The general message for when a parameter is not suitable with the verb (for example "That's not something you can attack"):

```
HAS illegal_parameter_sg "That's not something you can $v.".
HAS illegal_parameter_pl "Those are not something you can $v.".
```

The library accounts for singular and plural cases; that's why many messages have both a singular (sg) and a plural (pl) formulation.

In the following there are variations of the above message when a preposition is required after the verb (for example "That's not something you can ask about." or "That's not something you can cut things with."):

For verbs requiring *about* (the library verbs *ask_about*, *tell_about* and *think_about*):

```
HAS illegal_parameter_about_sg "That's not something you can $v about.".
HAS illegal_parameter_about_pl "Those are not something you can $v about.".
```

There are two ditransitive verbs requiring *at* in the library, *fire_at* (e.g"fire rifle at bear") and *throw_at* (for example "throw remote control at TV")*:*

```
HAS illegal_parameter_at "You can't $v anything at $2."
```

The following is needed for the verb *ask_for* (for example "ask servant for tea"):

```
HAS illegal_parameter_for_sg "That's not something you can $v for.".
HAS illegal_parameter_for_pl "Those are not something you can $v for.".
```

The verb *take_from* needs the following formulations:

```
HAS illegal_parameter_from_sg "That's not something you can take things from.
HAS illegal_parameter_from_pl "Those are not something you can take things
from.".
```

The verbs *dive_in*, *jump_in*, *lie_in* and *swim_in* use the following parameter messages:

```
HAS illegal_parameter_in_sg "That's not something you can $v in.".
HAS illegal_parameter_in_pl "Those are not something you can $v in.".
```

Climb_on, *jump_on*, *knock*, *lie_on*, *sit_on*, *stand_on*, *switch_on*, *turn_on*, for their part, use the following messages:

```
HAS illegal_parameter_on_sg "That's not something you can $v on.".
HAS illegal_parameter_on_pl "Those are not something you can $v on.".
```

For *get_off*, *switch_off* and *turn_of f*, the following parameter messages are used:

```
HAS illegal_parameter_off_sg "That's not something you can $v off.".
HAS illegal_parameter_off_pl "Those are not something you can $v off.".
```

The preposition *to* is needed in the verbs *listen_to* and *talk_to*:

```
HAS illegal_parameter_to_sg "That's not something you can $v to.".
HAS illegal_parameter_to_pl "Those are not something you can $v to.".
```

A slightly different message is needed for *give*, *show*, *tell*, *tie_to*, *throw_to* which are ditransitive verbs with the second parameter preceded by *to*:

```
HAS illegal_parameter2_to_sg "That's not something you can $v things to.".
HAS illegal_parameter2_to_pl "Those are not something you can $v things to.".
```

For *with*, we have two separate messages. The verbs *kill_with*, *shoot_with* and *play_with* use the following formulation:

```
HAS illegal_parameter_with_sg "That's not something you can $v with.".
HAS illegal_parameter_with_pl "Those are not something you can $v with.".
```

while a somewhat bigger group of verbs - *attack_with*, *break_with*, *burn_with*, *close_with*, *cut_with*, *fill_with*, *lock_with*, *open_with*, *pry_with*, *push_with*, *unlock_with* - are accompanied with a message one word longer: *things* is added, for no other reason than that it sounds better than if left out, as far as these verbs are concerned:

```
HAS illegal_parameter2_with_sg "That's not something you can $v things with.".
HAS illegal_parameter2_with_pl "Those are not something you can $v things
with.".
```

The communication verbs *ask, ask_for, say_to, talk_to* and *tell* use a message of their own:

```
HAS illegal_parameter_talk_sg "That's not something you can talk to.".
HAS illegal_parameter_talk_pl "Those are not something you can talk to.".
```

We have a separate individual default parameter message for a handful of verbs.

For *consult*, we have the following:

```
HAS illegal_parameter_consult_sg "That's not something you can find
                                        information about."
HAS illegal_parameter_consult_pl "Those are not something you can find
                                      information about."
```

You'll find this message at *examine* :

```
HAS illegal_parameter_examine_sg "That's not something you can examine.".
HAS illegal_parameter_examine_pl "Those are not something you can examine.".
```

The reason why *examine* doesn't use the general default message (scroll above) is that when the player types for example'x 34' the response would be "That's not something you can x." which isn't such pretty-looking as when the verb is printed in full.

The verbs *look_out_of* and *look_through* use prepositions other verbs don't, and that's why they need their own messages:

```
HAS illegal_parameter_look_out_sg "That's not something you can look out of.".
HAS illegal_parameter_look_out_pl "Those are not something you can look out
of.".
HAS illegal_parameter_look_through "You can't look through $+1.".
```

Other illegal parameter messages

The above are the default messages and their variations. There are, however, other illegal parameter messages needed at places. They are described below.

The following message is displayed when the player tries to for example put something into an actor instance. The verbs in which this message is found are *empty_in*, *pour_in*, *put_in*, and *throw_in.*:

```
HAS illegal_parameter_act "That doesn't make sense.".
```

The following message is displayed when the player tries to use the verbs *give, put, put_in, put_on, put_against, put_near, put_behind, put_under, throw_at, throw_in, throw_to, use* and *use_with* with actors as direct objects:

```
HAS illegal_parameter_obj "You can only $v objects.".
```

The verbs *answer*, *say*, *say_to* and *write* require that what we wish to answer, say or write is put into a string (= surrounded by quotes).

```
HAS illegal_parameter_string "Please state inside double quotes (""""") what
you want to $v.".
```

The verbs *look_behind*, *look_in* and *look_under* have the following message when the player tries to look somewhere that is not suitable object for these verbs:

```
HAS illegal_parameter_there "It's not possible to $v there.".
```

The verb *go_to* has its own message:

```
HAS illegal_parameter_go “It’s not possible to go there.”
```

The following is a variation of the above and is used when the second parameter of a ditransitive verb is not suitable.

The verbs *empty_in*, *empty_on*, *pour_in*, *pour_on*, *put_in*, *put_on*, *put_against*, *put_behind*, *put_near*, *put_under*, *throw_in*, *throw_to*, *tie_to* and *write* use this message:

```
HAS illegal_parameter2_there "It's not possible to $v anything there.".
```

Finally, there are some messages for the information “verbs” *what_is*, *where_is* and *who_is*. (The first two messages below also apply to *where_is* besides *what_is*.)

```
HAS illegal_parameter_what_sg "That's not something I know about.".
HAS illegal_parameter_what_pl "Those are not something I know about.".
HAS illegal_parameter_who_sg "That's not somebody I know about.".
HAS illegal_parameter_who_pl "Those are not somebody I know about.".
```

5) Default verb check messages

All these check messages can be individually changed by listing them under the *my_game* instance in your game source file. They are also listed in the file ‘mygame_import.i’ in the library distribution package, for easy modification. These check messages are used in verb definitions, mainly in ‘lib_verbs.i’. Changing one check message will affect all verbs where that particular check is found. Again, as with parameter messages, edit these messages directly in ‘mygame_import.i’ if you need to change a great number of them, otherwise redefine them within the *my_game* instance in your own source file. You’ll quickly notice that the list is quite long, and listing any number greater than just a few under the *my_game* instance would be a rather frustrating task.

a) attribute checks

The general check message for when an instance cannot be used with the verb :

```
HAS check_obj_suitable_sg "That's not something you can $v.".
HAS check_obj_suitable_pl "Those are not something you can $v.".
```

Thus, if the player tries to for example eat something that is not edible,

```
>eat book
That’s not something you can eat.
```

the check message will be displayed.

