Yib’s Guide to MOOing

Getting the Most from

Virtual Communities on the Internet

Elizabeth Hess

Table of Contents

Foreword....iii
Acknowledgements....v
Introduction....1
Part I Fundamentals....5
Chapter 1 – The Basics....7
  Getting Started....7
  Basic Communications....9
  Requesting a Character and Getting Settled In....14
Chapter 2 – How Do They Do That?....21
  Overview....21
  A Very Brief Introduction to Objects....21
  Exploring an Object-Oriented World....22
  Moving Around in a MOO....25
  The Give and Take of a Multi-User Environment....28
Chapter 3 – What's Going On, Here?....37
  Objects....37
  Moving Objects....38
  Feature Objects....41
  Player Classes....43
  Setting Messages....46
Chapter 4 – Using the Mail System and the Editors....51
  Reading Mail....51
  Sending Mail (and getting a start on using the in-MOO editors)....57
  Mail Options....59
  Using the in-MOO Editors....66
Chapter 5 – Extending the Virtual Reality: Building....79
  Overview....79
  Room Integration and Exit Messages....94
  Determining a Room or Object's Contents Definitively....98
Chapter 6 – Programming....101
  A Brief Overview of What it Is and How it All Works....101
  Yib's Pet Rock: A Programming Tutorial for Beginners....108
  MOO Programming Reference....156
Part II LambdaMOO....173
Chapter 7 – Yib's Guide To Interesting Places....175
Chapter 8 – LambdaMOO-Specific Reference Information....217
  A Short Compendium of LambdaMOO Feature Objects....217
  Popular LambdaMOO Player Classes....226
  An Overview of LambdaMOO's Political System, and How to Use It....232
Glossary of Terms....243
  Conversational Typing Abbreviations....255
Appendix A – Summary of Commands....259
Appendix B – Verbs in $Utils Packages....293
Appendix C – Text of "LambdaMOO Takes a New Direction" (LTAND)....303
Appendix D – Text of "LambdaMOO Takes Another Direction" (LTAD)....307

Appendix E – A Compendium of LambdaMOO Ballots 311
Bibliography 385
Index 387

Foreword

Yib, the author of this volume, needs little introduction to those who participate in the virtual world of LambdaMOO. She is a familiar figure to both newbies and experienced MOOers as a wizard (administrator) on various MOOs, as a provider of advice and support to those interested in learning about MOOing, as a creative and industrious programmer, and as a witty and wise contributor to the social life of the communities in which she participates. Instead, I will use this foreword to say a few words about MOOs and about the origins of this excellent introduction to them.

For those unfamiliar with these virtual worlds, a MOO is a type of program that permits multiple users (or players or participants), typically from widely dispersed sites, to access a shared database simultaneously, via telnet or a client program, and to communicate and interact synchronically and asynchronically. The environment is characterized by a spatial metaphor and an architectural motif. Because of these features, Pavel Curtis, the founder of LambdaMOO, has described it as "an electronically- represented 'place' that users can visit."

A MOO is a world of words. Participants describe themselves (or, more precisely, the characters they control) and the objects they create, and it is the text of these descriptions that players see when they "look" at one another and at the other objects they encounter within the virtual environment. Users act and interact with one another by typing – the objects they create and manipulate and the messages they send and receive appear as words scrolling down a screen. The users who create, name, and describe objects may also program them to "do" something when an appropriate command is given, each object "responding" to a command implied in its description.

In MOOs, players communicate and interact with one another in various ways. They "talk" by typing a command and a message appears as text on their computer monitors. Players can also depict themselves as gesturing or emoting by typing a command and a message that displays nonverbal actions. To participate even minimally in a virtual world like LambdaMOO, it is necessary to learn the basic commands that enable communication and interaction in it. And to partake more fully of the possible experiences there, a player needs to master a more advanced set of commands.

For many years, I have taught courses about the ways in which people behave online, especially in virtual environments like MOOs, and as part of those courses, I have asked my students to enter into the life of LambdaMOO. They are expected to establish characters there and to learn about the community by becoming participant-observers in it. They are asked to communicate and interact with other players, and, as do other participants, they often form friendships with other MOOers. Some even become avid programmers. For these experiences, the students must learn the basics of MOOing and that is how *Yib's Guide to MOOing* came into existence. At my invitation, Yib began to participate in the online meetings of my classes, responding to students' questions about how to communicate and navigate in

LambdaMOO. She also helped them when they were logged into the community, responding to a barrage of requests for help.

In the summer of 1999, Yib audited my class in person. As Yib puts it, "I taught them about MOOing; they taught me about anthropology." As questions about MOOing came up, Yib would answer them briefly in class, then, later, she would post a more in-depth answer to a mailing list that the students could read at their leisure. This collection of essays was eagerly read by those taking the courses; it became the students' MOO bible. The topics ranged from technical questions to social conventions to historical background about some of the major developments in the social and political life of the community. One day I said to her, "You know, you have a book to write," and she said, "Yes, I do." The result is *Yib's Guide to MOOing*.

For both newbies and veteran players, *Yib's Guide to MOOing* offers the clearest and most comprehensive explanation of the concept of a MOO and overview of MOO commands and customs. It has illuminated the world of MOOing for my students and me, and facilitated our participation in and enjoyment of the life of LambdaMOO. We have all benefited greatly from her contribution. And now the *Guide* is available to a wider audience. I fully expect that others will get as much satisfaction and pleasure from her efforts as we have. Everyone interested in participating in LambdaMOO and in virtual worlds like it owes Yib a debt of gratitude.

Tower
February 25, 2003

Acknowledgements

A MOO is a gift economy, and myriad people have given unstintingly of their time, energy, and talent over the years to make MOOs what they are today. This book is my tribute to those who have made meaningful the phrase *user extensible*.

Any mistakes are solely my own.

The following people and MOO players have been of particular help:

Judy Anderson (yduJ)
Pavel Curtis
David Jacobson
Peter “Euphistopheles, Source Error Supreme” St.John
Chris Stacy

Audrey
Barth
Bartlebooth
Bear(tm)
Bits
Boo
Doci
eep
Klaatu
mockturtle
Nim
Ostrich
Pax
Shmool
Tartan_Guest
Tower
Werebull

The LambdaMOO wizards

B.

Introduction

A MOO is a computer program that enables people to interact with one another in a variety of ways through the use of text. One logs on (as one might log on to any computer system), initiates action with any of a very wide variety of typed commands, and reads the results as text displayed on one's screen. Often, the commands one types cause text to appear on the screens of other people who are using the program at the same time, and, similarly, commands that others type may cause text to be displayed to one's own screen – giving rise to interpersonal interaction.

The text that one sees immediately after logging on usually includes the description of a location, often a room within a building (though not necessarily so). One facet of MOOs is that each one is a textually represented world, such as one might encounter in a play, a novel, or a non-fiction work about a particular time and place. Within this represented world, called the *virtual reality* or *VR*, are many rooms and other places that one can explore using commands such as "`north`", "`out`", and "`enter cottage`", and *objects* (presented as textual descriptions) that one can manipulate using commands such as "`open box`", "`look in box`", and "`take surprise prize from box`". When you type one of these commands, appropriate text concerning the object will appear on your screen. Each person logged on to the MOO also appears as an object represented within the MOO's frame story or *theme*. Many of the available commands are used to depict one's represented self as saying or doing various things within the represented scene. It's something like improvisatory theater, except that you type what you would say or do instead of saying things out loud or actually gesturing – with the added fillip that because it's typed text rather than physical action, you can as easily depict yourself as walking across the ceiling as dropping a handkerchief.

Like persons performing in an improvised theater scene, the users of a MOO – usually called *players*, for historical reasons – can interact with one another on a variety of levels. They can behave *as if* the stage set, props, and costumed characters are what they appear to be. In the theater, one might refer to this as acting "in character". On a MOO, we would say that an action or statement was "consistent with the virtual reality." At any time, though, an actor can break from her role and speak or act instead as the *person* performing the role. In a rehearsal, she might do this to discuss a point of how the scene should be played; while not on stage, she might ask another player if he would like to go out for coffee later. During a performance, she might peek into the audience to see how full the house is. On a MOO, everyone is both a participant in the virtual reality (a *player*) and a person who is logged onto and using the MOO computer program (a *typist*).

Actions (typed commands) which are not consistent with or not intended to function as part of the MOO's virtual reality are referred to as *non-VR* or *meta-VR* actions. These actions, while outside the MOO's frame story, virtual reality, or theme, are still done within the MOO itself, that is, while logged on and typing commands, and they are as much a part of MOOing as actions that are consistent with the virtual reality. Some examples would include typing a command to see who

else was logged on to the MOO at a particular point in time, a command to change the description of one's represented self, or a command to add a new room or object to those already available for use by oneself or by other players. MOO's are *user-extensible.* That the players within a MOO are themselves able to modify and extend the virtual reality is a central feature of what MOOs are for and about, and sets MOOs apart from chat rooms, web sites, and other virtual environments.

Much of this book concerns itself with how to investigate and manipulate the MOO's virtual reality underpinnings. Let me begin, then, by introducing a few terms. The MOO *server* is the program that runs on a host computer, whose job it is to handle the connection process, receive typed commands from users, cause those typed commands to be executed appropriately (much more on this later), and display resulting text to users' screens. The MOO *database* is a structured record of all the represented items within the virtual reality and all their technical underpinnings as created and extended by MOO users. A MOO's database is divided into *objects*, which are data constructs that are used both to represent things in the virtual reality and in the meta-VR. By this I mean that an object might represent a couch that you would sit on, in the context of the virtual reality – or it might not represent anything in the VR but serve as a repository of commands for extending the VR. Every object has associated with it a set of attributes, called *properties*, which are named pieces of data that contain information about that object (e.g. its name, its location, etc.). Some properties are common to all objects, while others are specific to certain kinds of objects: for example, objects that represent players (people) have a property associated with them indicating gender, while objects that represent pieces of furniture typically do not. In addition, objects also have associated with them sets of coded instructions, called *verbs*, that control their behavior. By creating new objects within the MOO, and adding interesting properties and useful verbs to these objects, *every* player can expand the virtual reality. Players adept at programming can even create virtual tools designed to modify the virtual reality.

The objective of this book is to provide a comprehensive yet comprehensible explanation of the many commands, both VR and non-VR, available to users of MOOs, in hopes of helping people learn the ropes as quickly as possible and get the most from their MOOing experience. Part I of the book focuses on commands that are common to all MOOs based on a particular common starting database called "LambdaCore". Part II provides information specific to LambdaMOO, which is the original and largest MOO in existence today (having between 3000 and 4000 registered users at any given time).

A note about typography: this is partly a book about words that people type into a computer program and words that a computer program prints on people's screens.

In the beginning, for clarity, if I specify something that you are to type in, I will put it on its own line, thus:

```
connect Guest
```

In the interest of space, though, I sometimes place text that you are to type (or text that the computer displays) within a paragraph, thus: `connect Guest`.

I use angle brackets <> as a placeholder for specific information that you fill in yourself. For example, if I told you to type:

```
say Hi, my name is <your name>.
```

and your name were Sally, then you would type:

```
say Hi, my name is Sally.
```

A note about the “Spivak” pronouns I use in this book: The mathematician Michael Spivak developed a set of gender-neutral singular pronouns for use in his books. They are popularly used on LambdaMOO when one doesn’t know a person’s gender, or when one is referring to a generic person such as “the reader” or “a MOOer”. They correspond to the familiar pronouns as follows:

E:	He/She
Em:	Him/Her
Eir:	His/Her
Eirs:	His/Hers
Emself:	Himself/Herself

Any other terms may be found in the glossary at the end of the book.

One last bit of reassurance: a MOO is a very rich interactive environment, providing thousands of different commands that enable people to use the MOO in many different ways. Because the commands are so many and so varied, the initial learning curve can seem quite steep, and people who aren’t comfortable using computer programs whose interface is strictly text may feel daunted. Please be assured that MOOs are intended to be arenas of exploration, and that while you may find yourself confused or uncertain on some occasions, there is nothing you can do that will mess things up irrevocably. You will learn as much (and hopefully more) from your own experimentation as you will from this book. Go ahead: Try things!

Part I

Fundamentals

Chapter 1 – The Basics

Getting Started

Learning is active. No amount of reading about MOOs and MOOing can convey what it's really like as well as jumping in and doing it! The goal of this section is to present the basics of connecting to a MOO and starting to get your bearings.

First, you must connect to the MOO. The most primitive way to do so is using telnet. The syntax, from a command line host computer, is

```
telnet <MOO name> <port number>
```

For example:

```
telnet yib.moo.mud.org 7777
```

or

```
telnet lambda.moo.mud.org 8888
```

Some telnet applications may have you type the MOO name and port number into a dialog box, then select "Connect" or "OK".

This is sometimes referred to as *raw telnet*, because there is no additional software between you and the MOO. In some cases, the backspace key doesn't work while using raw telnet. Sometimes, even the text you type doesn't display! (Some telnet programs have menus to adjust these things.) Also with telnet, if text is sent from the MOO to your screen while you are typing (and it often is), the text will appear right in the middle of the line you are typing, and things get very confusing, very quickly. For this reason, most people try to obtain a kind of program known as a MOO (or MUD) *client*. There are a variety of client programs available, and your choice of a client will depend partly on what kind of machine or operating system you are using to connect to the MOO. Client programs are not part of the MOO per se, but greatly enhance one's MOOing experience. I strongly recommend them.

The most important thing that a client does is separate, in some way, the text that you are in the middle of typing from text that the MOO is sending to your screen. Some clients have a small window at the bottom of a main window; others keep shifting your line down and inserting received text above it. The rest is frosting: Some show the text that you type in bold face, for example. Most let you use a scroll bar or other command to review previously-displayed text; some provide fancy editing capability, and so forth, but the main thing is to be able to see straight (so to speak). Some well-known client programs include Pueblo, tkMOO, MUDDweller, and emacs with mud.el. To find a client that's compatible with your computer, I recommend using an Internet search engine such as `www.google.com` and searching for `MOO clients`.

The first time you visit a MOO, you will connect as a guest. Guests have limited privileges (they can't receive MOOmail, for example), but are equipped to do all the basics of communicating and getting around. It's usually a good idea to visit a given

MOO a few times as a guest before requesting a character, to get a feel for the place and the people and then decide if you actually *want* to get a character there. Characters can select their own names, and usually have at least limited building privileges.

Please read the welcome screen.

After doing so, type:

```
connect guest
```
[1]

If you do use a client, you will probably have to specify the MOO's address in a set-up window. The specifics of how to do this are different for each client and beyond the scope of this book.

A good place to find out about many of the MOOs that are available is Rachel's Super MOO List, at:

http://cinemaspace.berkeley.edu/~rachel/moolist/index.html.

On most MOOs there will be some information that you are invited and requested to read when you first connect. First, you may be instructed to type `help` at any time for assistance. You may then be informed that there is new news, and instructed to type `news` or `news new` to read it.[2] After that (on LambdaMOO), you

---

[1] Since you are logging in anonymously, you may be asked a few additional questions. Take the time to read and answer them.

Most MOOs permit more than one guest to be logged on simultaneously. Your actual guest name might be something like "Green_Guest", or "Blue_Guest".

[2] If you *don't* use a client program, there are certain commands that you will want to be aware of from the very start.

First, you will need to specify the size (height in lines and width in characters) of the screen or window you are using. To specify the height of your screen, type

```
@pagelen <number>
```

Many screens have 24 lines.

To specify the width of your screen, type

```
@linelen <number>
```

Many screens have a line length of 80 characters; if yours does but things still look funny, specify one fewer characters than are actually displayed on a line.

Finally, type

```
@wrap on
```

This will prevent words from being broken in the middle across line breaks.

If you do these things, then when your screen gets full, you will be prompted to type `@more` to cause more text to be displayed.

If, after getting a character (see the section on requesting a character and getting settled in, page 14), you *switch* from using raw telnet to using a client, you will probably need to type the following:

```
@pagelen 0
@wrap off
```

Doing this will prevent the MOO from prompting you to type `@more` every time your screen gets full (because now the client lets you scroll back). Most clients separate lines at word boundaries, so typing

are invited to type `@tutorial` for an introduction to basic MOOing, and requested to type `help manners` and read the text presented if you have not already done so. This is an awful lot of stuff to read. I recommend you proceed in the following sequence:

Skim `help manners` or `help rules` or any indicated text about the way you are expected to behave. There will likely be several things in it that may not make sense at first, but MOOers will often expect you to have read it anyway. On the first pass, your goal should be to get what you can from it, note the various *kinds* of things that are in it, for future reference, and to realize that the community *does* have formal expectations of its members and guests.

If a tutorial is indicated, do that next.

Then, type `help` just to see the basics of the online help system. If a topic grabs your interest, go ahead and read about it by typing:

```
help <topic>
```

If there is a newspaper, read it, by typing `news`. If an obvious exit is indicated, try it. Welcome!

When you are ready to disconnect from the MOO, type:

```
@quit
```

Basic Communications

This section explains various ways of interacting with other people on a MOO.

Say

The most fundamental communication command on a MOO is the `say` command, and this is what to use if you want to say something to someone who is in the same room as you. If Yib types:

```
say Hello.
```

Then Yib will see on her screen:

```
You say, "Hello."
```

And everyone else in the room with Yib will see on eir screen:

```
Yib says, "Hello."
```

`@wrap off` relieves the MOO itself of that task and may speed up the MOO's response time slightly, which is desirable.

The `say` command is so basic, and used so much, that it has an abbreviation, which is the double-quote character ("). If Yib wants to say that she likes to tap dance, she can type:

```
"I like to tap dance.
```

and she will see on her screen:

```
You say, "I like to tap dance."
```

and everyone else in room when Yib says this will see on eir screen:

```
Yib says, "I like to tap dance."
```

Notice that when using this form of the command, you do not type a close-quote character at the end. The system appends one automatically.

Emote

Sometimes you might want to express something non-verbal, such as a feeling or a gesture. It might seem awkward to say, "I feel happy and am smiling." For this situation, use the emote command. This command will prepend your name to whatever you type after it. For example, if Yib types:

```
emote feels happy.
```

Everyone in the room (including Yib) will see on eir screen the line:

```
Yib feels happy.
```

Notice that Yib typed her sentence in the third person. If she had typed:

```
emote feel happy.
```

Then everyone would have seen the line:

```
Yib feel happy.
```

The emote command, too, is used so much that it has a single-character abbreviation, which is the colon (:). So Yib could type:

```
:feels like dancing.
```

and everyone in the room (including Yib) would see on eir screen:

```
Yib feels like dancing.
```

There is a difference between emoting things and actually doing things, i.e. actually interacting with objects on the MOO. Suppose Yib is holding an object called "linen handkerchief". If Yib emotes the line:

```
:drops linen handkerchief.
```

Everyone in the room would see the text line:

```
Yib drops linen handkerchief.
```

But Yib would *not* in fact have dropped it. If one were to look at the room Yib was in at the time, one would not see the handkerchief there. If Yib had instead typed:

```
drop linen handkerchief
```

then the object, `linen handkerchief`, would actually move from Yib to the room that Yib was in. This is a fundamental concept (though perhaps a subtle one, especially at first): whether one "merely generates text", or whether one causes a change in the database, e.g. changes the location of an object. Yib could emote dropping an elephant, or lifting one, and the text would print out, regardless of whether or not there actually was an elephant in the vicinity.

A special form of `emote` is the double colon (`::`) It is just like a regular `emote` except that a space doesn't appear after your name. A typical usage would be:

```
::'s new hat is of truly astonishing dimensions!

Yib's new hat is of truly astonishing dimensions!
```

Many MOOs have a collection of short cut commands such as `wave <person>` or `hug <person>`, which can be used for frequently-depicted actions such as waving, hugging and so forth. People who get spoiled with these short cuts often neglect emote, but it remains one of the single most flexible and expressive commands on the MOO.

Directed Say

Typically, when a room has a large number of people in it, the conversation tends to break up into several conversations going on at once. It's impractical to listen to everyone talking at the same time, and people don't. On a MOO, every utterance and gesture appears on a line by itself, so, in theory, one could keep up with everything, even in a crowded, noisy room. In practice, however, MOO conversations also tend to break up in crowded rooms, and it's typical to follow what one or a few people are saying and tune out the rest. To facilitate this, there is a command that is generally referred to as *directed say*.[3] To direct a remark to a particular person, begin your line of text with a dash (-), then, without typing a space, type the name of the person you wish to address, then a space, then your remark. For example, Barth might direct a remark to Yib by typing:

```
-Yib My, what a lovely hat you have on!
```

Everyone in the room (including Barth and Yib) would see the line:

```
Barth [to Yib]: My, what a lovely hat you have on!
```

---

[3] Different MOOs may or may not provide this automatically to brand new players. LambdaMOO does, using a kind of object called a *feature object* (FO). (Feature objects are explained in the section beginning on page 41.)

Whisper

You might want to communicate something to one person only, without others in the room being aware of it. For this there is the *whisper* command. The syntax of this command is slightly different than the other communications commands, in that you have to put quotes around the text you wish to transmit. Here's an example. Nim types:

```
whisper "Meet me in the hot tub in two minutes." to Yib
```

Yib sees:

```
Nim whispers, "Meet me in the hot tub in two minutes."
```

Nim sees:

```
You whisper, "Meet me in the hot tub in two minutes." to
Yib.
```

No one else in the room sees anything of this exchange.

Page

It's often the case that one wishes to communicate with someone who is logged on but not in the same room. For this we have the `page` command. The syntax is `page <person> <message>`. If Boo were in a room called `A Quiet Place` and wanted to greet Yib from afar, she might type:

```
page Yib Hi!  Been up to any mischief lately?
```

Yib would see:

```
You sense that Boo is looking for you in A Quiet Place.
She pages, "Hi!  Been up to any mischief lately?"
```

Boo would see:

```
Your message has been sent.
```

Because this is virtual reality, and because the players themselves can change the way it works, even simple commands like `page` can give fun and interesting results. For example, Boo can adjust the text that appears in the first line that a person sees when she pages em. For example, she might set it to say, "Boo tosses you a poison dart with a message attached." Yib can adjust the text that someone paging her sees when the message is received. For example, she might set it to say, "Yib adjusts her glasses and reads your message." (How to do this is explained in the segment on setting messages, beginning on page 46.)

Remote-Emote

Sometimes, instead of paging, one wants to gesture from afar, and the name given to this is `remote-emote`. The syntax for this command is `+<player> <text>`. If Yib were on top of the observatory dome and wanted to wave to Tartan_Guest from there, she would type:

```
+Tartan_Guest waves.
```

Tartan_Guest would see:

```
(from On Top of the Observatory Dome) Yib waves.
```

Yib would see:

```
Tartan_Guest has received your emote.
```

Like emote, this command is very flexible, but doesn't actually *do* anything, i.e. it doesn't change the database. If I were to remote-emote to Klaatu:

```
+klaatu hooks you with her fishing pole and reels you in!
```

Klaatu would see the text, but would not in fact be reeled in anywhere, and would remain wherever he was. This distinction is important, because one *could* make a fishing pole object, and endow it with verbs that moved other objects (e.g. fish, or other players).

Channels

Channels on a MOO are kind of like channels on a CB radio. The idea is, you tune to a given channel, and you and everyone else tuned to that channel can communicate with one another even though you may all be in different locations. One difference between a channel and a real CB radio is that with a real CB radio, everyone present in the room can hear it. On a MOO, only those connected to a channel can listen to it, even though others may be present in the room. There are public channels that anyone can join, and private channels that have restricted access. Channels are quite popular and many MOOs have them, though they are not included as part of LambdaCore. Details on how to use channels on LambdaMOO are given in the section on LambdaMOO Feature objects (see page 217).

Recording Communication

All of these commands (say, emote, directed-say, whisper, page, remote-emote and communicating on channels) are examples of communication that occurs in real time, i.e., the information is received at the time of transmission rather than stored for retrieval at a later time. In general, communication of this sort is of a transient nature, although it is stored in the computer's Random Access Memory (RAM) and is

protected by copyright[4] . Players can issue a command (@paranoid) to keep the most recent several lines available for closer inspection. Furthermore, many players' client programs save arbitrary numbers of lines which can be viewed in the client window by scrolling back. It is also possible for players to save logs (transcripts) of MOO sessions to a file on disk. In theory, only the guests and players in a particular room are supposed to be privy to conversation that occurs in that room at that time, but such is rarely the case in practice. It is technically possible for conversations to be bugged. Occasional overt "bugging" is tolerated, (for example a player might participate in a conversation remotely via a surrogate called a "puppet"), but this practice is generally discouraged. Covert bugging, also called "spying", is generally discouraged, and is rare. The command @sweep is provided for those who wish to check to see whether a listening device is present.

Requesting a Character and Getting Settled In

The Initial Request

Different MOOs have different registration policies. LambdaMOO requires that each player provide a valid email address and its administrators take pains to ensure that there is only one character per typist, or else that multiple characters are duly cross-referenced to one another. Other MOOs may have less restrictive policies.

The usual syntax for requesting a character on any MOO is:

```
@request <name> for <email-address>
```

For example:

```
@request Cinderella for cindy@fireplace.com
```

Read the questions carefully, and answer straightforwardly. Depending on which MOO you're on, the system may check your connection site and compare it against the email address you give. If they don't match, it may ask you for a brief explanation. If you already have a character and are requesting a second (this is permitted on some MOOs, not on others), it may ask you to verify this fact, and/or explain how it might be that there is already a character with the email address you've specified. Some MOOs disallow certain email addresses, specifically those from providers that are known for giving free, anonymous email accounts. If you have multiple email addresses, you may be asked to provide them.

The questions that are asked of you in the @request process come in three forms: YES/NO questions, single line answers, or multiple line answers. If you change your mind about answering any question at all, type:

```
@abort
```

(Note that if you are in the middle of a multi-line answer, @abort must be on a line by itself.) For the multiline answers, you may type in as many lines as you need to. A line, for the purpose of this discussion, is an arbitrary amount of text

[4] David Jacobson, "Doing Research in Cyberspace," Field Methods, 1999, 11:2:;127-145.

terminated by the <enter> key. To end your answer, type a period (.) on a line all by itself (and then press the <enter> key again). Sometimes further explanation is needed. In such cases, you will be given a regular email address to use for providing it.

Soon you should receive email at the address you provided, containing a password and verification of the player name you requested, or notification that the name you requested was already in use.[5] In either case, the character you requested will also have an alias of the form `New-Player-<number>` e.g. `New-Player-58337`. The password is case-sensitive, i.e. you must type it in exactly as given. You are not stuck with the name `New-Player-<number>`. After you connect initially (and at any time thereafter, as often as you like), you have the option of changing your name to anything you like. Suppose the name "Cinderella" was already in use, and our intrepid typist received email saying that e had been given a character with the name "New-Player-58337" and password "OgkM2".

The first order of business is to log on. Our typist would connect to the MOO, see the welcome screen, then type:

```
connect New-Player-58337 OgkM2
```

There are a variety of things to do from this point, and the sequence isn't essential. Two obvious things are to decide on a different name, and to change the password to something that's easier to remember.

Changing Your Password, Changing Your Name, Adding Aliases

Passwords have to be more than four characters long, and are not permitted to be common English words. One scheme is to select two English words and run them together, for example, "RubySlippers". As always, passwords are case-sensitive. To change your password, type `@password <old-password> <new-password>`. So our typist would enter:

```
@password OgkM2 RubySlippers
```

Now let's consider a name change, and perhaps some aliases. First, you'll want to cast about and see if the name you're considering is already in use, since names have to be unique. To do this, you can use the `@who` command. Suppose we want to see if the names "Drusilla" and "Prunella" are taken.

You type:

---

[5] LambdaMOO is under certain population-growth restrictions, and so it is possible that your request will be added to a queue, there. At the end of the `@request` process, you will be told what your place in the queue is. Most times, the queue moves along fairly briskly. You can check your status as often as you like: To do so, log on as a guest, then type:

```
@go registrar
```

Then, `check status on <email-address>` using the email address you gave when you requested your character. The person requesting the character Cinderella, for example, would type

```
check status on cindy@fireplace.com
```

```
@who Drusilla
```

And you see on your screen:

```
Disconnected
Player              Last Disconnect              Location
------------------  ------------------------     -----------
Werebull (#58806)   Sun Sep 19 02:30:42 1999 EDT The Pasture
```

This indicates that the character Werebull already has "Drusilla" as an alias, and therefore that name is not available to you. Let's try Prunella:

You type:

```
@who Prunella
```

And you see on your screen:

```
"Prunella" is not the name of any player.
```

Success! To change your name, use the `@rename` command, as follows:

```
@rename me to Prunella
```

Names of players are not case-sensitive. Your name will appear as you type it in when you use the `@rename` command, but someone typing `@who pRuneLLa` would still find *you*.

You are not limited to one name. You might, for example, want to use "Prunie" as a nickname. As before, the first thing to do is to see if that name (alias) is already in use. You type:

```
@who Prunie
```

And if you're lucky, you'll see:

```
"Prunie" is not the name of any player.
```

To add a name (as opposed to changing your name), use the `@addalias` command:

```
@addalias Prunie to me
```

If you later decide you don't like that nickname so much after all, you can remove it as follows:

```
@rmalias Prunie from me
```

Describing Yourself

The description you give yourself is what people will see when they look at you. Use the `@describe` command as in the following example:

```
@describe me as "One of Cinderella's two wicked step
sisters.  Her beauty, such as it is, is as cold as her
scheming heart."
```

Type in the whole description as one long line (even though it shows as multiple lines in the example).

