LambdaCore Database User's Manual

For LambdaMOO version 1.3
April 1991

Mike Prudence (blip)
Simon Hunt (Ezeke)
Floyd Moore (Phantom)
Kelly Larson (Zaphod)
Al Harrington (geezer)

Introduction

The LambdaCore database provides the facilities needed to make a LambdaMOO server useful for Multi User Adventuring. If you compare the LambdaMOO server to a piece of computer hardware, then LambdaCore is the operating system needed to allow the user to do useful work.

This document gives a rundown on the commands of the LambdaCore database, providing examples of how they are used, and some of the ideas behind them. It does not cover how the commands work, nor how they are implemented as verbs in the MOO programming language. A companion manual, **The LambdaCore Programmer's Manual** covers those aspects of the LambdaCore database.

The user may find it useful to read the other companion manual to this one, *The LambdaMOO Programmer's Manual*. An understanding of MOO concepts and the MOO language can be useful when playing the game.

1 The LambdaCore Player Commands

A player interacts with the game using a large number of commands. Most of these commands are implemented within the database as verbs of object classes.

The following sections list the commands in the LambdaCore database, grouped roughly by function. Some commands, such as those used for manipulating notes, are defined by the particular class of object they work on, in this case the `Note` class. Other commands are defined for one or more classes.

This section intends to give a paper reference for the information given in the *help* system within the LambdaCore database, with some additional explanation of the concepts involved.

Note that, for commands that can be abbreviated, the form in which the command is specified in the database is shown. For example, the `inventory` command is written down as

```
i*nventory
```

which means that this command can be invoked by the player typing any of the following:

```
i
in
inv
inven
etc...
```

2 Commands That Affect Your Player

The `$player` class defines a number of verbs that allow the player to change and view certain of his/her characteristics. The following commands are available :

help/information/? Command
help *topic* Command
help index Command
help *object:verbname* Command
help *$something_utils* Command

This command is used to print out entries from the online documentation system. The first form prints out a summary table of contents for the entire help system.

The second form prints out the documentation available on the given *topic*. Many help system entries contain references to other entries accessible in this way. The topic name may be abbreviated in either of two ways: you may give only a prefix of the complete topic name (e.g., '`bui`' instead of '`building`') and you may omit an initial '`@`' character (e.g., '`who`' instead of '`@who`'). If the abbreviation you give is ambiguous, you will be presented with a list of the matching complete topic names.

The '`help index`' commands prints out a complete list of all help topic names, arranged alphabetically. It is sometimes easier to find the topics you're interested in this way, rather than tracing through the chain of cross references.

Finally, we have two addtional forms that are likely to be only of use to programmers:

`help` *object:verbname*

This is used to print any documentation strings that are present at the beginning of the program for that verb.

`help` $*foo_utils*

prints general information about one of the `$..._utils` objects (e.g., `$string_utils`, `$list_utils`, etc...), which are all libraries of generally used verbs.

The commands '`?`' and '`information`' (usually abbreviated '`info`') are synonyms for '`help`'.

@quit Command

This command is used to disconnect from the MOO. This breaks your network connection and leaves your player sleeping. Disconnecting in most parts of the MOO automatically returns your player to its designated home.

@gender *gender* Command
@gender Command

The first form, with an argument, defines your player to have the gender *gender*. If *gender* is one of the standard genders (e.g., '`male`', '`female`', '`neuter`',...), your various pronouns will also be set appropriately, making exits and certain other objects behave more pleasantly for you.

The second form tells you the current definition of your player's gender, your current pronouns, and the complete list of standard genders.

@password *old-password new-password* Command

Changes your player's password (as typed in the '`connect`' command when you log in to the MOO) to *new-password*. For security reasons, you are required to type your current (soon to be old) password as the first argument.

Your password is stored in an encrypted form in the MOO database; in principle, not even the wizards can tell what it is, though they can change it, of course.

@sethome Command

Sets your designated home (see '`help home`') to be the room you're in now. If the current room wouldn't allow you to teleport in, then the '`@sethome`' command nicely refuses to set your home there. This avoids later, perhaps unpleasant, surprises.

3 Exploring and Interacting With the Virtual World

The main purpose of the core classes in the LambdaCore database is to allow players to construct and explore a virtual world. This involves moving from one room to another, using designated pathways or *exits* and looking at objects and locations along the way. The command given in this section are used for exploring the virtual world, and interacting with the game administrators, using `news` and `@gripe`. These verbs are defined by a variety of different classes.

3.1 Movement

The descriptions of most rooms outline the directions in which exits exist. Typical directions include the eight compass points ('`north`', '`south`', '`east`', '`west`', '`northeast`', '`southeast`', '`northwest`', and '`southwest`'), '`up`', '`down`', and '`out`'.

To go in a particular direction, simply type the name of that direction (e.g, '`north`', '`up`'). The name of the direction can usually be abbreviated to one or two characters (e.g., '`n`', '`sw`'). You can also type '`go` *direction*' to move; this is particularly useful if you know you're going to type several movement commands in a row.

In addition to such vanilla movement, some areas may contain objects allowing teleportation and almost all areas permit the use of the '`home`' command to teleport you to your designated home.

A couple of other commands are available to allow movement from one place to another.

go *direction ...* Command

Invokes the named exits in the named order, moving through many rooms in a single command.

```
blip types:
> go n e e u e e s e
```

and moves quite rapidly north, east, east, up, east, east south and east, all in one command.

home Command

Instantly teleports you to your designated home room. Initially, this room is the `$player_start` room. You can change your designated home using the `@`*sethome* command.

3.2 Commands for Manipulating Objects

Objects usually have verbs defined on them that allow players to manipulate and use them in various ways. Standard ones are:

`get` - pick an object up and place it in your inventory

`drop` - remove an object from your inventory and place it in the room

`put` - take an object from your inventory and place it in a container

`give` - hand an object to some other player

`look` - see what an object looks like

You can see what objects you're carrying with the '`inventory`' command.

Some specialized objects will have other commands. The programmer of the object will usually provide some way for you to find out what the commands are. One way that works for most objects is the '`examine`' command.

inventory Command

Prints a list showing every object you're carrying.

take *object* Command
get *object* Command
take *object from container* Command
get *object from container* Command
remove *object from container* Command

The first two forms pick up the named object and place it in your inventory. Sometimes the owner of the object won't allow it to be picked up for some reason.

The remaining forms move the named object from inside the named container into your inventory. As before, sometimes the owner of an object will not allow you to do this.

drop *object* Command
throw *object* Command

Remove an object you are carrying from your inventory and put it in your current room. Occasionally you may find that the owner of the room won't allow you to do this.

put *object* **into** *container* Command
insert *object* **in** *container* Command

Moves the named object into the named container. Sometimes the owners of the object and/or the container will not allow you to do this.

give *object* **to** *player* Command
hand *object* **to** *player* Command

Move an object from your contents to that of another player. This doesn't change the ownership of the object. Some players may refuse to accept gifts and some objects may refuse to be given.

look Command
look *object* Command
look *object* **in** *container* Command

This command is used to show a description of something. The first form, with no arguments, shows you the name and description of the room you're in, along with a list of the other objects that are there.

The second form lets you look at a specific object. Most objects have descriptions that may be read this way. You can look at your own description using '`look me`'. You can set the description for an object or room, including yourself, with the '`describe`' command.

The third form shows you the description of an object that is inside some other object, including objects being carried by another player.

examine *object* Command
exam *object* Command

Prints several useful pieces of information about the named object, including the following:

- its full name, aliases, and object number
- its owner's name and object number
- its description
- its key expression (if it is locked and if you own it)
- its contents and their object numbers
- the *obvious* verbs defined on it

The *obvious* verbs are those that are readable and that can be invoked as commands. To keep a verb off this list, either make it unreadable using '`@chmod`'or, if it shouldn't be used as a command, give it `args` of '`this none this`'.

4 Interacting With Other Players

There are several commands available to allow you to communicate with your fellow players. Other commands are available to affect the way communication occurs. The following list shows the commands used for these functions:

`say` - talking to the other connected players in the room

`whisper` - talking privately to someone in the same room

`page` - yelling to someone anywhere in the MOO

`emote` - non-verbal communication with others in the same room

`@gag, @listgag, @ungag`
- screening out noise generated by certain other players

`news` - reading the wizards' most recent set of general announcements

`@gripe` - sending complaints to the wizards

`@typo @bug @idea @suggest`
- sending complaints/ideas to the owner of the current room

`whereis` - locating other players

`@who` - finding out who is currently logged in

`mail` - the MOO email system

`@paranoid, @check, @sweep`
- the facilities for detecting forged messages and eavesdropping.