Note that the illegal parameter messages (above) mostly report cases where the player tried to use a *wrong kind of instance* with a verb:

```
>take 5
That’s not something you can take.
```

The verb *take* only works with objects, not with any other instances. Thus, if you try to take something else than an object (for example a numerical value in the above case), an illegal parameter message is shown. This restriction is defined in the syntax of the verb. Checks, on the other hand, are used to ensure that an instance has *the proper attribute* needed with the verb, for example edible, takeable, NOT open, and so forth.

Variations of the above message, needed for example when a preposition is required after the verb, are listed below:

fire_at, throw_at, throw_to:

```
HAS check_obj_suitable_at “You can’t $v anything at $+2.”
```

ask_for :

```
HAS check_obj2_suitable_for_sg "That's not something you can $v for.".
HAS check_obj2_suitable_for_pl "Those are not something you can $v for.".
```

turn_off, switch_off:

```
HAS check_obj_suitable_off_sg "That's not something you can $v off."
HAS check_obj_suitable_off_pl "Those are not something you can $v off.".
```

knock, switch_on, turn_on:

```
HAS check_obj_suitable_on_sg "That's not something you can $v on.".
HAS check_obj_suitable_on_pl "Those are not something you can $v on."  .
```

play_with:

```
HAS check_obj_suitable_with_sg "That's not something you can $v with.".
HAS check_obj_suitable_with_pl "Those are not something you can $v with.".
```

break_with, burn_with, close_with, cut_with, fill_with, lock_with, open_with, pry_with, push_with, touch_with, unlock_with:

```
HAS check_obj2_suitable_with_sg "That's not something you can $v things
with.".
HAS check_obj2_suitable_with_pl "Those are not something you can $v things
with.".
```

Again, we have a separate message for *examine*, *look_out_of* and *look_through*:

```
HAS check_obj_suitable_examine_sg "That's not something you can examine.".
HAS check_obj_suitable_examine_pl "Those are not something you can examine.".

HAS check_obj_suitable_look_out_sg "That's not something you can look out
of.".
HAS check_obj_suitable_look_out_pl "Those are not something you can look out
of.".

HAS check_obj_suitable_look_through "You can't look through $+1.".
```

Checks for open, closed and locked objects

open, open_with:

```
HAS check_obj_not_open_sg "$+1 is already open.".
HAS check_obj_not_open_pl "$+1 are already open.".
```

close, close_with:

```
HAS check_obj_open1_sg "$+1 is already closed.".
HAS check_obj_open1_pl "$+1 are already closed.".
```

empty, empty (in/on), look_in, pour (in/on):

```
HAS check_obj_open2_sg "You can't, since $+1 is closed.".
HAS check_obj_open2_pl "You can't, since $+1 are closed.".
```

empty_in, pour_in, put_in, throw_in:

```
HAS check_obj2_open_sg "You can't, since $+2 is closed.".
HAS check_obj2_open_pl "You can't, since $+2 are closed.".
```

unlock, unlock_with:

```
HAS check_obj_locked_sg "$+1 is already unlocked.".
HAS check_obj_locked_pl "$+1 are already unlocked.".
```

lock, lock_with:

```
HAS check_obj_not_locked_sg "$+1 is already locked.".
HAS check_obj_not_locked_pl "$+1 are already locked.".
```

Checks for "not reachable" and "distant" objects

A large number of verbs have the following checks:

```
HAS check_obj_reachable_sg "$+1 is out of your reach.".
HAS check_obj_reachable_pl "$+1 are out of your reach.".

HAS check_obj_not_distant_sg "$+1 is too far away.".
HAS check_obj_not_distant_pl "$+1 are too far away.".
```

In addition, the verbs *empty_in, fill_with, pour_in, put_in, take_from* and *tie_to* have the following check for the reachability of the second parameter:

```
HAS check_obj2_reachable_sg "$+2 is out of your reach.".
HAS check_obj2_reachable_pl "$+2 are out of your reach.".
```

and the verb *ask_for* has the following check:

```
HAS check_obj_reachable_ask "$+1 wouldn't be able to reach $+2.".
```

which is triggered when the hero asks an NPC for something that the NPC cannot reach. (This happens when the object in question has the attribute ‘NOT reachable’.)

The verbs *throw_at, throw_in, throw_to* allow the action to succeed if the second parameter is reachable, but not if the second parameter is distant:. Thus, the way things are defined in the library, it is possible to e,g, throw something in a container if that container is otherwise not reachable. But if the container is distant, the action will fail.

```
HAS check_obj2_not_distant_sg "$+2 is too far away.".
HAS check_obj2_not_distant_pl "$+2 are too far away.".
```

Checks for the hero sitting or lying_down

Numerous verbs in the library have one of the following checks for sitting:

```
HAS check_hero_not_sitting1 "It is difficult to $v while sitting down.".

HAS check_hero_not_sitting2 "It is difficult to $v anything while sitting
down.".

HAS check_hero_not_sitting3 "It is difficult to $v anywhere while sitting
down.".
```

and for lying down:

```
HAS check_hero_not_lying_down1 "It is difficult to $v while lying down.".

HAS check_hero_not_lying_down2 "It is difficult to $v anything while lying
down.".

HAS check_hero_not_lying_down3 "It is difficult to $v anywhere while lying
down.".
```

If the player uses the verbs *sit* or *sit_on*, and the hero is already sitting, the following check message is displayed:

```
HAS check_hero_not_sitting4 "You're sitting down already.".
```

If the player uses the verbs *lie_down* or *lie_in*, and the hero is already lying down, the following check message is displayed:

```
HAS check_hero_not_lying_down4 "You're lying down already.".
```

Other attribute checks

Checking that the object of the action has the ability to talk; verbs *ask, ask_for, say_to, tell:*

```
HAS check_act_can_talk_sg "That's not something you can talk to.".
HAS check_act_can_talk_pl "Those are not something you can talk to.".
```

Checking that the object is allowed to be emptied/poured/put/thrown in the container (*empty_in, pour_in, put_in, throw_in*):

```
HAS check_obj_allowed_in_sg "$+1 doesn’t belong in $+2".
HAS check_obj_allowed_in_pl “$+1 don’t belong in $+2.”
```

Checking that something is broken; the verb *fix:*

```
HAS check_obj_broken_sg "That doesn't need fixing.".
HAS check_obj_broken_pl "Those don't need fixing.".
```

Checking that the object of the action is inanimate, because normally the action would be considered improper if done to a person: *pull, push, push_with, scratch, search*

```
HAS check_obj_inanimate1 "$+1 wouldn't probably appreciate that.".
```

With some verbs, the above message is slightly altered; *rub, touch, touch_with:*

```
HAS check_obj_inanimate2 "You are not sure whether $+1 would appreciate
that.".
```

Checking if something is movable; the verbs *lift, pull, push, push_with, shake, take, take_from:*

```
HAS check_obj_movable "It's not possible to $v $+1.".
```

Checking whether something is scenery; the verbs *examine, take, take_from:*

```
HAS check_obj_not_scenery_sg "$+1 is not important.".
HAS check_obj_not_scenery_pl “$+1 are not important.”.
```