Notice that the description is enclosed in double-quote marks. This is optional, except that it happens to be the case that if you omit them, there will only be one space between sentences, even if you type two spaces between each sentence. If you use the double quotes, there will be as many spaces between sentences as you type.

If you find that you've made a typographical error in describing yourself, you have two options. One (easiest if your description isn't too long) is to `@describe` yourself all over again, as above. You can revise and re-enter your description as many times as you like. The other alternative is to learn to use the note editor (see the section that begins on page 66) and *edit* your description

As you meet other players, you will notice that some of them have multi-line descriptions. You can't get a multi-line description with `@describe`. This is another incentive to learn to use the editor, as it is the best way to create a description that has more than one line (paragraph).

Sooner or later you will undoubtedly encounter someone who reacts or responds to the fact that you looked at em. Certain player classes provide the ability to be notified when someone looks at you; this is generally referred to as *look detection*, and it can be disconcerting the first few times you encounter it. In general, it is considered poor form to give someone grief for looking; some people assert that it is rude even to mention that one knows someone looked. In contrast, some players not only notice when someone else looks at em, but in addition broadcast a message to the room, such as, "Oliver notices Prunella's glance and smiles at her," or worse, "Prunella looks at Oliver and smiles." The second form is worse because it should be up to Prunella to decide whether she smiles at the sight of Oliver or not. Perhaps it is Prunella's nature to sneer, instead, and she should have the right to specify that. MOOers are not united in their opinions on this issue, although most would probably concede that broadcasting a message to the room every time someone looks at you becomes boring fairly quickly.

Setting Your Gender

MOOs offer a variety of gender options. You can use the `@gender` command in two ways. If you type it on a line by itself:

```
@gender
```

The system will tell you your current gender setting, the pronouns it associates with that gender, and will show you a list of available genders. To set your gender, use the `@gender` command with an *argument* (see the glossary for an explanation of what an argument is), e.g,

```
@gender female
```

In this case, the system would display:

```
Gender set to female.
Your pronouns:
  she,her,her,hers,herself,She,Her,Her,Hers,Herself
```

Since there is no way to tell, online, whether someone is telling the truth about eir gender, we sometimes speak of a player as "presenting as male" or "presenting as female".

Setting a Home

When you disconnect from the MOO (using the command `@quit`), your player object is returned to its `home` on the MOO. Players who have not otherwise specified a home are returned to a default location. (On LambdaMOO it's the Linen Closet.) There is nothing wrong with keeping the default player start as your home. Many players do this.

Another option is to find a room that will permit you to set your home there (to do so, go to the room and type `@sethome`). There is no penalty for trying to set your home to a location where that is not permitted. You will simply be given a message instructing you to ask the owner to make you a resident of that room. If you are the owner of a room and someone else wishes to set eir home there, and you wish to let em do so, the command is:

```
@resident <player or object>
```

To see a list of your room's residents, type:

```
@residents
```

To remove someone from the list of residents type:

```
@resident !<player or object>
```

A third option is to create a home for yourself. Most players do this eventually. To do it, you would use the `@dig` command. For an in-depth discussion of `@dig` and related commands, see the section on building, starting on page 79.

Suppose Prunella wanted to create a home for herself named "The Crystal Palace". She could type:

```
@dig The Crystal Palace
```

The system would then create a new room object, would set Prunella to be the owner of this room object, would set the name of this room object to "The Crystal Palace", and would print a message to Prunella saying that the room had been created and informing her of its object number. A detailed discussion of objects begins on page 37.

```
The Crystal Palace (<object-number>) created.
```

To get to her new room, Prunella will have to teleport there, using the object number that the system just printed out for her. If the object number were #29370, for example, she would type:

```
@go #29370
```

After arriving, Prunella could type:

```
@sethome
```

and then that location would be her new home. She would be in that room when she connected, and would be returned to that room when she logged off. She could also go there at any time by typing the `home` command:

```
home
```

The room would start off with no description. Prunella could give it one with the `@describe` command. First she would `@go` there, then she could type:

```
@describe here as "You are in a magnificent palace, as warm
and inviting as its name suggests."
```

If Prunella ever forgot the object number of her room, she could type:

```
@audit
```

And that would show her a list of objects she owns, including herself and the room she had just created.

Unexpected Greetings

Sooner or later, someone will greet you (typically with a page or a remote-emote) within moments of your logging on. How do they do this? There is a *feature object* (see page 41) that lets one designate interesting players, and which will notify one when an interesting player connects or disconnects. These are called *login watchers.* Most MOOs have some version of a login watcher.[6]

Another thing that may seem disconcerting is that people may know that you are new without your having told them. It's more than just being an unfamiliar face. The `@age` command tells how old a person is in terms of the MOO.[7]

Eventually, you will become familiar with these commands, and take such spontaneous greetings in stride, or even start making them yourself!

---

[6] On LambdaMOO, it's feature object #24222.

[7] On LambdaMOO, other commands to find out about players' ages are on Carrot's Social Interaction Feature (#36714).

Chapter 2 – How Do They Do That?

Overview

After you've visited a MOO a few times, requested a character, and started to get the hang of communicating with others, you will undoubtedly begin to notice that there's a whole lot more to MOOing than `say` and `emote`:

- You will find that MOOs have different rooms and locations to explore.
- You will learn that there are commands you can type that will cause some of the depicted objects to behave in various ways.
- You will undoubtedly have seen many *object numbers* (indicated by the #-sign).
- People may arrive and depart in showers of sparks, clouds of smoke, or a burst of flame, and you may begin to wonder, "How do they do that?"
- You'll soon discover that the more experienced players have a great many commands available to them that you do not.

While you don't have to know the particulars of how a clutch works in order to drive a stick-shift car, a basic knowledge of how the clutch pedal, gas pedal and gear shift interact will make the going much easier. Similarly, you don't have to know how to program to enjoy MOOing, but a knowledge of some of the underpinnings will enhance your MOOing experience.

This chapter introduces several of these advanced techniques; the following chapter discusses some of these things in greater depth.

A Very Brief Introduction to Objects

A *MOO* is a multi-user domain that includes a programming language that anyone may use to extend the domain. The particular kind of programming language that MOOs use is called an *object-oriented* language, and as you explore the MOO, you will encounter *objects* everywhere.

What you need to know about objects at this juncture is that they are there. It is also helpful to know that every object has a unique number that identifies it. If you buy something from a mail order catalog, you might call and say, "I'd like to buy one frizzlebopper, please," or you might say, "I'd like to place an order. The first item number is #4612." Object numbers on a MOO are like item numbers in a mail order catalog.

There is an in-depth discussion of objects beginning on page 37.

Exploring an Object-Oriented World

Text-based virtual worlds generally have an underlying metaphor or *theme*. When you first connect, you typically see a description of where you are within that world. It might be an open field, the deck of a space ship, a room in a house, or the reception area of an office building. Within the world described by the words on your screen you can explore and interact with the various things and people you encounter there.

To see a description of the room you are in at any time, type `look`, which can be abbreviated to the single-letter command, `l`.

Let's suppose that you've logged onto LambdaMOO for the first time, and have opted to begin in the quiet location. After seeing all the information about news, the tutorial, and `help manners`, you may want to have another look your surroundings. Type:

```
look
```

You will see the following text:

```
The Linen Closet

The linen closet is a dark, snug space, with barely enough
room for one person in it.  You notice what feel like
towels, blankets, sheets, and spare pillows.  One useful
thing you've discovered is a metal doorknob set at waist
level into what might be a door.  Another is a small button,
set into the wall.
```

Now let's look at some things that are in this linen closet. If you type, `look towels` (or `l towels`), you will see:

```
You can make out the outline of some towels, but it's too
dark to tell what color they are.
```

This is nice, but not especially interesting; how about the blankets?

```
l blankets
```

```
You can make out the outline of some blankets, but it's too
dark to tell whether they are flannel, wool, or electric.
```

Well, so far, so boring. But not as boring as it could be, actually. Someone, in fact, went to the trouble of writing text for you to see if you look at the towels, the blankets, the sheets, the pillows, the button, or the door. Most rooms, as a matter of practicality, have some things that are merely mentioned in the room's description, and others that actually exist as things with descriptions in their own right. Suppose the author had elected not to include the pillows as something you could look at. In response to `l pillows`, you would see:

```
I see no 'pillows' here.
```

The author of the linen closet mentions two things as "useful": a doorknob and a button. But *how* are you to make use of them? You might think to try typing `turn doorknob` or `push button`, and if you were to do so, something would happen in either case. Before you leave the linen closet, though, it's worth knowing that you don't have to guess about which objects are interactive and how to interact with them; the command `examine` exists to enable you to determine that. If, instead of typing `look button` you were to type `examine button`, you would see the following:

```
button (aka #53344 and button)
Owned by Groundskeeper.
A small black button, set into a tarnished bronze plate on
the wall.
Obvious verbs:
  press/push/poke button
```

The `examine` command can be abbreviated to `exam`. `Exam doorknob` yields:

```
doorknob (aka #79708, doorknob, and knob)
Owned by Groundskeeper.
You see a plain metal doorknob.
Obvious verbs:
  turn doorknob
```

Let's consider each of these two instances. Both the button and the doorknob are *objects*, which means that they exist as things within the MOO and aren't merely mentioned in the text of a room's description. The first line you see after examining an object is a list of the object's aliases, and the object's number. (For now, it's perfectly fine to ignore the object numbers, but referring to an object by its number always works.) The button has only its object number and its name ("button") as aliases, but the doorknob also has the alias "knob", so the casual user could type `turn knob` and get an appropriate result. The second line you see tells you who owns the object. This, too, you can ignore for now, but if the button or doorknob were to malfunction in some way, the appropriate person to notify would be Groundskeeper, who owns these objects. Then there is a list of "obvious verbs". (It's a mystery to me why objects aren't called "nouns" or why verbs aren't called "commands", but that's the way it is.) Verbs are commands that you can use to manipulate objects. (The word "verb" has a broader definition in MOOs, but it's sufficient, for now, to think of it as the name of a command – something you can do with or to an object.) In these two cases, you could either turn the doorknob or push the button. It is an idiosyncrasy of MOO syntax that definite and indefinite articles are usually omitted.

I will note here that there are inconsistencies within the MOO as to how you can tell, within a room, what objects are in it that might be interesting or useful, and this is because different rooms are programmed differently. Every room has a mechanism (i.e. a program) associated with it to display its contents. Early on, most rooms displayed their descriptions and contents like this:

```
Guest Cottage Porch
You are on a breezy, screened-in porch.  A rocking chair and
a porch swing invite you to stay and relax for a while.  A
```

```
screen door leads west into the cottage; steps lead down to
the lawn.
You see glass of lemonade and harmonica here.
Yib and Bartlebooth are here.
```

First is the name of the room, then the room's description, then a list of non-player objects in the room prefaced by the words, "You see", then a list of the players present. With such a scheme, a player might reasonably pass up trying to examine the rocking chair, the porch swing, the screen door and the steps, and would zero in on (and examine) `glass of lemonade` and `harmonica`, hoping, perhaps, to be able to `drink lemonade` and `play harmonica`. Said visitor to the guest cottage porch might also be moved to greet Yib and Bartlebooth.

At some point, it became possible for rooms to be dark, because some rooms are. The coat closet and the linen closet on LambdaMOO are two examples of dark rooms. These rooms specifically don't list the items or people who are present, unless they're also mentioned as part of the text of the room's description (like the towels in the linen closet).

Later developments in room technology enabled statements about some objects' presence to be integrated into a room's text description. So, for example, if the guest cottage porch were an integrating room, and a model of the gazebo had an integrating message on it, it might look like this:

```
Guest Cottage Porch
You are on a breezy, screened-in porch.  A rocking chair and
a porch swing invite you to stay and relax for a while.  Off
to one side is a model of the gazebo.  A screen door leads
west into the cottage; steps lead down to the lawn.
You see glass of lemonade and harmonica here.
Yib and Bartlebooth are here.
```

Anytime the model of the gazebo was moved to an integrating room, the sentence, "Off to one side is a model of the gazebo," would appear in that room's description rather than in the list of objects that conventionally follows the text description.

Another type of room is the detailed room, whereby an owner can add extra descriptions, so, for example, if a room's description mentioned wallpaper, you could type `look wallpaper` and get some extra detail, even though there wasn't actually an *object* named `wallpaper` in the room. So in some rooms there are objects whose presence is not obvious because a room is dark, or there are objects whose presence is not obvious because the objects are integrated into the room's text description, *and* there are "things you can look at" (in detailed rooms) that aren't actually objects at all. Each of these developments was seen as an improvement at the time. Detailed rooms were thought to be "richer" than rooms that had single, simple descriptions. Integrating rooms were thought to "read more naturally" than rooms that appended, "`You see <stuff> here`," at the end of their descriptions. All I can say is, bear with it, and when in doubt, `examine`. Eventually you get a feel for it.

To summarize, then: `look` or `l` to see a description of the room you are in at any time. `Look <object>` or `l <object>` to see the description of an object, if there is

such an object in your vicinity. Examine <object> or exam <object> to see a list of its aliases, find out who owns it, and what, if anything, you might try doing with it.

Moving Around in a MOO

MOOs generally depict a spatial or architectural metaphor. When you first connect, you are in a room – either the room designated as $player_start (by definition the place where players start), or in another room that you've created and/or set as your home.

There are two basic modes of moving around. These are generally referred to as *walking* and *teleporting*. Another way of thinking about them might be *VR* (Virtual Reality) and *non-VR* or *meta-VR*: ways that transcend or break the frame story of the virtual reality.

In the computer games Adventure and Zork, in which MUDs and MOOs have their roots, most travelling was done using compass directions (and sometimes "up" and "down"). You might see, for example, words to the effect that there was a house to the east, and you would type the command east or e to enter the house. Although many people find the notion of moving in compass directions non-intuitive – when I leave my house I simply go out the front door without thinking "northeast" – nonetheless compass direction exits are still in frequent use in MOOs today.

Most room descriptions whose exits are named for compass directions will give cues to that effect. For example, part of the description of LambdaMOO's library alcove reads, "The library itself is back to the east. A spiral staircase leads up." You might reasonably expect that typing east would move you to the library and that typing up would move you up the stairs, and you would be right. It is possible to look before you leap, so to speak. You can type look east or l east before typing east. Failing to look is rarely of consequence, but LambdaMOO builders are strongly encouraged to give their exits descriptions, and from time to time there's an especially good one. It's worth knowing that the descriptions are there, at any rate.

On some MOOs, you can type help map to see a map of the MOO's main area or areas[8]. There are also some non-VR ways to check for available exits and get your bearings. You can type @ways for a list of obvious exits from the room you are in.

Exits can have other names besides compass points and "up" and "down". Some are intended to be intuitively obvious ("out", for example, from LambdaMOO's coat closet and linen closet), and some are obscure ("vent" from LambdaMOO's kitchen

---

[8] On LambdaMOO, there is an atlas on the mantel in the Living Room (to use it, go to the Living Room, type take atlas from mantel, and examine atlas to see its commands). There is also a copy of it on the geography shelf in the library.

You can also add the Compass Rosette Feature Object (#23824) (see pages 41 and 224) and then type @rose. Obvious Features (#41975) has the command @lrs for "long-range-scan" which lists all the rooms within three exits of your current location.

will move you into the vent system from there). Though not the norm on LambdaMOO, some areas there and on other MOOs use non-compass exits primarily, and often the names of these exits are capitalized within the text of a room's description.

The `go` command lets you string multiple exits together. Typing:

```
go north west
```

is the same as typing `north` and *then* typing `west`. You can also abbreviate exits in combination with the `go` command. From LambdaMOO's living room, for example, you can type `go n w` and wind up in the dining room as if you had typed each exit name separately.

A later development was the concept of a room-within-a-room, and there are two major implementations of this idea: one is a *container room*, that – as you might expect – is a hybrid between a room and a container. Like other containers, you can open and close it, and can put things into it (including people!). In addition, since such a room is *inside* another room, instead of entering it via a conventional exit (i.e. an exit that you simply type the name of to use), you must enter the container room explicitly. The dishwasher in LambdaMOO's kitchen is an example of this kind of room. If no one is around to stuff you in, you can check out the inside by typing:

```
open dishwasher
enter dishwasher
```

Sometimes it doesn't make sense to open and close a room as if it were a container, and so someone developed a portable room that wasn't container-like. By convention, you go into such a room by using the `enter` command, e.g. `enter helicopter`.

In either case, with portable rooms or container rooms, you can leave by typing either `exit` or `out`. (Most of them will also let you type `look out` from within, to see what's outside.)

There are other ways of getting around that are said to be "consistent with the VR". These include elevators, trains, planes, paintings that transport you to the places they depict when you gaze at them, and so forth. Rooms that can be reached exclusively by conventional exits or in ways that are consistent with the VR are said to be *connected.*

Teleporting

Teleporting is a way of moving around that "breaks", "violates", or "bypasses" the virtual reality, which is another way of saying that it is a way of going from one room to another without using conventional exits or other VR means. The three most commonly used commands to do this are `@join`, `@go`, and `home`[9]. Notice that

[9] In the early days of LambdaMOO, teleportation in the form of `@join` and `@go` didn't exist. Morpheus made a generic ring of teleportation which functioned in a way somewhat similar to the way `@addroom` works today. Eventually everyone had one. At that time, quota was object-based rather than

two of these commands begin with the @-sign, a common signal that a command is not strictly VR.

Suppose you are in the LambdaMOO library and your friend Werebull pages you from the pool. He pages, “Come on down, we’re having a party!” Maybe you don’t want to take the trouble of going south to the corridor, west (four times) to the entrance hall, south to the living room, southeast to the deck, and then south to the pool deck. Or maybe you don’t know the way. At any rate, on this particular occasion, you just want to be there already and not take all the time and trouble of walking. You can type `@join Werebull` and voilà! You instantly join him wherever he is (in the swimming pool, in this case).

On another occasion, you might want to teleport to a particular room (regardless of who might already be there) without going to the trouble of walking, and for this there is the `@go` command. In this case, you will need to specify the room to which you wish to teleport, and this is one case where all those pesky object numbers beginning with the #-sign come into play. To teleport to a distant room, you will generally need to specify that room’s number. For example, to go to the foyer of the LambdaMOO Museum, you would type:

```
@go #50827
```

How do you find out a room’s number? Several ways. You can ask around. If it’s a popular room, many players will know the room’s number by heart and will be able to tell you. If you are in a room, you can find out its number for future reference by typing `exam here`. You might read a room’s object number in other written references to it, and it is for this reason that written references to rooms (and other objects of general interest, for that matter) usually include the object number.

Remembering Rooms’ Object Numbers

Since teleporting is so common, and since remembering all those object numbers is so cumbersome (for most people, anyway), there is a facility for you to associate the name of a room with its object number and store that information, and the `@go` command is able to use that list. The three commands for maintaining your list of rooms are `@rooms`, `@addroom`, and `@rmroom`. These commands are provided as part of LambdaCore.

Suppose you are exploring, find the foyer of the LambdaMOO museum, and think that you think you’d like to return to that location to from time to time. While there, you could type:

---

byte-based, which means that people were limited to a certain fixed *number* of objects (ten, I think it was), regardless of their size, rather than an indefinite number of objects limited by the total amount of storage they take up. Creating a ring of teleportation, then, used up a significant percentage of your building allotment. Since everyone had one, and since it was perceived as being expensive, someone lobbied to have the code added to a player class, instead. This was done. To “encourage” people to recycle their teleportation rings, a note was posted on the refrigerator telling people to be sure to wash their hands after a food fight. When a person did so, e lost eir ring down the drain and into the garbage disposal, with suitably appropriate sound effects. (With thanks to Doug (#3685) who shared this story with me in September, 1999.)

```
@addroom museum
```

The system will then add the name “museum”, paired with its object number, to your player object, and thereafter if you want to teleport there, you can type `@go museum` instead of `@go #50827`. The place where this information is stored is called a player’s `.rooms database`. (I usually pronounce the period as “dot”, so if I were speaking, I would say, “a player’s dot-rooms database.”) On most MOOs, guests and new players are provided with a preliminary list of rooms the wizards think they might want to teleport to by name, and players can add to this list as desired (using the aforementioned `@addroom` command). To see a list of the rooms you have specified so far, along with those that have been specified for you, type:

```
@rooms
```

To remove a room from your `.rooms database`, you can type:

```
@rmroom <name>
```

Finally, you can type `@go home` to teleport to your home.

You can also type:

```
home
```

on a line by itself to teleport to your home. This is an exception to the @-sign convention, for historical reasons (some MUDs only had one teleport command, and that was `home`).

As a point of etiquette, it can be awkward for all concerned if you teleport somewhere only to find that you’ve barged in on a private conversation or a tryst, and some players, even if they’re alone, will react negatively if you teleport to their location without first paging them to find out whether you might be welcome at that particular time. It is possible for players to lock rooms that they own in order to prevent unwanted or uninvited entry. It is the case, however, that players don’t always do this, yet still complain if someone teleports in. Private and public rooms are addressed briefly in the section on privacy that begins on page 31, but if in doubt, page before teleporting.

The Give and Take of a Multi-User Environment

One of the draws of a multi-user environment is that you get to meet and interact with other people whom you might otherwise never have encountered. And one of the drawbacks of a multi-user environment is that you may encounter people who not only do not embrace the same norms of behavior that you do, but whose behavior may be annoying at best. Different MOOs have different themes, aims, and local customs – thus different conventions of acceptable behavior. Some MOOs are family oriented, some are educational, some are Dungeons and Dragons games, some are professional, and some are explicitly “anything goes”. It would be presumptuous to assert that any particular way of behaving is generally acceptable or generally unacceptable. Many MOOs have documentation available like `help manners` or

`help rules`, and I would urge you to seek this document out on your MOO and acquaint yourself with its contents.

This section details a variety of defense mechanisms, built in to the MOO, that you can use to counter various annoyances. From these, one may infer that certain behaviors carry the risk that you may annoy others if you engage in them – two sides of the same coin.

Noise Abatement

@gag

There may come a time when you wish you could just "turn someone off", so to speak, and you can, after a fashion. The `@gag` command prevents you from seeing text generated by a specified player or object.[10] The syntax is:

```
@gag <player or object>
```

It remains in effect until you type:

```
@ungag <player or object>.
```

You can see a list of people and objects you are `@gagging` with the `@gaglist` command. `@gaglist all` will show you a list of people who are `@gagging` *you*, though on large systems that command may be slow to execute, because it has to look at each player in the database individually.

Sometimes it is just as effective, if you have the self-discipline for it, to ignore someone *as if* you were `@gagging` em even though you aren't. The principle, here, is not to reward obnoxious behavior with a reaction. (Whom you are `@gagging` is readable by others. The information is kept in a player's `.gaglist` property, and some players take umbrage at *being* `@gagged`, which is one situation where simulated gagging might be one way to go.)

The flip side of `@gag` is generating unwanted noise, either generally or directed at one person in particular. Typing in all uppercase letters is sometimes construed as shouting. A different definition of shouting is writing a program that broadcast's text to all connected players (or almost all – leaving out one or two people doesn't exactly exonerate you). *Spamming* refers to generating so much text that its sheer quantity is offensive regardless of its content. Spam can be more than just offensive – it can be disabling for another user who has a very slow communications link to the MOO. You might also be `@gagged` just for making a nuisance of yourself – it's the prerogative of the person hearing the noise to `@gag` you or not, as e sees fit. If you have been `@gagged` by someone and would like to ask em to reconsider, you might ask a mutual acquaintance to intervene on your behalf.

---

[10] On LambdaMOO, new players are created with a lag-reduction feature pre-installed, and must first type the command `@rmlag` before `@gag` will work.

@paranoid and @check-full

Sometimes it's hard to tell whom to `@gag`, because someone may cause *unattributed* text to be displayed to your screen. Unattributed text is text whose origin is unclear. Falsely attributed text is text that appears to have been generated by one person when in fact it was generated by someone else – this is called *spoofing*. The `@paranoid` command records not only the last <number> lines that have been displayed to your screen, but their origins as well. The syntax is any one of:

```
@paranoid
@paranoid off
@paranoid immediate
@paranoid <number>
```

If you see some baffling bit of text, you can then type `@check-full <text>` and the system will print out a trace from which you can usually make a fair guess as to who initially typed in the command.[11]

The sort of behavior that occasioned `@paranoid` and `@check-full` is producing unattributed or falsely attributed text, especially with the intention of confusing or deceiving others. For example, if Klaatu causes the text, "`Yib farts loudly,`" to be displayed, Yib would have cause to complain and Klaatu is probably in the wrong (depending, as always, on local custom).

@refuse

The `@refuse` command lets you refuse certain actions from all players or from a specified player. The syntax is `@refuse <action> [from <player>] [for <duration>]`. The parts in square brackets `[]` are optional. The actions that you can refuse are:

- `page` – prevent someone from paging you
- `whisper` – prevent someone from whispering a message to you
- `mail` – prevent someone from sending you a message via MOOmail
- `move` – prevent someone from teleporting you (Note, this can affect teleportation – @go, @join, etc. – because if you @refuse move without specifying a particular player, you refuse to be moved by *anyone*, including yourself!)
- `join` – prevent someone from entering the same room as you (only works in a few rooms that support this functionality)
- `accept` – prevent someone from handing you an object (or teleporting it to you)
- `flames` – posts from the refused player(s) are suppressed on mail lists
- `politics` – refuse programmatic campaign solicitations (LambdaMOO only)

---

[11] As with `@gag`, on LambdaMOO you must type `@rmlag` before being able to use this facility.

- all – all of the above

The flip side of @refuse is rather general, but you might deduce that moving someone without eir consent might result in an @refusal, as might sending em obnoxious mail, teleporting unwanted things into eir inventory, etc.

You can type @refusals to see what actions you are currently @refusing, and can type:

```
@unrefuse <action> from <player>
```

to cease refusing an action. As with your gaglist, your @refusals list is readable by others. To see someone else's refusals, type @refusals for <player>.

Privacy

Just because it's pseudonymous, doesn't mean that it's private, and in fact, privacy on a MOO is almost impossible to guarantee. People may not know your off-MOO identity, but they can and do seem to pay a surprising amount of attention to what other people are doing. If you have something truly sensitive to discuss, it really is better to take it off the MOO.

@lock

You can use the @lock command to prevent unexpected and/or unwanted entry into a room you own. The syntax is @lock here with <key> e.g., @lock here with me || <someone else>. You can specify as many people as you wish, separated by "||" which translates to "or" in the locking syntax. To unlock your room again, type @unlock here. Various room classes on different MOOs may provide more elaborate programming for ease of controlling access.

The flip side of locking and room security is joining people when you are uninvited, unannounced, or unwanted. There may be MOOs where there is a way to designate a room as public or private, but I am not acquainted with any, which means that in many cases you may have to guess, and/or learn from experience. It is probably a pretty good bet to guess that a room named "Yib's Room" is private, and that you might receive a less than enthusiastic welcome if you teleport in when Yib is in the middle of a private conversation with someone else. It might be argued that the onus is on Yib to lock her room if she doesn't want surprise visitors, but the reality is that people often forget to do this. I can't think of a situation where it would be *un*acceptable to page someone, first, and ask if you may join em.

@sweep

The `@sweep` command checks a room for potential listening devices, (any object that has a `:tell` verb on it). If you own the room, you might `@move` or `@eject` the unwelcome item. If you don't own the room, you might elect to have your conversation somewhere else.

The flip side of `@sweep` is bugging and/or recording conversations when your doing so is not obvious to the participants. Similarly, teleporting silently into the midst of a conversation such that the participants don't realize you're there can also land you in the dog house.

General Awareness

People seem to have a natural tendency towards nosiness. I am continually surprised by the many ways that people find, on MOOs, to look in on and keep track of others' doings. Being aware of some of these may help you make more informed choices about where, when and how you do or say things of a potentially sensitive or confidential nature.

- logging – Most client programs permit people to record some or all of their MOOing session with the option of saving it to a file for later review. It is not unusual for people to do this, typically for their own reference or review at a later time. In my experience, it's the exception rather than the rule for logs to be published without permission, but you should be aware that *anything* you say or do in the presence of another *could* come back to haunt you one day.
- Anyone can tell your MOO age, i.e. how long it has been since you first connected.
- People can view your description even if they aren't in the same room with you.
- Anyone can `@audit` you and view your possessions.
- Whom you're `@gagging` or `@refusing` is publicly accessible.
- People can detect when you look at them. Some people have a message that broadcasts to the room when you look at them.
- It is possible to detect your checking on someone's connection status when you use `@who`.
- Feature Object owners can and do keep track of how often their various feature verbs are called, and theoretically could keep track of who calls them.
- Mailing list owners can detect when you `@read` or `@peek` at messages on a mailing list.
- Player class owners are in a position to snoop in a variety of ways, including intercepting pages that you send or receive, intercepting MOOmail that you send or receive, and listening to conversations. There is usually enough social pressure to keep player class owners from doing these things, but one should be aware of

what's theoretically possible, and understand that this is what's behind the admonition to trust the owner(s) of your player class parent and ancestors.