4.1 Communicating With Other Players

Several commands are available for communicating with other players in the way you might do in real life.

say *anything ...* Command
"*anything ...* Command

Says *anything* out loud, so that everyone in the same room hears it. This is so commonly used that there's a special abbreviation for it: any command-line beginning with a double-quote (‘`"`’) is treated as a ‘`say`’ command.

For example, suppose that blip types the following command:

```
"This is a great MOO!
```

He would see this printed on his terminal screen:

```
You say, "This is a great MOO!"
```

Others in the same room see this:

```
blip says, "This is a great MOO!"
```

whisper "*text*" *to player* Command

This command sends the message "*yourname* whispers, "*text*" to you " to *player*, if they are in the room. This is used to send a private message to a another player in the room. The message is passed to *player* only. No-one else can hear or detect the message. For example, the command

```
whisper "Hello there" to blip
```

sends the following message to blip:

```
Ezeke whispers, "hello there" to blip.
```

page *player* [[*with*] *text*] Command

This verb is a player command used to send messages between players who are not physically located in the same room in the virtual world. You can imagine a *page* to be a worldwide form of shouting. Without an argument, a message like

```
You sense that blip is looking for you in The Venue Hallway.
```

is sent to the recipient of the page. If an argument is given, it is treated as a message to send to the other player. This results in the recipient getting a message like

```
You sense that blip is looking for you in The Hallway
He pages, "Hello - are you busy ?"
```

Paging is used primarily to attract the attention of a player, or to pass short messages between players in different locations. It is not intended to be used for conversation.

The following commands can be used to set messages referred to by '`@page`':

@page_origin *message* Command

The *page origin* message determines how the recipient is told of your location. The default value of this message is '`You sense that %n is looking for you in %l.`'

@page_echo *message* Command

The *page echo* message determines the response received by anyone who pages you. The default value of this message is '`Your message has been sent.`'

@page_absent *message* Command

This message determines the response received by anyone who tries to page you when you aren't connected. The default value of this message is '`%n is not currently logged in.`'

All of these undergo the usual pronoun substitutions except that in both cases the direct object '`%d`' refers to the recipent of the page and the indirect object '`%i`' refers to the sender.

emote *anything ...* Command
:*anything ...* Command

Announces *anything* to everyone in the same room, prepending your name. This is commonly used to express various non-verbal forms of communication. In fact, it is so commonly used that there's a special abbreviation for it: any command-line beginning with '`:`' is treated as an '`emote`' command.

For example, if blip types the following:

```
:wishes he were much taller...
```

Everyone in the same room would see the following message:

```
blip wishes he were much taller...
```

4.2 Gagging - How to Ignore Other Players and Objects

Occasionally, you may run into a situation in which you'd rather not hear from certain other players. It might be that they're being annoying, or just that whatever they're doing makes a lot of noise. Gagging a player will stop you from hearing the results of any task initiated by that player. You can also gag a specific object, if you want to hear what the owner of that object says, but not the output from their noisy robot. The commands to use gagging are described below:

@gag *player or object* [*player or object...*] Command

Add the given players to your *gag list*. You will no longer see any messages that result from actions initiated by these players. In particular, you will not hear them if they try to speak, emote, or whisper to you.

For example, if blip types in the following command:

```
@gag geezer
```

and no longer hears anything that geezer says.[1]

If you specify an object, then any text originating from that object will not be printed.

For example, suppose `Noisy Robot` prints '`Hi there`' every 15 seconds. In order to avoid seeing that, blip types the following command:

```
@gag Noisy
```

and no longer hears that robot! Note that blip must be in the same room as `Noisy Robot` for this to work, or know its object number.

@ungag *player or object* Command
@ungag everyone Command

Remove the given player or object (or, in the second form, **everyone**) from your '`gag list`'. You will once again see any messages that result from actions initiated by the ungagged player(s) or objects. In particular, you will once again be able to hear them if they speak, emote, or whisper to you.

For example, suppose that blip types the following:

[1] What a relief!

```
@ungag geezer
```

and is once again able to hear geezer's witty remarks.[2]

@listgag Command

Shows you a list of the players and objects currently on your *gag list*. You don't see any messages that result from actions initiated by the players or objects on this list. In particular, you will not hear them if they try to speak, emote, or whisper to you.

4.3 Communicating With The Game Administrators

Several commands are provided for communicating with the people that run the game. The `news` command is used by the wizards to let players know of anything that is globally interesting. Players can use '`@grip`' to complain to the wizards, and commands like '`@typo`' to report defects to builders and programmers.

The following section describes these commands in detail.

news Command

Read the latest edition of the LambdaMOO server news, which carries articles concerning recent changes to the MOO server or to the main public classes, or whatever else is important for players to know.

@gripe *anything . . .* Command

Puts you into the MOO mail system to register a complaint (or, conceivably, a compliment) with the wizards. The rest of the command line (the *anything . . .* part) is used as the subject line for the message. More information on using the MOO mail system is given once you're in it.

You may hear back from the wizards eventually. For example:

```
>@gripe The Fruitbat
>"How come I can't ever see the fruitbat in the Venue Clock?
>"        -- A frustrated player
```

sends it, and, somewhat later, the wizards reply with a note about being sure to look while the Clock is chiming.

@typo [*text*] Command
@suggest [*text*] Command
@bug [*text*] Command
@idea [*text*] Command

If *text* is given, a one-line message is sent to the owner of the room, presumably about something that you've noticed. If *text* is not given, we assume you have more to say than can fit comfortably on a single line; the usual mail editor is invoked. The convention is that @typo is for typographical errors on the room or objects found therein, @bug is for anomalous or nonintuitive behaviour of some sort, and @idea/@suggest for anything else.

[2] Ah well, it could be worse. . .

The usual mail editor is only invoked for this command when in rooms that allow free entry, i.e., rooms that are likely to allow you back after you are done editing your message. Otherwise these commands will require *text* and only let you do one-line messages. Most adventuring scenario rooms fall into this latter category.

4.4 Locating Other Players in the Virtual World

Two commands are available for finding out where other players are hiding in the virtual world, as follows:

whereis *player* [*player*...] Command
@whereis *player* [*player*...] Command

Returns the current location of each of the specified players. whereis refers to each player's '`@whereis_location`' message to determine what should be printed. This message defaults to

```
"%N (%#) is in %l (%[#l])."
```

and the usual pronoun substitutions are done.

For example the default message could expand to

```
"blip (#42) is in The Venue Manager's Office (#47)
```

@who Command
@who *player* [*player* ...] Command

The first form lists all of the currently-connected players, along with the amount of time they've been connected, the amount of time they've been idle, and their present location in the MOO.

The second form, in which a list of player names is given, shows information for just those players. For any listed players that are not connected, we show the last login time instead of the connect/idle times.

'`@who`' refers to the '`@who_location`' on each of the players to be listed in order to determine what should be printed in the location column. Pronoun substitutions are done on this string in the usual manner. The default value is "%l" (i.e., `player.location`).

4.5 Checking the Security of Your Communication

There are several commands available that allow you to check that your communications with other players are secure. The following commands are available:

@sweep Command

Used when you wish to have a private conversation, and are concerned someone may be listening in. `@sweep` tries to list the avenues by which information may be leaving the room. In a manner analogous to @check, it assumes that you don't want to hear about your own verbs, or those belonging to wizards, who presumably wouldn't stoop to bugging.

@paranoid Command
@paranoid off Command
@paranoid immediate Command
@paranoid *number* Command

In immediate mode, the monitor prepends everything you hear with the name of the character it considers responsible for the message. Otherwise, it keeps records of the last *number* (defaults to 20) lines you have heard. These records can be accessed by the @check command.

@check *options* Command

Used when you are suspicious about the origin of some of the messages your character has just heard. Various *options* can be specified:

- the number of lines to be displayed
- a player's name, someone to be "trusted" during the assignment of responsibility for the message.
- a player's named prefixed by !, someone not to be "trusted".

Output from @check is in columns that contain, in order, the monitor's best guess as to:

- what object the message came from,
- what verb on that object that was responsible,
- whose permissions that verb was running with, and the beginning of the actual message.

'@check' operates by examining the list of verbs that were involved in delivering the message, and assigning responsibility to the first owner it sees who is not *trusted*. By default, it trusts you and all the wizards. It uses the records maintained by '@paranoid', so you must have used that command before you received the message.

5 Using Pronoun Substitutions

Some kinds of messages are not printed directly to players; they are allowed to contain special characters marking places to include the appropriate pronoun for some player. For example, a builder might have a doorway that's very short, so that people have to crawl to get through it. When they do so, the builder wants a little message like this to be printed:

```
Zaphod crawls through the little doorway, bruising his knee.
```

The problem is the use of '`his`' in the message; what if the player in question is female? The correct setting of the '`oleave`' message on that doorway is as follows:

```
crawls through the little doorway, bruising %p knee.
```

The '`%p`' in the message will be replaced by either '`his`', '`her`', or '`its`', depending upon the gender of the player.