In the verbs *ask_for* and *take_from* there is also a check for whether the second parameter in the command happens to be a scenery object:

```
HAS check_obj2_not_scenery_sg "$+2 is not important.".
HAS check_obj2_not_scenery_pl “$+2 are not important.”.
```

For some verbs, the target of looking is checked with the following message: *look_behind, look_under:*

```
HAS check_obj_suitable_there "It's not possible to $v there.".
```

The verbs *throw_in* and *tie_to* has a slightly different formulation from the above:

```
HAS check_obj2_suitable_there "It's not possible to $v anything there.".
```

The following check is found in verbs in which implicit taking is possible but the present instance is not takeable:

```
HAS check_obj_takeable "You don't have $+1.".
```

fill_with has the following check:

```
HAS check_obj2_takeable1 "You don't have $+2.".
```

while *ask_for* has:

```
HAS check_obj2_takeable2 "You can't have $+2.".
```

Checking that an object is not too heavy (*lift*, *take*, *take_from*):

```
HAS check_obj_weight_sg "$+1 is too heavy to $v.".
HAS check_obj_weight_pl "$+1 are too heavy to $v.".
```

Checking that an object can be written in/on:

```
HAS check_obj_writeable "Nothing can be written there.".
```

b) location and containment checks for actors and objects

Location and containment checks for actors other than the hero (checks for the hero are listed separately below):

For the verb *follow* to work successfully, the actor to be followed should be in an adjacent location to the hero. The following check will verify this:

```
HAS check_act_near_hero "You don't quite know where $+1 went.
           You should state direction where you want to go.".
```

If the hero tries to take something from an NPC and the NPC doesn't have the stated object, the following check is triggered (*take_from*):

```
HAS check_obj_in_act_sg "$+2 doesn't have $+1.".
HAS check_obj_in_act_pl "$+2 don't have $+1.".
```

Similarly, if the player types `>give object to actor`, and the actor already has that object, the following check message is displayed:

```
HAS check_obj_not_in_act_sg "$+2 already has $+1.".
HAS check_obj_not_in_act_pl "$+2 already have $+1.".
```

Location and containment checks for the hero

The following checks deal with where the hero is or what (s)he is carrying.

The verb *shoot* has the following check:

```
HAS check_count_weapon_in_hero "You are not carrying any firearms.".
```

while the following is need for *exit:*

```
HAS check_hero_in_cont "But you aren't in $+1 at present.".
```

enter:

```
HAS check_hero_not_in_cont "But you are already in $+1!".
```

find, follow, go_to, where_is:

```
HAS check_obj_not_at_hero_sg "$+1 is right here.".
HAS check_obj_not_at_hero_pl "$+1 are right here.".
```

drop, fire, fire_at, put, show:

```
HAS check_obj_in_hero "You don't have the $+1.".
```

The following check is used in many verbs, typically ditransitive ones such as *break_with*, *cut_with* etc:

```
HAS check_obj2_in_hero "You don't have the $+2.".
```

In the following, the action tried out by the player is targeted at something the hero is holding, and the action would not make sense (verbs *attack, attack_with, kick, lift, shoot* and *shoot_with*):

```
HAS check_obj_not_in_hero1 "It doesn't make sense to $v something you're
holding.".
```

The following check ensures that the hero is not trying to get something (s)he already has (the verbs *take, take_from*):

```
HAS check_obj_not_in_hero2 "You already have $+1.".
```

The throwing verbs (*throw_at, throw_in throw_to*) have this check to prohibit the hero from throwing something at, to or into something that (s)he is holding:

```
HAS check_obj2_not_in_hero1 "You are carrying $+2.".
```

For "putting" verbs other than *put_in* and *put_on*, the following check ensures that the player cannot succeed in putting something against, behind, near, on or under something else when (s)he carries the object referenced by second parameter (the verbs *put_against*, *put_behind*, *put_near*, *put_under*):

```
HAS check_obj2_not_in_hero2 "That would be futile.".
```

Thus, if the player is for example carrying a book, the command

```
>put apple near book
```

wouldn't be successful.

If the hero already is carrying an object that (s)he asks for, the following check message is displayed:

```
HAS check_obj2_not_in_hero3 "You already have $+2.".
```

Checking whether an object is in a container or not

When the following check fires, the player tried to empty the contents of an object into a container that already was contained by the object (for example if there is a bottle in a box, and the player types "empty box in bottle"). This applies to the verbs *empty_in* and *pour_in*:

```
HAS check_cont_not_in_obj "That doesn't make sense.".
```

If the player tries to take something from a container and that something is not there to begin with, the following check message is displayed (*take_from*):

```
HAS check_obj_in_cont_sg "$+1 is not in $+2.".
HAS check_obj_in_cont_pl "$+1 are not in $+2.".
```

If the player tries to put or throw something into a container but the object is already in the container, the following message is displayed *(put_in, throw_in):*

```
HAS check_obj_not_in_cont_sg "$+1 is in $+2 already.".
HAS check_obj_not_in_cont_pl "$+1 are in $+2 already.".
```

The following check message is displayed when the player tries to fill a container with something that the container already is full of (*fill_with*):

```
HAS check_obj_not_in_cont2_sg "$+1 is already full of $+2.".
HAS check_obj_not_in_cont2_pl "$+1 is already full of $+2.".
```

Checking whether an object is on a surface or not (*take_from*):

```
HAS check_obj_on_surface_sg "$+1 is not on $+2.".
HAS check_obj_on_surface_pl "$+1 are not on $+2.".
```

Putting something on a surface (*put_on*):

```
HAS check_obj_not_on_surface_sg "$+1 is already on $+2.".
HAS check_obj_not_on_surface_pl "$+1 are already on $+2.".
```

Checking whether an object is worn by the hero or not:

You can't take off something you're not wearing *(remove, take_off)*:

```
HAS check_obj_in_worn "You are not wearing $+1.".
```

The following check is for cases when the hero tries to put on something (s)he is already wearing *(put_on, wear)*:

```
HAS check_obj_not_in_worn1 "You are already wearing $+1.".
```

Here, the action is stopped if the hero tries to attack, kick or shoot something (s)he's wearing *(attack, attack_with, kick, shoot, shoot_with):*

```
HAS check_obj_not_in_worn2 "It doesn't make sense to $v something you're
wearing.".
```

Lastly, it's not possible to drop a piece of clothing if it is worn. It will have to be removed first *(drop)*:

```
HAS check_obj_not_in_worn3: "You'll have to take off $+1 first."
```

c) checking location states

The following check is found in numerous verbs. It prohibits actions requiring seeing when the location is not lit:

```
HAS check_current_loc_lit "It is too dark to see.".
```

d) logical checks

The checks in this group a) prohibit the action from being directed at the hero, and 2) prohibit the action in ditransitive verbs where both the first and the second parameter refer to the same instance.