- Wizards can look at anything on the MOO, including properties and verbs that you have set to unreadable. Wizards can enter locked rooms. Wizards can view your forked tasks. They can (though traditionally do not) read your private MOOmail. Wizards have access to your registration email address(es) and all the site addresses from which you connect. Strictly speaking, if you don't trust the wizards of a particular MOO, then you shouldn't get an account there. It's an interesting paradox, then, that anyone gets an account anywhere, because there really isn't a practical way to assess a group of wizards' trustworthiness and integrity before the fact. As with many things in life, it's a calculated risk.

Theft and Trespass

Sometimes people steal things on a MOO, but in point of fact it is impossible to truly hide a thing's whereabouts. I generally think that complaints of people stealing things are overblown – most of the time you can just `@audit` yourself and `@move` an object you own back to where you want it to be. When you *can't* just `@move` an object you own to a location of your choosing, things get more interesting. Trespass is the opposite problem: It's possible for someone to move something into your inventory or a room you own and make it difficult to get rid of. (Imagine a pernicious "Kick me!" sign.)

@lock

In addition to using the @lock command to keep intruders out of a room, you can also `@lock` objects in place to prevent people from taking them.[12] See `help locking`.

Theft Prevention

Some player classes provide an option that prevents others from moving things out of your inventory. This sounds fine, but becomes problematic if you pick up something that I own, and then I can't move my own object back to where it belongs. (In my experience, this situation has usually turned out to be an oversight on the part of the player class author, and been corrected upon request.)

The way theft prevention works is that when an item leaves a player's inventory, a special verb on that player called `:exitfunc` is called, notifying the player object

[12] On LambdaMOO you can ask the housekeeper to return an item to a specified place when it is no longer in use. This technology enables people to borrow things without your having to keep constant track of them manually and repeatedly put them back where they belong, and is a nice alternative to locking things in place.

that the item is leaving. A theft-preventing :exitfunc verb would fork a task and move the object right back again. A civic-minded :exitfunc verb would probably check to see if the exit was initiated by the object's owner (and/or a legitimate housekeeping task) and, if so, permit the item to be moved.

The flip side of @lock and programmatic theft prevention is taking stuff that doesn't belong to you. Sometimes you just have to guess as to whether an item is one that anyone is welcome to borrow or one that someone will miss if you take it, but be advised that this can be an issue that annoys people.

@eject

This command is for getting rid of things that are unresponsive to drop and/or @move. The syntax is `@eject <item> from <location>`, and you have to own `<location>` for it to work. (Typical usage would be `@eject <item> from me` or `@eject <item> from here`.) Successively forceful versions of this are @eject! and @eject!!, which provide the item being moved progressively less information about its being moved, thus giving it less opportunity to move itself right back again. If an item continues returning and `<location>` is a room, you might need to @lock your room against the item (`@lock <room> with !<item>`) and *then* @eject it.

@ban

This verb prevents a designated player or item from entering *any* room you own, as opposed to @lock, which only works on one individual room at a time.

@spurn

This command works like @ban, except it prevents a designated item from entering or re-entering your inventory. With a particularly pernicious object, it might be necessary to @spurn it before using @eject. (@Spurn is a relatively recent addition to the anti-trespass verbs available. MOOs based on a LambdaCore dated prior to 1998 probably don't have it.)

The flip side of @eject, @ban, and @spurn is trespass, either in person or with an object. Realize that if a player doesn't want to carry an object of yours or have it in a room e owns, then you are on risky territory if you try to impose your presence or program an object's "stickiness".

Sexual Advances

Some people who MOO like to engage in conversations with overt sexual content, either to shock people in public rooms, or privately, in order to become sexually aroused. If you aren't expecting it, receiving a sexually explicit page can be very disconcerting. So, first, just be mindful that such an event might occur. Second, be aware that most people who ask for MOOsex are looking for a *consenting* partner. A politely paged, "Not interested, thanks anyway," is enough to deter the vast majority of people cruising for action. If someone persists despite your declining politely, your best course of action is probably to utilize the noise abatement procedures detailed above. If you do decide to accept someone's invitation, be prudent: In particular, consider very carefully before giving anyone any real-life information about yourself.

If you are the person looking for a sexual conversation or encounter, you should be prepared (and willing) to take "No" for an answer, and be aware that not everyone is amenable to being approached in this way. I recommend politeness first, lasciviousness later.

Harassment

Different people have different thresholds of annoyance beyond which they feel harassed. Each MOO has its own policies and procedures for dealing with harassment or alleged harassment. On some MOOs, one is advised to contact a wizard. On other MOOs, the wizards explicitly state that they expect players to manage on their own. It is an inconvenient fact that there *are* people who seem to derive enjoyment from annoying others. If you are on a MOO whose wizards encourage you to notify them if someone is harassing you, by all means contact them. If you are on a MOO whose wizards have a more hands-off policy, understand that no amount of programmatic effort can prevent every instance of harassment, particularly a perpetrator's first attempt to harass a particular target. While an assault is never the victim's fault, there are some things that you can do to reduce the chances of being selected as a target, and to minimize the likelihood of repeated occurrences.

- Try asking the person to desist. Sometimes supposed "harassment" is truly unintentional, and it's a shame to escalate something that's just a simple misunderstanding.
- Realize that perpetrators want you to react. This is why they attack people and not inanimate objects. The more distress you show, the more you reward them for their offensive behavior. This is not to say that you shouldn't be upset, but that a flamboyant display of your distress to the perpetrator is likely to generate more unwanted attention from em. The calmer you can manage to seem in the face of a textual attack, the sooner the perpetrator will give up and go pick on someone else. The most effective response is no response at all, as if you had already `@gagged` whoever or whatever was generating the unwelcome text.
- If you feel panicked, `@quit` immediately. This is equivalent to simply hanging up the telephone if you receive an obscene phone call. Letting yourself be hounded

off the MOO may be unpalatable, but in most cases it is effective. In other instances, it may be sufficient simply to relocate to another room.

- Check `@who` *before* you connect. (Most MOOS permit this, though some do not). Return when the person bothering you isn't connected emself, then `@gag`, `@ban`, and `@refuse all from <em> for <several> years`.
- Do everything you can to rise above it, stay out of the fray, and move on. As my mother used to say, "Don't get in a pissing contest with a skunk."

Outing

MOOs vary in their privacy policies, but a good rule of thumb is that if a MOO does not in some way programmatically identify the typists behind the characters (e.g. by incorporating site information into guest descriptions or providing an automated registry of players' email addresses), then it is impolite to disclose information about another player's offline identity or particulars without that person's permission. Unfortunately, there is no program one could write to prevent this from occurring. Your best defense, as always, is to keep private information private, and not disclose it to others.

Summary

MOOers vary in their cultural and behavioral norms and expectations, and MOOs vary in the kinds of interactions that are encouraged and tolerated. Getting along is something of a balancing act. There are few hard and fast rules, but it's hard to go wrong if you treat others with respect, then learn a few techniques to cope effectively with the few troublemakers that are out there. Good luck.

Chapter 3 – What's Going On, Here?

Objects

MOO stands for "MUD, Object-Oriented"[13]. This section discusses in depth what an object actually is.

Objects are the building blocks of a MOO. They are things; in a sense, they are nouns: A noun is a word that represents a person, place, or thing; an object is a data construct within the computer program that is the MOO (i.e. the server) that represents a person, place, or thing. Some nouns represent concrete things, such as chairs, cats, and candy, while others represent intangible things, such as news, knowledge, and abilities. Likewise, some objects represent concrete things within the MOO (chairs, cats, candy) while other objects represent intangible things (news, knowledge, and abilities). But intangible things are still things, and therein lies the nature of an object.

Every object in a MOO is assigned a number upon creation. This number is unique within the MOO and immutable. If absolutely every characteristic of an object were changed – its name, its owner, its location, its description – its number would still be the same.

There are certain pieces of information that are attached to every valid object without exception. (An object that has a number but doesn't have these pieces of information associated with it is an invalid object, by definition.) These pieces of information include the object's owner (itself identified by object number), its location (identified by object number), its contents (a list of one or more object numbers), and its parent (identified by object number).

Object parenthood is a special concept. An (imperfect) analogy is the taxonomy of animals. There are animals, and then there are vertebrates and invertebrates, and there are mammals and reptiles and insects, and there are felines and canines, and there are tigers, and then there's this tiger that happens to have a litter of adorable tiger kittens, among which is a particular tiger kitten whose name is "Stripes". The structure (which is also called the *object hierarchy*) is like a family tree (or a root system), where everything is descended from a thing (an object) that came before it. The root object (with a unique number: #1) is unusual in that it has no parent.

Some objects are used so often that the system provides a way to refer to them by name instead of by object number, and we use a $-sign to designate those. Some especially common ones are `$thing`, `$container`, `$room`, and `$note`. But each of them also has a unique object number on the MOO (#5, #8, #3, and #9 respectively, on LambdaMOO).

---

[13] "MUD" has come to stand for "Multi-User Domain", though it's original meaning was "Multi-User Dungeon". MUDs have their roots in a role-playing game called "Dungeons and Dragons" and in some single-user computer games along the same lines that were popular in the 1970's and 1980's, notably *Adventure* and *Zork*.

When you create an object, you begin by specifying its parent and its name, for example:

```
@create $thing named "rock"
```

The system will respond with something like:

```
You now have rock with object number #1614 and parent
generic thing (#5).
```

And your rock will have all the attributes and characteristics of the generic thing that is its parent, until such time as you modify it, for example by giving it a description, or maybe by programming it to behave in some rock-like way.

You can see a list of all the objects you own by typing:

```
@audit me
```

You can see a list of objects Yib owns by typing `@audit Yib`. When you examine an object, you are told its number, among other things. You can check your parent object and its parent object(s) (i.e., its ancestors) by typing `@parents me`, and you can check the parents of any object by typing `@parents <object>`.

When must you refer to an object by its number, and when can you just use its name? In general, if you are holding an object, or are in the same room as an object, then you can refer to it by its name. If you are at some remove from an object (i.e. neither holding it and nor in the same room) then you generally have to refer to that object by its number in order for the system to know which object you mean. There are some exceptions: Objects that can be designated with the $-sign plus a name ($thing, etc.). Mailing lists, which begin with the asterisk symbol. Some commands let you specify a player by name even if you aren't in close proximity (`@who Klaatu`). And often you can specify a distant player by name with the tilde (~):

```
look ~yib
```

will yield the same result as:

```
look #58337
```

(if #58337 is Yib's object number).

On a MOO, everybody who's anybody, and everything that's anything, is an object.

See also `help objects`.

Moving Objects

Chapter 2 included a discussion of how to move (yourself) around the MOO. This section discusses moving other objects from one place to another.

A little about what it *means* to move an object: The system keeps meticulous track of every object's location. Specifically, every object has a property (a named piece of data) which stores that object's location in the form of an object number. An object in the LambdaMOO Living Room, say, would have the Living Room's

object number in its `.location` property. (When naming a property, it is conventional to precede it with the "`.`" character.) Reciprocally, while that object is in the LambdaMOO Living Room, that object's number will appear in the Living Room's `.contents` property. So: every object stores its location, and every location stores a list of its contents. When an object is successfully moved in a MOO, three things happen: The object's `.location` property is changed to reflect the object's new location. The object is removed from the old location's list of contents. The object is added to the new location's list of contents.

The most straightforward way to move an object is to take it or drop it.

Suppose I own an object named `bright sparkly thing`, but leave it lying about in the driveway where anyone can find it. Shmool comes along and sees:

```
Driveway
A circular driveway, in front of LambdaHouse.  The
LambdaHouse front door is to the south.  The drive curves
away to the northeast and northwest; there is a spur to the
west, curving back around the house to the garage.
You see bright sparkly thing here.
```

Shmool types:

```
take bright sparkly thing
```

Shmool now has the bright sparkly thing in his *inventory* (the list of stuff he's carrying), and anyone looking at Shmool will see not only his description, but his inventory as well:

```
l Shmool

Shmool
A 3 1/2 foot tall squirrel, bald but for a 3 foot ponytail,
with large and luminous violet eyes.  A silver locket
dangles by a gossamer chain around his neck.
Carrying:
  bright sparkly thing
```

When Shmool took the bright sparkly thing, he changed the database, meaning the `.location` of the bright sparkly thing changed, the `.contents` of the driveway changed, and the `.contents` of Shmool changed. Note that if Shmool had merely emoted,`:takes bright sparkly thing`, on the other hand, while it might *appear* that he had taken it, it would in fact still be lying in the driveway.

Shmool might then type:

```
@go home
drop bright sparkly thing
```

to add it to his growing collection of stuff. The bright sparkly thing would be removed from Shmool's inventory and added to the contents of his room. If Shmool keeps this up, his room will become quite cluttered! He might be moved to create a treasure chest to put things in. Putting things into containers is another way to move things. Shmool would type:

```
put bright sparkly thing in treasure chest
```

and the bright sparkly thing would be moved again: Its new location would be the chest, and the contents of Shmool's room and his treasure chest would be changed appropriately, too.

Eventually I notice that my bright sparkly thing is missing. Where could it be? I look everywhere for it. Last seen in the driveway! I go to the driveway, but the bright sparkly thing is gone. Some rascal has taken it!

I might search high and low, and on a MOO as big as LambdaMOO, or even on a much smaller one, I might never find my object. If I cared to get my object back, this is an occasion where I might choose to break the VR and start working with object numbers. Specifically, I might want to teleport my object back to a location of my choosing.

Remember that if you aren't holding an object or in the same room with it (my predicament), then you must identify it by its number. For a while, I just ignored all those object numbers, and got along fine without them, but now I would *really* like to know the object number of my bright sparkly thing! There is a way, by using the `@audit` command. I type:

```
@audit
```

and the system prints out a list of objects (by number!) that I own, along with their size, name, and location. Now I have the information necessary to teleport my bright sparkly thing, either to myself or to my location or to a named object in my vicinity or to a location whose number I know (aha, those numbers, again), using the `@move` command. The syntax is:

```
@move <object> to <location>
```

Here are some of my choices, assuming that the object number of my bright sparkly thing is `#4612` and that I am in a room with a walnut desk that I use in lieu of a treasure chest:

```
@move #4612 to here
@move #4612 to me
@move #4612 to a walnut desk
@move #4612 to #6193
```

(On LambdaMOO, the last example would move my bright sparkly thing back to the driveway.)

Getting Rid of Unwanted Objects

You may find yourself in the strange predicament that you are carrying something, or something is attached to you, or something is in a room you own which you would like to be rid of but which doesn't respond to the `@move` command. For this we have `@eject`. The syntax is:

```
@eject <item> from <location>
```

If you're holding it, then `<location>` would be `me`; if you want to eject a thing from a room you own, then `<location>` would be `here`.

Information is power. In this case, if you know the number of a thing, you can teleport that thing. There are some exceptions. (Aren't there always?) An owner can lock a thing in place, preventing people from taking it or otherwise moving it. An owner can put conditions on moving an object. The owner of a room can prevent an object from being dropped or moved there. (See locking, page 88) Lastly, on LambdaMOO, the owner of an object can let people borrow it, and have the housekeeper return it to a designated location when certain conditions are met.

Taking things that don't belong to you falls into a category I call, "risky behavior". It isn't strictly forbidden, and if you know how, it's easy to undo. But it also annoys some people, so think before you take.

Feature Objects

This section discusses Feature Objects, including an explanation of just what sort of objects they are and how they work.

When you are connected to a MOO, you type things in and read text that appears on your screen. The lines you type (except when you are being prompted for data, for example when you are working within an editor) are *commands*. When you type a command, you expect something to happen, either a change to the database (e.g. changing your location) or perhaps simply the display of some informational text from the database (e.g. looking at a room or a player). MOOs are user-extensible, which means that users can create objects and can define (i.e. program) commands associated with those objects, which you and others can then use. It is part of the server's job to consider the command you type in and divide it into its component parts (command name plus optional arguments or direct object, preposition and indirect object). The process of separating a command line into its component parts is called *parsing*, and the parser is the part of the server that does this. Once the command has been divided into its component parts, the server then tries to identify an object that defines the command (verb) that you want to run. It conducts its quest for an object defining the verb in a very particular order, as follows: (1) your player object or player class or any of its ancestors, (2) any of a player's feature objects, (3) the room you are in, (4) the direct object (if specified and identifiable), and (5) the indirect object (if present). If the parser finds that the command is valid (i.e. defined on one of player, a feature object, the room, the direct object or the indirect object), then the server tries to execute it. If the parser can't find any definition of the command on any of the objects it is supposed to consider, then the system prints the text, `I don't understand that.`

A feature object (commonly referred to as a/an FO) is an object that exists solely to serve as a repository for a set of commands that you might want to use, so that when you type the command, the system executes it rather than displaying everyone's favorite, `I don't understand that.` The beauty of feature objects is that for commands that lend themselves to this method implementation (some

commands don't), anyone can use them *regardless of what player class e has selected.* This is why feature objects tend to have broad appeal.

Unlike objects that represent tangible things, such as chairs, candy, or rocks, you don't need to be holding an FO or in the same room with it in order to use it. Rather, one *adds* a feature to oneself, which is another way of saying that one adds its number to a list of feature objects one wishes to be able to use. To add a feature, you have to know its object number. To find out a feature's object number, you can try asking the person who just used it, or you can type:

```
@features for <so-and-so>
```

to see a list of that person's feature objects and their numbers. Then you can add it by typing:

```
@addfeature <object-number> to me
```

So, if you notice that mockturtle is a thoughtful guy, and in particular much of his text appears in typographical thought balloons instead of between double quote marks:

```
mockturtle . o O ( Can she read my toughts? )
```

You might say, "mockturtle, is that thought balloon verb on an FO?" And mockturtle might quickly type `@features for me` to jog his memory and then say, "Yes, it's the 'think' verb on the Thinking FO, #10392." And then you might type `@addfeature #10392` (and become more contemplative, yourself).

It's possible to use a feature object command quite often and forget which feature object it's actually on. Then when someone asks you what FO a verb is on, you might look at your list of features and still not know which one has the command in question. In such a case you might take advantage of a verb called `@find`. So instead of listing his feature objects, mockturtle might instead type:

```
@find :think
```

(Note that the colon is part of the command), and the server would print on his screen, `The verb :think is on Thinking Feature(#10392).`

Most feature objects have help text (e.g. `help #10392`) that list the commands they offer and explain briefly what they do and how to use them.

On LambdaMOO, there is an exhibit in the museum dedicated to feature objects, where you can read each one's help text and then pick and choose the ones you want. On other MOOs, you can type `@kids $feature` to see a list of all direct children of the generic feature object, and then read the help text for those feature objects that look like they might be of interest. It might be tempting to add all the feature objects you can find, but remember: When parsing a command, the server must consider all the verbs on you, your player class, its ancestors, *all your feature objects*, the room you're in, and possibly direct and indirect objects. The more feature objects you add to yourself, the longer this process takes. It's better to read the help text for various features and then add only those features that you think you'll actually use.

The sequence in which you add feature objects can matter. If two feature objects define the same command, the first one in your list of features is the one that will be

executed. To change the order of your features, use the `@rmfeature` command to remove the one that comes first, then re-add it to move it to the end of the list.

Player Classes

A *player class* is an object that provides or expands a set of commands available for a player to use. To use a player class, a player changes eir parent to that player class object, thus inheriting all its properties and verbs and all its ancestors' properties and verbs.

Recall that every valid object (except #1, the root object) has another object as its parent. Players can change the parent of objects they own, *including themselves*, with the `@chparent` command:

```
@chparent <object> to <new-parent-object>
```

The list of an object's parents and ancestors is called its *parent hierarchy*. You can look at an object's parent hierarchy using the `@parents` command. The syntax is `@parents <object>`. To see a list of your own parent and ancestors, type:

```
@parents me
```

(If the system responds with `I don't understand that`, ask a wizard to make you a builder.)

Someone who has a fairly fancy player class (in this case on LambdaMOO) might have a parent hierarchy that looks like this:

```
Yib(#58337)
Sick's Sick Player Class(#49900)
Sick's Slightly Sick Player Class(#40099)
Sick's Sick of Spam player class(#59900)
Detailed Player Class(#6669)
Generic Super_Huh Player(#26026)
Politically Correct Featureful Player Class Created Because
Nobody Would @Copy Verbs To 8855(#33337)
Player Class hacked with eval that does substitutions and
assorted stuff(#8855)
Experimental Guinea Pig Class with Even More Features of
Dubious Utility(#5803)
Generic Player Class With Additional Features of Dubious
Utility(#7069)
generic programmer(#217)
generic builder(#630)
Generic LambdaMOO Citizen(#322)
Frand's player class(#3133)
Generic Mail Receiving Player(#100068)
generic player(#6)
Root Class(#1)
```

A brand new player on LambdaMOO would have a parent hierarchy that looks like this:

```
Bit_Blaster(#200119)
generic builder(#630)
Generic LambdaMOO Citizen(#322)
Frand's player class(#3133)
Generic Mail Receiving Player(#100068)
generic player(#6)
Root Class(#1)
```

A brand new player on another MOO would likely have a parent hierarchy that looks like this:

```
New_Grrl_On_The_Block(#1438)
generic builder(#4)
Frand's player class(#90)
Generic Mail Receiving Player(#40)
generic player(#6)
Root Class(#1)
```

(Note, however, that this may differ, as the wizards of each MOO can change the default starting player class for new players.)

In principle, it is vitally important to understand the workings of a player class you adopt, and to trust its owner and the owners of its ancestors. You can use the `examine` command to see an object's owner; many MOO's also provide a specific command to do this, e.g. `@owner`. At the very least, you should be aware that the owners of your player class and its ancestors are in a position to intercept and monitor your pages, intercept and read your private MOOmail, change your name and/or remove or change your aliases, and any number of other acts that you might normally expect to be your prerogative alone.

In practice, people often select a player class based on the recommendation of friends or experienced acquaintances, and trust the various player class authors by reputation. The purist in me would like to say that one should acquaint oneself with all the possible player class alternatives, read the help text and the verbs of each, understand what each does, and then select a player class based on the trustworthiness of the authors, and which is no fancier than one's particular needs at the moment. (The more elaborate the player class, the more quota your player-object takes up.) Note that it is trivial to change your parent to a fancier descendent of your current player class. It is less trivial to change to a different branch of the player class "tree", because doing so can mean having to give up messages or morphs (see glossary) in which one may have invested a fair bit of time and creativity.

There is generally sufficient social pressure on player class owner/authors not to spy or otherwise abuse their privileged position that it isn't a common problem. There was an incident several years ago on LambdaMOO in which the author of a popular player class was accused of intercepting pages which his girlfriend received from a perceived rival. There was an outcry when this was discovered, and the code was revised to the satisfaction of all concerned. In another instance, a player dared all comers to do their worst (the point being to demonstrate that in the face of any

attack one could still MOO serenely without needing an arbitration system). The owner of one of this person's player class ancestors removed the challenger's name and aliases and changed them to something highly unflattering. The names and aliases were subsequently returned to their original owner. These instances are rare, but one should still be advised that adopting a player class entails a certain degree of calculated risk.

One should be especially wary if a player class has unreadable verbs. Type:

```
@display <player-class-object>:
```

to list the verbs. (The colon is part of the command.)

Knowing the commands available to you will give you better use of them. For more information about a particular command, try either of:

```
help me:<command>
help <command>
```

(Help text had not been standardized at the time these early player classes were written.)

Documentation for the oldest player classes that nearly every player on every MOO has in common is sometimes non-existent or difficult to find; here is a brief summary of what these basic player classes do. For a detailed explanation of any of these commands, see the command summary in Appendix A

- **Generic Player**: Provides the most basic set of commands, including `home`, `help`, `@describe`, `@gender`, `@quit`, `@password`, and `@mail-options`.
- **Generic Mail Receiving Player**: With the exception of `@mail-options` which is (incongruously) provided on the generic player, this player class provides all the commands for reading and sending MOOmail, including `@mail`, `@send`, `@read`, `@next`, and `@subscribe`.
- **Frand's Player Class**: Frand is one of LambdaMOO's oldest and most venerable and innovative players. So much so, in fact, that this player class is now included in LambdaCore itself, and has been integrated into JHCore. Many of the verbs on Frand's player class can be said to "break" or "transcend" the VR, including `@go`, `@join`, `@addroom`, `@rooms`, `@ways`, and `@refuse`.
- **Generic Builder**: This player class provides the minimum set of commands needed to add objects to the MOO's database, e.g. build rooms. They include `@create`, `@dig`, `@recycle`, `@chparent`, `@audit`, `@parents`, `@lock`, and `@contents`.
- **Generic Programmer**: In order to write programs on a MOO, you must have this player class in your ancestry and have gotten a programmer bit (the usual procedure for getting one is to ask a wizard). The programmer bit gives you the authority to program; the generic programmer player class gives you the necessary commands to do so. These commands include: `@property`, `@verb`, `eval`, `@program`, `@display` and `@dump`.

Setting Messages

```
Yib arrives in a shower of sparks.
```

Q: How does she *do* that?

A: With messages.

Recall that a *property* is a named piece of data associated with an object. A *message* is a special kind of property whose name ends with `_msg`. The purpose of messages is to give users a way to customize themselves and objects they own without having to learn how to program. Many kinds of objects have messages.

To see a list of the messages on an object, type:

```
@messages <object>
```

Player objects have a great many messages, and each player class in a player's ancestry typically adds a few more, so typing:

```
@messages me
```

will probably generate a long list! Let's look at two player messages:

```
@self_arrive #3133 is "%<teleports> in."
@oself_port #3133 is "%<teleports> out."
```

If you transport yourself within the MOO using either the `@go` or `@join` command, the system will process these messages (prepending your name (if it is absent) and conjugating the verb "teleport") and then display them to the appropriate viewers – at the location you are leaving and at your destination. To customize my departures and arrivals, then, I would change these messages. There are two syntaxes for doing so:

```
@set me.self_arrive_msg to "Yib arrives in a shower of sparks."
@set me.oself_port_msg to "Yib disappears in a sudden puff
of smoke."
```

or:

```
@self_arrive me is "Yib arrives in a shower of sparks."
@oself_port me is "Yib disappears in a sudden puff of
smoke."
```

Because *silent teleporting* (sneaking into a room without people knowing you are there) is frowned upon in most MOOs, the system displays your name at the beginning of these messages if it isn't otherwise present. Furthermore, because some players may have their gender set to plural, the default messages use the syntax `%<teleports>` to signal that "teleports" is a verb that should be conjugated.

The messages on different player classes (including the ancestors of *your* player class!) were created by different people at different times, and many of them were created before some now-common conventions were established. Unfortunately, many of them aren't documented, or are documented poorly. Historically, people figured out and set most of their own messages by listing them (`@messages me`),

looking at the combination of their names and content, and (often) trying them out with a friend.

Here, then, is a list of all the messages that you are likely have defined on your player object as a new arrival on a MOO that uses LambdaCore, along with a few comments about some of the messages' idiosyncrasies. They are presented as if you had typed `@messages me`, and the syntax for setting them is the same as that displayed by the system when it lists them.

```
@more me is "*** MORE ***  %n lines left.  Do @more
[rest|flush] for more."
```

`@more` is the message displayed if you have used the `@pagelen` command to set a fixed page length. The syntax `[rest|flush]` means that you have the option of typing `@more rest` or `@more flush` to print all of the remaining output, rather than just one additional page. Type `@more` by itself to see just the next page.

```
@page_absent me is "%N is not currently logged in."
```

`@page_absent` is the message that someone sees if e pages you when you are logged off. In this message, your name will be substituted for `%N`.

```
@page_origin me is "You sense that %N is looking for you in
%l."
```

`@page_origin` is the message that someone sees as when you page em, before the actual content of your paged text is displayed. Your name is substituted for `%N` and the name of your location is substituted for `%l`.

```
@page_echo me is "Your message has been sent."
```

`@page_echo` is the acknowledgement message that someone sees when e pages you.

```
@join me is "You join %n."
```

This message is displayed to you when you use the `@join` command to teleport to another player's location. The name of the player you are joining is substituted for `%N`.

```
@object_port me is "teleports you."
```

This message is transmitted to an object it when you teleport (i.e. `@move`) it. (See the section on moving objects, page 38.) Your name (followed by a space) is automatically added at the beginning.

```
@victim_port me is "teleports you."
```

This message is displayed to a player if you teleport em somewhere. Your name (followed by a space) is automatically added at the beginning.

```
@thing_arrive me is "%T teleports %n in."
```

This message is displayed to your location when you teleport an object either to yourself or to the room you are in. Your name is substituted for `%T` and the name of the object being teleported is substituted for `%n`.

@othing_port me is "%T teleports %n out."
This message is displayed to the room a thing is in (if it is in a room) when you teleport that thing to a different location. Your name is substituted for %T and the name of the object being teleported is substituted for %n.

@thing_port me is "You teleport %n."
This message is displayed to you when you teleport an object. The name of the object you are teleporting is substituted for %n.