As it happens, you can also refer to elements of the command line (e.g., direct and indirect objects) the object issuing the message, and the location where this is all happening. In addition one can refer to arbitrary string properties on these objects, or get the object numbers themselves.

The complete set of substitutions is as follows:

`%%` => '%' (just in case you actually want to talk about percentages).

`%n` => the player

`%t` => this object (i.e., the object issuing the message,... usually)

`%d` => the direct object from the command line

`%i` => the indirect object from the command line

`%l` => the location of the player

`%s` => subject pronoun: either '`he`', '`she`', or '`it`'

`%o` => object pronoun: either '`him`', '`her`', or '`it`'

`%p` => posessive pronoun (adj): either '`his`', '`her`', or '`its`'

`%q` => posessive pronoun (noun): either '`his`', '`hers`', or '`its`'

`%r` => reflexive pronoun: either '`himself`', '`herself`', or '`itself`'

`%(foo)` => player.foo

`%[tfoo], %[dfoo], %[ifoo], %[lfoo]`
=> this.foo, dobj.foo, iobj.foo, and player.location.foo

`%#` => player's object number

`%[#t], %[#d], %[#i], %[#l]`
=> object numbers for this, direct obj, indirect obj, and location.

In addition there is a set of capitalized substitutions for use at the beginning of sentences. These are, respectively,

- %N, %T, %D, %I, %L for object names,
- %S, %O, %P, %Q, %R for pronouns, and
- %(Foo), %[dFoo] (== %[Dfoo] == %[DFoo]),. . . for general properties

Note that there is a special exception for player name's (the `.name` propertywhich are assumed to already be capitalized as desired.

There may be situations where the standard algorithm, i.e., upcasing the first letter, yields something incorrect, in which case a *capitalization* for a particular string property can be specified explicitly. If your object has a `.foo` property that is like this, you need merely add a `.fooc` (in general `.(`*propertyname*`+"c")`) specifying the correct capitalization. This will also work for player `.name`'s if you want to specify a capitalization that is different from your usual .name

For example, Phantom makes a hand-grenade with a customizable explode message. Suppose someone sets `grenade.explode_msg` to:

```
"%N(%#) drops %t on %p foot.  %T explodes.
%L is engulfed in flames."
```

If the current location happens to be #1234("blip's house"), the resulting substitution may produce, eg.,

```
Phantom(#42) drops grenade on his foot.  Grenade explodes.
Blip's house is engulfed in flames.
```

which contains an incorrect capitalization (The name '`blip`' cannot be capitalized. blip may remedy this by setting

```
#1234.namec="blip's house".
```

A special nore for programmers: in programs, use `$string_utils:pronoun_sub()`. Using the substitution `%n` actually calls `player:title()` while `%(name)` refers to `player.name` directly.

6 The MOO Mail System

The MOO email system allows you to send and receive messages to and from other players. Whilst not approaching the complexity of a fully fledged email system, it does allow players to send short messages between themselves, normally on game related topics. The `Mail Room` class defines a set of verbs that can be used by other MOO programs to send mail. This is used, for example, by the `@gripe` command, which uses the MOO Mail system to deliver gripes to the game administrators.

The following commands are used to activate portions of the MOO mail system:

`@mail` - seeing a table of contents for your collection of email messages

`@read` - reading individual messages in your collection

`@next` - reading the 'next' message in your collection

`@prev` - reading the 'previous' message in your collection

`@send` - composing and sending a message to other players

`@answer` - replying to one of the messages in your collection

`@rmmail` - discarding one or more messages from your collection

`@renumber`
- renumbering the messages in your collection

`@mail` Command
`@mail` *new* Command

Shows a table of contents for your MOO email message collection. You are notified when you connect to the MOO if there are any such messages. A little arrow indicates the mail system's notion of your *current message*. The first form lists all of your messages; the second form lists all messages after your *current message*.

If you have a large number of mail messages, you can give arguments so that `@mail` only lists the messages you're interested in. The general format is

`@mail` *message-sequence*

where *message-sequence* is some combination of the following

`cur` - the current message

`new` - all messages after the current message

`#` - (where # is a number) the message numbered # if there is one.

`#-#` - all messages in the given range, if any.

`last:#` - the last # messages

`-#` - the last # messages

You may use as many of these at once as sanity permits, e.g.,

```
@mail 1 4 7 last:10 2-3 15 cur
```

@read *message-number message-number . . .* Command
@read Command

Prints the contents of the indicated messages from your MOO email collection. You get the message numbers for use here by typing the '`@mail`' command, which prints a table of contents for your entire MOO email collection. If no arguments are given to '`@read`', then the *current message* in your collection is printed. In any case, the *current message* after '`@read`' finishes is the last one printed.

@next Command

Print the '`next`' message in your MOO email collection. The mail system's notion of your *current message* is incremented. Thus, one can read all of one's new messages one-by-one simply by typing '`@next`' repeatedly.

@prev Command

Print the '`previous`' message in your MOO email collection. The mail system's notion of your *current message* is decremented. Thus, one can review all of one's previous messages one-by-one simply by typing '`@prev`' repeatedly.

@send *recipient* [*recipient . . .*] Command

Prepares for you to compose a MOO email message to the recipients named on the command line. A recipient can be specified by giving a player name or object-id, or a '`*`' followed by the name or object-id of some non-player mail recipient (e.g., a mailing list or a mail folder). A list of such non-player recipients is available from within the mailroom with the '`showlists`' command.

When the '`@send`' command is typed, the usual line editor is invoked. The '`subject`' command is used to set a `Subject:` line. Use '`say`' ('`"`') to insert lines in the body of your message.

Giving this command without arguments resumes editing the previous unsent draft message if one exists.

@reply [*message-number*] [*sender*] [*all*] [*incl*] [*noincl*] Command
@answer [*message-number*] [*sender*] [*all*] [*incl*] [*noincl*] Command

Prepares for you to compose a MOO email message to the players who either received or composed the indicated message from your collection. The usual editor is invoked.

The subject line for the new message will be initialized from that of the indicated message. If you leave off the message number, the reply will be to your current message, if that exists.

If there is a `Reply-to:` field in the message you are answering, its contents will be used to initialize the `To:` line of your reply. Otherwise, a `To:` line is determined depending on whether you specified '`sender`' or '`all`' in the command line (or your `.mail_options`).

'`incl`' includes the text of the original message in your reply, '`noincl`' does not.

Defaults are '`sender`' and '`noincl`', but you can change this by setting your `.mail-options` property.

@rmmail *message-number message-number . . .* Command

Deletes the indicated messages from your MOO email collection. There is no confirmation for this action, so be careful; deleted messages are really gone, irrecoverable. You get the message numbers for use here by typing the '`@mail`' command, which prints a table of contents for your entire MOO email collection.

You may specify **cur** in place of a number to specify your current message. '`@rmm`' with no arguments deletes your current message.

@renumber Command

Renumbers the messages in your collection to go from 1 to however many you have at the moment.

6.1 Mail Options

There are 3 personal properties that you can use to customize how your mail is composed and forwarded

6.1.1 .mail_forward

This property is a list of people (object reference numbers) who will receive any mail that gets sent to you. This list may include non-person recipients (i.e.,descendants of `$mail_recipient`). If this list is nonempty, you will not receive any mail yourself unless you are on it. E.g., if blip is #42 and ur-blip is #43

```
#43.mail_forward={}         -- usual case; ur-blip gets his own mail.
#43.mail_forward={#42}      -- blip gets ur-blip's mail instead.
#43.mail_forward={#43,#42}  -- ur-blip gets mail and blip gets a copy.
#43.mail_forward={#-1}      -- ur-blip's mail disappears without a trace.
```

6.1.2 .mail_notify

This property is a list of people (object reference numbers) to be notified whenever mail is sent to you. This list may include anything that has a `:tell()` verb. Notification will take place regardless of whether or how your mail is forwarded. Thus, in the previous example

```
#42.mail_notify={#43}
```

means that ur-blip will be told whenever blip is sent new mail.

6.1.3 .mail_options

This property is a list of options that is consulted by `@`send and `@`answer to determine how messages should initially be composed

The following options are available for `@`answer:

`sender` - replies go only to the message's author (`From:` line).

`all` - replies go to everyone who got the original (`From:` + `To:` lines)

`incl` - include original message in the body of your reply.

`noincl` - don't include original message in the body of your reply.

The following option affects the behaviour of `@`send and `@`answer:

`replyto` ="*list of people*". A `Reply-to:` field will be added containing these people.