1) prohibiting the action from being directed at the hero:

The following check is triggered when the player tries something like "attack me"*(ask, ask_for, attack, attack_with, catch, follow, kick, listen, pull, push, push_with, take, take_from, tell)* :

```
HAS check_obj_not_hero1 "It doesn't make sense to $v yourself.".
```

For the verbs *fire_at, kill, kill_with, shoot, shoot_with* there is a specific message when the target of the action is the hero:

```
HAS check_obj_not_hero2 "There is no need to be that desperate.".
```

For a couple of actions where the hero is the target, the action might make sense but it is anyway not deemed fruitful. This applies to the verbs *scratch* and *touch*:

```
HAS check_obj_not_hero3 "That wouldn't accomplish anything.".
```

The verbs *find* and *go_to* have the following check triggered when the player types "find me" or "go to me":

```
HAS check_obj_not_hero4 "You're right here.".
```

If the player tries "free me", the following check message is displayed *(free)*:

```
HAS check_obj_not_hero5 "You don't have to be freed.".
```

The verbs *kiss, play_with* and *rub* have the following check:

```
HAS check_obj_not_hero6 "There's no time for that now.".
```

The verb *look_behind* has the following check for cases when the hero looks behind him-/herself *:*

```
HAS check_obj_not_hero7 "Turning your head, you notice nothing unusual behind
yourself.".
```

while *look_under* has the following one*:*

```
HAS check_obj_not_hero8 "You notice nothing unusual under yourself.".
```

Many ditransitive verbs have the following check when the hero tries to perform these actions to her-/himself *(say_to, show, take_from, touch_with, throw_at, throw_in, throw_to)*:

```
HAS check_obj2_not_hero1 "That doesn't make sense.".
```

Lastly, some other cases:

put_against, put_behind, put_near, put_under:

```
HAS check_obj2_not_hero2 "That would be futile.".
```

give, tie_to:

```
HAS check_obj2_not_hero3 "You can't $v things to yourself.".
```

2) prohibiting the action in ditransitive verbs where both the first and the second parameter refer to the same instance:

The following checks prohibit actions like "cut rope with rope", "throw stone at stone" and "put bottle in bottle":

fire_at, throw_at:

```
HAS check_obj_not_obj2_at "It doesn't make sense to $v something at itself.".
```

take_from:

```
HAS check_obj_not_obj2_from "It doesn't make sense to $v something from
itself.".
```

empty_in, pour_in, put_in, throw_in:

```
HAS check_obj_not_obj2_in "It doesn't make sense to $v something into
itself.".
```

empty_on, pour_on, put_on:

```
HAS check_obj_not_obj2_on "It doesn't make sense to $v something onto
itself.".
```

give, show, throw_to, tie_to:

```
HAS check_obj_not_obj2_to "It doesn't make sense to $v something to itself.".
```

attack_with, break_with, burn_with, close_with, cut_with, fill_with , lock_with, open_with, pry_with, push_with, shoot_with, touch_with, unlock_with, use_with:

```
HAS check_obj_not_obj2_with "It doesn't make sense to $v something with
itself.".
```

put_against, put_behind, put_near, put_under:

```
HAS check_obj_not_obj2_put "That doesn't make sense."  .
```

e) additional checks for classes

Lastly, there are some checks that apply only to a specific class. Most of these are found in 'lib_classes.i'.

The first one checks that a male character doesn't put on women's clothing by default, and vice versa:

```
HAS check_clothing_sex "On second thoughts you decide $+1 won't really suit
you.".
```

The following check ensures that it won't be possible to put something inside a supporter object:

```
HAS check_cont_not_supporter "You can't put $+1 inside $+2.".
```

If the player tries to turn off a device that is already off, the following check is triggered (*turn_off, switch_off*):

```
HAS check_device_on_sg "$+1 is already off.".
HAS check_device_on_pl "$+1 are already off.".
```

The following message is triggered if the player tries to turn on a device which is already on (*device: turn_on, switch_on*)

```
HAS check_device_not_on_sg "$+1 is already on.".
HAS check_device_not_on_pl "$+1 are already on.".
```

If the player tries to unlock or lock a door with something that is not the matching key of the door in question (*lock_with, unlock_with*):

```
HAS check_door_matching_key "You can't use $+2 to $v $+1.".
```

The following message is for situations where the player tries to turn off or extinguish a lightsource that is not lit (*lightsource: extinguish, turn_off*):

```
HAS check_lightsource_lit_sg "But $+1 is not lit.".
HAS check_lightsource_lit_pl "But $+1 are not lit.".
```

while the following is for the opposte case *(lightsource: light, turn_on)*:

```
HAS check_lightsource_not_lit_sg "$+1 is already lit.".
HAS check_lightsource_not_lit_pl "$+1 are already lit.".
```

Checking that the verb switch won't work with a natural lightsource (*lightsource: switch*):

```
HAS check_lightsource_switchable_sg "That's not something you can switch on
and off."  .
HAS check_lightsource_switchable_pl "Those are not something you can switch on
and off.".
```

When there is some liquid in a container, for example some juice in a bottle, and the player types *>take juice from bottle*, the following check is triggered (*liquid: take_from*):

```
HAS check_liquid_vessel_not_cont "You can't carry $+1 around in your bare
hands.".
```

When the player tries to turn on a device or light a lightsource which is broken, the following check message is displayed (*device, lightsource: light, turn_on*):

```
HAS check_obj_not_broken "Nothing happens.".
```

6) Implicit taking message

```
HAS implicit_taking_message "(taking $+1 first)$n".
```

The following verbs use implicit taking:

bite, drink, eat, empty, empty_in, empty_on, give, pour, pour_in, pour_on, put_in, put_on, throw, throw_at, throw_in, throw_to, tie_to.

In ditransitive verbs, only the first parameter (the direct object) is taken implicitly. For example,

```
>push door with pole
```

won't work if the hero is not carrying the pole (= the pole is not taken implicitly).

Have the game banner show at the start

To show the game banner at the start, after an optional intro text, you must add the text "`DESCRIBE banner.`" after the START AT clause, for example:

```
START AT room1.
DESCRIBE banner.
```

or:

```
START AT room1.
"This is the (optional) intro text at the start of the game, before the first
location description."
DESCRIBE banner.
```

The following attributes should be added to the *my_game* instance, for example:

```
HAS title "The Baffling Case Of Mrs Wells".
HAS subtitle "An interactive mystery".
HAS author "Sam".
HAS year 2015.
HAS version 1.
```

Leaving the subtitle line out and setting the version number to "0" will omit these lines from the banner. As it stands now, these attributes would produce the following kind of banner text:

```
The Baffling Case Of Mrs Wells
An interactive mystery
© 2015 by Sam
Programmed with the ALAN Interactive Fiction Language v3.0
Version 1
All rights reserved
```

The philosophy used in deciding successful and unsuccessful outcomes for action in the library verbs

If you try different actions in-game, with the library imported, you will notice that some actions are successful and result in what the player commanded, while other actions do nothing (= the action is unsuccessful). For example the response to > drop [object] will be "Dropped.", the object being rejected from the hero's inventory and ending up in the location, while the response to "attack [thing]" is "Resorting to brute force is not the solution here.". Which actions are allowed to succeed and which are not is based on what is the most reasonable and expected outcome for the action – the outcome that the game author most unlikely needs to edit except for special circumstances. Please

experiment with different verbs in-game to see whether the outcome of a particular action is suitable for your game – otherwise redefine the outcome of the verb in the *my_game* instance.