@player_arrive me is "%T teleports %n in."
If you teleport (i.e. @move) a player, this message is displayed to the room to which that player is moved. Your named is substituted for %T and the name of the player being teleported is substituted for %n.

@oplayer_port me is "%T teleports %n out."
If you teleport (i.e. @move) a player, this message is displayed to the room from which that player is removed. Your named is substituted for %T and the name of the player being teleported is substituted for %n.

@player_port me is "You teleport %n."
This message is displayed to you when you teleport (i.e. @move) a player. The name of the player you are moving is substituted for %n.

@self_arrive me is "%<teleports> in."
This message is displayed to your destination when you teleport somewhere (e.g. with @join or @go). Notice that it is not a complete sentence. Your name is automatically prepended to the beginning of the message if it doesn't appear somewhere else within it. Setting this message to the empty string ("") will result in a silent arrival, which is contrary to good manners, and in some places might be construed as a form of spying. The construct %<teleports> causes the verb "teleport" to be conjugated according to your .gender property.

@oself_port me is "%<teleports> out."
This message is displayed to your point of departure when you teleport out. Again, your name is prepended if it doesn't appear somewhere else in the message. Setting this message to the empty string will result in a silent departure. While not considered as bad as a silent arrival, it can be unsettling to others in the room not to know that you have left. . The construct %<teleports> causes the verb "teleport" to be conjugated according to your .gender property.

@self_port me is ""
This message is displayed to you when you teleport somewhere. It is blank by default.

@page_refused me is "%N refuses your page."
This message is displayed to someone if e tries to page you but you have refused pages from em. (See page 30) Your name is substituted for %N.

```
@whisper_refused me is "%N refuses your whisper."
```

This message is displayed to someone if e tries to whisper to you and you have refused whispers from em. Your name is substituted for `%N`.

```
@mail_refused me is "%N refuses your mail."
```

This message is displayed to someone if e tries to send you mail and you have refused mail from em. Your name is substituted for `%N`.

Chapter 4 – Using the Mail System and the Editors

Reading Mail

This section explains how to read mail, how to examine and set mail options that pertain to reading mail, and *message sequencing*, which is a way of specifying that you wish to read or see a list of mail messages that meet certain criteria that you specify. Sending mail is addressed in the section that begins on page 57.

```
You have new mail (1 message).  Type 'help mail' for info on
reading it.
```

The simplest way to read mail that someone has sent you is to type:

```
@next on me
```

and to keep doing that until the system responds, `You have no next message.`

The MOO supports a generalized concept of a *mail recipient*, which is any object that can receive mail. The two categories of mail recipient are players and mailing lists. Each stores a collection of mail messages, and the commands for reading mail on yourself and on a mailing list are the same. Mailing list names are preceded by the asterisk character (`*`). In offline discussions, mailing lists are sometimes referred to as "star-lists".

When you log on, the MOO will tell you how many new messages you have (if any), and how many new messages are on *lists that you subscribe to. This implies two things: one, that there is a concept of subscribing to *lists, and two, that the system keeps track of which messages on a *list you've read and which you haven't. This information – what lists you're subscribed to and how far you've read in them – is stored on your player object in the `.current_message` property. Only you and wizards can examine this property, which means that you have some privacy concerning what *lists you choose to read. However, *lists themselves can record the fact that you read messages on them (or even @peek at messages on them), and that information is then available to the owner of the *list if e chooses to access it.

To start reading mail on a *list, you use the same command described above for reading mail on yourself, except that instead of typing `@next on me`, you'll type:

```
@next on <*list>
```

Note that reading any mail message on a list subscribes you to it. If you want to read all or part of a *list without subscribing to it, use the `@peek` command (see page 55).

When you first get your MOO character, you will not be subscribed to any lists. To see what mailing lists exist that you might subscribe to, type:

```
@unsubscribed
```
[14]

or:

```
@subscribe
```

To subscribe to a *list, type:

```
@subscribe <*list>
```

for example, `@subscribe *News`. Or read a message on it. To read the first message on a list to which you aren't subscribed, type:

```
@next on *list
```

To see what mailing lists you're currently subscribed to, type:

```
@subscribed
```

You can get a sense of a *list's intended topic by reading its description. The system is able to recognize mailing lists by the asterisk, so you don't need to know a list's number to do this:

```
look *News
look *Chatter
```

If you've been logged on for a while and want to see if there are any new posts on lists you're subscribed to, you can type:

```
@rn
```

An extremely useful command is:

```
@nn
```

(Think "next new".) This command will read the next new message on yourself or on any *list you are subscribed to. Unlike `@next`, if it comes to the last new message on one list, it will start with the first new message on your next list, and so on until all new messages on all lists have been read. An alternative, if you want to read all your new mail in one shot, is the command:

```
@ranm
```

(Think "read all new mail".) This will display the contents of all new messages on all your *lists non-stop. At the end, you will be presented with a `yes/no` prompt asking whether you got it all. If you didn't (for example, if you were disconnected in the middle) the system won't update the list of messages you've read, so that the same ones will still appear as new/unread messages when you next log on.

Now let's take a more detailed look at the `@next` command, along with its counterpart, `@prev`, combining them with the concept of your *current folder* and your

[14] On LambdaMOO this is *very long*. You might want to use a logging facility (this would be provided by a client program) or, alternatively, a command called `@netsend-mail-catalog` on feature object #27325.

current message on each folder. A *folder* is another name for a mail recipient, either yourself or a *list. (Think of a folder with a set of messages in it.)

If you type `@next` or `@prev` with no other arguments (specifiers), e.g,

```
@next
@prev
```

the system will print the next (or previous) message from your current folder, and will update its record of your current message on that folder. Suppose you have five old messages (on yourself), numbered 1-5, and five new messages, numbered 6-10. (Your current folder is and will always be yourself unless you change it, using the `@mail-options` command, detailed below.) Your current message is 5 (the last one you read). If you type `@next`, you will see message 6 on yourself, and your current message will be set to 6. If you type `@next` again, you will see message 7 on yourself, and your current message will be 7. If you type `@prev`, the system will print message 6, and will reset your current message to 6, and so on.

Now suppose you decide to read some messages on a *list. For the sake of illustration, we'll call it *chatter. You would type:

```
@next on *chatter
```

and the system would print the next message on `*chatter`, where "next" means "the next message that you haven't read yet", or, to put it another way, the next message after your current message on that folder. Note that your next message on that folder might be different from my next message on that folder, because you and I might not have read up to the same point. To continue reading messages on `*chatter`, you would type `@next on *chatter` repeatedly until you either came to the end of the list or had read as many messages as you wanted to.

You can also use `@next` and `@prev` to read more than one message at a time, either on yourself or on another list:

```
@next 3
@next 3 on *chatter
@next <number> on <*list>
```

You can also tell the system to treat the folder on which you most recently read a message as your current folder, instead of your current folder always being yourself. To do this, you would set one of your mail options using the `@mail-options` package. Options packages are simply groups of settings that you can use to customize your MOOing experience so that certain system behaviors are more to your liking. Suppose that you prefer not to have to type `@next on *chatter` each time, but would rather access the next message on that folder with a plain `@next` command with no arguments. To do this, type:

```
@mail-option +sticky
```

This makes the pointer to your current folder "stick" to the folder you last read a message on, rather than always reverting to yourself, and all mail commands will apply to that folder until you specify a different one. Some people find this more convenient. If you try this option and don't care for it, you can reverse it by typing:

```
@mail-option -sticky
```

Besides reading messages, you can also list message headers. A message header consists of a message number, the date it was sent, who sent it, and either its subject heading or the first few words of the first line (if there is no subject heading). One reason for listing message headers is to take advantage of the fact that you are not stuck just reading the next message on every list you're subscribed to. Rather, you can read messages selectively, either singly or in groups, if you so choose. Of course, you can list message headers selectively, too. This ability is a powerful tool when you want to search a list for a particular message, set of messages, or kind of message.

@mail

The `@mail` command shows you a list of messages, with summary headers. It can be used by itself, with no arguments, but it is an especially powerful searching tool when used with a specifier called a *message sequence*, which is a word or set of words that specifies a set of criteria that a message may or may not satisfy.

`@mail` typed by itself will show you the summary headers of the last 15 mail messages (you can change the number, with another mail-option) on your current mailing list. You could also type `@mail on *chatter` to peruse the last 15 posts on that mailing list, and so on.

Two typical uses for `@mail` are to preview messages on a list before reading them, or to try to identify a particular post that you want to re-read, respond to, or cite.

There are many useful message sequences that you can use to limit or shape the output you get. Here are a few examples:

`@mail new on me`	lists new messages for you
`@mail 43 on *think-tank`	lists message 43 on `*think-tank`
`@mail 43-53 on *think-tank`	lists messages 43-53 (inclusive) on `*think-tank`
`@mail next3 on me`	lists the next three messages on you (note, no colon or space between `next` and the number)
`@mail prev4 on me`	lists the previous 4 messages on you
`@mail first:5 on *theme`	lists the first 5 messages on `*theme`
`@mail last:6 on *B:Shutdown`	lists the last 6 messages on `*B:Shutdown`
`@mail 1-last on *newbies` `@mail 1-$ on *newbies`	lists all the messages on `*newbies` (The $ sign stands for "last".)
`@mail cur-$ on *geography`	lists messages headers from your current message through and including the last message on `*geography`

The above are based mainly on message numbers, your current folder, and your current message in a given folder. Here are some others that are even more flexible:

`@mail cur`	lists your current message on your current *list
`@mail from:Tower on *Research`	lists messages that Tower has sent to `*Research`
`@mail subj:infrastructure`	lists messages on your current *list with “infrastructure” in the subject heading
`@mail before:21-Mar`	lists messages on your current folder that were sent before March 21
`@mail until:30-Jun`	lists messages on your current folder that were sent on or before June 30
`@mail after:04-Jul`	lists messages on your current folder sent after July 4
`@mail since:31-Oct`	lists messages on your current folder sent on or after October 31
`@mail since:01-Dec body:unthemely on *Public-ARB`	lists messages on `*Public-ARB` posted since December 1 that have “unthemely” in the body of the message. (Note, body searches take a long time and should have an additional qualifier to narrow the search.)

If you do not specify a folder when using the `@mail` command, you will see the specified message headers on your current folder. If you do specify a folder (*list), you will see the specified message headers from that list (and if your mail options are set to +sticky, your current folder will be set to that *list).

See also `help mail` and `help message-sequences`.

@read, @peek

The `@read` and `@peek` commands display the full text of one or more messages rather than just the headers. They can take the same message sequence specifiers that the `@mail` command does. So, for example, you might do:

```
@read new on me
@peek last:5 on *think-tank
```

The differences between @read and @peek are these:

- @read subscribes you to the specified *list if you were not already subscribed. @peek does not.
- @read sets your current folder and current message to the last message read. @peek does not.
- @read suppresses the message if you have refused flames from the sender. @peek does not. (See help @refuse.)

@rmm

The @rmm command (think ReMove Message) removes one or more messages. It takes the same message sequence specifiers as @mail, @read and @peek. You can remove messages from yourself. You can remove messages that you have sent to a public *list (unless its owner has specified to the contrary) and you can remove messages from *lists that you own.

The preposition for the @rmm command is different: Whereas you @read <message-sequence> ON <folder>, you @rmm <message-sequence> FROM <folder>.

Messages that you have removed using the @rmm command aren't *really* gone until you either log out or @rmm some other messages, and you can "unremove" them with the @unrmm command. While they are in this indeterminate state, they are called *zombie messages.*

See also help @unrmmail and help zombie-messages.

@skip

Skip to the end of your current list. A good example of when you might choose to use this command is when you've been away from the MOO for a while and have a lots and lots of new messages on a *list. Some lists you may want to read all of, but with others you may just want to skip to the end without reading all the intervening messages.

@renumber

Some people like to renumber their messages from time to time, while others never do. When you remove a message (with the @rmm command), its number is not re-used. If you were to remove a message and subsequently list your messages, you would see a gap. Renumbering messages assigns new numbers to the messages (starting with 1) and removes the gap. So, if your current messages were numbered 1, 3, 5 and 17, and you typed @renumber, the same set of messages would now be

numbered 1-4. It's a matter of taste, mostly. Note, though, that while it is fine to @renumber yourself, it can cause problems if you @renumber a *list that you own. The reason is that the record of everyone's current message on that list is kept not on the list, but on the players who subscribe to that list. If you @renumber a *list, then the @rn command might indicate to a player that the list has new mail, but the @nn command would say that there are no new messages. This is hardly catastrophic, but if you own a *list, it's better to let gaps in the message numbering stay.

*news

The *news mailing list is a regular list, like any other, except that only wizards can send to it, and wizards can cause designated messages to appear when a player types the command news. But you can read it like a regular mailing list. This would come in handy, for example, if you wanted to review an old *news message that wasn't actually appearing in the newspaper any more (because a wizard had removed its current-news designation).

Sending Mail (and getting a start on using the in-MOO editors)

It's good to get comfortable with the mail editor, because then you can send mail to individuals and compose posts to mailing lists and focus on what you have to say (write) rather than on the mechanics of writing it. Furthermore, the note editor and the verb editor work almost identically, so you can apply your mail editor skills later if you choose to program your own objects.

Practice is key. But when people practice, especially at the beginning, they make mistakes, and making mistakes feels embarrassing (to many people), and so they don't practice. My proposed remedy for this is to have you start by sending mail to *yourself* until you feel ready to go public, so to speak.

A bit of reassurance: You can't break the editor. If you get *completely* stuck, just log off – exit your client application or simply disconnect. In a minute or two, you'll be moved from the editor back to your home, and no harm has been done. So relax.

Beginning and Ending a Mail Session

A normal mail editing session begins with @send and ends with send. The difference is in the @-sign. If you change your mind about sending, you can type abort and your editing session will be thrown away. You can @send to a person, to a mailing list, or to any combination. Here are a few examples:

```
@send Yib
@send Yib Tower Drippy lovecraft Veren skeptopotamus
@send *Think-Tank
```

```
@send Yib *Think-Tank
@send me
```

(The last of these is what you would use to send mail to yourself.)

There is no case-sensitivity here, as in most things within the MOO except for passwords.

After beginning with `@send`, you will be prompted for a subject line. You may leave it blank if you wish by typing the `<enter>` key.

Once in the editor, you can type `look` at any time to see a list of available commands. Basic commands fall into three categories: adding text, displaying text, and changing text.

Adding Text

There are two basic ways to add text. In the editing rooms, what you say (with the `say` command) gets added to the text you are editing. That's one way. The other way is to type *the word*

```
enter
```

on a line by itself, followed by any number of lines of text, followed by a period on a line by itself. You can paste in lines of text from your computer's clipboard, this way, if you want to. This is the method that I prefer, but it's strictly a matter of taste. NOTE: Most client programs will wrap words to fit a reader's screen width, and trying to do that manually makes the text funny-looking for some people, and some of those will complain. As a point of style, you should type each paragraph as if it were one very long line, and separate your paragraphs with a blank line (by pressing the <enter> key twice in a row).

Displaying Text

After you've entered some text, you'll probably want to take a look at it and see whether you like it, or whether you want to adjust it. You can do either of the following:

```
print
```

or:

```
list
```

The first, `print`, displays all the text as it would be seen by the recipient. This is especially good for proofreading. The second, `list`, shows you the lines with their numbers. This is helpful because when making changes, you need to specify a line or lines by number. If the post is long, `list` may only print out some of the lines. You can force it to list all the lines by typing:

```
list 1-$
```

In this context the $-sign is a symbol that means “the end”. You can list any range of numbers.

Changing Text

To change something, use the `subst` command (think “substitute”), which can also be abbreviated as `s`:

```
subst /kind of fun/a whole lot of fun/5
subst /pretty good/fabulous/1-$
s /so-so/great/3-16
```

The first form will make the substitution in line 5, the second will make the substitution in all lines, the third in lines 3 through 16. The “/” character is a separator. It can actually be any character that doesn’t appear in the text you’re substituting.

To delete one or more lines, use the `delete` command (`del` for short):

```
delete 5
del 3-8
del 1-$
```

The editor will print out the line or lines deleted, for verification. Heads up: When you delete a line, the editor will move the insertion point to that line. This means that if you `enter` text again, it will appear *in place of* the line or lines you just deleted. To enter text elsewhere, use the `insert` command (`ins` for short):

`insert 1`	Get ready to insert text before line 1.
`insert 5`	Get ready to insert text before line 5.
`ins $`	Get ready to insert text at the end.

After the `insert` command, you must still type enter to actually add text to your message. Again, type a period on a line by itself when you’re finished entering lines.

Send it Off!

Type `send` to send the mail, or `abort` if you change your mind.

Mail Options

You can customize many aspects of how your mail messages are displayed and handled through an interface called an *options package*, which is simply a group of settings plus some commands to view and alter those settings. In the section on reading mail, you were shown `@mail-option +sticky`. In this section, I’ll explain

what each mail option means. You can type `help @mail-options` go get a brief reminder of this section.

To list all your current mail options, type:

```
@mail-options
```

You will see something like this:

```
Current mail options:

-include            Original message will not be included in
                    replies.
-all                Replies will go to original sender only.
-followup           No special reply action for messages with
                    non-player recipients.
-nosubject          Mail editor will initially require a
                    subject line.
-expert             Novice mail user.
-enter              Mail editor will not start with an implicit
                    'enter' command.
-sticky             Teflon folders:  mail commands always
                    default to 'on me'.
-@mail              Default message sequence for @mail:
                    last:15
-manymsgs           Willing to be spammed with arbitrarily many
                    messages/headers
-replyto            No default Reply-to: field
-netmail            Receive MOO-mail here on the MOO
-expire             Unkept messages expire in 30 days (default)
-resend_forw        @resend puts player in Resent-By: header
-rn_order           .current_message folders are sorted by last
                    read date.
-no_auto_forward    @netforward when expiring messages
-expert_netfwd      @netforward confirms before emailing
                    messages
-news               the 'news' command will show all news
-no_dupcc           I want to read mail to me also sent to
                    lists I read
-no_unsend          People may @unsend unread messages they
                    send to me
-@unsend            Default message sequence for @unsend:
                    last:1
```

The items in the first column are the options themselves, which you can set. Most options are either *on* or *off*. In the example above, every option is preceded by a `-`, so all options are turned off. A few must be set to one of a selection of permissible

text values. The syntax for setting (turning on) or resetting (turning off) options is as follows:

```
@mail-option +<flag>
@mail-option -<flag>
@mail-option !<flag>
```

(The last two are equivilent.)

OR (if required):

```
@mail-option <option>=<value>
```

Let's go through them one by one.

```
@mail-option +include
@mail-option -include
```

Sooner or later you are bound to see a mail message that looks something like this:

```
Message 122 on *Features (#42343):
Date:    Tue Jan 18 17:41:37 2000 PST
From:    Yib (#58337)
To:      *Features (#42343)
Subject: Re: @parties

>  Date:     Tue Jan 18 14:24:01 2000 PST
>  From:     Goldmund (#96860)
>  To:       *Features (#42343)
>  Subject:  @parties
>
>  Good work Yib.
>  It would be nice if the rooms would be sorted by
>  the number of occupants rather than by alphabet.
>  ... and you should add it to the @help.
>
>  thanks

Done, and done.  Thanks for the suggestions.
--------------------------
```

Embedded in this message is a prior message, each line of which is preceded by the ">" symbol. Sooner or later you'll wonder, "How does she do that?" The `@reply` command does this automatically if the `include` mail option is turned on. Goldmund's message was message 121 on *features. When I typed `@reply 121 on *features`, the mail editor automatically created the header and included the text of the message I was replying to. I then appended my own lines of text.

Note: I also typed `to *features` to send my post to the mailing list as part of an ongoing discussion (in this case, discussion of a feature verb I'd written) instead of only to Goldmund. The `@reply` command directs the reply to the author of the post

you're replying to – and not the list you read it on – *unless* you have either the `all` option or the `followup` option turned on.

```
@mail-option +all
@mail-option -all
```

If I had set the `+all` mail option, then my reply would automatically have been to Goldmund *and* `*features`.

```
@mail-option +followup
@mail-option -followup
```

It gets better. If the `+followup` option is set, and I `@reply` to a message, the message will be to "the first non-player recipient, if any". What's a non-player recipient? A mailing list. What if you're replying to a private message from a player and not a message on a mailing list (i.e. what if there *are* no non-player recipients)? Then it uses whatever you've set the `all` flag to, either the original author (`-all`) or all the original recipients (`+all`).

Reminder: You can override the designated recipients from within the mail editor at any time with the `to` command. `To` Goldmund would direct a post just to Goldmund. `To` *features would direct it just to *features.

```
@mail-option -nosubject
@mail-option +nosubject
```

When you type the `@send` command, the system prompts you to enter a subject line, unless you have the `+nosubject` option set.

```
@mail-option +expert
@mail-option -expert
```

The online documentation tells you that the `+expert` mail option suppresses "varying annoying messages". What are these various annoying messages? It turns out that there's only one. If you are `-expert` and receive mail, you will see something like this:

```
You have new mail (22) from Yib (#58337).
Type 'help mail' for info on reading it.
```

If you are `+expert`, you won't see the line that says, `Type 'help mail' for info on reading it.` Rog (the author of this code) got tired of seeing this message all the time, and so added this option to suppress it plus any other messages he decided to add later. No other messages suggested themselves, so that's all, folks.

```
@mail-option +sticky
@mail-option -sticky
```

This option was described in the segment on reading mail. It affects the `@mail`, `@read`, `@peek`, `@next`, `@prev`, and `@answer` (same as `@reply`) commands. If the option is set to `+sticky`, any of the above commands will apply by default to the mail recipient (yourself or a *list) you accessed most recently. If its set to `-sticky`, your current mail recipient (current folder) will always be yourself.

```
@mail-option +netmail
@mail-option -netmail
```

You can choose whether to receive your MOOmail in-MOO (-netmail), or have your MOOmail automatically forwarded to your registration email address (+netmail). The latter saves space (for the MOO), but the trade-off is that if you receive mail while logged on, you don't get to read it until it reaches your regular email address, and for some people that's a hassle.

```
@mail-option +resend_forw
@mail-option -resend_forw
```

You can either @forward or @resend a message (on yourself or a *list) to another player or another *list. These two commands are awfully similar, differing only in what the header information looks like, possibly modified by the resend_forw mail option. Here are some examples to show you what I mean: I started by sending a message to myself, then forwarded that message to another character, Yib's_Assistant.

Here is the original message that I sent to myself:

```
Date:     Wed Jan 19 16:28:08 2000 PST
From:     Yib (#58337)
To:       Yib (#58337)
Subject:  Secret

La plume de ma tante est dans le jardin.
--------------------------
```

When you @forward this message, you get:

```
Message 7:
Date:    Wed Jan 19 16:28:30 2000 PST
From:    Yib (#58337)
To:      Yib's_Assistant (#61050)
Subject: [Yib (#58337):  Test]

Date:     Wed Jan 19 16:28:08 2000 PST
From:     Yib (#58337)
To:       Yib (#58337)
Subject:  Secret

La plume de ma tante est dans le jardin.
--------------------------
```

Notice that there are two header blocks, and brackets around the original subject line. If you @resend the message, with -resend_forw, you get only one header block:

```
Message 8:
Date:    Wed Jan 19 16:28:38 2000 PST
From:    Yib (#58337)
To:      Yib (#58337)
Subject: Secret
Resent-By:      Yib (#58337)
Resent-To:      Yib's_Assistant (#61050)
Original-Date: Wed Jan 19 16:28:08 2000 PST

La plume de ma tante est dans le jardin.
--------------------------
```

@resend with +resend_forw generates just one header, but this time there are two "Original" lines instead of "Resent-" lines:

```
Message 9:
Date:    Wed Jan 19 16:28:56 2000 PST
From:    Yib (#58337)
To:      Yib's_Assistant (#61050)
Subject: Secret
Original-Date: Wed Jan 19 16:28:08 2000 PST
Original-From: Yib (#58337)

La plume de ma tante est dans le jardin.
--------------------------
```

```
@mail-option +no_auto_forward
@mail-option -no_auto_forward
```

Mail messages are one of the bigger database hogs on a MOO. Therefore, on some MOOs, mail is automatically deleted from players and *lists after a certain amount of time (usually 30 days). By default, when your messages expire, they are forwarded to your registration email address. Setting this option inhibits that forwarding.

```
@mail-option +expert_netfwd
@mail-option -expert_netfwd
```

You can forward MOOmail messages on yourself or on *lists to your registration email address with the @netforward command. By default, the system will display your current registration email address and prompt you for confirmation. Setting this option inhibits that.

```
@mail-option manymsgs is <number>
@mail-option manymsgs <number>
@mail-option manymsgs=<number>
@mail-option -manymsgs
```

Suppose you typed @read new on *chatter and suddenly found yourself spammed with 300 messages! The above option gives you some control over that kind of situation. If you specify a number for the manymsgs option, you will be

given a yes/no prompt to continue. The last form, -manymsgs, inhibits this behavior.

```
@mail-option @mail <message-sequence>
@mail-option @mail is <message-sequence>
```

For example:

```
@mail-options @mail=new
```

On most MOOs, typing the command @mail is equivalent to typing @mail last:15 to show the last 15 messages on your current folder. There are other possibilities, such as @mail new or @mail 1-last, and you can specify how you want plain @mail to behave by using this option.

```
@mail-option replyto <recipient> [<recipient>...]
@mail-option replyto is <recipient> [<recipient>...]
@mail-option -replyto
```

The above option automatically inserts a "Reply to:" field in any messages that you compose using @send or @reply to indicate that replies should be sent to someone other than the original sender. The last form resets this option so that no "Reply to: " field is initially inserted.

```
@mail-option rn_order=<order>
@mail-option rn_order <order>
```

This option controls the order in which folders will be listed when you use the @rn and @subscribed commands. The options for <order> are:

read	Folders are sorted by last read date. This is the default.
send	Folders are sorted by the date you last sent to them.
fixed	Folders are not sorted initially.

If you specify rn_order=fixed, you can sort your folders using the @subscribe command, and they will stay in that order afterwards. For example, you can type @subscribe *random after *life, or @subscribe *random before *life. By doing this several times, you can re-order all your folders.

```
@mail-option expire <time-interval>
@mail-option expire is <time-interval>
@mail-option expire=<time-interval>
```

By default, mail messages expire after a certain time period (usually 30 days). You can use the above option to change this. <time-interval> is a number of seconds unless you specify units, e.g. 5 weeks or 2 months.

A negative number or:

```
@mail-option +expire
```

disables expiration entirely. You can keep a particular mail message from expiring with the @keepmail command.

```
@mail-option -expire
```

sets your message expiration time to MOO's current default.

```
@mail-option +no_unsend
@mail-option -no_unsend
```

LambdaMOO provides the ability to take back a MOOmail message sent to an individual if the message has not yet been read, and has not been netforwarded. Setting this option prevents people from being able to @unsend messages they have sent to you.

```
@mail-option @unsend=<message sequence>
@mail-option -@unsend
```

If you wish to retract a message you just sent to someone, you can try @unsend from <player>. If you don't specify which message(s) you wish to @unsend, the system will use last:1, meaning the last message you sent. You can't look at another person's mail headers to specify message numbers, but you can specify other parameters such as subj:<part of a subject heading> or since:yesterday, and so on. You can do this for a single invocation of @unsend, or you can set such a filter to be the default, by using this mail option.

Using the in-MOO Editors

There are three editors provided with LambdaCore: the mail editor, for editing and sending MOOmail, the verb editor, for editing and compiling verbs (MOOcode), and the note editor, for editing all other bodies of text. Each editor has a few unique commands, e.g. you send a message, compile a verb, and save other text, but other than that, they are fundamentally the same, and if you can use one, you can use the others.

Editors in the MOO are implemented as special rooms! These rooms simulate a player being alone (if other people are using the editor simultaneously, you will not see them), and in these rooms, the verbs say and emote are programmed to work differently – specifically, say and emote are ways of inserting or appending text to whatever you are editing.

These are line-based editors, which means that text is added a line at a time. There isn't a cursor, as there is in most screen editors. Instead, there is the concept of an *insertion point*, and this number represents where new text will be added: between two lines, before the first line, or after the last line. Lines can in fact be quite long – as long as a conventional paragraph, in fact, comprising many sentences. They may take up more than one horizontal row of text on a person's screen. A *line*, then, in this context, is a logical unit of text. Typical usage is to type in one entire paragraph of text per line in the editor, and use blank lines to separate paragraphs. Most people use some sort of client software that formats text to the width of their screen or window as needed, and the one paragraph per line method makes it easiest for most people's clients to present text in a coherent way.

I'll begin by talking about how to begin an editing session, and how to end one, i.e. how to get out of the editor once you're there. Then I will describe some different ways to enter text, ways to view the text you've entered so far, and ways to modify the text you've entered.