So, for example, if ur-blip usually wants his replies to go to everyone and always start out with the text of the original included, he should do

```
;#43.mail_options = {"all", "incl"};
```

7 Building and Creating Objects

There are a number of commands available to players for building new parts of the MOO. The primary means for players to extend the MOO is for them to create new objects with interesting behavior. There are convenient commands for creating and recycling objects and for keeping track of the objects you've created.

The following commands are used in the creation of objects:

`@dig` - conveniently building new rooms and exits

`@create` - making other kinds of objects

`@recycle` - destroying objects you no longer want

`@quota` - determining how many more objects you can build

`@count` - determining how many objects you already own

`@audit` - listing all of your objects

`@classes` - listing all of the public classes available for your use

`@move` - moving your objects from place to place

They are described in detail in the following paragraphs.

`@create` *class-name named* "*names*" Command
`@create` *parent-object named* "*names*" Command

The main command for creating objects other than rooms and exits, for which '`@dig`' is more convenient.

The first argument specifies the *parent* of the new object: loosely speaking, the *kind* of object you're creating. *class-name* is one of the four standard classes of objects:

- $note
- $letter
- $thing
- $container

As time goes on, more *standard classes* may be added. If the parent you have in mind for your new object isn't one of these, you may use the parent's name (if it's in the same room as you) or else its object number (e.g., #1234).

The *names* are given in the same format as in the '`@rename`' command, as follows:

name:*alias*,...,*alias*

or alternatively

name-and-alias,*alias*,...,*alias*

@describe *object* **as** *description* Command

Sets the description string of *object* to *description*. This is the string that is printed out whenever someone uses the '`look`' command on *object*. To describe yourself, use '`me`' as the *object*.

For example, if blip types the following:

```
@describe me as "A very fine fellow, if a bit on the short side."
```

People who type '`look blip`' now see this:

```
A very fine fellow, if a bit on the short side.
```

The description of an object is kept in its `.description` property. For multi-line descriptions, `.description` can be a list of strings.

@rename *object* **to** *name:alias,. . .,alias* Command
@rename *object* **to** *name-and-alias,alias,. . .,alias* Command
@rename *object:verb* **to** *new-verb-name* Command

The first two forms are used to change the name and aliases of an object. The name is what will be used in most printed descriptions of the object. The aliases are the names by which players can refer to the object in commands. NOTE that the name of a player may not include spaces and that no two players may have the same name at the same time.

For example, if blip names his dog using the following command:

```
@rename #4237 to "Rover the Wonder Dog":Rover,dog
```

Now we'll see '`Rover the Wonder Dog`' if we're in the same room as him and we can refer to him as either '`Rover`' or just '`dog`' in our commands, like '`pet dog`'.

The third form of the `@`rename command is for use by programmers, to change the name of a verb they own. If the *new-verb-name* contains spaces, the verb will have multiple names, one for each space-separated word.

@recycle *object-name-or-number* Command

Destroys the indicated object utterly and irretrievably. Naturally, you may only do this to objects that you own.

@quota Command

Each player has a limit as to how many objects that player may create, called their *quota*. Every object they create lowers the quota by one and every object they recycle increases it by one. If the quota goes to zero, then that player may not create any more objects (unless, of course, they recycle some first).

The '`@quota`' command prints out your current quota.

To get a larger quota, talk to a wizard. They will take a look at what you've done with the objects you've built so far and make a determination about whether or not it would be a net gain for the MOO community if you were to build some more things. If so, they will increase your quota; if not, they will try to explain some ways in which you could build things that were more useful, entertaining, or otherwise interesting to other players.

The quota mechanism is intended to solve a long-standing problem in many MUDs: database bloat. The problem is that a large number of people build a large number of dull objects and areas that are subsequently never used or visited. The database becomes quite large and difficult to manage without getting substantially more interesting. With the quota system, we can make it possible for players to experiment and learn while simultaneously keeping random building to acceptable levels.

It is expected that some will find the quota system distasteful or otherwise controversial. It was invented by Haakon and he is always interested in hearing your constructive comments, suggestions and protests.

@count Command

Prints out the number of objects you own. Do not be surprised if this is one larger than you think it should be: remember that your player object is owned by you as well, even though you didn't create it in the usual way.

@audit Command
@audit *player* Command
@audit *player from number* Command

The first form prints out a list of every object you own, giving each one's name and object number.

The second form does the same for the named player.

The third form does the same for the named player, but begins searching the database with the numbered object. *player* may be replaced by "me" to restrict the audit of yourself. This can be useful if you know the player does not own any objects below a certain number (typically the player's number itself).

Interestingly, due to a quirk of the code, "@audit me from me" will show you objects owned by you starting with your object number, an unexpected shorthand.

@classes Command
@classes *class-name . . .* Command

The wizards have identified several useful classes of objects in the database. The '@classes' command is used to see which classes exist and what their member objects are.

The first form simply lists all of the defined classes along with short descriptions of the membership of each.

The second form prints an indented listing of that subset of the object parent/child hierarchy containing the objects in the class(es) you specify.

@move *thing to place* Command

Move the specified object to the specified location. This is not guaranteed to work; in particular, the object must agree to be moved and the destination must agree to allow the object in. This is usually the case, however. The special case where *thing* is 'me' is useful for teleporting yourself around.

If @move doesn't work and you own the room where the object is located, try using '@eject' instead.

@eject *object* Command
@eject *object from place* Command

This command is used to remove unwanted objects from places you own. Players thus removed are unceremoniously dumped in the default player starting place. Other kinds of objects get thrown into `#-1` or `$nothing`. Unlike '`@move`', '`@eject`' does *not* check to see if the object wants to be moved, and with the destination being what it is, there is no question of the destination refusing the move, either. Generally, you should only resort to '`@eject`' if '`@move`' doesn't work.

The first form of the command removes the object from the current room. The second form removes the object from the specified place (which, in most cases, you'll have to specify as an object number). In either case, this command only works if you own the room/entity from which the object is being ejected.

The form of the command

```
@eject ... from me
```

suffices to get rid of some unwanted object in your inventory.

On any given room, one may user the following commands to set the messages used for ejection:

@ejection *message* Command

This message is Printed to player issuing the `@`eject command. The default *message* is '`You expel %d from %i.`'

@oejection *message* Command

This message is Printed to others in the room from which the ejection occurs. The default *message* is '`%D is unceremoniously expelled from %i.`'

@victim_ejection *message* Command

This message is Printed to the victim being ejected. The default *message* is '`You have been expelled from %i.`'

7.1 Rooms and Exits

Rooms and exits are the stuff from which the landscape of the virtual world is created. A *room* is generally an instance of the generic room class, also referred to as `$room`. An exit is an instance of the generic exit class, `$exit`. An exit can be thought of as a one way tunnel leading from one room to another. If you wish to have a two way exit, you have to use two exits: one going from the *source* to the `destination` and one going from the `destination` to the `source`.

The following commands are used for creating and managing rooms and exits:

@dig "*new-room-name*" Command
@dig *exit-spec* **to** "*new-room-name*" Command
@dig *exit-spec* **to** *old-room-object-number* Command

This is the basic building tool. The first form of the command creates a new room with the given name. The new room is not connected to anywhere else; it is floating

in limbo. The '`@dig`' command tells you its object number, though, so you can use the '`@move`' command to get there easily.

The second form of the command not only creates the room, but one or two exits linking your current location to (and possibly from) the new room. An *exit-spec* has one of the following two forms:

names
names`|`*names*

where the first form is used when you only want to create one exit, from your current room to the new room, and the second form when you also want an exit back, from the new room to your current room. In any case, the *names* piece is just a list of names for the exit, separated by commas; these are the names of the commands players can type to use the exit. It is usually a good idea to include explicitly the standard abbreviations for direction names (e.g., '`n`' for '`north`', '`se`' for '`southeast`', etc.). DO NOT put spaces in the names of exits; they are useless in MOO.

The third form of the command is just like the second form except that no new room is created; you instead specify by object number the other room to/from which the new exits will connect.

NOTE: You must own the room at one end or the other of the exits you create. If you own both, everything is hunky-dorey. If you own only one end, then after creating the exits you should write down their object numbers. You must then get the owner of the other room to use '`@add-exit`' and '`@add-entrance`' to link your new exits to their room.

For example,

```
@dig "The Conservatory"
```

creates a new room named "The Conservatory" and prints out its object number.

```
@dig north,n to "The North Pole"
```

creates a new room and also an exit linking the player's current location to the new room; players would say either '`north`' or '`n`' to get from here to the new room. No way to get back from that room is created.

```
@dig west,w|east,e,out to "The Department of Auto-Musicology"
```

creates a new room and two exits, one taking players from here to the new room (via the commands '`west`' or '`w`') and one taking them from the new room to here (via '`east`', '`e`', or '`out`').

```
@dig up,u to #7164
```

creates an exit leading from the player's current room to #7164, which must be an existing room.