Runtime messages

Many of the runtime messages built into ALAN have been altered in the library from their default wording as stated in the ALAN manual. This is to ensure that plural is handled correctly and that there are no clashes between first and second person. The first person of some default wordings (for example "I don't know the word "$1") is changed to a more passive or impersonal formulation. To edit these for your game, open 'lib_messages.i' and edit the wanted message(s) there.

```
MESSAGE
      AFTER_BUT: "You must give at least one object after '$1'."
      AGAIN: ""
      BUT_ALL: "You can only use '$1' AFTER '$2'."
      CAN_NOT_CONTAIN: "$+1 can not contain $+2."
      CANT0: "You can't do that."
            -- note that the fifth token in CANT0 is a zero, not an 'o'.
      CARRIES:
            IF parameter1 = hero
                  THEN "You are carrying"
                  ELSE
                        IF parameter1 IS NOT plural
                              THEN "$+1 carries"
                              ELSE "$+1 carry"
                        END IF.
            END IF.

      CONTAINMENT_LOOP:
            "Putting $+1 in"
                  IF parameter1 IS NOT plural
                        THEN "itself"
                        ELSE "themselves"
                  END IF.
            "is impossible."
      CONTAINMENT_LOOP2:
            "Putting $+1 in $+2 is impossible since $+2 already"
                  IF parameter2 IS NOT plural
                        THEN "is"
                        ELSE "are"
                  END IF.
           "inside $+1."
      'CONTAINS':
            IF parameter1 IS NOT plural
                  THEN "$+1 contains"
                  ELSE "$+1 contain"
            END IF.
```

```
CONTAINS_COMMA: "$01,"
CONTAINS_AND: "$01 and"
CONTAINS_END: "$01."
EMPTY_HANDED:
      IF parameter1 = hero
            THEN "You are empty-handed."
            ELSE
                  IF parameter1 IS NOT plural
                        THEN "$+1 is empty-handed."
                        ELSE "$+1 are empty-handed."
                  END IF.
      END IF.
HAVE_SCORED: "You have scored $1 points out of $2."
IMPOSSIBLE_WITH: "That's impossible with $+1."
IS_EMPTY:
      IF parameter1 IS NOT plural
            THEN "$+1 is empty."
            ELSE "$+1 are empty."
      END IF.
MORE: "<More>"
MULTIPLE: "You can't refer to multiple objects with '$v'."
NO_SUCH: "You can't see any $1 here."
NO_WAY: "You can't go that way."
NOT_MUCH: "That doesn't leave much to $v!"
NOUN: "You must supply a noun."
NOT_A_SAVEFILE: "That file does not seem to be an Alan game save
      file."
QUIT_ACTION: "Do you want to RESTART, RESTORE, QUIT or UNDO? "
      -- these four alternatives are hardwired to the interpreter and cannot be changed.
REALLY: "Are you sure (press ENTER to confirm)?"
RESTORE_FROM: "Enter file name to restore from"
SAVE_FAILED: "Sorry, save failed."
SAVE_MISSING: "Sorry, could not open the save file."
SAVE_NAME: "Sorry, the save file did not contain a save for this
     adventure."
SAVE_OVERWRITE: "That file already exists, overwrite (y)?"
SAVE_VERSION: "Sorry, the save file was created by a different
     version."
SAVE_WHERE: "Enter file name to save in"
SEE_START:
      IF parameter1 IS NOT plural
            THEN "There is $01"
            ELSE "There are $01"
      END IF.
SEE_COMMA: ", $01"
SEE_AND: "and $01"
SEE_END: "here."
NO_UNDO: "No further undo available."
UNDONE: "'$1' undone."
UNKNOWN_WORD: "The word '$1' is not understood."
```

```
WHAT: "That was not understood."
WHAT_WORD: "It is not clear what you mean by '$1'."
WHICH_PRONOUN_START: "It is not clear if you by '$1'"
WHICH_PRONOUN_FIRST: "mean $+1"
WHICH_START: "It is not clear if you mean $+1"
WHICH_COMMA: ", $+1"
WHICH_OR: "or $+1."
```

Verb syntaxes used in the standard library

Verb	Synonyms	Syntax
about	(+ help, info)	about
again	(+ g)	again
answer	(+ reply)	answer (topic)
ask	(+ enquire, inquire, interrogate)	ask (act) about (topic)
ask_for		ask (act) for (obj)
attack	(+ beat, fight, hit, punch)	attack (target)
attack_with		attack (target) with (weapon)
bite		bite (obj)
break	(+ destroy)	break (obj)
break_with		break (obj) with (instr)
brief		brief
burn		burn (obj)
burn_with		burn (obj) with (instr)
buy	(+ purchase)	buy (item)
catch		catch (obj)
clean	(+ polish, wipe)	clean (obj)
climb		climb (obj)
climb_on		climb on (surface)
climb_through		climb through (obj)
close	(+ shut)	close (obj)
close_with		close (obj) with (instr)
consult		consult (source) about (topic)
credits	(+ acknowledgments, author, copyright)	credits
cut		cut (obj)
cut_with		cut (obj) with (instr)
dance		dance
dig		dig (obj)
dive		dive
dive_in		dive in (liq)
drink		drink (liq)
drive		drive (vehicle)
drop	(+ discard, dump, reject)	drop (obj)
eat		eat (food)
empty		empty (obj)

```
empty_in                                          empty (obj) in (cont)
empty_on                                          empty (obj) on (surface)
enter                                             enter (cont)
examine          (+ check, inspect, observe, x)   examine (obj)
exit                                              exit (cont)
extinguish       (+ put out, quench)              extinguish (obj)
fill                                              fill (cont)
fill_with                                        fill (cont) with (substance)
find             (+ locate)                       find (obj)
fire                                              fire (weapon)
fire_at                                           fire (weapon) at (target
fix              (+ mend, repair)                 fix (obj)
follow                                            follow (act)
free             (+ release)                      free (obj)
get_up                                            get up
get_off                                           get off (obj)
give                                              give (obj) to (recip)
go_to                                             go to (dest)
hint            (+ hints)                        hint
inventory       (+ i, inv)                       inventory
jump                                              jump
jump_in                                           jump in (cont)
jump_on                                           jump on (surface)
kick                                              kick (target)
kill            (+ murder)                       kill (victim)
kill_with                                         kill (victim) with (weapon)
kiss            (+ hug, embrace)                 kiss (obj)
lie_down                                          lie down
lie_in                                            lie in (cont)
lie_on                                            lie on (surface)
lift                                              lift (obj)
light           (+ lit)                          light (obj)
listen0                                           listen
listen                                            listen to (obj)
lock                                              lock (obj)
lock_with                                         lock (obj) with (key)
look            (+ gaze, peek)                   look
look_at                                           look at (obj)
look_behind                                       look behind (bulk)
look_in                                           look in (cont)
look_out_of                                       look out of (obj)
look_through                                      look through (bulk)
look_under                                        look under (bulk)
look_up                                           look up
no                                                no
notify (on, off)                                  notify.
                                                  notify on.
                                                  notify off.
open                                              open (obj)
open_with                                         open (obj) with (instr)
```

```
play                                                play (obj)
play_with                                           play with (obj)
pour          (= defined at the verb 'empty')       pour (obj)
pour_in       (= defined at the verb 'empty_in')    pour (obj) in (cont)
pour_on       (= defined at the verb 'empty_on')    pour (obj) on (surface)
pray                                                pray
pry                                                 pry (obj)
pry_                                                pry (obj) with (instr)
pull                                                pull (obj)
push                                                push (obj)
push_with                                           push (obj) with (instr)
put               (+ lay, place)                    put (obj)
put_against                                         put (obj) against (bulk)
put_behind                                          put (obj) behind (bulk)
put_down                                            put down (obj)
put_in            (+ insert)                        put (obj) in (cont)
put_near                                            put (obj) near (bulk)
put_on                                              put (obj) on (surface)
put_under                                           put (obj) under (bulk)
read                                                read (obj)
remove                                              remove (obj)
restart                                             restart
restore                                             restore
rub                                                 rub (obj)
save                                                save
say                                                 say (topic)
say_to                                              say (topic) to (act)
score                                               score
scratch                                             scratch (obj)
script                                              script. script on. script
                                                          off.
search                                              search (obj)
sell                                                sell (item)
shake                                               shake (obj)
shoot (at)                                          shoot at (target)
shoot_with                                         shoot (target) with (weapon)
shout             (+ scream, yell)                  shout
show              (+ reveal)                        show (obj) to (act)
sing                                                sing
sip                                                 sip (liq)
sit (down)                                          sit.  sit down.
sit_on                                              sit on (surface)
sleep             (+ rest)                          sleep
smell0                                              smell
smell                                               smell (odour)
squeeze                                             squeeze (obj)
stand (up)                                          stand.  stand up.
stand_on                                            stand on (surface)
swim                                                swim
swim_in                                             swim in (liq)
```

```
switch_on     (defined at the verb 'turn_on')    switch on (app)
switch_off    (defined at the verb 'turn_off')   switch off (app)
take         (+ carry, get, grab, hold, obtain)  take (obj)
take_from       (+ remove from)                  take (obj) from (holder)
talk                                             talk
talk_to         (+ speak)                        talk to (act)
taste           (+ lick)                         taste (obj)
tear            (+ rip)                          tear (obj)
tell            (+ enlighten, inform)            tell (act) about (topic)
think                                            think
think_about                                      think about (topic)
throw                                            throw (projectile)
throw_at                                      throw (projectile) at (target)
throw_in                                        throw (projectile) in (cont)
throw_to                                 throw (projectile) to (recipient)
tie                                              tie (obj)
tie_to                                           tie (obj) to (target)
touch           (+ feel)                         touch (obj)
turn            (+ rotate)                       turn (obj)
turn_on                                          turn on (app)
turn_off                                         turn off (app)
undress                                          undress
unlock                                           unlock (obj)
unlock_with                                      unlock (obj) with (key)
use                                              use (obj)
use_with                                         use (obj) with (instr)
verbose                                          verbose
wait            (+ z)                            wait
wear                                             wear (obj)
what_am_i                                        what am i
what_is                                          what is (obj)
where_am_i                                       where am i
where_is                                         where is (obj)
who_am_i                                         who am i
who_is                                           who is (obj)
write                                            write (txt) on (obj)
yes                                              yes
```