Invoking the Editor

When you start up one of the editors, you are moved to one of the special editing rooms, and we call that *invoking* that editor. Thus, one would *invoke* the note editor, the mail editor, or the verb editor. The in-MOO editors require you to specify what you're editing before you can begin. (This differs from most commercial editor or word processing programs which provide the option of opening a new file, then specifying where the text will go, for example with a "Save As" command.) So you begin editing by indicating the thing you want to edit. Here are some examples:

Command	Description
`@send Werebull`	Invoke the mail editor, working on a message to the player Werebull.
`@send *chatter`	Invoke the mail editor, working on a message to the list `*chatter`.
`@send Werebull *chatter`	Invoke the mail editor, working a message for both Werebull and `*chatter`.
`@notedit me.description`	Invoke the note editor, working on your own `.description` property.
`@edit me.description`	Invoke the note editor, working on your own `.description` property (same result as `@notedit me.description`).

`@notedit here.exterior_description`
 	
`@edit here.exterior_description`
	Invoke the note editor, working on the `.exterior_description` property of the room that you are in. Note that you have to own an object in order to edit properties on it.

`@notedit test note`
 	
`@edit test note`
	Invoke the note editor, working on the text of an object named "`test note`".

`@notedit #1234`
 	
`@edit #1234`
	Invoke the note editor, working on the description of object `#1234` (which you must own).

@edit rock:drop	Invoke the verb editor, working on the verb :drop on an object named “rock”.
@edit #1234:drop	Invoke the verb editor, working on the verb :drop on object #1234

Notice that the same command, @edit, can invoke either the note editor or the verb editor, depending on whether you use a period (.) to specify a property, or a colon (:), which indicates a verb. Furthermore, if you @notedit or @edit a descendent of either the generic note or the generic letter, you will be editing that note or letter’s text, but if you @notedit or @edit any other object (without specifying a property or verb), then you will edit that object’s description.

```
@edit #1234
```

will invoke the note editor to edit the *description* of object #1234.

Leaving an Editor

There are three ways that you can leave an editor after you have finished entering and editing text.

One is to exit and discard all text you may have entered or modified during that editing session. The command is the same in all three editors. Type the command:

```
abort
```

The second is to save your changes or send your message and then exit, and this is different in each of the editors. In the note editor, type save, and then done. In the mail editor, type send. In the verb editor, type compile and then done.

The third way is to leave your work in progress. In any of the editors, you can type done (without saving, sending, or compiling). Or you can teleport from the editor (remember, an editor is a room) to any other location. Or you can log off. Your work is saved on the equivalent of a scratch pad, and when you return to the editor, you can either resume that work, or abort out of it. To return, you can type @edit by itself (for the note editor or the verb editor) or @send by itself (for the mail editor). If you try to edit something other than what you were working on, the system will ask you if you want to resume the original editing session or abort it, before letting you edit the new note verb, or MOOmail.

Listing Text, Finding and Moving the Insertion Point

When you enter new lines of text, they are added after the current line. So the first order of business might be to determine which line is current and perhaps move to another line. The place where new text will be added is called the *insertion point*.

The easiest way to see where the insertion point is, and get your bearings in the text, is to type:

```
list
```

This will show you the current insertion point, plus some of the context, i.e. a few of the lines above and a few below. Here's one example:

```
  1: Fee,
  2: Fie,
__3_ Fum!
^^^^
```

In this case, the insertion point is after line 3, and if I begin entering text, it will go in at the bottom of the document. If the insertion point were between lines 2 and 3, then `list` would show:

```
  1: Fee,
__2_ Fie,
^^3^ Fum!
```

The combination of underscores and up-arrows is intended to depict a spot between two lines, where new text will go, if and when you start entering it.

There are a few ways to move the insertion point. Perhaps the easiest is with the `insert` command, which can be abbreviated `ins`. Counter to what you might think, the `insert` command doesn't insert text. It gets you ready to insert text by specifying *where* you want to insert it. Used alone, the `insert` command will show you the insertion point with just the line above and the line below. If you specify a number, though, that moves the insertion point before that line:

`ins 1`	Get ready to insert text before line 1, i.e. at the very beginning of the document.
`ins _1`	Get ready to insert text after line 1, i.e. between lines 1 and 2.
`ins 3`	Get ready to insert text before line 3, i.e. between lines 2 and 3.
`ins $`	Get ready to insert text at the end of the document, i.e. after the last line.

Entering Text

There are four ways to input new text into your document, mail message, or verb. These are: `say`, `emote`, `enter`, and `yank`.

When you say something in an editor, the text you say is added as a new line at the insertion point. So if, in our example, the insertion point were between lines 2 and 3, and you typed:

```
"Foo,
```

and then you typed `list`, you would see:

```
  1: Fee,
  2: Fie,
```

```
__3_ Foo,
^^4^ Fum!
```

(Notice that the double-quote character (") didn't get added, because it is just the abbreviation for the say command.)

When you emote something in an editor, it appends that text to the end of the line that is *before* the insertion point. Suppose you wanted to append some text to line 4. First you would position the insertion point after line 4 by typing either ins _4 or ins $. Then you might type:

```
:  I smell the blood of an Englishman!
```

and then if you typed list, you would see:

```
  1: Fee,
  2: Fie,
  3: Foo,
__4_ Fum!  I smell the blood of an Englishman!
^^^^
```

Another way to enter text is with the enter command. This entails typing the word enter on a line by itself. The system prompts you to enter lines of text. You terminate this text entry by typing a period (.) on a line by itself, or else the command @abort. The first keeps (and inserts) the lines of text, the second throws them away. Suppose you wanted to add some text to the beginning of the document. First, position the insertion point so that it is at the beginning: ins 1. Then type (exactly):

```
enter
The giant looked around...
Then the giant sniffed the air...
Then the giant began to roar:
.
```

Now if you type list, you will see:

```
  1: The giant looked around...
  2: Then the giant sniffed the air...
__3_ Then the giant began to roar:
^^4^ Fee,
  5: Fie,
  6: Foo,
  7: Fum!  I smell the blood of an Englishman!
```

You can also compose text in a word processing window on your own computer and paste it from the clipboard after you type the word enter and before you type the period on a line by itself.

The last way to enter text is with the yank command. It's used to bring in the text of a note object in its entirety. So if you had the entire story of Jack and the Beanstalk in a note object, and wanted to pull it into a mail message, for example,

you would start the mail editor with the `@send <person>` command, then (after typing the subject line, if prompted to do so) you would type:

```
yank from <note>
```

The entire text from the specified note would be entered as if you had typed it. You could then edit it further, or save or send it as it was.

In addition to listing text, you can display it without the numbers, by typing the command:

```
print
```

Modifying Text

Once you have some text and looked it over, you might want to make some modifications. For this we have the `subst` command, which can be abbreviated as `s`. If you type `look` while in any of the editors, you'll see a short synopsis of commands with their syntaxes. The one for `subst` looks like this:

```
s*ubst      /<str1>/<str2>[/[g][c][<range>]]
```

The heart of the `subst` command is to substitute one string for another. The simplest version is to make the substitution in the current line. Let's list out our text, again:

```
  1: The giant looked around...
  2: Then the giant sniffed the air...
__3_ Then the giant began to roar:
^^4^ Fee,
  5: Fie,
  6: Foo,
  7: Fum!  I smell the blood of an Englishman!
```

The insertion point is after line 3, which makes line 3 the current line for substitution. Let's change "roar" to "bellow":

```
subst /roar/bellow/
```

The editor will print out the modified line:

```
__3_ Then the giant began to bellow:
```

Now let's change that "Foo" to "Fo" in line 6:

```
subst /Foo/Fo/6
```

The editor will again print out the modified line:

```
6: Fo,
```

Let's change the giant into an ogre, through the whole range of lines. Remember that the way to designate "after the last line" is with the "`$`" sign:

```
subst /giant/ogre/1-$
```

We will see:

```
  1: The ogre looked around...
  2: Then the ogre sniffed the air...
__3_ Then the ogre began to bellow:
```

By now you might want to take a look at the full text again, to see what we've got:

```
print
```

You will see:

```
The ogre looked around...
Then the ogre sniffed the air...
Then the ogre began to bellow:
Fee,
Fie,
Fo,
Fum!  I smell the blood of an Englishman!
--------------------------
```

For fun, let's change all instances of the letter "f" to the letter "r". This incorporates everything at once:

```
subst /f/r/gc 1-$
```

The "g" (think "global") in the command means every letter "f" on a line, not just the first, so we get "snirred" rather than "snirfed". The "c" means ignore capitalization. So we get:

```
The ogre looked around...
Then the ogre snirred the air...
Then the ogre began to bellow:
ree,
rie,
ro,
rum!  I smell the blood or an Englishman!
```

At this point, we've pretty well mangled our example. How to restore it? More substitutions. Or type `abort` and start over.

Before we leave substitutions, let's consider what to do if you wanted to use the slash character (/) itself as part of a substitution: You can use any separator at all that doesn't appear in either the string you wish to replace or the new string. So to change "and or" to "and/or" you could type any of:

```
subst |and or|and/or|
subst ,and or,and/or,
subst xand orxand/orx
```

The `subst` command is used so much that it finds its way into casual conversation. You might see something like this:

```
Ostrich says, "I will now sing the national anthem while
standing on my head and drinking root beer through a straw."
Ostrich says, "subst /now/not/"
```

Ostrich is indicating that he made a typographical error and intended to say that he will *not* sing the national anthem while standing on his head and drinking root beer through a straw.

Re-arranging Text

Let's go back to an earlier version of our text:

```
   1: The giant looked around...
   2: Then the giant sniffed the air...
   3: Then the giant began to roar:
   4: Fee,
   5: Fie,
   6: Foo,
 __7_ Fum!  I smell the blood of an Englishman!
 ^^^^
```

The `move` command moves a line of text, or a range of lines, to a new point. If we wanted to change the sequence of our giant's utterances, for instance, we could do this:

```
move 6 to 5
```

and get this:

```
   1: The giant looked around...
   2: Then the giant sniffed the air...
   3: Then the giant began to roar:
   4: Fee,
   5: Foo,
   6: Fie,
 __7_ Fum!  I smell the blood of an Englishman!
 ^^^^
```

We could do this:

```
move 5-6 to 4
```

and get this:

```
   1: The giant looked around...
   2: Then the giant sniffed the air...
   3: Then the giant began to roar:
   4: Foo,
   5: Fie,
   6: Fee,
```

```
__7_ Fum!  I smell the blood of an Englishman!
^^^^
```

The `copy` command copies a line or range of lines in like manner:

```
copy 4 to 4
```

would yield:

```
  1: The giant looked around...
  2: Then the giant sniffed the air...
  3: Then the giant began to roar:
  4: Foo,
  5: Foo,
  6: Fie,
  7: Fee,
__8_ Fum!  I smell the blood of an Englishman!
^^^^
```

The `join` command joins lines together. For example:

```
join 4-8
```

gives:

```
  1: The giant looked around...
  2: Then the giant sniffed the air...
  3: Then the giant began to roar:
__4_ Foo, Foo, Fie, Fee, Fum!  I smell the blood of an
Englishman!
^^^^
```

The `fill` command combines joining and splitting lines so that they are less than or equal to a specified number of characters (columns) in length. For example:

```
fill 4 @25
```

gives:

```
  1: The giant looked around...
  2: Then the giant sniffed the air...
  3: Then the giant began to roar:
  4: Foo, Foo, Fie, Fee, Fum!
  5: I smell the blood of an
__6_ Englishman!
^^^^
```

Deleting lines

You can delete a line or range of lines with the `delete` command, which can be abbreviated as `del`. When you do this, the editor prints out the deleted text *and* the

insertion point is automatically moved to the point where the lines were deleted, so if you then start entering text, it will directly replace the lines you just deleted. Here's a quick example:

```
del 4
Foo, Foo, Fie, Fee, Fum!

"Fee, Fie, Fo, Fum!
Line 4 added.

list

  1: The giant looked around...
  2: Then the giant sniffed the air...
  3: Then the giant began to roar:
__4_ Fee, Fie, Fo, Fum!
^^5^ I smell the blood
  6: of an Englishman!
```

A Few Miscellaneous Commands

If you somehow forget what you are editing, you can use the `what` command. I edited an actual note on LambdaMOO as I wrote this segment. Here's what I see:

```
what

You are editing "test note"(#92413).
Your insertion point is before line 5.
There are changes.
```

Sometimes you want to verify a line number before doing a substitution, and for this we have the `find` command. By itself, it will find the next instance (i.e. after the current insertion point, wherever that is) of the string you specify. Or you can specify a starting point. You can also ignore capitalization. The `find` command moves the insertion point, so you can use the command sequentially until you find the particular instance of a word or string of characters that you're looking for. Here are some examples:

`find /f`	Find the next line with an "f" in it.
`find /f`	Find the next-next line with an "f" in it.
`find /f/1`	Find the first "f" in or after line 1.
`find /f/c1`	Find the first "F" or "f" in or after line 1.
`find /fo/c1`	Find the first instance of "fo" or "FO" or "Fo" or "fO" in or after line 1.

The `mode` command is probably of interest only to programmers. If you are not a programmer and don't understand this part, don't worry; you may ignore it in complete safety. If your text has several lines in it, it will be stored, when you save it, as a list of strings. If your text has only one line of text in it, then it can either be stored as a single text string *or* as a list with one element (a string) in it.

`mode`	Find out what mode you're currently in.
`mode string`	Switch to string mode.
`mode list`	Switch to list mode.

The `publish` and `view` commands work together. Normally, no one can see your text as you edit it. But suppose you were struggling with something in the verb editor, and I wanted to help you but couldn't quite get a handle on the situation based on paging each other back and forth. First I would go to a "blank" editing session by typing `@go $verb_editor` (remember, it's a room). Then I would page you to type the `publish` command. Then I could type `view <your name>` and the system would list out what was in your editing session. I might then page you with something like, "Try adding a semicolon to the end of line 3."

In the mail editor, you can change the recipient of your post with the `to` command. Suppose you had typed `@reply cur on *chatter`, and because of your mail-option settings, your post was addressed to the author when what you really wanted to do was reply on the *list itself. You'd spot this (one hopes) when printing your message before sending it. You could change the recipient by typing `to *chatter`, or `to <list of recipients>`.

Edit Options

Just as there are mail-options you can set, there are also edit-options you can set. Type `@edit-options` to see what they are. Probably the most significant one is `local`. If you type `@edit-option +local`, then whenever you `@edit` anything, the text you wish to edit will be uploaded to your client program, and when you are finished, you can download the text again. Each client is different, as are the commands to edit in those clients, and not all clients support this feature. To change back to using the in-MOO editors, type `@edit-option -local`.

Summary

If you forget the collection of commands available to you, you can type `look` while in any of the editors and see a reminder list of them. There is detailed help text for each individual command as well.

```
look

Note Editor
Commands:

say         <text>                        y*ank       from <text-source>
emote       <text>                        w*hat
lis*t       [<range>] [nonum]             mode        [string|list]
ins*ert     [<ins>] ["<text>]             e*dit       <note>
n*ext,p*rev [n] ["<text>]                 save        [<note>]
enter                                     abort
del*ete     [<range>]                     q*uit,done,pause
f*ind       /<str>[/[c][<range>]]
s*ubst      /<str1>/<str2>[/[g][c][<range>]]
m*ove,c*opy [<range>] to <ins>
join*l      [<range>]
fill        [<range>] [@<col>]

----  Do `help <cmdname>' for help with a given command.  ----

<ins> ::= $ (the end) | [^]n (above line n) | _n (below line n) | . (current)
<range> ::= <lin> | <lin>-<lin> | from <lin> | to <lin> | from <lin> to <lin>
<lin> ::= n | [n]$ (n from the end) | [n]_ (n before .) | [n]^ (n after .)
'help insert' and 'help ranges' describe these in detail.
```

Chapter 5 – Extending the Virtual Reality: Building

Overview

This section details the various ways to create rooms and other objects. It amplifies what you already leaned about `@dig`. On nearly all MOOs, players start out as members of the "generic builder" player class, which means that the building commands are available to them.

Between the basics of communication, moving around, and interacting with various objects on the one hand, and the intricacies of programming on the other, is building. Building is the business of creating new objects on a MOO and modifying them in certain limited ways. When you create an object, you first specify what kind of object it is to be (a room or a container or a note, for example), and the object's name, with optional aliases. Then, typically, you describe your object, maybe set some of its messages (see page 46), and the object is then ready to use. Some fancier kinds of objects let you specify some additional information which might also govern how the object behaves. An example of this might be where an object should go if it needs to be "sent home".

@create

```
@create <generic> named "<name>"
```

or:

```
@create <generic> named
"<name>","<alias1>","<alias2>",…,"<last alias>"
```

`@create` and `@dig` are the two quintessential commands of building. Each creates a new object where there was none before.

There are a couple of ways to specify the kind of thing (i.e. the *parent* or *generic*) you want to create. As always, you can specify the object by its object number, if you know it. If the generic you want to create a copy of is in your vicinity (i.e. you are either holding it or in the same room with it), then you can specify it by name. A few objects are used so commonly that we can refer to them even if they are not nearby, using the "$" sign. The generic thing is called `$thing`. The generic container is called `$container`. The generic note is called `$note`. In this way, you can create things and containers and notes without having to know the generics' object numbers or have them in your vicinity. Let's look at an example of creating a very simple object, a paper weight:

```
@create $thing named "paper weight","paperweight","pw"
```

The system will print out something like:

```
You now have paper weight (aka paperweight and pw) with
object number #63555 and parent generic thing (#5).
```

The actual number of your paper weight will be different. Do you have to remember this number? Yes and no. As long as the paper weight is either in the same room that you are in or is in your inventory (i.e. you are holding it), then you can refer to it by name. And when you first create something, you are holding it. That's one reason why we use `@dig` (described further on) instead of `@create` for rooms and connecting exits: so that you won't be holding these things when they are created. There's nothing actually *wrong* with holding rooms or exits, but it doesn't make sense and serves no practical purpose.

@describe

The next logical thing to do is to describe your paper weight:

```
@describe pw as "An ovoid paper weight made of onyx.  Though
perfectly smooth, it has the curious property that it gives
no reflection, almost as if it were an oddly-shaped black
hole.  It does have the expected flat spot on the bottom."
```

There isn't much you can do with a $thing. You can hand it to someone. You can drop it. You can take it. That's about it. To be sure, type:

```
examine paper weight
```

What you *can* do is change the text that you and/or others see when you hand it to someone or drop it or take it. (Note that you don't have to be a builder to set messages on objects you own; I review it here because it's the logical next thing to do when crafting an object.) This is done through messages, and changing the messages on an object is a quick and easy way to give it a bit of character. First, you might want to list the existing messages:

@messages

```
@messages paper weight
```

The system will respond with the following list:

```
@drop_failed paper weight is "You can't seem to drop %t
here."
@drop_succeeded paper weight is "You drop %t."
@odrop_failed paper weight is "tries to drop %t but fails!"
@odrop_succeeded paper weight is "drops %t."
@otake_succeeded paper weight is "picks up %t."
@otake_failed paper weight isn't set.
@take_succeeded paper weight is "You take %t."
@take_failed paper weight is "You can't pick that up."
```

Because your object is a child of `$thing`, it has inherited all its properties, including all the messages that `$thing` has.

The purpose of most of these messages should be pretty clear from the combination of their names and content. By convention, messages beginning with “o” are told to others, while messages not beginning with “o” are told to the player initiating the relevant action. The `%t` (think “this”) in each message substitutes the actual name of the object. In general, most generics have their messages set so that they make sense without any customization, but let’s change a few of these to demonstrate the method. The syntax for setting messages is the same as the way the messages are printed out above. Here are a couple of examples:

```
@drop_succeeded pw is "You drop %t.  It lands with a thud,
then rolls a short distance before coming to a stop."

@odrop_succeeded pw is "drops %t.  It lands with a thud,
then rolls a short distance before coming to a stop."
```

@recycle

Should you decide that you no longer want this object, you can get rid of it by typing:

```
@recycle paper weight
```

and the system will respond with a line like:

```
paper weight (#63555) recycled.
```

You can only recycle objects that you own. If the object you wish to recycle is not in your vicinity, then you can recycle it by object number instead of by name. It’s good to recycle objects that you don’t use, as this conserves system resources.

Generic Objects

$things are useful as stage props. If you want to be seen carrying a feather duster, for example, but don’t need to do any actual dusting, `@create $thing named "feather duster"` will probably suit your purposes adequately. Other generics, however, are capable of doing more. A good example of this is the generic container, abbreviated `$container`. To make one, type (for example):

```
@create $container named "leather pouch","pouch","lp"
@describe lp as "A simple-looking leather pouch, of
remarkable capacity."
```

You can type `examine pouch` to see what actions you can actually do with it, and `@messages pouch` to see if you want to change any of the associated messages. Then if you want to you can put your paper weight in the pouch.

Normally, one would (and should) investigate a generic before making an instance of it. People tend to practice varying levels of due diligence in this regard, but the appropriate steps (the sequence isn't all that important) are as follows:

```
examine <generic>
help <generic>
@parents <generic>
```

Then repeat the above for each listed parent until you come to an object you already know about.

Generics can be extremely complex. By using `@create` to make an instance (think "clone") of an existing generic, you get all its functionality without having to do any programming. Building things based on existing generics also takes up less space in the database than programming a new object from scratch, which is desirable, too.

So then the question arises, "How does one know what generics there are to make kids of?" Some of the basics are `$thing`, `$container` (you can open it, close it, put things into it and take things from it), `$note` (you can edit its text and others can read it), `$letter` (like a `$note`, but a designated recipient can burn it when e has finished reading) and `$mail_recipient` (a MOO mailing list). These generics are part of the LambdaCore, and are included in every database based on it. Players who are programmers can create additional generics. These usually can't be referenced by a name beginning with a "`$`"; they have to be referenced by object number.

@parents

One way to learn about existing generics is to explore the MOO, examining many objects, and when you find one that interests you, type:

```
@parents <object>
```

This command shows an object's ancestry, and after the `examine` command it is one of the most basic tools at your disposal to investigate an existing object. Suppose, for example, you took an interest in a hot air balloon that you came across in your explorations. You could type:

```
@parents Royal Blue Balloon

Royal Blue Balloon(#68806)   Generic Hot Air Balloon(#66549)
Generic Aircraft(#42055)   Generic Magnetic Portable Secure
Seated Integrated Detail Room(#58237)   Generic Portable
Secure Seated Integrated Detail Room with Sensible
Locking(#17524)   Generic Portable Secure Seated Integrated
Detail Room(#36643)   Generic Secure Seated Integrating
Detailed Room(#9805) Area/Seat-Conscious Room(#5531)
Generic Secure Integrating Detailed Room (without
seats)(#156)   Integrating Detail Room with Features(#21311)
Integrating Detail Room with Exit-Verb Matching(#8801)
```

```
Integrating Detail Room Mark III(#17755)   Modified Detail
Room(#11825)   Frand's generic detailed room(#6464)   Self-
Cleaning Room(#27777)   generic room(#3)   Root Class(#1)
```

Whew, that's a lot! (You'll find that some objects have very long pedigrees.) You could then read the help text for the generic hot air balloon by typing `help #66549` and make an informed decision as to whether you actually wanted to create one for yourself. Or you might decide that instead of a hot air balloon, a flying carpet is more to your taste, in which case the generic aircraft might be a better choice.

LambdaMOO also has a museum, as do some other MOOs. This is a room or set of rooms whose purpose is to provide information about various generic objects that are available, and it is a valuable resource.

Not every item is available as a generic; sometimes a programmer might make an object but want to limit it to being one-of-a-kind. Before people can make their own copies of an object the programmer of that object must make it *fertile*. (The command is `@chmod <object> +f`, but that's properly in a section about programming rather than building. It's included here for completeness.) If an object is not fertile, you can't make a copy of it, even if it has the word "generic" in its name. You are out of luck unless you can persuade the owner to make it fertile.

@kids

This command is the opposite of `@parents`. You use it to list the children of a given object. Note that it does not list kids of kids. Different MOOs have different commands to list all descendents with one single command, but you can always use `@kids` sequentially to explore various branches of the *object hierarchy*.

@audit

It is a common predicament to create an object (or a room, see `@dig`, below), then misplace it and not be able to find it or use it because you've forgotten its object number. For this we have the `@audit` command. There are two forms:

```
@audit
@audit <player>
```

The first audits yourself. It prints out a list of all the objects that you own, along with their object numbers and sizes. The second form prints the same information, except that it lists another player's objects instead of your own. Other players can `@audit` you and see a list of things that you own. Suppose I mislaid my paper weight, and typed `@audit` to find it. I might see something like this:

```
Objects owned by Yib (from #0 to #118569):
 54K  #58337 Yib                        [Yib's Study]
  4K  #23920 Yib's Study
```

```
 <1K  #57744 a walnut desk              [Yib's Study]
  2K  #71354 white dendrobium orchids   [Yib's Study]
  3K  #32504 a linen handkerchief       [Yib]
 <1K  #35487 west                       Yib's Study->*Library Turret
  3K #107539 east                       *Library Turret->Yib's Study
  1K  #71176 a few lucky Bits           [Yib]
  1K   #4612 bright sparkly thing       [a walnut desk]
  3K #101204 a lady's pocket watch     *[Boo]
 <1K  #63555 paper weight              *[Under the Couch Cushions]
-- 11 objects.  Total bytes: 74,927.------------------------------
```

The numbers in the first column are the sizes of the objects in kilobytes of quota. The second column is the object number and name of each thing I own, including myself. The third column tells the name of each object's location (in square brackets); an asterisk(*) indicates that I don't own the indicated location. If the object is an exit, then the third column shows the names of the two rooms each exit connects together, instead. Here you can see that my paper weight has somehow found its way to a place called "Under the Couch Cushions". I can retrieve it easily by typing:

```
@move #63555 to me
```

@dig (rooms)

As mentioned previously, when you `@create` something, its initial `.location` is you – that is to say, you will be holding it right after creating it. In the case of a room, this is awkward, because holding a room doesn't make sense (unless perhaps it's a portable room) and also, you can't teleport into or otherwise enter a room if you are holding it, because it would violate the *containment hierarchy.* (`A` can't be located inside `B` if `B` is located inside `A`). So, for rooms, we use the `@dig` command instead. We also use `@dig` to create the exits which connect two rooms, because it conveniently automates the administrative work of setting the exit's source and destination. Here are some examples of each form:

```
@dig Home Sweet Home
```

The system will respond with, `Home Sweet Home (#113415) created`, thus informing me of the object number of my new room. (Note that the object number of your room will be different.) Now I can teleport there, and describe it. If I ever forget its number, I can always `@audit` myself to find out again.

The location of this new room is `#-1 <$nothing>`. You can think of it floating free in the ether. Until you connect it to another room with exits (see below), you can only get there by teleporting. There is nothing wrong with this, by the way – many rooms are unconnected, and that's just fine.

@dig (exits)

An exit is a special kind of object that connects one room to another. It's special in that it isn't located *in* either of the rooms it connects, but can be referenced by name in the room that is its `.source`. The other room is the exit's `.dest` (think "destination"). To have a two-way connection between a pair of rooms, you have to have two exit objects, one for each direction.

Digging an exit starts in the source room. You can dig exits one at a time, or two at a time (one for each direction), you can specify aliases at the time you `@dig` or add them later, and you can dig to an existing room or create a new room simultaneously. Here are some examples:

```
@dig "east" to #115
```
would dig an exit named "east" to the room with the object number #115.

```
@dig "east","e","out" to #115
```
would dig an exit with the aliases "east", "e", and "out" to the room with object number #115.

```
@dig "east","e","out"|"west","w","in" to #115
```
would dig two exits in opposite directions, connecting room #115 to the room you were in when you typed the `@dig` command. The vertical bar (|) character separates the aliases of the exit going *to* the room from the aliases of the exits leading back *from* the room.

```
@dig "east","e","out" to The Back Porch
```
would create a new room named "`The Back Porch`", and simultaneously create an exit, `east`, from the room where you were to the new room.

```
@dig "east","e","out"|"west","w","in" to The Back Porch
```
would create a new room named "The Back Porch" and exits in both directions connecting the porch to the room you were in when you typed the `@dig` command.

The system will print a line informing you of the object numbers of the new exits (and, if applicable, the new room) that you've created with the `@dig` command. As always, you can also get the object numbers of these newly-created objects using the `@audit` command.

To make your building richer, it's good to describe your exits and set their messages. This is addressed in the section on room integration and exit messages, starting on page 94.

Exit objects must designate the rooms they lead from and to; this is normally done automatically as a side effect of the `@dig` command if you own both rooms. If you don't own the room the exit leads from, the owner of the source room should use the following command to attach it:

```
@add-exit <exit-object>
```

And if you don't own the room that is the exit object's destination, then the owner of the destination room must use the following command to complete the connection:

```
@add-entrance <exit-object>
```

Exit attachment matters more for the source room, since without the attachment, nobody can use the exit. If the exit leads *to* a room that has been set not to allow teleportation via the `.free_entry` property, the exit can't work unless it's attached. The principle, here, is that if a room permits a person to teleport in, then the message generated by an exit is no worse than a teleport message. On the other hand, if the owner of the room won't let you teleport in, then you can't dig your own exit to the place, either. Or rather you can `@dig` the exit, but you can't use it unless the destination's owner "blesses it" with the `@add-entrance` command.