`@add-exit` *exit-object-number* Command

Add the exit with the given object number as a conventional exit from the current room (that is, an exit that can be invoked simply by uttering its name, like '`east`'). Usually, '`@dig`' does this for you, but it doesn't if you don't own the room in question. Instead, it tells you the object number of the new exit and you have to find the owner of the room and get them to use the '`@add-exit`' command to link it up.

@add-entrance *exit-object-number* Command

Add the exit with the given object number as a recognized entrance to the current room (that is, one whose use is not considered teleportation). Usually, '`@dig`' does this for you, but it doesn't if you don't own the room in question. Instead, it tells you the object number of the new exit and you have to find the owner of the room and get them to use the @add-entrance command to link it up.

@exits Command

Prints a list of all conventional exits from the current room (but only if you own the room). A conventional exit is one that can be used simply by uttering its name, like '`east`'.

@entrances Command

Prints a list of all recognized entrances to the current room (but only if you own the room). A recognized entrance is one whose use is not considered to be teleportation.

8 Notes and Letters

Notes and letters are objects that can have text written on them to be read later. They are useful for leaving messages to people, or for documenting your creations.

Note that, like most objects, only the owner of a note can recycle it. If you'd like to make it possible for a reader of your note to destroy it (this is a common desire for notes to other individual players), then you might want to look at using a `$letter` instead.

8.1 Using Notes

You can make a note by creating a child of the standard note, `$note`. The following commands are available for interacting with notes:

read *note* Command
: Prints the text written on the named object, usually a note or letter. Some notes are encrypted so that only certain players may read them.

write "*any text*" **on** *note* Command
: Adds a line of text to the named note or letter. Only the owner of a note may do this.

erase *note* Command
: Deletes all of the text written on a note or letter. Only the owner of a note may do this.

delete *line-number from note* Command
: Removes a single line of text from a note. The first line of text is numbered 1, the second is 2, and so on. Only the owner of a note may do this.

@notedit *note-object* Command
@notedit *object.property* Command
: Enters the MOO Note Editor to edit the text on the named object For the first form, *note-object* must be a descendant of $note. For the second form, *object.property* can be any text-valued (i.e., list of strings) property on any object.

  The standard MOO editor is used to perform editing operations.

encrypt *note with key-expression* Command
: Restricts the set of players who can read the named note or letter to those for whom the given key expression is true. Only the owner of a note may do this.

decrypt *note* Command
: Removes any restriction on who may read the named note or letter. Only the owner of a note may do this.

8.2 Using Letters

A letter is a special kind of note with the added feature that it can be recycled by anyone who can read it. This is often useful for notes from one player to another. You create the letter as a child of the generic letter, `$letter`, encrypt it so that only you and the other player can read it and then either give it to the player in question or leave it where they will find it. Once they've read it, they can use the '`burn`' command to recycle the letter.

The following command is available for letters, in addition to those used for notes.

burn *letter* Command

Destroy the named letter irretrievably. Only players who can read the letter can do this.

9 Using Containers

Containers are objects that allow you to store other objects inside them. Containers may be open or closed, using the verbs '`open`' and '`close`' on the container. Containers have a separate lock to determine if a player may open them. You can make a container by creating a child of the standard container, `$container`.

Containers have a large number of messages which get printed when players act upon them.

Containers have opacity. This is manipulated using the following command :

`@opacity` *container is integer* Command

The opacity can take on one of three values:

`0` - The container is transparent and you can always see into it.

`1` - The container is opaque, and you cannot see into it when closed

`2` - The container is a black hole, and you can never see into it whether closed or open.

The default opacity is '`1`' - the container is opaque.

10 Messages on Objects

Most objects have messages that are printed when a player succeeds or fails in manipulating the object in some way. Of course, the kinds of messages printed are specific to the kinds of manipulations and those, in turn, are specific to the kind of object. Regardless of the kind of object, though, there is a uniform means for listing the kinds of messages that can be set and then for setting them.

The '`@messages`' command prints out all of the messages you can set on any object you own.

To set a particular message on one of your objects use a command with this form:

`@`*message-name object* `is "`*message*`"`

where '*message-name*' is the name of the message being set, *object* is the name or number of the object on which you want to set that message, and *message* is the actual text.

For example, consider the '`leave`' message on an exit; it is printed to a player when they successfully use the exit to leave a room. To set the '`leave`' message on the exit '`north`' from the current room, use the command

```
@leave north is "You wander in a northerly way out of the room."
```

This class of commands automatically applies to any property whose name ends in '`_msg`'. Thus, in the example above, the command is setting the '`leave_msg`' property of the named exit. You can get such a command to work on new kinds of objects simply by giving the appropriate properties names that end in '`_msg`'.

Messages of this type are used on the following objects:

- containers
- exits
- things

`@messages` *object* Command

List all of the messages that can be set on the named object and their current values.

10.1 Setting Messages for Exits

Several kinds of messages can be set on an exit object ; they are printed to various audiences at certain times whenever an attempt is made to go through the exit. The ones whose names begin with '`o`' are always shown prefixed with the name of the player making the attempt and a single space character. The standard pronoun substitutions (with respect to the player) are made on each message before it is printed.

The following commands can be used to set the corresponding messages on an exit:

`@leave` *message* Command

This command sets the message printed to the player just before they successfully use the exit. The default message is ''.

`@oleave` *message* Command

This command sets the message printed to others in the source room when a player successfully uses the exit. The default message is ' `has left.`'.

@arrive *message* Command
This command sets the message printed to the player just after they successfully use the exit. The default message is ‘’.

@oarrive *message* Command
This command sets the message printed to others in the destination room when a player successfully uses the exit. The default message is ‘`has arrived.`’.

@nogo *message* Command
This command sets the message printed to the player when they fail in using the exit. The default message is ‘ `You can't go that way.`’.

@onogo *message* Command
This command sets the message printed to others when a player fails in using the exit. The default message is ‘’.

10.2 Setting Messages for Things

Several kinds of messages can be set on *things*, that is, objects that have `$thing` as an ancestor. They are printed to various audiences under various circumstances when an attempt is made to ‘`take`’ or ‘`drop`’ a thing. The ones whose names begin with ‘`o`’ are always shown prefixed with the name of the player making the attempt and a single space character. The standard pronoun substitutions (with respect to the player) are made on each message before it is printed.

The following commands can be used to set the corresponding messages on things:

@take_failed Command
This command is used to set the message printed to a player who fails to take the object. The default message is ‘`You can't pick that up.`’.

@otake_failed Command
This command is used to set the message printed to others in the same room if a player fails to take the object. The default message is ‘’.

@take_succeeded Command
This command is used to set the message printed to a player who succeeds in taking the object. The default message is ‘ `You take %t.`’.

@otake_succeeded Command
This command is used to set the message printed to others in the same room if a player succeeds in taking the object. The default message is ‘ `picks up %t.`’.

@drop_failed Command
This command is used to set the message printed to a player who fails to drop the object. The default message is ‘ `You can't seem to drop %t here.`’.

@odrop_failed Command

This command is used to set the message printed to others in the same room if a player fails to drop the object. The default message is '`tries to drop %t but fails!`'.

@drop_succeeded Command

This command is used to set the message printed to a player who succeeds in dropping the object. The default message is ' `You drop %t.`'.

@odrop_succeeded Command

This command is used to set the message printed to others in the room if a player succeeds in dropping the object. The default message is ' `drops %t.`'.

10.3 Setting Messages for Containers

Several kinds of messages can be set on a container object; they are printed to various audiences at certain times whenever an attempt is made to use the container. The ones whose names begin with 'o' are always shown prefixed with the name of the player making the attempt and a single space character. The standard pronoun substitutions (with respect to the player) are made on each message before it is printed.

The following commands can be used to set the corresponding messages used with containers.