To see the outcome for all verbs above, check the file ‘mygame_import.i’ where you’ll find a list of all verb outcomes. The syntaxes of these verbs are defined in the library file ‘lib_verbs.i’.

Note that the ‘exit’ and ‘enter’ verbs won’t have successful outcomes by default; after all, it is impossible to place an actor (like the hero) inside a container in the current version of ALAN. To make for example the command *>enter car* work, you should make the car a separate location and then locate the hero there at the DOES ONLY part of the ‘enter’ verb in the car instance. In other words, simulate entering and exiting by locating the hero in between locations.

Default attributes used in the standard library

The attributes in the following list are pre-defined in the library. When you coin your own attributes for your game, please be aware that these attributes already exist.

This attribute is added to every ‘entity’:

```
NOT plural.
```

These attributes are added to every ‘thing’:

```
IS examinable.
   inanimate.
   movable.
   open.
   reachable.
      -- See also 'distant' below
   takeable.


HAS allowed {null_object}.
     -- You can only put an object in a container if the object
     -- is in the ‘allowed’ set of the container.
HAS matching_key null_key.
     -- All lockable objects need a matching key to lock/unlock them.
     -- "null_key" is a default dummy that can be ignored.
HAS text "".
HAS weight 0.
     -- Actors and objects will have different weight values, see below


NOT broken.
NOT distant.
     -- Usage: you can for example talk to a "not reachable" actor but -
     -- not to a "distant" one.
     -- You can also throw things in, to or at a not reachable target
          -- but not to a distant one.
     -- The other verbs where the action succeeds if the object is
          -- not reachable are: dive_in, fire_at, kill_with, read, and
          -- shoot
     -- Default response for not reachable things: "The [thing] is out
          -- of your reach."
     -- Default response for distant things: "The [thing] is too far
          -- away."
NOT drinkable.
NOT edible.
NOT fireable.
     -- can (not) be used as a firearm
```

```
NOT lockable.
NOT locked.
NOT openable.
NOT readable.
NOT scenery.
     -- has special responses for ‘ask_for’, 'examine', 'take' and
     -- ‘take_from’, behaves like a normal object otherwise.
NOT wearable.
NOT writeable.
CAN NOT talk.
```

These attributes are added to every ’actor’:

```
IS wearing {null_clothing}.
     -- By default, actors are not described as wearing any specific
     -- clothing. null_clothing is a default dummy value that can be
     -- ignored.

HAS weight 50.
     -- If something has the weight value of 50 or more, it cannot
     -- be lifted or taken.


NOT following.
     -- not following the hero character by default

NOT inanimate.
NOT named.
NOT compliant.
NOT sitting.
NOT lying_down.
```

The code for clothing objects adds these attributes, used only internally in the library, to every actor:

```
IS tempcovered 0.
IS wear_flag 0.
IS sex 0.
```

These attributes are added to every object:

```
HAS weight 5.
     -- This is the default weight of every object, whether takeable
     -- or NOT takeable. However, the library by itself
     -- doesn’t define any limit for containers. If the game author
     -- wants to have a limit to how many objects a container can hold,
     -- the author must set this limit by themselves.
```

Attributes added to specific classes of objects:

These attributes are added to every clothing object:

```
IS wearable.
IS NOT donned.     -- = not worn by an NPC
IS sex 0.
IS headcover 0.
IS handscover 0.
IS feetcover 0.
IS topcover 0.
IS botcover 0.
```

The following attribute is defined for every door object:

```
HAS otherside door.
```

The following attribute is added to every device object:

```
IS NOT ‘on’.
```

The following attributes are added to every lightsource object:

```
IS natural.
IS NOT lit.
```

The following attribute is added to every weapon:

```
IS NOT fireable.
```

The following attributes are added to every location:

```
IS lit.
HAS visited 0.
HAS described 0.
HAS nested {nowhere}.
```

Finally, the score notification coding uses the following attributes:

```
HAS oldscore 0.
IS notify_on.
IS NOT seen_notify.
```

Translating to other languages

To translate the ALAN system and library to other languages, you should

1) translate all the messages in the file 'lib_definitions.i':
- the two messages for the hero
- the two messages for dark locations
- all illegal parameter messages
- all verb check messages
- the message for implicit taking

- the message lines for the banner instance where applicable

2) translate the verb syntaxes in 'lib_verbs.i' (not parameters and the ELSE parts).