@chparent

The `@chparent` command is typically used to change the parent of an object from its current parent to another generic, usually (though not necessarily) an object of a similar type. One might `@chparent` a room, for example, to a fancier -- or merely different -- room generic. Likewise one might change the parent of an exit to a transparent exit, or a generic gate, and one might `@chparent` oneself to a different player class. For example:

```
@chparent me to #191
```

or:

```
@chparent here to #9805
```

@recreate

The `@recreate` command is basically a combination of `@create` and `@chparent`. When you use `@chparent`, any properties and messages that you've set that are in common with the new parent keep the values you've set them to. If you really want to start over from scratch, but keep the same object number, then `@recreate` is the command to use. The syntax is:

```
@recreate <object> as <parent> named <new name>
```

@set

Some objects permit some customization by letting you set the value of one or more properties which in turn affect the object's behavior. You can, for example, modify the way a room's contents are displayed by setting its `.ctype` property to

different values. To the best of my knowledge, there isn't any documentation about the various possible values of `.ctype`, so I present a rather long illustration here.

```
@set <room>.ctype to 0
```

will list out the room's contents, one item per line, in the order in which the items entered the room:

```
The Front Veranda
A gloriously spacious covered veranda, painted all in white.
To the east, a large door leads into the mansion.  Wide
steps lead west and down to the front lawn.
Contents:
  YibCo Muffle-Matic Soundproof Energy Field
  Rocking Chair
  a small wicker basket
  a white-washed wooden porch swing
  Yib
```

```
@set <room>.cytpe to 1
```

will put all non-player objects into separate sentences of the form, "You see <item> here," on separate lines, and all players into separate sentences of the form "<So-and-so> is here," on separate lines. The order will be the order in which items and players entered the room:

```
The Front Veranda
A gloriously spacious covered veranda, painted all in white.
To the east, a large door leads into the mansion.  Wide
steps lead west and down to the front lawn.
You see YibCo Muffle-Matic Soundproof Energy Field here.
You see Rocking Chair here.
You see a small wicker basket here.
You see a white-washed wooden porch swing here.
Yib is here.
```

```
@set <room>.ctype to 2
```

will list all the contents in a single sentence:

```
The Front Veranda
A gloriously spacious covered veranda, painted all in white.
To the east, a large door leads into the mansion.  Wide
steps lead west and down to the front lawn.
You see YibCo Muffle-Matic Soundproof Energy Field, Rocking
Chair, a small wicker basket, a white-washed wooden porch
swing, and Yib here.
```

```
@set <room>.ctype to 3
```

will list items in one sentence, and players separately, in another:

```
The Front Veranda
A gloriously spacious covered veranda, painted all in white.
```

```
To the east, a large door leads into the mansion.  Wide
steps lead west and down to the front lawn.
You see YibCo Muffle-Matic Soundproof Energy Field, Rocking
Chair, a small wicker basket, and a white-washed wooden
porch swing here.
Yib is here.
```

If you set a room's `.ctype` to anything else, the contents won't display at all, *unless* you have changed its parent to a room generic that supports additional `.ctypes`. (Ideally, the room's help text will mention this. To read a room's help text, type `help here` while in the room or `help <room object number>` if you are not in the room.)

Using @set with Messages

A message is a property on an object that ends with "_msg". These properties are special in that they can be set as described in the larger section on messages (see page 46), and that is the way it is usually done. But message properties can also be set and/or changed using the `@set` command, just like any other property of an object. If you had a paper weight with the alias "pw", then the following two commands would have an identical result:

```
@drop_succeeded pw is "You drop %t.  It lands with a thud,
then rolls a short distance before coming to a stop."

@set pw.drop_succeeded_msg to "You drop %t.  It lands with a
thud, then rolls a short distance before coming to a stop."
```

@contents

One advantage of being a `$builder` is that you don't have to depend on a room's description (or any object's description) to find out what its contents are. You can type:

```
@contents <object>
```

and get a list of its contents by name and number. Like all of the commands presented in this segment, it is a meta-VR command, which crosses the boundary of a MOO's theme into its underlying structure. The trade-off is some of the VR charm for increased informational accuracy.

@lock

As a builder, you can control some of the ways your objects are used. The easiest way is with the `@lock` command. `@lock` works differently with different kinds of

objects, and that can make it seem a bit tricky, but it's easy once you get the hang of it. Briefly, if you lock a room, that governs what can and cannot enter it. If you lock a thing or a container, that governs locations to which it can and cannot be moved. And if you lock an exit, that governs who (or what) can and cannot pass through it. The syntax of the command is:

```
@lock <object> with <key>
```

To use `@lock` effectively, you need to understand the concept of a key. A key is a string of text that represents objects and ways that they can be combined. Briefly, "`&&`" means "and", "`||`" means "or", and "`!`" means "not". These can be combined in various ways. For example, if my object number is #97, and Ostrich's object number is #891, then "`#97 && #891`" means "Yib and Ostrich", "`#97 || #891`" means "Yib or Ostrich", and "`!#97`" means "not Yib". You can combine any number of objects in any number of ways. Use parentheses to clarify complicated expressions. (See also `help locking` and `help keys` online.)

The reverse of locking an item is:

```
@unlock <item>
```

Containers offer the additional option of:

```
@lock-for-open <container> with <key>
```

and:

```
@unlock-for-open <container>
```

This governs who can open a container, as opposed to who may take a container or where a container may be dropped.

@build-options

There is an interface called an *options package* that lets you customize the way some of the building commands work. To list your current option settings type:

```
@build-options
```

Most MOOs support four builder options; this section explains how to set and clear each of them, and what each of them means.

```
@build-option dig_room=<room-generic>
```

When you `@dig` a room, you are creating a child of a particular room. In most cases, this is `$room`, which is the generic room provided by the MOO. After digging a room, you may wish to use the `@chparent` command to select a different room-generic as the room's parent. If you have a favorite room generic and want all the rooms you dig to have that generic as a parent, you can specify that with this option. To clear this option and make it so that all rooms that you subsequently `@dig` use the system default, type `@build-option -dig_room`.

```
@build-option dig_exit=<exit-generic>
```

As with rooms, when you `@dig` an exit, its parent is set to a particular generic exit, usually `$exit`. If you wish to specify a different generic exit as your default, you can use the `dig_exit` builder option to specify a generic, and all exits you `@dig` after that will be kids of the generic you specified. To clear this option and make it so that all exits you subsequently `@dig` use the system default, type: `@build-option -dig_exit`

```
@build-option create_flags=<flags>
```

This option governs the permission settings that will be associated with every object you create. An object can be readable by others or not, writable by others or not, and fertile or not. Readable means that others can list the properties on your object. Writable means that others may add or remove properties and/or verbs from your object. CAUTION! It is almost never a good idea to set an object to be writable. Better to use available facilities or get a wizard to change ownership of an object to a different person if you want to let someone else assist you with your building. Fertile means that other people may make kids of your object. The value for `<flags>` in this builder option can be any substring of `"rwf"`, or it can be the empty string `""`).

```
@build-option -bi_create
@build-option +bi_create
```

When you create a new object, the system will either re-use a previously-recycled object, or it will create a new object with a higher object number than all previously-created objects. It is better for the database if you use recycled objects, which is the "`-bi_create`" option.

@quota

Quota is the term we use to measure the amount of space that objects take up in the computer's memory. Building things takes up space in the database, and players are usually granted a fixed amount of quota to start with. In LambdaMOO's early days, players were allotted a fixed number of objects that they could create. An unforeseen consequence of this was that people programmed fancier and fancier objects which took up more and more space, culminating in the generic multi-room, which was a room that simulated many rooms but which was, in fact, still a single object. LambdaMOO then converted to what is called "byte-based quota". A player may create as many objects as e wishes, except that the total size of all the objects may not exceed a specified limit. (If it does, then the player can't create new objects or add properties to existing objects.) The command:

```
@quota
```

will display how much quota you have used up with the objects you have created, and how much you still have available for creating new objects. You can also type:

```
@quota <player>
```

to see how much quota someone else has available, and how much e has used up.

Different MOOs will have different policies regarding whether they use object-based quota or byte-based quota, how much quota players are allotted when they first register, and how to get more quota. LambdaMOO has an elected Architecture Review Board (ARB) which reviews quota requests against certain criteria; in addition, LambdaMOO players can transfer quota directly to one another. On other MOOs, wizards will set quota policy.

Trimming Down Your (Quota) Size

@measure

The amount of quota an object takes up can change. Consider a note that has only a few lines of text in it. Then its owner edits it to be an extremely long note. Now it takes up more quota. The system has a single object-measurement task, and individual objects are generally measured only once every few days. Thus, the information provided by `@quota` may not be up-to-date. This shouldn't matter unless you are over quota and are trying to "slim down", as we say, perhaps in order to be able to create another object. In addition, you are only permitted to have a certain maximum number of unmeasured objects at a time (ten, on LambdaMOO), and after that you may not create more objects until the new objects have been measured. The `@measure` command, in its several variations, is provided so that players don't have to wait for the automatic measurement task to run if they need or want to have an object measured sooner than that. The tradeoff is that measuring things takes up computational resources, contributing to lag, and players are asked to use this command sparingly.

```
@measure object <object>
```

This command is used to measure a single object at a time, in lieu of waiting for the background measurement task to get to it.

The first step for trimming down is to use the `@rmm` command to remove any MOOmail messages that you don't need to keep. (You can use the `@netforward` command to forward messages to your registration email address before removing them, too.) These removed messages are not entirely gone, yet. The next step is to expunge the removed messages. You can do that in either of the following ways: `@renumber me` renumbers all your MOOmail messages and expunges deleted messages. `@unrmm expunge on me` expunges removed messages without renumbering them. Last, type `@measure object me` and `@measure summary`, to measure and record your new (smaller) size.

```
@measure summary
@measure summary <player>
```

The `@quota` command does not itself measure objects. Rather, it prints out summary information that was computed at an earlier time. `@measure summary` will tally up the current total of your object sizes for reporting by the `@quota` command.

```
@measure new
@measure new [<player>]
```

In a MOO that uses byte-based quota, you can only have a fixed number of unmeasured objects. After that, you can't create any new objects until the current ones have been measured. This can be a problem if you need to create a large number of small objects. They don't take up more than your allotment of quota, but still you can't `@create` more until they've been measured. The symptom of this particular problem is an error message that reads, "`Resource limit exceeded`." `@measure new` alleviates this. Once the new objects have been measured, you can go on to create more if you want to. If you are assisting someone with this dilemma, you can type `@measure new <person>` to measure that person's new objects instead of your own.

```
@measure recent [<number of days>] [<player>]
```

This measures those objects (yours or the specified player's) that haven't been measured either in the specified number of days, or, if no number of days is specified, the ordinary cycle of the measurement task. If no player is specified, then it measures your objects.

```
@measure breakdown <object>
```

If you just can't figure out why an object is SO BIG, the `@measure breakdown` command will print a list of how much space each property and verb takes up, and, hopefully, provide you with some clues. You can optionally have the output sent to you via MOOmail, but be aware that the message itself takes up quota.

@newmessage, @unmessage

In general, only programmers can add properties to and remove properties from an object, but builders can add and remove message properties. This is a pretty obscure aspect of building, and I'm only going to treat it briefly. There are a very few occasions where different things will happen depending on the presence or absence of a certain message properties. Some room descriptions will incorporate an object's `.look_msg` property, if present. If an object has a `.carried_msg` property, some player classes will integrate the message into a player's description.

A builder who does not have programming privileges can still add a message property as follows:

```
@newmessage <message-name> [<message>] [on <object>]
```

and remove a message from an object if it is no longer wanted or needed:

```
@unmessage <message-name> [from <object>]
```

@check-chparent

This command would be useful if you had added a message to an object, then tried to change the parent of that object to a generic that also had the same message. An object can define a property or inherit it, but not both. Programmers probably find more use for this verb than builders.

Creating a Mailing List

Building a mailing list is a popular thing to do, but requires extra steps so others besides yourself can use your mailing list, too. First, create the list, using the generic mail recipient as a parent:

```
@create $mail_recipient named <new list name>
```

Next, give your list a description, explaining what its intended topic is:

```
@describe <your list> as "<description>"
```

Then, to make it a public list that anyone can read and to which anyone can post, do:

```
@set <your list>.readers to 1
```

Last, to make your mailing list publicly available so that people can subscribe to it, do:

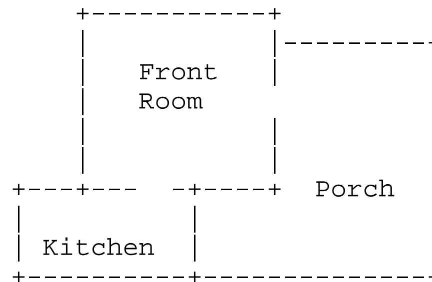
```
@move <your list> to $mail_agent
```

Note that $mail_agent won't accept a list that lacks a description. See also `help $mail_recipient`.

Room Integration and Exit Messages

Integrating objects into a room's description, adding messages to exits, and (to a lesser extent) describing exits can enrich a user's VR experience at the expense of relatively little effort on the part of a builder. This extended example illustrates each of these techniques.

Suppose you have a front room and a porch, situated east-west relative to each other, and a screen door in-between:




```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.  A small kitchen is to the
south.  There is a screen door to the east.


Porch
You are on a breezy, screened-in porch.  A rocking chair and
a porch swing invite you to stay and relax for a while.  A
screen door leads west into the cottage, steps lead down to
the lawn.
```

Here are the steps you would follow to make an integrating room with exit messages:

1. Start by making the rooms' descriptions clearly mention the obvious exits, so that people don't have to guess or use meta-VR commands such as `@ways`, since this is intended to be a welcoming place for people to visit, and not a puzzle.

2. Describe the exits. This means describing what someone would see if they looked in the direction of the exit, for example, `look east`.

Stand in the front room and type:

```
@describe east as "You see a weathered but sturdy screen
door, held closed by a spring.  The top screen has a small
tear in the lower left-hand corner."
```

Stand on the porch and type:

```
@describe west as "You see a weathered but sturdy screen
door.  The handle and hinges are rusty but serviceable.  The
top screen has a small tear in the lower right-hand corner."
```

Notice that in writing these descriptions, I have implicitly decided that the door opens outwards onto the porch -- that's the side that the hinges are on. We can use this detail later to intensify the "VR feel" of things. The descriptions don't have to be elaborate, but it's nice if they add some new information to what's already there in the rooms' descriptions.

3. (Optional, but nice) Use `look_msgs` for the exits' descriptions instead of describing them in the room's description proper. (Yes, this contradicts step 1.) The reason for doing this is so that the exits will consistently be mentioned at the end of the description, no matter how many other objects' `.look_msgs` are included.

Some rooms can integrate objects and exits into their descriptions.[15] By convention (on LambdaMOO and YibMOO at least), an integrating room checks to see which objects and exits have a `.look_msg` property and/or a `:look_msg` verb, and, if so, incorporates those messages into the description instead of baldly listing the them in the room's contents afterwards (in the case of objects).

Why bother with this? Lets start by redescribing our Front Room, which had exits leading southwest to the kitchen, stairs leading up, and our east exit onto the porch. And let's put in a fireplace object, so we can see how the `.look_msg` properties interact. Starting with no `look_msg` properties on any objects or exits:

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.
You see fireplace here.

@prop south.look_msg "A small kitchen is to the south"
@prop east.look_msg "There is a screen door to the east."
```

Now if we were to look at the room, we'd see this:

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.  A small kitchen is to the
south.  There is a screen door to the east.
You see fireplace here.
```

[15] One integrating room generic on LambdaMOO is #17755 (Integrating Detail Room Mark III). See also #9805. On YibMOO, you can use the YibCo(tm) Multi-Media Modular Room (#237) in conjunction with the Integrating Description Module (#259).

This is strikingly like what we had before, but watch this. Now we'll put a `.look_msg` on the fireplace, too:

```
@prop fireplace.look_msg "Against the west wall is a large
stone fireplace."
```

Now the description becomes:

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.  Against the west wall is a
large stone fireplace.  A small kitchen is to the south.
There is a screen door to the east.
```

We could add a painting (assuming you have an object that is a painting):

```
@prop painting.look_msg "A portrait of someone, vaguely
familiar, hangs on the north wall."
```

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.  Against the west wall is a
large stone fireplace.  A portrait of someone, vaguely
familiar, hangs on the north wall.  A small kitchen is to
the south.  There is a screen door to the east.
```

The beauty of it is that you can integrate any number of objects into the middle of the description, and still have the exits described at the end, where they are easy to find.

4. Give the exits messages. Exit messages govern what the player sees and what others see when a person goes through the exit. There are six to set (besides `.look_msg`):

`@leave <exit>`	This is what the player sees when e leaves by an exit.
`@oleave <exit>`	This is what other people in the room see when a player leaves by an exit.
`@arrive <exit>`	This is what the player sees after passing through the exit and arriving at the new location.
`@oarrive <exit>`	This is what others at the destination see when the player arrives.
`@nogo <exit>`	This is what a player sees if he can't get through the exit for any reason.
`@onogo <exit>`	This is what other people in the room see if a player tries an exit but can't get through.

Pronoun and verb substitutions are addressed at length in the programming tutorial (see page 108), or you can read the online help text in help pronouns.

Briefly, `%n` substitutes the name of the player, `%s` is the subject pronoun (he/she/e/etc.), `%p` is the possessive pronoun (his/her/eir/etc.), `%r` is the reflexive pronoun (himself/herself/emself/etc.), `%o` is the object pronoun (him/her/em/etc.). These are the ones most commonly used for exit messages. There are others. Third person singular verbs, when surrounded by angle brackets (`<>`) and preceded by the percent sign (`%`) will agree with the gender of the player. Specifically, the messages will print correctly even if player's gender is set to plural. For example:

```
%N %<goes> through the door.
```

would yield:

```
Yib goes through the door.
```

but:

```
Bits go through the door.
```

Let's start with the east exit from the front room to the porch, then do the west exit from the porch to the front room. It's easiest to add the messages if you are in the room that is the exit's source, rather than the destination. (If you are elsewhere, you'll have to refer to the exits by their object numbers rather than by name.)

Starting in the Front Room:

```
@leave east is "You push open the screen door and head out
to the porch."
@oleave east is "%N %<pushes> open the screen door to the
east and %<heads> out to the porch.  The door slams shut
behind %o."
@arrive east is "The screen door slams shut behind you with
a bang."
@oarrive east is "%N %<comes> out through the screen door to
the west.  The door slams shut behind %o."
@nogo east is "You push on the screen door, but someone
seems to have nailed it shut."
@onogo east is "%N %<pushes> on the screen door, but someone
seems to have nailed it shut."
```

Now for messages on the exit going the other way. From the Porch:

```
@leave west is "You pull open the screen door and head into
the cottage."
@oleave west is "%N %<pulls> open the screen door to the
west and %<goes> inside.  You wince as the door slams shut
behind %o."
@arrive west is "The door slams shut behind you."
@oarrive west is "%N %<comes> in through the door to the
east.  You wince as it slams shut behind %o."
@nogo west is "You try the door, but it seems to be nailed
shut."
@nogo west is "%N %<tries> the screen door, but it seems to
be nailed shut."
```

A few comments:

It's good for the oleave (and sometimes oarrive) messages to mention the compass direction (if there is one). This helps others keep their bearings of where they are and what's beyond. It's especially good for when one player is following another.

Often it's sufficient to set only the leave message or only the arrive message, rather than both, such as when a player is going through an open doorway, for example. In the above example, I used the screen door slamming for the arrive message, to add to the effect.

If the exit is likely never to be locked or otherwise impassable, it's acceptable to omit the `nogo` and `onogo` messages.

Give some thought to who hears/sees what. I wanted to embellish the feel of the door slamming, and did that by having people wince, but didn't want to bombard them with it by using it in every single message. I chose to have people on either side of the door wince when someone goes in, but no one wincing when someone goes out. The choice was arbitrary, but deliberate. Paying attention to small details will give your work a richness that it might otherwise lack.

This may seem like a lot of work on top of `@dig` (voilà, you have an exit), but exit descriptions and messages give areas on the MOO a much stronger VR feel, and make any area more fun and interesting to explore. The messages don't have to be fancy, but they should be appropriate to the situation

Determining a Room or Object's Contents Definitively

Let's revisit the fact that every object has a property called `.contents`, which is either a list of object numbers or the empty list. Each of those listed objects will reciprocally have this object in its `.location` property.

Normally, when you look at a player, you will be presented with a list of things e is carrying, and when you look at a room, you will be told about things you see there. There are ways, however, for a programmer to camouflage or conceal what is in a room, container or player, and there are ways to circumvent such programming. First I will discuss camouflaging, then circumvention.

Integration – The basic room class shows you the room's description, then lists the non-player objects in it, then the players present:

```
The Conservatory
You are in a glass-sided room filled with orchids,
bromeliads, and other tropical plants.
You see trowel here.
Gardener is here.
```

A room class that supports object integration, on the other hand, will check for a special property on the objects within it and integrate that text into the description itself. The property usually has the name `.look_msg`. If our conservatory were an

integrating room, and if the trowel had a `.look_msg` saying, "Off to one side is a trowel," then the room's description would look like this:

```
The Conservatory
You are in a glass-sided room filled with orchids,
bromeliads, and other tropical plants.  Off to one side is a
trowel.
Gardener is here.
```

Integrating objects into a room's description has advantages and disadvantages. The main advantage is that the text is more pleasing to read, and many builders choose integration for this reason, rather than from any intent to deceive. The disadvantage is that the hapless explorer might now overlook the fact that the trowel is an object (which might be interactive or relevant to the scene at hand) *or* would have to spend time trying to examine the orchids and bromeliads which are merely a part of the description and not actual objects. A practiced MOOer *might* realize that in integrated rooms there is typically one sentence per integrated object, and therefore guess that "orchids, bromeliads, and other tropical plants" are so-called *tiny scenery* and that the trowel is a bona fide object, but there is no reliable way to tell for sure just by looking.

Darkness – Rooms have a `.dark` property, which, if set to a non-zero value suppresses the display of the room's contents. Such a room might have a clue in its description about how to turn on the light, such as, `"As you grope around in the dark, your hand encounters a string,"` that, when pulled, turns on a light.[16]

The way to circumvent such techniques is to use the `@contents` verb. It isn't as pretty, but it's useful if you're on the prowl for objects that might do something. Consider this example:

```
Undertaker's Cottage
This front room of the cottage is reminiscent of an old-
fashioned parlor, the kind one never actually went into.  At
one end, an overstuffed couch, at the other a stone
fireplace.  In between, six French Empire chairs, facing
each other gloomily, three and three.  Off in a corner sits
an ancient pump organ.  On one of the walls is a collection
```

[16] Here's a bit of behind-the-scenes technology. It may not be of immediate interest if you are a beginner (and you can safely skip over it if it doesn't make sense right now), but it may be of interest further down the road, or if you are an intermediate-level MOOer.

When you enter a room, that room's `:look_self` verb is called. This verb in turn calls the room's `:description` verb which assembles the text which will be displayed as the description, and it also calls the room's `:tell_contents` verb which does the actual work of printing out what objects and players you "see" when you enter or look at the room. The `:tell_contents` verb calls the `:contents` verb which *usually* returns the value of the room's `.contents` property BUT a room's owner, can, if e wishes, cause the `:contents` verb or the `:tell_contents` verb to give incomplete or spurious information. In the case of a room with integrated objects, the `:description` verb tells you more (the integrated objects are integrated) and the `:tell_contents` verb tells you less (integrated objects are omitted, so as not to be mentioned redundantly). So in the quest for more pleasant prose, information is regrettably lost (i.e. which nouns in the description represent actual objects of possible interest).

```
of portraits.  Doors to the northeast and southwest stand
slightly ajar, as if someone were beyond them, watching...
or waiting.  Everything is covered with a layer of dust.
Cold stone steps lead down into darkness.
You see The Undertaker and Epitaph Registry here.
Yib is here.

@contents here
Undertaker's Cottage(#101792) contains:
The Undertaker(#666)   a fireplace(#78070)   pump
organ(#73881)   Collection of portraits(#14627)   Epitaph
Registry(#15588)   Yib(#58337)
```

If you were exploring this room to see what might be done here, you would examine the undertaker, the fireplace, the pump organ, the collection of portraits, and the epitaph registry. You wouldn't bother with the overstuffed couch, the empire chairs, the dust or the cold stone steps.

To summarize, then, objects can contain other objects. The contained objects are stored as a list of object numbers in a property named `.contents`. There is a `:contents` verb which usually returns a complete list of an object's contents, but which can also be programmed not to do so. The `@contents` command displays a definitive list of an object or room's contents, irrespective of other programming.

Chapter 6 – Programming

A Brief Overview of What it Is and How it All Works

Building transcends the VR in that when you `@create` objects and `@dig` rooms you are using the system *as* a computer system rather than acting strictly within the bounds of the MOO's frame story or virtual reality. Programming takes things one step further, in that you create new and original ways for objects to behave. This section presents an abstract overview of how that process works. The following section, "Yib's Pet Rock", is a hands-on tutorial.

First Principles

I take on faith the mechanics of how text travels from your keyboard to the MOO and from the MOO to your screen, and ask you to do so as well. At some later time you may wish to investigate these for yourself, but they are beyond the scope of this book.

The MOO is made of two principal parts, the *server* and the *database*.

The Server

The *server* is the program that runs on the MOO's host machine. It accepts new connections, interprets the *commands* you type, causes various things to happen in the database because of what you type, and causes some text *output* to appear on your screen. A *command* is precisely a line of text that you type with the intention of getting a response from the MOO. This cycle, you typing a command, the server executing it, and output (usually) displaying on your screen is referred to as a *task*, or, more accurately, a *foreground task.*

The Database

The *database* is the entirety of all the *objects* on the MOO, including all their *properties* and *verbs*. (The *core database* is the database that is there when a new MOO first starts, before any new objects, properties, or verbs have been added.) A *property* is a named piece of data associated with a particular object. Where do properties come from? They are added to objects on an as-needed basis by programmers, using the `@property` command. A *verb* is a named sequence of instructions that the server carries out (or executes). On MOOs, the terms *program*, *command*, and *verb* are often used interchangeably.

The Parser

A command consists of one or more words that you type, separated by spaces. One of the things that the server does is analyze (parse) the line of text that you type, and try to identify which verb on which object it should execute. The part of the server that does this is called the *parser*. The first word of the command you type is the name of the verb. The rest the words you type (if any) are called *arguments*, which are items of information that the verb needs in order to work properly. Let's consider a few example commands and talk about their arguments:

```
take rabbit from hat
put hat on table
pet rabbit
page help I'm trying to understand parsing, can anyone
explain it to me?
home
```

For every command, the parser tries to identify an object with a verb of that name on it that it can run, and it considers sets of objects and their associated verbs in a particular sequence. This sequence is the player emself, any feature objects that the player has, the room the player is in, the direct object of the command, and the indirect object. Looking at these examples, you might intuitively figure out that if there is a `hat` nearby, with a verb that lets one `take` things `from` it, that might be an appropriate verb to run, and you would probably be right. Then, if there is a `table` in the vicinity, and a verb that lets one `put` things on it, that might be an appropriate choice, and so on. Some commands, like `page` and `home` don't take direct objects, prepositions, and indirect objects. They take other items of information instead, or no information.

When a programmer creates a verb, e must specify what arguments (if any) the verb uses. These are called *argument specifiers*. When the parser identifies an object and a verb with argument specifiers that are appropriate to the command that was typed in, we say that it *matches* the object or *matches* the verb on the object.

If the parser can't identify an object with an appropriate verb to run, the server sends the following text to your screen:

```
I don't understand that.
```

Tasks

As mentioned above, the cycle of your typing in a command, the parser matching an object and a verb to run, and then output (usually) appearing on your screen is called a *foreground task*. There are three basic kinds of things that any task can do: It can send information (text) to be displayed on your screen. It can modify the database in one or more ways, including changing the values of properties or even creating new verbs to run in future tasks. And *it can start up another task* that does something else, either later or at the same time, but independently. These tasks are called *background tasks.* They can do the same three things that foreground tasks

do, including starting up additional background tasks. Every background task has a unique numerical identification number called its `task_id`. A list of background tasks (identified by `task_id`) that are scheduled to run at a later time is called a *queue*.

How Are Properties and Verbs Created?

There are two commands that are fundamental to the programming process, and these are `@property` and `@verb`. Like any other command, someone types them in (with some arguments), the parser figures out which object is to receive the new property or verb, and then the server runs the verb that causes new properties or verbs to be added to an object.