@empty Command

This command is used to set the message printed in place of the contents list when the container is empty. The default message is '`It is empty.`'

@open Command

This command is used to set the message printed to the player who successfully opens the container. The default message is '`You open %d.`'

@oopen Command

This command is used to set the message printed to others in the same room if the player successfully opens the container. The default message is '`opens %d.`'

@open_fail Command

This command is used to set the message printed to the player who cannot open the container. The default message is '`You can't open that.`'

@oopen_fail Command

This command is used to set the message printed to others in the room when a player fails to open a container. The default message is ''

@close Command

This command is used to set the message printed to the player who closes a container. The default message is '`You close %d`'

@oclose Command

This command is used to set the message printed to others in the room when a player closes a container. The default message is '`closes %d`'

@put Command

This command is used to set the message printed to a player when an object is successfully placed in a container. The default message is '`You put %d in %i`'

@oput Command

This command is used to set the message printed to others in the room when a player successfully places an object in a container. The default message is '`puts %d in %i.`'

@put_fail Command

This command is used to set the message printed when a player fails to put an object in a container. The default message is '`You can't put %d in that.`'

@oput_fail Command

This command is used to set the message printed to others in the room when a player fails to place an object in a container. The default message is ''

@remove Command

This command is used to set the message printed when a player succeeds in removing an object from a container. The default message is '`You remove %d from %i`'

@oremove Command

This command is used to set the message printed to others in the room when a player succeeds in removing an object from a container. The default message is '`removes %d from %i.`'

@remove_fail Command

This command is used to set the message printed when a player fails to remove an object from a container. The default message is '`You can't remove that.`'

@oremove_fail Command

This command is used to set the message printed to others in the room when a player fails to remove an object from a container. The default message is ''

11 Using Locks With Objects

It is frequently useful to restrict the use of some object. For example, one might want to keep people from using a particular exit unless they're carrying a bell, a book, and a candle. Alternatively, one might allow anyone to use the exit unless they're carrying that huge golden coffin in the corner. LambdaMOO supports a general locking mechanism designed to make such restrictions easy to implement, usually without any programming.

Every object supports a notion of being *locked* with respect to certain other objects. For example, the exit above might be locked for any object that was carrying the coffin object but unlocked for all other objects. In general, if some object '`A`' is locked for another object, '`B`', then '`B`' is usually prevented from using '`A`'. Of course, the meaning of *use* in this context depends upon the kind of object.

The various standard classes of objects use locking as follows:

- Rooms and containers refuse to allow any object inside them if they're locked for it.
- Exits refuse to transport any object that they're locked for.
- Things (including notes and letters) cannot be moved to locations that they're locked for.

There are two sides to locking:

- How is it specified whether one object is locked for another one?
- What is the effect of an object being locked?

Note that these two questions are entirely independent: one could invent a brand-new way to specify locking, but the effect of an exit being locked would be unchanged.

Programmers should note that the interface between these two sides is the verb

x`:is_unlocked_for(`*y*`)`

which is called by *x* to determine if it is locked for the object *y*. The way in which `:is_unlocked_for` is implemented is entirely independent of the ways in which *x* uses its results. Note that you can play on either side of this interface with your own objects, either defining new implementations of `:is_unlocked_for` that match your particular circumstances or having your objects interpret their being locked in new ways.

The following commands are used to specify locks on objects.

@lock *object* **with** *key expression* Command
Set a lock on *object* to restrict its use.

@unlock *object* Command
Clear any lock that might exist on the given object.

@lock_for_open *container* **with** *key expression* Command
Set the lock on *container* which restricts who can open it.

@unlock_for_open *container* Command
Clears the lock which restricts who may open *container*.

11.1 Keys

LambdaMOO supports a simple but powerful notation for specifying locks on objects, encryption on notes, and other applications. The idea is to describe a constraint that must be satisfied concerning what some object must be or contain in order to use some other object.

The constraint is given in the form of a logical expression, made up of object numbers connected with the operators '`and`', '`or`', and '`not`' (written '`&&`', '`||`', and '`!`', for compatibility with the MOO programming language). When writing such expressions, though, one usually does not use object numbers directly, but rather gives their names, as with most MOO commands.

These logical expressions (called *key expressions*) are always evaluated in the context of some particular *candidate* object, to see if that object meets the constraint. To do so, we consider the candidate object, along with every object it contains (and the ones those objects contain, and so on), to be '`true`' and all other objects to be '`false`'.

As an example, suppose the player blip wanted to lock the exit leading to his home so that only he and the holder of his magic wand could use it. Further, suppose that blip was object #999 and the wand was #1001. blip would use the '`@`lock' command to lock the exit with the following key expression:

```
me || magic wand
```

and the system would understand this to mean

```
#999 || #1001
```

That is, players could only use the exit if they were (or were carrying) either #999 or #1001.

To encrypt a note so that it could only be read by blip or someone carrying his book, his bell, and his candle, blip would use the '`encrypt`' command with the key expression

```
me || (bell && book && candle)
```

Finally, to keep players from taking a large gold coffin through a particularly narrow exit, blip would use this key expression:

```
! coffin
```

That is, the expression would be false for any object that was or was carrying the coffin.

There is one other kind of clause that can appear in a key expression:

`?` *object*

This is evaluated by testing whether the given object is unlocked for the candidate object; if so, this clause is true, and otherwise, it is false. This allows you to have several locks all sharing some single other one; when the other one is changed, all of the locks change their behavior simultaneously.

The internal representation of key expressions, as stored in .key on every object, for example, is very simple and easy to construct on the fly.

11.1.1 Key Representation

The representation of key expressions is very simple and makes it easy to construct new keys on the fly.

Objects are represented by their object numbers and all other kinds of key expressions are represented by lists. These lists have as their first element a string drawn from the following set:

```
"&&"     "||"     "!"     "?"
```

For the first two of these, the list should be three elements long; the second and third elements are the representations of the key expressions on the left- and right-hand sides of the appropriate operator. In the third case, '`!`', the list should be two elements long; the second element is again a representation of the operand. Finally, in the '`?`' case, the list is also two elements long but the second element must be an object number.

As an example, the key expression

```
#45  &&  ?#46  &&  (#47  ||  !#48)
```

would be represented as follows:

```
{"&&", {"&&", #45, {"?", #46}}, {"||", #47, {"!", #48}}}
```

12 The MOO Editor

One can always enter an editor by teleporting to it, or you can use one of the commands provided

`@edit` *object*`:`*verb*
: invokes the Verb Editor (edits verb code)

`@notedit` *note_object*
: invokes the Note Editor (edits note text)

`@notedit` *object*`.`*prop*
: invokes the Note Editor (edits text property)

`@send` *list of recipients*
: invokes the Mailer (edits a mail message)

`@answer` [*msg_number*] [*flags*`...`]
: invokes the Mailer (edits a reply)

This will transport you to one of several special rooms that have editing commands available. These editors are admittedly not as good as EMACS, but for those with no other editing capability on their host systems, it is better than nothing.

In addition to the commands provided by the generic editor, individual editors provide their own additional commands for loading text from places, saving text to places, and various specialized functions.

Note that a given editor only allows you one session at a time (ie. one verb, one note, or one mail message). If you leave an editor without either aborting or compiling/saving/sending the item you're working on, that editor remembers what you are doing next time you enter it, whether you enter it by teleporting or by using the appropriate command. Note that editors are periodically flushed so if you leave stuff there for sufficiently long, it will go away.

A player may have his own `.edit_options` property which is a list containing one or more (string) flags from the following list

`quiet_insert`
: suppresses those annoying '`Line n added.`' or '`Appended...`' messages that one gets in response to '`say`' or '`emote`'. This is useful if you're entering a long list of lines, perhaps via some macro on your client, and you don't want to see an equally long list of '`Line n added...`' messages. What you do want, however is some indication that this all got through, which is why the '`.`' command is an abbreviation for insert.

There will be more options, some day.

12.1 Editor Ranges

Most editor commands act upon a particular range of lines. Essentially, one needs to specify a first line and a last line. Line numbers may be given in any of the following forms

`n` (i.e., the nth line of text)

`n^` n-th line after/below the current insertion point

`n_` n-th line before/above the current insertion point

`n$` n-th line before the end.

In the latter three, n defaults to 1, so that '`^`' by itself refers to the line below the current (i.e., the line that gets '`^`' printed before it), and likewise for '`_`' while '`$`' refers to the last line. Note that the usage depends on whether you are specifying a line or an insertion point (space between lines). '`^5`' is the space above/before line 5, while '`5^`' is the fifth line after/below the current insertion point.

Ranges of lines may be specified in any of the following ways:

line just that line

`from` *line* `to` *line*
what it says; the following two forms are equivalent:
line–*line*
line *line*

With the '`from l to l`' form, either the **from** or the **to** can be left off and it will default to whatever is usual for that command (usually a line above or below the insertion point).

12.2 Editor Commands

The following commands are provided by the editor classes. Those that are not generic are specifically marked so.

say *text* Command
"*text* Command

Adds *text* to whatever you are editing. The second form is equivalent to the first except in that it doesn't strip leading blanks off of *text* (just as with the normal '`say`' and '`"`' commands).