For example for the verb 'attack' when translated into French:

```
SYNTAX attaquer = attaquer (target)
    WHERE target ISA THING
      ELSE
            IF target IS NOT plural
                  THEN SAY illegal_parameter_sg OF my_game.
                  ELSE SAY illegal_parameter_pl OF my_game.
            END IF.
```

Also, translate the verb names, for example `VERB attack DOES` … becomes, translating into French, `VERB attaquer DOES` … etc.), and the verb outcomes (what happens after DOES).

3) translate the verb outcomes for class objects (what happens after DOES or DOES ONLY) in 'lib_classes.i'.

4) translate the direction names, their synonyms and the few marginal verb outcomes for indoor and outdoor objects in 'lib_locations.i'

5) translate the runtime messages in 'lib_messages.i'.

Now, every possible response and message in the game is shown in the target language, and it is possible for the player to issue commands in the target language.

Reference guide

This reference guide will deal with subjects some of which are already handled previously in this manual, while other subjects are dealt with here for the first time.

I want to…

… override the library response to a verb

Define the verb outcome with a DOES ONLY section within the *my_game* instance:

```
THE my_game ISA DEFINITION_BLOCK

   VERB examine
          DOES ONLY “Nothing special really.”
    END VERB.

 END THE.
```

… override the library response to a verb within a specific class:

```
EVERY cat ISA ACTOR

      VERB examine
            DOES ONLY “It’s just an ordinary cat.”
      END VERB.

END EVERY.
```

This will override the default library message for *examine* for all cats in the game.

If you want to change the verb outcome for a class predefined in the library, do like below. Here, the verb outcome for *examine* has been modified for all windows in the game:

```
THE my_game ISA DEFINITION_BLOCK

      VERB examine
            DOES ONLY
                  IF obj ISA WINDOW
                        THEN “Better not be looking out of windows.”
                               -- this applies for all windows
                        ELSE “Nothing special really.”
                              -- this applies for all other obejcts
                  END IF.
      END VERB.

END THE.
```

… override the library response to a verb within a specific instance only

Use DOES ONLY at the instance:

```
THE little_cat ISA ACTOR AT garden

      VERB examine
          DOES ONLY “It’s a little black and white cat with yellow
                     eyes.”
      END VERB.

END THE.
```

… to add a check to a library verb

Add the check to the verb under the *my_game* instance and *not* in the library file:

```
THE my_game ISA DEFINITION_BLOCK

      VERB take
           CHECK COUNT ISA ACTOR, AT hero = 1
                 -- ( = only the hero, and nobody else, is present)
           ELSE “Remember that in this game you are a thief.
                 You shouldn’t take anything while there is
                 somebody else in the same location.”
      END VERB.

END THE.
```

Note that there is no DOES ONLY part above. (The hero will be counted as one actor “AT hero”. That’s why above, the count check is formulated the way it is.)

… to add a check to a verb for a specific class

```
ADD TO EVERY cat
     IS takeable.

     VERB take
          CHECK nails OF THIS ARE cut
                ELSE “You might just get scratched.”
     END VERB.

END ADD.
```

Note that there is no DOES/DOES ONLY section here; the check is performed on the cat class only, and if the check is passed, the library outcome of the *take* verb will be carried out.

… to add a check to a verb for a specific instance

Add the check to the instance (and not to the library nor to the *my_game* instance):

```
THE soup ISA OBJECT AT kitchen
     IS edible.
     IS NOT hot.

     VERB eat
          CHECK soup IS hot
             ELSE "You must warm the soup first."
     END VERB.

END THE soup.
```

Note that there is no DOES/DOES ONLY section here; the check is performed on the soup instance only, and if the check is passed, the library outcome of the *eat* verb will be carried out.

… to add an attribute to the hero

The hero instance is predefined in the ALAN language and you don't have to define it in your own game source if you are not planning to add any attributes or other features, like specific verb responses, to it. If you need to add even a single attribute of your own to the hero instance, you should define the hero instance explicitly in your own game source:

```
THE hero ISA ACTOR
     IS NOT hungry.
     IS NOT sleepy.

     VERB examine
          DOES ONLY "You're you."
     END VERB.

END THE hero.
```

Note: the hero always has the container property (so that it is able to pick up and carry objects). The container property never needs to be stated explicitly for the hero; that's why there is no mention of the hero being a container in the above code snippet, either.

… to change the syntax of a library verb

a) without accessing the library:

Let's say that you want to for example change the syntax of the *talk_to* verb. Elsewhere in this manual you'll find all verb syntaxes listed. From there, you'll find out that the syntax of the *talk* verb is

```
talk_to = talk 'to' (act).
```
Let's imagine that you want to change this so that it's possible for the player to type

```
>talk man
```

or just

```
>t man
```

i.e. stating the character with whom you wish to talk, after the verb, without the preposition 'to'.

The easiest way to allow this is just to add an additional syntax for 'talk_to' in your own game file:

```
SYNTAX talk_to = talk (act).

       talk_to = t (act).
```

This syntax declaration should be *outside* the *my_game* instance, in your game file. This syntax declaration won't cancel the original syntax for 'talk_to' defined in the library; it would still be possible for the player to type >`talk to man`, as well.

If you wish to cancel the original syntax altogether, do like this in your own game file:

```
THE my_game ISA DEFINITION BLOCK

     VERB talk_to
          DOES ONLY "To talk to someone, type ""talk [person]"" or just
                    ""t [person]""."
     END VERB.

END THE my_game.
```

Then, outside the *my_game* instance, define your own *talk* verb, for example:

```
SYNTAX my_talk_to = talk (act)
        WHERE act ISA ACTOR
              ELSE …

VERB my_talk_to
      DOES
         IF act = mr_smith
             THEN…
         ELSIF…
END VERB.

SYNONYMS t = talk.
```

b) accessing the library:

Find the verb in the library file ‘lib_verbs.i’ and make the desired changes to the syntax. (If you add or change a parameter, make sure that the verb checks function properly.)

… to change the pre-defined illegal parameter messages in syntax statements

Please refer to a previous section in this manual where all illegal parameter messages are listed.
To change some of them, re-declare them within the *my_game* instance:

```
THE my_game ISA DEFINITION_BLOCK

     HAS illegal_parameter_sg “You can’t $v $+1.”
     HAS illegal_parameter_pl “You can’t $v $+1.”
     HAS illegal_parameter_there “You can’t $v there.”

END THE.
```

If you need to change a great number, or all, of the parameter messages (for example to adjust the messages for an unusual narrative perspective, or if you’re writing in a language other than English), it’s better to open ‘mygame_instance.i’ and find the list of illegal parameter messages, and do all the changes straight there. This will save you a lot of typing. Remember to import the ’mygame_instance.i’ file to your game.

… to change the pre-defined illegal parameter message of a single verb

The way the illegal parameter messages have been defined in the library, it is not usually possible to affect just one verb at a time. Most often, changing a default message will alter the outcome of at least a handful of verbs, because one default message is shared by many verbs. There are some default parameter messages that only affect one verb; you should check the list of parameter messages (above) for details. Anyway, the quickest way to accomplish this task would be to open ‘lib_verbs.i’, find the verb, then modify the appropriate parameter message in its syntax statement.