The `@property` Command

The syntax for adding a new property to an object is:

```
@property <object>.<property name> <initial value>
[<permission flags> [<owner>]]
```

The property name can be anything you want except that it may not contain any spaces. The initial value can be anything you want, but text should be enclosed within double quotes (`""`). The permission flags can be any combination of the letters "`r`", "`w`", and "`c`". They govern who else may access those properties. (A *flag* is a tiny bit of data, usually stored as a `1` or a `0` (often though not always signifying "yes" or "no") within a larger piece of data.) If you include the letter "`r`" in the permission flags, then anyone may read the value of this property, and anyone's verb may access and use the value of this property. If you include the letter "`w`" in the permission flags, then anyone may change the value of the property, and anyone's verb may change the value of the property. The "`w`" flag is hardly ever used; there are safer ways to permit others to vary the value of a property in limited ways that you control. The "`c`" flag controls who is allowed to change the value of the property in the case that someone else makes a child of your object. If you include "`c`" in the permission flags, then the owner of the child object can change it, and your verbs can't change it. If you don't include "`c`" in the permission flags, then your verbs can change the property's value, even on child objects owned by others, but the owners of the child objects can't change the value of the property directly. This is a concept that many people wrestle with, so don't be discouraged if it doesn't make sense right away. It's mentioned several times in the "Yib's Pet Rock" tutorial (page 108), and explained again in the programming reference section (page 156).

When a letter is included as a permission flag, we say that that flag is *set*. When a letter is omitted from the permission flags, we say that that flag is *clear*. For example, the "`r`" flag is usually set, and the "`w`" flag should almost always be clear.

The `@verb` Command

The syntax for adding a new verb to an object is:

```
@verb <object>:<verb name> <direct object specifier>
<preposition specifier> <indirect object specifier>
[<permission flags> [<owner>]]
```

The verb name can be anything you want except that it may not contain any spaces and should not begin with the asterisk character (`*`). The argument specifiers are three generalized expressions of the direct object, preposition, and indirect object that are used by the parser when trying to match a verb to run. The direct object specifier and indirect object specifier can be either `this` or `none` or `any`. The preposition specifier may be either `any`, `none`, or one of the list of permissible values (such as `in` or `on`) given in the Programmer's Reference Manual and the programming reference section (see page 163). The permission flags for a verb can be any combination of the letters "`r`", "`x`", or "`d`". If the "`r`" flag is set, then others may read the verb. If the "`x`" flag is set, then other verbs may use this verb as an intermediate step in their own execution. The "`d`" flag is obsolete but should always be set; it used to govern whether an error, if one was encountered, should cause the verb to cease executing immediately and produce a traceback or be ignored. A later version of the server provided other ways to handle error conditions without causing tracebacks; nevertheless, the programmer's manual indicates that while obsolete, the "`d`" flag should always be set[17]. A wizard can set the owner of the verb to be someone other than emself.

Here's an example of adding a new verb to an object:

```
@verb collage:paste any onto this rxd
```

(Such a verb might be used to program a collage object so that you could paste anything onto it to create a work of art.)

'Round and 'Round We Go...

After a verb is added to an object, a programmer then sets to programming it, i.e. specifying, in terms the server can understand, just what it is that the verb is to do, either with the `@program` command (explained in the programming tutorial, page 108) or using the verb editor (explained in the section on using the in-MOO editors). Programming is an "iterative process", which means that it usually takes several tries before a verb works just the way it was originally intended to.

---

[17] Pavel Curtis, The LambdaMOO Programmer's Manual, section 2.2.3.

The Nitty-Gritty: What Goes On Inside All Those Verbs?

In a nutshell, verbs start up, process data, and then finish.

Starting Up

When you type a command, the first word of what you type is the name of a verb, and you are said to be "invoking that verb from the command line". Sometimes these verbs are called *command-line verbs*. Other verbs, though, are *only* meant to be invoked (or *called*) from within other verbs – they perform some intermediate function and return a result, which the calling verb then uses as if the function had been written into the calling verb itself. These verbs, called from other verbs, are called *subroutines*. Regardless of whether a verb is a command-line verb or a subroutine, all verbs do some initial start up processing when they are called or invoked, and this consists of setting up some *variables*.

A *variable* is like a property in that it is a named piece of data. Unlike a property, however, a variable only exists and has meaning while a verb is running. Also unlike properties, variables aren't stored with objects – so they can't be accessed by other players or other verbs. We say that they are "internal to the verb" or *local*, whereas properties are "external to the verb" or *global*. In the MOO programming language, variables are said to be *dynamically allocated*, which is a fancy way of saying that as soon as a line of MOO-code assigns a value to a variable, voilà! that variable comes into being and contains the value that the verb just assigned to it.

There are different *kinds* of values that variables can hold, and in the computer world, "kinds of values" are referred to as *data types*. In some programming languages, you have to specify at the beginning of a program what variables will be used and what kind of data each will hold. A counter, for example, might be of type *integer*, while a variable intended to hold a person's name would be of type *character string*. In the MOO programming language, you don't have to declare in advance what type of data a variable will hold, and a variable can hold different types of data at different times. There is a way to ascertain what type of data a variable is holding at any particular time, if one needs to know that.

When a verb is first invoked, certain variables are automatically created right away, and are assigned values before anything else happens. These are called *built-in variables.* The data these variables hold are always available for use within the body of the verb itself. They include the object number of the player who typed the command, the direct object (if any), the indirect object (if any), and a special variable called `args`, which holds a list of any other pieces of information the verb or subroutine needs to do its work – in other words, the arguments. For a command line verb, the value of the variable `args` is a list of just those things the player typed – the direct object, the preposition, and the indirect object, or the content of a paged message, for example. Subroutines may need other pieces of information, however. If a subroutine's job is to take a list of numbers and sort them, for example, then it needs to be told what numbers to sort, and that's what would be in its `args` variable.

It is the job of the calling verb to send the right arguments to a subroutine so that the subroutine can do its job correctly.

Processing Data – The Very Stuff

The basic things that verbs do are:

- Change (directly or indirectly) the values of properties on objects in the database.
- Send information to be displayed on someone's screen (or several people's screens).
- Calculate intermediate results from given information and store them in variables. (The "given information" is received by the verb in the built-in variable `args`, which is sometimes also called the *argument list*.)

How is all this done? The server evaluates a sequence of *expressions*. An *expression* is a combination of letters, numbers, punctuation marks and white space which, when evaluated, generates a value. The value of an expression can then be assigned to a variable, or stored in a property, or ignored. Why would a value be ignored? Some expressions have *side effects*, which are actions that occur as a result of evaluating the expression. An example of this would be displaying some text on a player's screen. If all you care about is an expression's side effect(s), then you don't need to store or otherwise pay attention to its value, even though it has one.

The Finish

Calls to verbs are themselves expressions. When all the expressions within a verb have been evaluated, then the verb is said to *terminate*. Any variables that the verb used are removed from the computer's memory, and a value, the final value of the verb, is *returned*, either to the command line or to the verb that called it. If the verb was called from the command line, its return value is ignored. If the verb was called as a subroutine, then its return value may be ignored, or it may be used as a component of a more complex expression.

In Conclusion

The substance of any programmer's manual or programming language reference is an enumeration of the kinds of expressions that are available, what each one does, and (depending on how detailed the reference is) a synopsis of how to use them. Looking at a programming reference can seem daunting, at first, but it isn't an all-or-nothing proposition. If you know a few simple kinds of expressions, then you can write a few simple programs. If you know a wide variety of expressions, then you can write a wide variety of programs, and everything in between. Virtuoso programmers amass a knowledge of expressions and available subroutines the way master chefs

amass a knowledge of ingredients. Anyone who can read can cook, but the more you know, the more you can do.

Yib's Pet Rock: A Programming Tutorial for Beginners

Introduction

This tutorial was originally written on and for LambdaMOO. The overall objective is to give you a footing in the MOO programming environment and a feel for what the programming process is really like. I will take you through the steps of creating and programming a few objects to a fair degree of sophistication. Without the interaction afforded by a classroom setting (or a MOO!), it's all but impossible to avoid your doing at least some of the project by rote, but I hope you'll gain an intuitive sense of at least some of it as you go along. I go light on structured presentations and explain things as and when we need them, and (I hope) just enough so that you can get a handle on the concepts, but not so much that they distract from the project at hand. The point is not to teach each and every nuance in painstaking detail, but rather to give you some exposure to the *kinds* of things that can be done and how to begin doing them.

The MOO Programming Reference section that begins on page 156 is a more structured presentation, and those who prefer to begin with an overview should skip ahead and come back to this chapter afterwards. Both styles of presentation work in concert. A few things mentioned earlier in the book are repeated here, for review and for the benefit of those who may have skipped ahead to this chapter.

I will explain how to inspect the code on an object (yours or someone else's), because this is one of the major methods of expanding one's existing knowledge and horizons. I will suggest a polite way to ask for help from more experienced players.

Nobody writes bug-free code, including this author. If you write your code thinking that it will work perfectly the first (or second) time, you are setting yourself up for disappointment. Debugging is an adventure; it's detective work. If you find a bug, celebrate! You're that much closer to fixing it and moving on to the next one.

I hope to demonstrate what I consider to be good programming style, and to point out ways to improve the inherent quality of your objects by making them robust. It's one thing to whip up a prototype that works, and quite another thing to make a finely-crafted object that is easy for others to understand and use. Although we will be making fairly simple objects, what you learn in their making will transfer to making larger, more complex objects. Polishing is an important part of that process.

Last, by showing you a variety of tools and how to use them, I hope to launch you on a journey that is as much fun for you as mine has been for me.

Preamble and Prerequisites

If you don't already have one, you will need a programmer bit. There are several ways to get one, though in general you have to get one from a wizard. On some MOOs, the process of getting a programmer bit is automated. One common way is to

give yourself a gender and description, then send mail to `*wizards` containing the text, "May I please have a programmer bit?"

While this tutorial *can* be done without a copy of the Programmer's Manual, you'll get much more out of it if you have one. The programmer's manual is ordinarily available by FTP from ftp.lambda.moo.mud.org. The file names are:

```
pub/MOO/ProgrammersManual.txt
pub/MOO/ProgrammersManual.ps.Z.
```

The `.txt` one is readable as plain text. The `.ps` one is for printing on a PostScript® printer. The `.Z` means the file has been compressed (use `uncompress` to decode it). (How to use FTP is beyond the scope of this book.)

You will need to know how to edit things on the MOO, which is addressed in Chapter 4 – Using the Mail System and the Editors.

If you find this tutorial too laborious, and/or if you want to go on to do another supervised project after those presented here, see also yduJ's Wind-up Duck tutorial, which is found in the Library on LambdaMOO.

Some Conventions

Typically, though not universally, commands that call attention to the fact that we're working on a computer begin with an "`@`" sign. `@edit` is one, `@who` is another. Commands that are consistent with the Virtual Reality usually don't begin with an "`@`" sign, such as `look` or `drop`.

Angle brackets "`< >`" indicate a place-holder for an actual value that you must supply at the time. For example, if I instruct you to type `examine <object>`, you will type `examine rock` or `examine tree` or `examine book` depending on the actual object of interest. If you are holding an object or if an object is in the same room with you, then you can refer to the object by name. If you aren't holding or with an object, you can refer to it remotely by using its object number instead. One way to find out an object's number is to examine it. You can always get a list of objects you own, and their numbers, by typing `@audit me`. You can also type `@audit <player>` to see a list of objects that someone else owns.

It's nice when objects have help text, and the ones we make will. But if an object doesn't have help text, the system will tell you to try `examine <object>`. Some people prefer `@examine <object>` instead. What's the difference? `@examine <object>` will show you *all* the verbs associated with an object, whether they're meant for you or not. Some people prefer this completeness. `Examine <object>` can be tailored by the object's owner so that only relevant verbs appear, or so that all the verbs appear in a more logical order, and it's a more polished look. I will teach you how to tailor what someone sees when e examines an object you have programmed.

Asterisks in verb names: If you type `examine $thing`, you will see:

```
generic thing (aka #5 and generic thing)
Owned by Haakon.
(No description set.)
Obvious verbs:
g*et/t*ake $thing
d*rop/th*row $thing
gi*ve/ha*nd $thing to <anything>
```

The asterisks indicate ways in which you can abbreviate a verb when typing it. So `get $thing` and `g $thing` do the same thing. In a room, you can type `look` or `l` to the same effect. You may put asterisks in your verb names or leave them out. Does it matter? Sometimes. If you're going to embellish an existing verb, then you have to type in the verb name as given. More on this later.

A quick note about formatting. The MOO programming language will let you separate its different elements with any combination of spaces, tabs, and line breaks except that a quoted string, such as `"You have to be holding that to use it,"` may not have a line break in it. (Any line breaks within quoted strings in this tutorial are artifacts of typography.) When typing in commands or lines of MOO code, type in quoted strings as one long line, even if they are longer than a physical line on your screen.

Let's Go!

Well! Let's stop beating around the bush, and make something already! We're going to make a pet rock. Pet rocks don't do much, but it's good to start small. Think of it as a stepping stone to bigger and better things. Type in the following (as one long line):

```
@create $thing named "<your name>'s pet rock","pet
rock","rock","pr"
```

Whew. Let's look at that. `@create` is the command for making a new object What about that "$" sign in `$thing`? Some objects are so basic to the system that they have sort-of universal names, and we never have to remember their object numbers.[18]

When you created the rock, you gave it a list of names, in double quotes, separated by commas. The first one is its actual name. The others are aliases, and you can use any of them to refer to your rock if you are holding it or in the same room with it. If you want to change or adjust the name or aliases, see `help @rename`, `help @addalias` and/or `help @rmalias`. See also `help @create` for all the low-down nitty-gritty on creating things. If you wanted to, you could give it only a name, and no aliases, or you could name it Malcolm, or Fred, or whatever you like.

[18] Where do they come from? They're designated by wizards on object #0. If you were to look at #0.thing you would see #5 (generic thing). So `$thing` means `#0.thing`, and `$note` means `#0.note`, and so on.

Programming is about making choices, and it's the choices you make that make your programming yours.

You also don't have to put quotation marks around the aliases. (I do it out of habit – the two are equivalent.) Speaking of quotation marks, if you want to include the quotation mark character in a string, precede it with the backslash character "\". Names and aliases can't contain commas.

Well, at the moment, our rock doesn't look like much. Let's give it a description. I did mine this way:

```
@describe rock as "A small rock.  It looks friendly, but
doesn't do much."
```

Okay, let's cut to the chase! Time to put a verb on that rock:

```
@verb rock:pet this none none rxd
```

The `@verb` command is how we add commands to objects. Some are "obvious verbs" that show up in `examine`, and others aren't. If you examine your rock now, you won't see the verb you just added. And if you type `pet rock`, the system will respond with `I don't understand that.` (Try it yourself, just to be sure.) That's because we haven't programmed the verb, yet. But hang on, here we go.

There are two ways to accomplish this. One is to type in the verb all at one go, as follows:

```
@program rock:pet
player:tell("You pet the rock.  Nothing happens.");
.
```

Notice the period on a line by itself at the end. That's an essential part.

The other way to program the verb is to use the verb editor, like this:

```
@edit rock:pet
enter
player:tell("You pet the rock.  Nothing happens.");
.
compile
done
```

You should type in each line exactly as given. The semicolons at the end of some lines are part of the program, for example, and the word `enter` is an editor command.

If you get a compiler error message, don't panic. You probably made a typing mistake. See which line it had trouble with (there aren't a lot of choices, with this verb, but there will be, later), and check for a missing semi-colon, mismatched quote-marks, a period instead of a colon, etc. *How* do you check? With compiler errors, you have to do it by using verb editor as shown in the second example above. Type in your verb. Type `compile`. When you get the compiler error, type `list` or `list 1-$` to see the lines listed with their associated numbers. Lines are delimited by semicolons; if you leave one out, the compiler may report the problem as being on the prior or subsequent line than the one the error is actually on. So for an error on a

given line, also check the lines before and after the one that the compiler claims is problematic.

Now we're rolling! Examine the rock. Pet the rock. Celebrate!

Got a traceback? Don't panic. Read it, see which line is problematic, then check that line and any lines near it. Tracebacks are *good* (in an intermediate sort of way). They help us find and fix bugs faster.

To fix a program (or change it), you can either type in the corrected version from scratch (`@program rock:pet`) or use the verb editor (`@edit rock:pet`). Note that once you have created a particular verb with the `@verb` command, don't use `@verb` to add it again. That's a once-only operation. Just change it with `@program` or `@edit`.

But What Did I Just Do?

Before we move on, I want to explain a bit more each of the lines I just had you type in.

```
@verb rock:pet this none none rxd
```

`@Verb` tells the system to add a verb to an object. Let's look at `rock:pet`. "`Rock`" identifies the object. "`:`" signifies a verb, as opposed to a property. "`Pet`" is the name of the verb. The words "`this none none`" are the *argument specifiers* for the verb, and express in a formal way how the command will be typed in by the user. In this case, we want the command to be `pet rock` with no preposition or indirect object. "`rxd`" means that it's readable by others (`r`), callable by other verbs (`x`) (think "eXecutable"), and will generate a nice, informative traceback (`d`) (think "debug") if something goes wrong.

```
player:tell("Nothing happens.");
```

`Player` is a built-in variable that always refers to the player who typed in the command. `:tell` is one way to cause text to be displayed on someone's screen. The parentheses surround *what* we're going to tell that player. In this case, we've typed in a string, delimited by double quotes, that the player will see. Strings delimited with quotes, object numbers, and some arithmetic numbers are called *literals* (as opposed to *variables*). It is bad programming hygiene to put literals in your verbs (with a few exceptions), and we'll be cleaning that literal out and replacing it with something better shortly.

In fact, let's do that now. Instead of embedding the text in the verb (bad), we'll extract it into a property (good). And this is going to be a special kind of property called a message. Type this:

```
@property rock.pet_msg "Nothing happens." rc
```

Notice the period instead of the colon between the word "rock" and the string "pet_msg". This, along with the fact that we're using the `@property` command tells the system that we're adding a property and not a verb. `"Nothing happens."` is the

initial value of our property. We can change it later if we want to. "r" means that the property is readable. In general, specify "c" when you expect to change this property from the command line, and *not* from within a verb. Leave the "c" out if you do intend to change the property from within a verb. (A detailed discussion of what the "c" flag means and how to use it begins on page 165.)

Now, edit the verb to use this message. Either @edit rock:pet to use the verb editor, or @program rock:pet as follows:

```
@program rock:pet
player:tell(this.pet_msg);
.
```

Notice that instead of a string in quotation marks, we've put in a property name instead. "this" refers to the object on which the verb is defined, in this case, your rock. ".pet_msg" refers to the property we just created.

Now pet the rock:

```
pet rock
Nothing happens.
```

Hmm. Something's missing. Right! I forgot to put in "You pet the rock," first. Well, it's easier to change a property, especially a message property, than to re-edit the verb, and that's one of the reasons why we put messages in properties. Observe:

```
@set rock.pet_msg to "You pet the rock.  Nothing happens."
```

or:

```
@pet rock is "You pet the rock.  Nothing happens."
```

The second form works *because* the property's name ends in _msg.

Now pet the rock again. Voilà! See how easy this is? If you wanted to you could try:

```
@pet rock is You pet the rock.  Nothing happens.  What
foolishness!
```

Sharing the Experience

Now let's go public, so to speak, and make our petting action visible to others.

The business of programming has many phases. Among them I number deciding what you want the object to do, thinking up various ways one might do it, seeing if it can be done at all (prototyping), finding a better way to do it. The last one tends to be repeated, and deciding when one has gone far enough is part of what makes it an art.

So. What do I want it to do? I want it to announce to other players in the room that I'm petting the rock.

What are some ways that I might do it? New programmers may not have a clue where to begin. Experienced programmers may have a sense of "the usual way to do it", if it's a commonly-done thing, or will investigate how other people have done it, if it's a thing they've seen before. Geniuses may come up with a brilliant, new way to do it that may or may not be practical. For now, follow my lead. Later, I'll show you some ways to see how other people do things. Good programmers stand on the shoulders of giants (giving credit where credit is due, of course).

We'll do a prototype, first, to demonstrate that it can be done at all, then I'll show you some better ways to do it. Here we go:

```
@program rock:pet
player:tell(this.pet_msg);
player.location:announce(player.name +
  " pets the rock.  Nothing happens.");
.
```

`Player.location` is the room where the player is located. `Player.location:announce(<text>)` is a verb defined on all rooms, and it announces text to everybody in the room except `player`, the person who typed in the command. The parentheses contain the arguments to `player.location:announce`. Arguments are the incoming information a verb works with. `Player.name` is the `.name` property of the person typing in the command. Text in double quotation marks is plain text, also called a *literal.* The "+" sign joins, or *concatenates*, two strings of text to each another.

Now to test it. An interesting challenge, here, because you see the regular stuff, but want to know what other people see. You will either need to get a partner, or, if you have the capacity to MOO with two windows at once, log in as a guest, join yourself, and do two things at once. If you work with a partner, I recommend *both* of the following: Pet the rock, and ask your partner what e sees. Ask your partner to pet the rock and see for yourself.

Okay, that's the prototype. It works, but leaves much to be desired. The first step in improving the situation would be to extract the text into a second message. But wait! The message changes depending on who is petting the rock, so we can only put the fixed part of the message into the property:

```
@property rock.opet_msg " pets the rock.  Nothing happens."
rc
```

This is the part of the message which won't change.

It is a programming convention on the MOO to make messages in pairs, one for the player doing the action and one for everybody else, and, by convention, the message have the same name except that the one for everybody else is prefixed with an "`o`", for "others". To use the message:

```
@program rock:pet
player:tell(this.pet_msg);
player.location:announce(player.name + this.opet_msg);
.
```

Test it. It works, but is only a modest improvement. Modest improvements are progress, though. (If it doesn't work, debug it until it does. Probably a typographical error – we all make them.)

Just as we extracted message text into a property, before, now we're going to extract the business of constructing a message into a separate verb of its own. This may seem like a lot of work for one little message, but the fact is that complex objects have many messages, and we'll be able to generalize our work. So think of it as an investment of effort that will pay off later. (And it will, I promise.)

```
@verb rock:opet_msg this none this rxd

@program rock:opet_msg
return player.name + this.opet_msg;
.

@program rock:pet
player:tell(this.pet_msg);
player.location:announce(this:opet_msg());
.
```

Be sure to test your work.

What's new, here? What's new is that we have just created a verb with the same name as a property. This is fine, and in fact desirable, as I will demonstrate later. For now, just remember to pay close attention to the difference between "`.`" and "`:`".

Also, the new verb had argument specifiers of `this none this`. Because there is no natural English language construct in which a thing is both the direct and indirect object of a verb with no preposition in between, we use this combination of argument specifiers to designate an internal verb (called a *subroutine*) as opposed to a command-line verb. If you examine your rock again, you will note the absence of opet_msg from the list of obvious verbs, which is the way we want it. The new verb returns a result, which in turn is used by the verb that called it.

Plunging Into Pronoun Substitutions!

The time has come to learn to put pronouns into your messages.

At your leisure, you should skim `help pronouns` and `help $string_utils:pronoun_sub`. You don't have to understand every nuance now, but you should know that these help texts exist, and you should have a sense of their scope, for future reference. (I still refer to them from time to time, myself.)

Suppose that when I pet the rock, you wanted the system to announce, "Yib pets the rock. Nothing happens. Doesn't she look foolish!" And when Klaatu pets the rock, we would want it to say, "Klaatu pets the rock. Nothing happens. Doesn't he look foolish!" and when Bits (who use the plural gender) pet the rock, we'd like it to say, "Bits pet the rock. Nothing happens. Don't they look foolish!" (I hope you're beginning to detect a pattern, here.)

Here we go.

The tool that does the work for us is called `$string_utils:pronoun_sub`. It takes a string as an argument, and replaces certain special symbols with values that are meaningful at the time the verb is called. We'll do this in steps. Type:

```
@set rock.opet_msg to "%N pets the rock.  Nothing happens."
```

or:

```
@opet rock is %N pets the rock.  Nothing happens.
```

or:

```
@opet rock is "%N pets the rock.  Nothing happens."
```

Because the property name ends in "_msg" we can set it using any of these forms. (From now on I'm going to let that go without saying.) `%N` is the special symbol for which the player's name will be substituted. (In the "Learn Something New Every Day Department", after all these years of programming, I just noticed that that omitting the double quotes in the second form reduces the number of spaces between the two sentences from two to one, while using the double quotes preserves the two spaces between the two sentences. Fancy that! Since I'm picky about spacing, I'll use double quotes, as I always have (until I started writing this tutorial).)

Now we'll revise the `:opet_msg` verb:

```
@program rock:opet_msg
return $string_utils:pronoun_sub(this.opet_msg);
.
```

Now that the code is in place, lets fancy up that opet_msg a bit more:

```
@opet rock is "%N pets the rock.  Nothing happens.  Doesn't
%s look foolish?"
```

`%s` is the special symbol that substitutes the subject pronoun (he, she, e, they, etc.) Capitalization of the substitution symbols works as you might expect, by the way. Find a way to test this new message with different genders. Either find people of different genders to play with, or find an observer while you set your own gender to different things. Or, if your machine and/or client software has the capability, you can log on simultaneously as yourself and a guest for purposes of testing. Many programmers do this.

Did you find the problem with the plural gender? "Bits pets the rock. Doesn't they look foolish?" Who looks foolish now? The pronouns are good, but there's a little problem with verb agreement. Never fear, `$string_utils:pronoun_sub` will save the day again:

```
@opet rock is "%N %<pets> the rock.  %<Doesn't> %s look
foolish?"
```

Now this works for everybody and everything. (Ain't life grand?) The percent sign in combination with the angle brackets signals to `$string_utils:pronoun_sub` that some adjustment may be needed for verb agreement. From your end, just put the appropriate (English language) verbs between

the angle brackets, preceded by the "%" sign. Write them as if they were for a third person singular actor.

Generalizing the Message Verb

What we've done so far is work our way up to the .opet_msg property and a corresponding :opet_msg verb that does spiffy pronoun substitutions, and we've come quite a long way from where we started out. And we could do another verb, if we wanted, :feed, perhaps, with corresponding .feed_msg and .ofeed_msg properties and an :ofeed_msg verb. (Think about how you might do that.) The :ofeed_msg verb would look an awful lot like the :opet_msg verb, wouldn't it? In fact, it would bear a STRIKING RESEMBLANCE, except for the name of the message property it referred to. Well, well. Can we capitalize on this and do something more efficient? You bet we can:

```
@program rock:opet_msg
return $string_utils:pronoun_sub(this.(verb));
.
```

Instead of this.opet_msg, I wrote, this.(verb). Notice that the name of the verb is the same as the message. The variable verb is a system-provided or *built-in* variable that contains a string, the name of the verb as it was called. Now do this:

```
@addalias "pet_msg" to rock:opet_msg
```

Just as objects can have aliases, so, too, can verbs. But the important part is that when the verb gets executed, the message on which pronoun substitution is performed and which is ultimately displayed is *either* this.pet_msg or this.opet_msg, *depending on which alias was used to call verb.*

And then:

```
@program rock:pet
player:tell(this:pet_msg());
player.location:announce(this:opet_msg());
.
```

Ta-dah! We've now generalized it about as much as we can. Watch closely. Nothing up my sleeve, and presto!

```
@addalias "feed_msg" to rock:pet_msg
@addalias "ofeed_msg" to rock:pet_msg

@prop rock.feed_msg "You try to feed the rock.  Nothing
happens." rc

@prop rock.ofeed_msg "%N %<tries> to feed the rock.  Nothing
happens.  Natch!" rc

@verb rock:feed this none none rxd
```

```
@program rock:feed
player:tell(this:feed_msg());
player.location:announce(this:ofeed_msg());
.

examine rock
pet rock
feed rock
```

Heh.

Polishing the Rock

I am now going to address myself to issues of documentation. At the beginning, writing documentation may seem tedious, and it may seem silly to add comments to such simple programs. But comments and documentation are among the things that separate the good craftsman from the mere hack; adding them consistently is a very good habit to get into.

We'll start with plain help text, which turns out to be easier than you might think. Just create and edit a .help_msg property. (It can have one line or a list of several lines.) I usually start out with an empty list, then edit that with the note editor:

```
@property rock.help_msg {} rc

@edit rock.help_msg
enter
Rocks make great pets!  They're quiet, clean, and easy to
maintain.

Build one today!
.
save
done
```

If you wanted to, you could `@addalias "help_msg" to rock:pet_msg` and do pronoun substitutions in the help messages. Let's do that, just to see:

```
@addalias "help_msg" to rock:pet_msg

@edit rock.help_msg
list
ins 2
enter

This one's name is %t.
```

```
.
save
done

help rock
```

Recall that I asked you to read help $string_utils:pronoun_sub. $string_utils is an object, and has its own .help_msg property. And there is *also* help text for the *verb*, :pronoun_sub. How do they do that? If you typed @list $string_utils:pronoun_sub (it's rather long), you would see some lines at the top in double quotes, with semicolons at the end. You would probably recognize these as comments, and you would be right. Comments that appear at the top of a verb (before any other lines of code) will also appear as help text for that particular verb. Even though our verbs are small and simple, let's add comments to them:

```
@edit rock:pet
ins 1
enter
"Usage:  pet <this>";
.
compile
done

help rock:pet
```

Do the same for the :feed verb. At the top of the :pet_msg verb, put a comment that says, "This verb does pronoun substitutions on various messages." Test your work.