The added text appears as a new line at the insertion point. The insertion point, in turn, gets moved so as to be after the added text. For example:

```
>"first line
 ⊣Line 1 added.
```

```
>"  second line"
 ⊣Line 2 added.

>list
 ⊣1: first line
 ⊣__2_   second line"
 ⊣^^^^
```

emote *text* Command
:*text* Command

Appends *text* to the end of the line before the insertion point. The second form is equivalent to the first except that it doesn't strip leading blanks off of *text* (just as with the normal '`emote`' and '`:`' commands). The insertion point is left unmoved.

```
>list .
 ⊣_37_ Hello there
 ⊣^38^ Oh, I'm fine.

>:, how are you
 ⊣Appended to line 37.

>:?
 ⊣Appended to line 37.

>list .
 ⊣_37_ Hello there, how are you?
 ⊣^38^ Oh, I'm fine.
```

lis*t [*range*] Command

Prints some subset of the current verb text. The default range is some reasonable collection of lines around the current insertion point: currently this is '`8_-8^`', ie., 8 lines above the insertion point to 8 lines below it unless this runs up against the beginning or end of file, in which case we just take the first or last 16 lines, or just '`1-$`' if there aren't that many.

ins*ert [*ins*] ["*text*] Command
. Command

Many editor commands refer to an "insertion point" which is (usually) the place right below where the most recent line was inserted. The insertion point should really be thought of as sitting *between* lines. In listings, the line above the insertion point is marked with '`_`' while the one below is marked with '`^`'.

The '`insert`' command, when given an argument, sets the insertion point. If *text* is provided, a new line will be created and inserted as with '`say`'. *ins*, both here and in other commands that require specifying an insertion point (e.g., copy/move), can be one of

`^n`	above line n

`n`	above line n
`_n`	below line n
`$`	at the end
`^$`	before the last line
`n^$`	n lines before the end
`.`	the current insertion point (i.e., '`insert .`' is a no-op)
`+n`	n lines below the current insertion point.
`-n`	n lines above the current insertion point.

For the truly perverse, there are other combinations that also work due to artifacts of the parsing process, but these might go away. . .

A single dot '`.`' is the same as '`insert`' without any arguments ie, start insertion at the current insertion point.

n*ext [*n*] ["*text*] Command

Moves the insertion point down *n* lines. If *text* is provided, inserts a new line there just like '`say`'. Equivalent to '`insert +`*n*'. As one might expect, *n* defaults to 1.

p*rev [*n*] ["*text*] Command

Moves the insertion point up *n* lines. If *text* is provided, a new line is inserted as with '`say`'. Equivalent to '`insert -`*n*'. As one might expect, *n* defaults to 1.

del*ete [*range*] Command

Deletes the specified range of lines. Note that *range* defaults to the line *before* the current insertion point.

f*ind */ str*[*/*[*c*][*ins*]] Command

/ str[*/*[*c*][*ins*]] Command

Searches for the first line after *ins* containing *str*. *ins* defaults to the current insertion point . With the first form, any character (not just '`/`') may be used as a delimiter. For the second form, you must use '`/`'. The '`c`' flag, if given, indicates that case is to be ignored while searching

s*ubst */ str1 / str2*[*/*[*g*][*c*][*range*]] Command

Substitutes *str2* for *str1*, in all of the lines of *range*. Any character (not just '`/`') may be used to delimit the strings. If *str1* is blank, *str2* is inserted at the beginning of the line. (For inserting a string at the end of a line use '`emote`' or '`:`').

Normally, only one substitution is done per line in the specified range, but if the '`g`' flag is given, *all* instances of *str1* are replaced. The '`c`' flag indicates that case is not significant when searching for substitution instances. *range* defaults to the line *before* the insertion point.

You do *not* need a space between the verb and the delimiter before *str1*. [Bug: If you omit the space and the first whitespace in *str1* is a run of more than one space, those spaces get treated as one.]

m*ove [*range*] *to ins* Command

5Moves the range of lines to place specified by *ins*. If *ins* happens to be the current insertion point, the insertion point is moved to the end of the freshly moved lines. If the range of lines contains the insertion point, the insertion point is carried over to the range's new location.

c*opy [*range*] *to ins* Command

Copies the specified range of lines to place given by *ins*. If *ins* happens to be the current insertion point, the insertion point moves to the end of the inserted lines.

join [*range*] Command
joinliteral [*range*] Command

This command combines the lines in the specified range. Normally, spaces are inserted and double space appears after periods and colons, but '`joinliteral`' (abbreviates to '`joinl`') supresses this and joins the lines as is. *range* defaults to the two lines surrounding the insertion point.

fill [*range*] [`@` *c*] Command

This command combines the specified lines as in join and then splits them so that no line is more than *c* characters (except in cases of pathological lines with very long words). *c* defaults to 70. *range* defaults to the single line preceding the insertion point.

w*hat Command

Prints information about the editing session.

abort Command

Abandons this editing session and any changes.

q*uit Command
done Command
pause Command

Leaves the editor. If you have unsaved text it will be there when you return (and in fact you will not be able to do anything else with this editor until you '`abort`' or save the text).

reply-to [*recipients*] Command

This is a mail room command. It reports the current contents of the `Reply-to:` field of your message. With arguments, adds (or changes) the `Reply-to:` field.

When someone '`@answers`' a message, the `Reply-to:` field is checked first when determining to whom the reply should be sent.

To clear the `Reply-to:` field, enter the command

```
reply-to ""
```

edit *object:verb* Command

This is a verb editor command. It changes what verb you are editing and loads the code for that verb into the editor. Equivalent to

`@edit` *object*`:`*verb*`.`

edit *note-object* Command
edit *object.property* Command

This command is used for both note and verb editors. It changes to a different note or a different object text property and loads its text into the editor. These are equivalent to

`@notedit` *note*

or

`@notedit` *object*`.`*property*

respectively.

For both the verb-editor and note-editor commands, *object* will match on the room you came from, though if the room you came from was another editor, then all bets are off. . .

compile [*as object:verb*] Command

This is a verb editor command. It installs the new program into the system if there are no syntax errors. If a new *object:verb* is specified and actually turns out to exist, that *object*:*verb* becomes the default for subsequent compilations.

save [*note-object*] Command
save [*object.property*] Command

This is a note editor command. It installs the freshly edited text. If *note* or *object.property* is specified, text is installed on that note or property instead of the original one. In addition the new note or property becomes the default for future save commands.

subj*ect [*text*] Command

This is a mail editor command. It is used to specify a `Subject:` line for your message. If *text* is `""`, the `Subject:` line is removed.

to [*recipients*] Command

This is a mail editor command. Specifies a new set of recipients (the `To:` line) for your message. Recipient names not beginning with * are matched against the list of players. Recipient names beginning with * are interpreted as mailing-lists/archives/other types of non-person addresses and are matched against all such publically available objects. If the list you want to use isn't in the database (i.e., isn't located in the database ($mail_agent)) you need to refer to it by object id.

also-to [*recipients*] Command

This is a mail editor command. Adds additional recipients to the To: line of your message. Same rules apply as for the '`to`' command.

pri*nt Command

This is a mail editor command. Print your message as it is going to appear at the far end.

send Command

This is a mail editor command. Send your message and exit the mail room. If there are bogus addresses on your `To:` line, the message will not be sent. It may be, however, that valid addresses on your `To:` line will forward to other addresses that are bogus; you'll receive warnings about these, but in this case your message will still be delivered to those addresses that are valid.

who Command

who *rcpt...* Command

This is a mail editor command. Invokes `$mail_agent`'s mail-forwarding tracer and determines who (or what) is actually going to receive your message. The resulting list will not include destinations that will simply forward the message without `:receive_message()`'ing a copy for themselves.

The second form expands an arbitrary list of recipients, for if e.g., you're curious about the members of particular mailing list.

showlists Command

This is a mail editor command used to print a list of the publically available mailing lists or archives and other non-player entities that can receive mail.

subscribe to *list-name* Command

subscribe [*name...*] **to** *list-name* Command

This is a mail room command. Add yourself to the given mailing list. The second form adds arbitrary people to a mailing list. You can only do this if you own the list or if it is listed as [Public] and you own whatever is being added. Use the '`who`' command to determine if you are on a given mailing list.

unsubscribe *from list-name* Command

unsubscribe *name... from list-name* Command

This is a mail room command. It is used to remove yourself from the given mailing list. The second form removes arbitrary people from a mailing list. You can only do this if you own whatever is being removed or you own the list.

You can use the '`who`' command to determine if you are on a given mailing list.

13 Dealing with Verbs and Properties

Verbs and properties are the elements of objects that make them useful. A verb allows you to define things to do with an object, and properties are used to store state information about the object. A verb can be thought of as a MOO code program, executed when the verb on the object is invoked. This can happen if the LambdaMOO parser matches the user input with the verb on an object, or when another MOO code program explicitly calls the verb.

Several commands are available to allow manipulation of properties and verbs.

@show *object* Command
@show *object.prop-name* Command
@show *object:verb-name* Command
Displays quite detailed information about an object, property or verb, including its name, owner, permission bits, etc. The information displayed for an object can be quite long, but usually fits on most screens.

@chmod *object object-permissions* Command
@chmod *object.prop-name property-permissions* Command
@chmod *object:verb-name verb-permissions* Command
Changes the permissions of an object, property or verb, to those given. The following table shows what permission bits are allowed for each form of the command:

object-permissions
r, w

property-permissions
r, w, c

verb-permissions
r, w, x, d

See the LambdaMOO Programmer's Manual for their meanings.