… to change the wording of a library-defined verb check

Find the check in the list of check messages in this manual, or in ‘mygame_import.i’. Re-declare it in within the *my_game* instance. Here we change the wording for *check_obj_writeable* which in its default form is “Nothing can be written there.”:

```
THE my_game ISA DEFINITION_BLOCK

     HAS check_obj_writeable "You can't write anything on $2."

END THE.
```

… to remove a check from a verb

This requires accessing the library. Go to 'lib_verbs.i', find the verb you wish to remove a check from and remove the check. (Make sure the behavior of things in your game remains sensible; the library verb checks, after all, are there to ensure that everything functions in a reasonable and rational way.)

… to change the attribute of a class:

a) giving a predefined attribute to a class of your own:

```
 EVERY broken_vase ISA OBJECT

    IS broken.

 END EVERY.
```

Objects are by default NOT broken. If we want a class of objects that all share the same attribute, like "broken", we can override the default this way.

b) changing the predefined attribute of a class defined in the library.

Use INITIALIZE in the *my_game* instance. Here, the game author wants (for some reason) to make all CLOTHING objects not wearable:

```
THE my_game ISA DEFINITION_BLOCK

     INITIALIZE

          FOR EACH c ISA CLOTHING DO
               MAKE c NOT wearable.
          END FOR.

END THE my_game.
```

… make a new verb for your game

Declare a new verb in the normal manner in your own game source file, outside any instances:

```
SYNTAX test = test.

VERB test
     DOES “Test successful.”
END VERB.
```

or

```
SYNTAX test = test (obj)
      WHERE obj ISA OBJECT
              ELSE “That’s not something you can test.”



ADD TO EVERY OBJECT
      VERB test
            DOES “You test” SAY THE obj. “successfully.”
      END VERB.
END ADD.
```

… make a verb apply to one instance only (for example > `cross the street`, with *cross* meant to be working with the street instance only:

Place the verb within the needed instance:

```
THE street ISA OBJECT AT town

      VERB cross
            DOES “There’s too much traffic.”
      END VERB.

END THE.
```

This way, the verb only applies to the street instance. Its syntax won’t need to be defined anywhere else. If the player tries to use it somewhere else, for example >cross table, the outcome will be “You can’t do that.” (the default outcome for the MESSAGE CANT0), unless the player defines an individual outcome for the verb under another instance where a successful outcome for the verb is wished:

```
THE brook ISA OBJECT AT forest
     VERB cross
          DOES “You would just get your feet wet.”
     END VERB.
END THE.
```

… add a synonym for an existing verb

Declare the synonym in your own game source file, outside any instance declarations, and outside the *my_game* instance, like this:

```
SYNONYMS peruse = read.
```

… **override automatic implicit taking**

Locate the verb(s) for which you want to override implicit taking in ‘lib_verbs.i’ or in ‘mygame_import.i’, find their DOES ONLY sections and delete the implicit taking code.

… edit default runtime messages

This is not possible without accessing the library. Please open the file ‘lib_messages.i’ in the library files and edit the wanted messages there.

Short examples

1) A very short complete game using minimal obligatory imports and coding. Here, the hero must go from room1 north to room2 and eat an apple to win the game.

```
IMPORT 'library.i'.

THE my_game ISA DEFINITION_BLOCK
END THE.

THE room1 ISA LOCATION
    DESCRIPTION "North to room2."
    EXIT north TO room2.
END THE.

THE room2 ISA LOCATION
    DESCRIPTION "South to room1."
    EXIT south TO room1.
END THE.

THE apple ISA OBJECT AT room2
    IS edible.
    VERB eat
        DOES "Congratulations!" QUIT.
    END VERB.
END THE.

START AT room1.
DESCRIBE banner.
```

(This game wouldn't actually need the library at all; it would be even shorter to code:)

```
THE room1 ISA LOCATION
     DESCRIPTION "North to room2."
     EXIT north TO room2.
END THE.



THE room2 ISA LOCATION
     DESCRIPTION "South to room1."
     EXIT south TO room1.
END THE.
```

```
THE apple ISA OBJECT AT room2
      VERB eat
            DOES “Congratulations!” QUIT.
      END VERB.
END THE.

START AT room1.
```

In this latter case, though, the player wouldn’t for example be able to examine him-/herself, take inventory, try various things with the apple, quit properly, etc.

Examples 2-4 below show mainly different variations of the *my_game* instance and not complete games:

2) In this example of defining the *my_game* instance, the author has changed the default verb responses for ‘eat’, ‘climb’ and ‘take_from’. In addition, the author has added a check and a response of his/her own to ‘take_from’:

```
THE my_game ISA DEFINITION_BLOCK

      VERB eat
            DOES ONLY “You don’t feel like eating anything in this game.”
      END VERB.

      VERB climb
            DOES ONLY “Let’s just stay on the ground, shall we?”
      END VERB.

      VERB take_from
            WHEN obj
            CHECK COUNT ISA ACTOR, AT hero = 1   -- ( = the hero himself)
                  ELSE “You don’t want to take anything while somebody
                        might be looking.”
            DOES “Triumphantly, you fish” SAY THE obj. “out of”
                  SAY THE holder. “.”
      END VERB.

END THE.
```

3) Here, the author uses the automatic formulation for the game title, author, and other information:

```
THE my_game ISA DEFINITION_BLOCK

      HAS title "The Lost Treasure".
      HAS subtitle "An interactive treasure hunt".
      HAS author "Sam".
      HAS year 2015.
      HAS version 1.

END THE.

THE garden ISA LOCATION
     DESCRIPTION “…”
END THE.

START AT garden.
DESCRIBE banner.
```

4) Here, the game author has added a check of his own to the library-defined ‘drink’ verb and changed an illegal parameter message for the verbs *look_behind*, *look_in*, and *look_under*:

```
THE my_game ISA DEFINITION_BLOCK

      VERB drink
            CHECK hero IS thirsty
                  ELSE “You don’t feel like drinking anything right now.”
      END VERB.

      HAS illegal_parameter_there “You can’t $v there.".

END THE.
```

5) A complete example game with locked doors and keys. This code reintroduces the situation used in example 1, with a locked door and two keys added.

```
IMPORT 'lib_classes.i'.
IMPORT 'lib_definitions.i'.
IMPORT 'lib_locations.i'.
IMPORT 'lib_messages.i'.
IMPORT 'lib_verbs.i'.

THE my_game ISA DEFINITION_BLOCK
END THE.
```

```
THE room1 ISA LOCATION
   DESCRIPTION "North to room2."
    EXIT north TO room2
      CHECK locked_door_1 IS open
            ELSE "The door to the north is on the way."
    END EXIT.
END THE.

THE locked_door_1 ISA DOOR AT room1
      DESCRIPTION ""
      NAME door
      HAS otherside locked_door_2.
      IS lockable. IS locked.
      HAS matching_key silver_key.
END THE.

THE silver_key ISA OBJECT AT room1
      NAME silver key
END THE.

THE brass_key ISA OBJECT AT room1
      NAME brass key
END THE.

THE room2 ISA LOCATION
DESCRIPTION "South to room1."
      EXIT south TO room1
            CHECK locked_door_2 IS open
                  ELSE "The door to the south is on the way."
      END EXIT.
END THE.

THE locked_door_2 ISA DOOR AT room2
      DESCRIPTION ""
      NAME door
END THE.

THE apple ISA OBJECT AT room2
      IS edible.

      VERB eat
            DOES "Congratulations!" QUIT.
      END VERB.

END THE.

START AT room1.
DESCRIBE banner.
```