To clear all of the permissions for an object, verb, or property, use `""` as the second argument.

@chparent *object to new parent* Command
Changes the parent of the named object to be the named parent. The object acquires all the verb and property definitions of its parent, as well as the parent's values for any newly-defined properties. The parent object must be readable by the player in order to use it as a new parent.

13.1 Dealing with Verbs

The following commands are used for creating and manipulating verbs.

@verb *object:verb-name(s)* Command
@verb *object:verb-name(s) dobj [prep [iobj]]* Command
@verb *object:verb-name(s) dobj prep iobj permissions* Command
@verb *object:verb-name(s) dobj prep iobj permissions owner* Command

Adds a new verb with the given name(s) to the named object. If there are multiple names, they should be separated by spaces and all enclosed in quotes:

```
@verb foo:"bar baz mum*ble"
```

The direct and indirect object specifiers (*dobj* and *iobj*) must be either '`none`', '`this`', or '`any`'; their meaning is discussed in the LambdaMOO Programmer's Manual. The preposition specifier (*prep*) must be either '`none`', '`any`', or one of the prepositional phrases possible. (a prepositional phrase with more than one word must be enclosed in quotes (`""`)). All three specifiers default to '`none`'.

It is also possible to specify the new verb's permissions and owner as part of the same command (rather than having to issue separate '`@chmod/@chown`' commands), using the third and fourth forms above.

permissions are as with @chmod, i.e., must be some subset of '`rwxd`'. They default to '`rxd`' (specifying '`w`' for a verb is highly inadvisable). The owner defaults to the player typing the command; only wizards can create verbs with owners other than themselves.

@rmverb *object:verb-name* Command

Removes the named verb from the named object.

@args *object:verb-name dobj* Command
@args *object:verb-name dobj prep* Command
@args *object:verb-name dobj prep iobj* Command

Changes the direct object, preposition, and/or indirect object specifiers for the named verb on the named object. Any specifiers not provided on the command line are not changed. The direct and indirect object specifiers (*dobj* and *iobj*) must be either '`none`', '`this`', or '`any`'. The preposition specifier (*prep*) must be either '`none`', '`any`', or one of the prepositional phrases that are possible.

.program *object:verb-name* Command

Provides or changes the MOO program associated with the named verb on the named object.

This command works differently from all other MOO commands, in that it actually changes how the server will interpret later lines that you type to it. After typing the '`.program`' line, you are in *programming mode*. All lines that you type in this mode are simply saved away in the server until you type a line containing only a single period ('`.`'). At that point, those lines are interpreted as a MOO program and are checked for syntax errors. If none are found, a message to that effect is printed and the code you typed is installed as the program for the verb in question. In any case, after typing the '`.`' line, you are returned to the normal input-handling mode.

@list *object:verb* Command
@list *object:verb* **with parentheses** Command
@list *object:verb* **without numbers** Command
@list *object:verb* **with parentheses without numbers** Command

Prints out the code for the MOO program associated with the named verb on the named object. Normally, the code is shown with each line numbered and with only those parentheses that are necessary to show the meaning of the program. By specifying options as shown in the last three forms above, you can have the numbers omitted and/or all parentheses included.

For example,

```
@list $room:@move
```

to see the code for the '`@move`' command, or even

```
@list $prog:@list
```

to see the code implementing `@list` itself.

@edit *object:verb-name* Command

Enters the MOO Verb Editor for the named verb on the named object.

eval *MOO-code* Command
; *MOO-code* Command

Evaluates the given piece of MOO code and prints the resulting value. If the MOO code begins with one of the MOO language keywords ('`if`', '`for`', '`while`', '`fork`', or '`return`') or with the character '`;`', then the entire piece of code is treated as the program for a verb, with '`;`' appended to the end. Otherwise, '`return`' is appended to the front and '`;`' is appended to the end and that string is treated as the code for a verb. In either case, the resulting verb is invoked and whatever value it returns is printed.

For programmers, this is such a mind-bogglingly useful thing to do that there is a simple abbreviation for this command; any command beginning with a semicolon ('`;`') is treated as a use of '`eval`'.

For example:

```
>eval 3 + 4
⊣7

>;3+4
⊣7

>;for x in (player.aliases) player:tell(x); endfor
⊣Haakon
⊣Wizard
⊣ArchWizard
⊣0

;;l = {}; for i in [1..10] l = {@l, i}; endfor return l
⊣{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

13.1.1 Prepositions

The complete list of prepositions recognized by the command-line parser is shown in the list below:

- with/using
- at/to
- in front of
- in/inside/into
- on top of/on/onto/upon
- out of/from inside/from
- over
- through
- under/underneath/beneath
- behind
- beside
- for/about
- is
- as
- off/off of

13.2 Dealing with Properties

The following commands are used for dealing with properties.

@property *object.prop-name* Command
@property *object.prop-name initial-value* Command
@property *object.prop-name initial-value permissions* Command
@property *object.prop-name initial-value permissions owner* Command

Adds a new property named *prop-name* to the named object. The initial value is given by the second argument, if present; it defaults to 0.

Normally, a property is created with permissions '`rc`' and owned by whoever types the command. However, you may also specify these explicitly, using the third and fourth forms. Only wizards can create properties with owners other than themselves.

'`@property`' can be abbreviated as '`@prop`'.

@rmproperty *object.prop-name* Command

Removes the named property from the named object. '`@rmproperty`' may be abbreviated as '`@rmprop`'.

14 Using Tasks

A task is an execution of a MOO program. There are three ways for tasks to be created in LambdaMOO:

- Every time a player types a command, a task is created to execute that command; we call these *command tasks*.
- Whenever a player connects or disconnects from the MOO, the server starts a task to do whatever processing is necessary, such as printing out '`blip has connected`' to all of the players in the same room; these are called *server tasks*.
- The `fork()` statement in the programming language creates a task whose execution is delayed for at least some given number of seconds; these are *forked tasks*.

To prevent a maliciously- or incorrectly-written MOO program from running forever and monopolizing the server, limits are placed on the running time of every task. One limit is that no task is allowed to run longer than one minute; this limit is, in practice, never reached. The reason is that there is a second limit on the number of operations a task may execute.

Every task has an associated *clock* that counts down *ticks* as the task executes. The server counts one tick for every expression evaluation (other than variables and literals) and one for every time through the body of a loop. If a task's clock winds all the way down to zero, the task is immediately and unceremoniously aborted.

Command and server tasks are given brand-new clocks with an initial store of 20,000 ticks; this is enough for almost all normal uses.

A forked task inherits the clock of the task that forked it, with however many ticks remain on it. To allow objects like cuckoo clocks and other recurring tasks that do a little bit of work every once in a while forever, clocks also regain ticks at the rate of 25 ticks per second, up to the maximum of 20,000 ticks. The seconds are counted from the end of the time that one task was counting down that clock to the time when the next user of that clock actually begins execution.

Because forked tasks may exist for long periods of time before they begin execution, there are commands to list the ones that you own and to kill them before they execute. These commands are covered in the following section.

`@forked` Command

Gives a list of all of the forked tasks you own, along with detailed information about each one. The information includes the following:

`Queue ID:`
: A numeric identifier for the task, for use in killing it .

`Start Time:`
: The time after which the task will begin execution.

`Owner:` You, if you're not a wizard.

`Clock:` The number of ticks left on the clock for the task right this moment; if
the task does not execute immediately, this number will grow, up to a maximum of 20,000 ticks.

`Clock ID:`
It is possible for several tasks to share a single clock. The clock ID is a numeric identifier for each clock, so that you can tell if one is being shared.

`Verb:` The object and verb-name of the code that forked the task.

`Line:` The line number of the first statement that the task will execute when
it starts. Note that the code for the verb in question may have changed since the task was forked; the forked task will use the version that was being executed when it was forked.

@kill *queue-id* Command
Immediately kills the forked task with the given numeric queue ID. The '`@forked`' command is useful for finding out these queue IDs. Only the owner of a task may kill it.

15 Miscellaneous

The following verbs are useful, but not easily categorisable.

@version Command

Prints out the version number for the currently-executing MOO server.

@lastlog Command

@lastlog *player* Command

The first form prints out a list of all players, roughly sorted by how long it's been since that player last connected to the MOO. For each player, the precise time of their last connection is printed.

The second form only shows the last-connection time for the named player.

@memory Command

Prints out all information available on the current memory-usage behavior of the MOO server. Probably only a wizard, if anyone, cares about this.

Verb Index

"

A

B

C

D

E

F

G

H

I

J

L

M

N

P

Q

R

S

T

U

W

Table of Contents