And now, for the icing on the cake, because we can, we're going to customize the output when someone types examine rock:

```
@prop rock.obvious_verbs {} rc

@edit rock.obvious_verbs
enter
  pet %<what>
  feed %<what>
  give/hand %<what> to <anyone>
  get/take %<what>
  drop/throw %<what>
  help %<what>
 .
save
done


@verb rock:examine_verbs tnt rxd

@program rock:examine_verbs
```

```
"Returns a list of obvious verbs, substituting the string
the player typed in for %<what>";
what = dobjstr;
vrbs = {};
for vrb in (this.obvious_verbs)
  vrbs = {@vrbs, $string_utils:substitute(vrb,
    {{"%<what>", what}})};
endfor
return {"Obvious verbs:", @vrbs};
.

examine rock
examine pet rock
```

The `.obvious_verbs` property should seem fairly obvious to you by now. Let's take a quick look at that `:examine_verbs` verb. It's an internal verb that's called when you `examine` something. It has a comment at the top. `dobjstr` (think "direct object string") is the string the user typed as the direct object in a command. If the player typed `examine rock` then `dobjstr` will be "`rock`". If the player typed `examine pet rock`, then dobjstr will be "`pet rock`" and so on. The next few lines are building a construct to return as a result. We start with an empty list. Then there is a *for loop* that does something with every item of `.obvious_verbs`. What does it do? It does a substitution of `dobjstr`, and appends the new result to the existing list. For the use of the `@` sign in this context, I refer you to `help listappend`. For the rest, parentheses, brackets, and braces nest. It's just a fancy call to a fancy verb, `$string_utils:substitute`, which has its own help text. If you want to, you can take this one at face value, and model subsequent `:examine_verbs` on other objects after this one. Mimicking is a tried and true technique that gets you into hot water sometimes but often works. I'm not above doing it myself when I'm trying to learn how to do something: Mimic something similar, and in the process of getting it to do exactly what I want, I learn how it actually works. Last, notice the return statement: Because of the curly braces {}, it's returning a *list* of things, which the calling verb is expecting.

Programming can be wild and woolly, sometimes, but that's part of the fun.

What Can't You Do With a Rock?

Well, you can throw a rock. Throwing rocks isn't very nice. You can, if you wish, prevent people from throwing your rock.

In this case, it's less about programming (more of that later), and more about controlling your environment and the things you own, so let's learn to exercise a bit more of that control.

In MOOs, throwing and dropping are more or less considered synonymous. But it doesn't have to stay that way. `Examine rock`. You will see, among other obvious

verbs, `d*rop/th*row rock`. Initial concept: We want dropping the rock to behave normally, but throwing the rock to give the player a message, instead.

Type `@messages rock`. You will see all the _msg properties defined on your rock (and all of its object ancestors), including the messages that we added. Notice: No throwing messages. Concept: Make a separate throw verb, and a set of throw messages to go with it. Revision: Actually, they should be no_throw messages, or perhaps, to blend in with what's already there, throw_failed messages. So:

```
@prop rock.throw_failed_msg "Throwing rocks isn't nice, and
besides, this rock likes you, so it stays nestled safely in
your hand." rc

@prop rock.othrow_failed_msg "%N %<makes> a throwing motion
with %t, but can't quite seem to bring %r to let go." rc

@addalias "throw_failed_msg" to rock:pet_msg
@addalias "othrow_failed_msg" to rock:pet_msg
```

So far this should seem familiar. I often start with what I want the messages to be, then verbs to control how, when, and to whom they'll be displayed:

```
@verb rock:throw this none none rxd
```

But wait! The verb name is "th*row", and if we want to override it, we have to name it exactly the same as the verb on its parent. If you went ahead and typed in the line above, remove the verb with:

```
@rmverb rock:throw
```

To be absolutely sure of how to add it, we'll check with this:

```
@display rock:throw
```

We use `@display rock:throw` rather than `examine rock` because who knows how the previous programmer may have gussied up the examine verbs, eh? Based on what we see, then, we'll add the verb like this:

```
@verb rock:th*row this none none rxd

@program rock:throw
"Throwing stones isn't nice.  Thwart that impulse.";
player:tell(this:throw_failed_msg());
player.location:announce(this:othrow_failed_msg());
.
```

Well, on looking at it, that throw_failed_msg is a bit patronizing, and furthermore, unhelpful to someone who actually wants to rid emself of your rock. So let's adjust it:

```
@throw_failed rock is "Throwing rocks isn't nice.  Try
dropping it, instead."
```

Test everything. Try throwing the rock. Try dropping the rock. Examine the rock. Hmm. I found, in my play-testing, that you can't drop a rock if you aren't

holding it, but you can throw a rock (or try) if you aren't holding it. We can do better. Let's start by looking at what the original code does:

```
@list rock:drop
```

This will show us the original drop verb. It's pretty old code, and written in an older style. You'll notice the difference between the way that verb handles messages and the way ours do. Our verbs are new and improved. And you'll see some things that you don't understand, perhaps, which may or may not turn out to be relevant. Get what you can out of it and don't worry about the rest, right now. What we're looking for, though, is the part that generates the text, "You don't have that," so that we can do ours in a similar, if not identical way. And what it does is check the location of the rock, and display different messages depending on where it is. Our verb will be simpler, but will behave in a similar way. I'm going to show you two versions, and I hope you can tell by inspection (and maybe by consulting the programmer's manual) what the difference is. They are both equally good, so you can choose which way you want to implement it.

Version 1:

```
@program rock:throw
"Throwing stones isn't nice.  Thwart that impulse.";
if (this.location == player)
  player:tell(this:throw_failed_msg());
  player.location:announce(this:othrow_failed_msg());
else
  "You can't throw a rock if you don't have it in the first
place.";
  player:tell("You don't have that.");
endif
.
```

Version 2:

```
@program rock:throw
"Throwing stones isn't nice.  Thwart that impulse.";
if (this.location != player)
  "You can't throw a rock if you don't have it in the first
place.";
  player:tell("You don't have that.");
else
  "You have it, but you can't throw it....";
  player:tell(this:throw_failed_msg());
  player.location:announce(this:othrow_failed_msg());
endif
.
```

Here is a case where I've chosen *not* to extract a message into a property, so let me tell you why. I put in a perhaps-superfluous comment, "You can't throw a rock if you don't have it in the first place," mostly to set a good example. But in a case where we're just telling the player some sort of error message, the embedded string

can serve *as* the comment. So a leaner but just-as-readable version might look like this.

Version 3:

```
@program rock:throw
"Throwing stones isn't nice.  Thwart that impulse.";
if (this.location != player)
  player:tell("You don't have that.");
else
  "You have it, but you can't throw it....";
  player:tell(this:throw_failed_msg());
  player.location:announce(this:othrow_failed_msg());
endif
.
```

Expediency is fine, if it isn't cryptic. If you have to squint to follow what the code is doing, add a comment. You'll be glad later that you did. Trust me.

Before we move on, I want to take a quick moment to point something out in Version 1. Notice the line, `if (this.location == player)`. Notice especially the double-equals sign "`==`". There is a very big difference between a single equals sign and a double equals sign. The first is an assignment statement. `a = b` means, "Set the variable `a` equal to the current value of the variable `b`." `a == b` means, "Is the current value of `a` equal to the current value of `b`?" These are two very different things. Confusing the two is a classic mistake. Heads up.

Here's another way to control your rock and what people do with it.

Suppose you want to lock your rock in place, so that people can't take it. Maybe it's a boulder!

```
drop rock
@lock rock with here
take rock
```

Heh, you can't pick that up, and neither can anybody else. You might change your rock's description to show that it's a boulder. Or you might just change .take_failed_msg and otake_failed_msg to say something like, "It's heavier than it looks, isn't it!"

If you decided to make your rock portable again, type:

```
@unlock rock
```

See also `help @lock` and `help locking`.

Rockin' and Rollin'

So far, we've done a variety of things with our rock, but the rock itself hasn't changed, much. Hardly surprising, I suppose, but the phrase comes to mind, "A

rolling stone gathers no moss," and I was thinking, wouldn't it be fun if our rock gathered moss?

How shall we start thinking about this? Well, what if the description had a bit tacked onto the end saying how mossy the rock is? The amount of moss can depend on how long since the rock was last moved.

So. We'll need a list of different amounts of moss. We'll need to write a verb to make the description change over time. We'll need a way to see how long it has been since the rock was last moved, and we'll need a way to convert that amount of time into a phrase about moss.

Let's start with the easy part to get our juices going:

```
@property rock.moss_list {} rc
```

It's going to be a list of text strings, and {} is the symbol for the empty list. We start with that. Then:

```
@edit rock.moss_list
enter
It has gathered no moss.
If you were to look at it closely with a magnifying glass,
you would see a tiny bit of moss on it.
There is just a wee bit of moss growing on it.
It has gathered a little bit of moss.
There is some moss growing on it.
It is about half-covered with moss.
It has gathered quite a bit of moss.
It has gathered a great deal of moss.
It is almost completely covered with moss.
It is covered with moss.
.
save
done
```

Time on MOOs is measured in seconds since midnight on 1 January 1970, Greenwich Mean Time. How do I remember that? I don't. It's in `help time()`, which we will be using.

```
@prop rock.last_moved_time 0 r
```

It's `r` (and not `rc`) because we'll be changing it from within our verb (only). Every time the rock moves, we'll record the time. Every time someone looks at the rock, we'll compare the current time to the `.last_moved_time`. Any time an item moves or is moved, its `:moveto` verb is called. We want to take advantage of this by adding a bit to the existing `:moveto` verb, and we do it like this:

```
@verb rock:moveto tnt rxd
```

`tnt` is an abbreviation for `this none this`, our designation for an internal verb.

```
@program rock:moveto
"Reset the reference time (clearing off any moss)";
```

```
this.last_moved_time = time();
"Then do all the usual stuff that the parent does.";
return pass(@args);
.
```

Can we test out this much? You bet. Drop the rock (if you have it) or take the rock (if you don't). You can manually inspect the .last_moved_time property this way:

```
#rock.last_moved_time
```

(See also `help #`.) Move it again, then check `.last_moved_time` again. It changed, by about the number of seconds between moves. Don't be daunted by that great big number. There are plenty of verbs to help us make sense of it. (See `help $time_utils` if you're curious right now.)

We don't care what the number means. It's the *difference* between `time()` and `rock.last_moved_time` – the number of seconds that have elapsed since it was last moved – that interests us. Next we have to pick an interval (measured in seconds) during which a new amount of moss grows. Maybe a day, or a week, but who wants to wait that long to test it? We'll start with, say, a minute, then change the number later after we've tested it:

```
@prop rock.moss_interval 60 rc
```

The next bit is rather a lot of complexity all at once, but try to stay with me, here.

First, let's look at some of those moss descriptions one at a time, to get a feel for them. You'll need to know your rock's object number for this. If you've forgotten it, type `#rock`. Mine is `#70217`, so I'll type that here, but you should use your own rock's object number. (Angle brackets are just too cumbersome for this demonstration.) We're going to play with `eval` a bit. `eval` lets you evaluate a tiny bit of MOO-code on the fly, as it were, without having to write an entire verb to do it. `#` and `;` are abbreviations for `eval`. It's extremely useful! Try some of these:

```
#70217.moss_list
#70217.moss_list[1]
#70217.moss_list[5]
#70217.moss_list[0]
#70217.moss_list[30]
```

Oho, if we give it too high or too low a number, we get an error. We'll want to keep this in mind when we write our verb. Here are some more examples of `eval`:

```
;length(#70217.moss_list)
;#70217.moss_list[length(#70217.moss_list)]
;#70217.moss_list[$]

;ctime()
;ctime(#70217.last_moved_time)
;time() - #70217.last_moved_time
```

```
;(time() - #70217.last_moved_time) /
   #70217.moss_interval
```

If your moss interval is 60, like mine, the last one will show you how many minutes since the rock was last moved.

Now we're going to add a `:description` *verb* to the rock, which we will then customize. When you look at a thing, the system executes that thing's `:description` verb. For most things, all the verb does is return the value of the `.description` property, very like our message verbs were doing before we got fancy with pronoun substitutions. It's not unusual for objects to have customized `:description` verbs:

```
@verb rock:description tnt rxd

@program rock:description
"Start with the original description, then add to it.";
"Some moss, perhaps.";
base_description = pass(@args);
"What time is it now?";
now = time();
"How long has it been since it was last moved?";
how_long = now - this.last_moved_time;
"How many .moss_intervals is that?";
index = how_long / this.moss_interval;
"If it has been a very short time, index will be 0, but list
elements always start at 1 in MOO code, so we'll add 1.";
index = index + 1;
if (index > length(this.moss_list))
  "It has been so long, index is too high.";
  "So just use the last one.";
  index = length(this.moss_list);
endif
return (base_description + "  " +
  this.moss_list[index]);
.
```

Test your work. Isn't this fun? If you don't like waiting an entire minute each time, set your rock's `.moss_interval` to something smaller, like 10 or 15. Then when you're satisfied, set it to something longer. How many seconds in a day? In a week? In a month? Find out like this:

```
"One hour
;60 * 60
"A day
;60 * 60 * 24
"A week
;60 * 60 * 24 * 7
"And so on.
```

And now, a stationary stone gathers moss.

Looking Under Various (Other) Rocks

Learning a new language is always a challenge, and a programming language is no different. It's nice, as an adult, to attend a language class and have a professor or instructor take you through the material in a logical sequence, so that first you learn to express simple things, then more complex things, until you get a sufficient understanding of the grammar and a sufficiently large vocabulary that you can start to express your own ideas independently. Children learn languages, on the other hand, by being immersed, by imitating those around them, by trying things and seeing which utterances get results and which get perplexed looks from their elders. I don't think anyone learns a language by reading a dictionary.

The Programmer's Manual is a good reference book, but there are other tools that will help you explore the MOO around you and learn (by example) from what's out there already. I am going to detail some of those tools now.

Suppose you want to investigate an object to find out what makes it tick. First, (I hope) you would `examine` the object and perhaps play with it a bit.

Then you might type `help <object>`, to see what the owner or creator wants you to know about it. Then, perhaps, `@parents <object>` to get a handle on what sort of object it is. You'll get a list of object numbers and names, and it may (or may not) be fruitful to check the help text on the object's ancestors, as well.

Then, perhaps, you'd like to know whether this object has any special programming of its own that makes it different from its ancestors. This is where the `@display` verb comes in. It's one of my favorites. There's extensive help text on it, but the form of `@display` that I use most often is `@display <object>.:` which gives a list of verbs and properties defined on that object. (Note that if none are defined, however, you may see the verbs and properties defined on its immediate parent, instead. You can fix that by typing `@display-options +thisonly`.) Often you can get a good start on understanding an object just by looking at the names of the verbs and properties defined on it. Trick: If an object is set as a whole to be unreadable, but some or all of its verbs *are* readable, you can get a handle on those by typing `@verbs <object>` instead of using `@display`. (See also `@show <object>`.)

If a particular verb catches my eye, I might list it, using the command `@list <object>:<verbname>`, just to see what's in there.

Another thing I might do is type `@messages <object>` to get a sense of the scope of its output. If there are messages that I haven't found yet in the course of my playing, then I know that the programming on the object is richer than first meets the eye, and that it might well be worth exploring the object further.

Some people like to use either `@dump <object>` or `@dump <object> with create` to get a complete listing of everything on it. As a rule, I don't care for the large quantity of text that `@dump` generates, but some people like to print out

everything there is to know about an object, send it to a printing device, and read the hard copy at leisure. If that's your cup of tea, by all means do that.

Suppose you see a line of text, that you know to be from a particular object, and you want to home in on it to see what verb generates it, and what's going on in the vicinity (if you will) of the line of text that has caught your interest. For example, "Yib tries to feed the rock. Nothing happens. Natch!" The `@grep` command can be helpful here. (Note, `@grep` requires an object number, not the name of an object, even if you are in proximity.)

```
@grep "Natch!" in <object>

Searching for verbs in <object> containing the string
"Natch!" ...

Total: 0 verbs.
```

Well, in this case we struck out. But if you typed `@messages rock` and found that the phrase, "Natch!" was part of the `.ofeed_msg` property, then you could type:

```
@grep "ofeed_msg" in <object>
```

and you would see:

```
Searching for verbs in #<object> containing the string
"ofeed_msg" ...

 <object>:feed [<verb owner>]:
player.location:announce(this:ofeed_msg());

Total: 1 verbs.
```

So in this case you would have learned that `ofeed_msg` (which contains the string we're interested in) is used in the `:feed` verb, so then you might list that verb out to see the larger context.

Last, suppose you are MOOing along, minding your own business (more or less), and out of the blue you see the text, "A black magpie flies in and looks greedily at bright sparkly thing." (You happen to have dropped said sparkly thing recently.) Suppose you made that sparkly thing yourself, and know for sure that there is nothing in its verbs or properties that refers to a black magpie. Where did this line of text come from? What's going on here? You can type `@check-full <text>` and get a trace of the verbs that were called in the process of delivering this choice tidbit of text to your baby blue eyes:

```
@check-full magpie

Traceback for:
A black magpie flies in and looks greedily at bright sparkly thing.
This           Verb               Permissions    VerbLocation   Player
-------------- ------------------ -------------- -------------- ------------
#58337(Y)      tell(1)            #67(Rincewind) #7069(generic) #5720(blue)
#58337(Y)      tell(29)           #3920(Jay)     #33337(PC Clas #5720(blue)
#58337(Y)      tell(4)            #57140(SSO)    #40099(SSSPC)  #5720(blue)
#58337(Y)      tell(1)            #58337(Y)      #58337(Y)      #5720(blue)
#6193(Driveway announce_all(3)    #2(wiz)        #3(generic roo #5720(blue)
#6193(Driveway announce_all(3)    #24442(rw3)    #17755(Integra #5720(blue)
#77522(magpie) make_the_rounds(27 #61050(Y_A)    #77522(magpie) #5720(blue)
#77522(magpie) wake_up(19)        #61050(Y_A)    #77522(magpie) #5720(blue)
-------------- ------------------ -------------- -------------- ------------
```

Well, in this example, you can find out the object number of the magpie, the names of a couple of verbs on the magpie that might be worth looking into, and then you're off and running. Heads up: If you are doing this on LambdaMOO and have the Lag Reduction FO of Godlike Powers, you will have to type `@addlag` and `@paranoid <number>` in order to get anything useful from `@check-full`. `<number>` in the `@paranoid` command refers to a number of lines to keep tracebacks for. There is help text for all of these functions.

Read lots and lots of verbs. Even if you don't understand everything in them, you will gain exposure, start to pick up on patterns, and, as you are ready, absorb new concepts and learn new tricks. Leave no stone unturned.

Asking Others for Help

It is very important that you give a problem the old college try before asking others for help, for a couple of reasons.

First, it's inconsiderate to ask someone else to put more time and work into investigating a bug of yours than you are willing to do yourself.

Second, the very act of trying everything you can think of and checking every reference you know about makes you more receptive to and able to understand the answer when you finally get it.

Before asking for help:

- Read any tracebacks and look at the verb/lines that seem to be causing a problem.
- Read any compiler errors and look at the verb/lines that seem to be causing a problem.
- See if you can find any online help text that addresses your difficulties. (Don't forget `help index`.)
- See if there is any other documentation relevant to your project.

When you ask for help, put together a summary of the problem:

- Include a brief description, and a copy of a traceback (if any).
- Describe in detail how to duplicate the problem.
- Include object numbers -- don't just say, "My pet rock doesn't work."
- Indicate what you've tried so far, both to show that you *have* tried, and to save your helper the trouble of trying things that you've already checked.

DO ask for help if you're genuinely stuck. Most people are happy to assist (or at least try) if you ask nicely and demonstrate respect for their time.

Is It Covered With Moss, Yet? (A Small Side Project)

While my rock was gathering moss, I was wanting to check it each and every .moss_interval, just for the fun of reading all the messages. But I felt silly typing `look rock` over and over again, waiting to see whether it had changed yet. So I thought, "I wish I had a timer, so I could set it and forget it, and be reminded when it was time to look at the rock again."

Let's make one.

Concept: This will not be a complicated object to use. All we need is a verb, `set timer for <some duration>`. One verb ought to do it. We'll get more practice using `$time_utils`:

```
@create $thing named timer
@describe timer as "A simple timer, such as you might find
in the kitchen."
```

Now we'll start with a prototype, to see if we can make it work at all, then refine it and make it more robust. With programming (as with many kinds of projects) the trick is, "Divide and Conquer." If a task seems too big and overwhelming, divide it into subtasks. If those are still too complicated, divide those further. I'm going to show you many versions of one verb, to demonstrate that these programs don't emerge full blown from my head. After years and years of programming, I still work simple-to-complex. Here we go.

First, I typed `@display $time_utils:` (note the colon), which gives me a list of all the verbs on that particular utility package. I have something in mind that I'm looking for, and that just comes with experience and lots of exploring. You could also type `help $time_utils`. Aha, here is what I'm looking for, `$time_utils:parse_english_time_interval`. This will let me take input like, "1 minute" and turn it into a number of seconds. Let's try out just that much:

```
@verb timer:set this for any rxd

@program timer:set
duration =
$time_utils:parse_english_time_interval(iobjstr);
```

```
player:tell(tostr(duration));
.
```

I created the verb, and then put the bare minimum of programming on it. `Iobjstr` is the string that someone types in as the indirect object, and, since it can be anything (it will be a duration, in English words), I gave `this for any` as the arguments. In the program I set up a variable called `duration`, which stores the output from one of the `$time_utils` verbs. My only goal at the moment is to satisfy myself that I can take a string like "10 seconds" and get a number 10 out of it. The second line of the verb is a simple output line, to player (that's me, the developer) to see if I did, in fact, get something intelligible. Note the `tostr(duration)`. `duration` is a number. `player:tell` takes a string. `tostr()` turns a number into a string. So I've turned the number into a string (inner set of parentheses), then I'm telling that string to `player` (me):

```
set timer for 10 seconds
  10
```

Success! But let's test further:

```
set timer for 10
Incorrect number of arguments
set timer for Fred
Incorrect number of arguments
```

Hmm. I want to know if $time_utils:parse_english_time_interval is sending back the STRING "Incorrect number of arguments", in which case I can work with it, or whether the system is giving me this message, in which case I'll have to scratch my head considerably more. I'm going to adjust my verb to answer this question:

```
@program timer:set
duration = $time_utils:parse_english_time_interval(iobjstr);
player:tell("Result: " + tostr(duration));
.
```

All I've done is insert the word "Result: ", before the result I get back from $time_utils. If I'm getting the string back from the utility, then I'll see the word, "Result: " at the beginning. If I'm getting the message from the system, then I won't:

```
set timer for 10
Result: Incorrect number of arguments
set timer for Fred
Result: Incorrect number of arguments
```

Yay! The message is coming from $time_utils, so I'll be able to snag it easily and deal with it gracefully. (In fact, there are ways to intercept system error messages, too. See the note for the ERR datatype on page 158).

```
@program timer:set
duration =
  $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
```

```
  "We got a good value for duration.";
  player:tell(tostr(duration));
else
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

This version of the verb tests to see whether the result returned by `$time_utils:parse_english_time_interval` was a number or not. If it's a number, then we got good input. If we didn't get a number back, then we'll give the player a polite and instructive error message. The `typeof()` built-in function is what I use to find out what sort of thing I'm working with. See `help typeof()`. At this stage, I choose not to extract that error message into its own verb, so I do the pronoun substitution on the fly. With more than a handful of lines, it's also time to put help text at the top of the verb, so I'll be sure to do that in the next round. Test the code as before. When you've elicited every message for every contingency you've accounted for, then you've done enough.

Now it's time to make the timer actually do its thing (drum roll, please):

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration =
  $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  fork (duration)
    player:tell("Ding!");
  endfork
  player:tell("The timer starts ticking.");
else
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

You should be able to follow most of this. The only new items are those `fork` and `endfork` statements. When you set a timer, you set it down and go off and do something else, while the timer does its thing independently. And that's what's going on here. Everything between the lines, `fork....endfork` will be done at a later time. How much later? The number of seconds that we give to `fork` as an argument, in this case, duration. Anything after the `endfork` statement gets done right away. Try this out, now. Use a duration like `1 minute`. An hour is probably longer than you want to wait.

You can check on background tasks (as these are called) with the `@forked` command. See `help @forked` and also `help @kill`. Forked tasks hog system

resources, and should only be used in moderation. If you work on a MOO that has a task scheduler, you should make a point of learning to use it (when you feel ready).

Well. When I tried it, it worked, but the “Ding!” got lost in some other text I was displaying on the screen at the time. So on the next round, I’m going to indent it, among other things. The only thing new is that I’m going to make the output prettier. Pretty output is more important than you might think. What I want is, “Ding! 10 seconds are up”, indented so that I’ll notice it more. I also want it to differentiate between “60 seconds *are* up”, and “1 minute *is* up”:

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration = $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  fork (duration)
    "Indent the message, for better visibility.";
    "Add text to indicate how much time has elapsed.";
    be = ((iobjstr[$] == "s") ? "are" | "is");
    message = iobjstr + " " + be + " up.";
    message = $string_utils:capitalize(message);
    player:tell("  Ding!  " + message);
  endfork
  player:tell("The timer starts ticking.");
else
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

The new material is within the fork/endfork statement. I construct a message using smaller strings and the “+” sign to concatenate them together. Here’s the mystery line:

```
be = ((iobjstr[$] == "s") ? "are" | "is");
```

Here it is in pseudo-code:

```
if (the last letter of iobjstr is "s")
  set the variable 'be' to the string "are"
else
  set the variable 'be' to the string "is"
endif
```

Divide and conquer. Working from the inner-most parentheses outwards: `Iobjstr` is going to be something like, “10 minutes”, or “1 hour”. I want to know whether the last character is an “s”. `iobjstr[5]` means the fifth letter of iobjstr. `iobjstr(length(iobjstr))` is the last letter of `iobjstr`. `iobjstr[$]` is a way to abbreviate that. “$” in this context means “last”. Note the double “`==`” sign. If you are assigning a value to a variable, use one: `x = 3`. If you are testing for equality, use

two: `if (x == 3)...` These two different usages lend themselves to typographical errors. Be sure to look for a mistake in the number of “=” signs when you are debugging.

Now what about that question mark? Am I uncertain of what I’m coding? No. The paired symbols, “?” and “|” are an abbreviated way to do a simple `if...then` statement:

```
result = (x ? a | b);
```

is the short way of writing:

```
if (x)
  result = a;
else
  result = b;
endif
```

“x” can be an arbitrarily complicated expression.

So:

```
be = ((iobjstr[$] == "s") ? "are" | "is");
```

sets up a variable, `be` to be either the string `"are"` or the string `"is"` depending on whether `iobjstr` ends in the letter `"s"` or not.

If you’ve played around with your timer, you may have noticed that you can set it more than once. It’s actually a multiple timer. We could call this a bug, and add code to see if the timer is ticking, and, if it is, tell the player that it’s currently in use. Or we could call this a feature, and add code (and documentation!) to take advantage of the fact. I choose the latter.

Good Times

We have a timer, and it times things. It dings when the time is up, and we can set it more than once -- it’s a multiple timer. Being the forgetful sort, now I would like to be able to type in an optional reminder message, so that when the timer dings, I’ll know what it was I had set it for.

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration = $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  "Ask for an optional reminder message.";
  message = $command_utils:read("an optional reminder
message");
  if (!message)
    be = (iobjstr[$] == "s" ? "are" | "is");
    message = iobjstr + " " + be + " up.";
```

```
    message = "  Ding!  " +
$string_utils:capitalize(message);
  endif
  fork (duration)
    player:tell(message);
  endfork
  player:tell("The timer starts ticking.");
else
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

The new line here is:

```
message = $command_utils:read("an optional reminder
message");
```

Oho! Another utilities package, `$command_utils`. Check out its help text to see what sorts of things are in this box of tools. Don't worry about understanding it all; just get acquainted a little bit, for future reference. We are going to read a line of input from the user (`player`), and store its value in the variable `message`. If the user typed `<enter>` without any text, then we'll just build the message as before.

Notice that I took some of the message building out of the fork...endfork construct and did it all ahead of time. This is a matter of programming style, which is hard to teach. My reason was, put all the message building code in one place, before the fork statement. Then when the forked duration is up, just tell the player the message. In a situation where I wouldn't know what the message should be until after the time had elapsed, it would be appropriate to construct or select message text within the body of the fork/endfork block.

Edit or type in the new version, and try it out. This is not bad, but I missed the `Ding!` if I typed in a message. So I'm going to adjust it so that the output is more to my taste. The next changes aren't really substantive, so I've just noted the changes with comments in the code itself.

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration = $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  "Ask for an optional reminder message.";
  message = $command_utils:read("an optional reminder
message");
  if (!message)
    be = (iobjstr[$] == "s" ? "are" | "is");
    message = iobjstr + " " + be + " up.";
    message = "  Ding!  " +
$string_utils:capitalize(message);
```

```
  else
    "Add a Ding!  because I can.";
    message = "  Ding!  " + message;
  endif
  fork (duration)
    player:tell(message);
  endfork
  player:tell("The timer starts ticking.");
else
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

This version does *almost* exactly what I want. The last step (before doing up the examine verbs and the help text) is to take a step back and look at the code and see if I can make it any better. I notice that I am prepending "Ding!" in two places, and can consolidate that. Now that you understand what more of the code means, some of the comments are superfluous, and I'm going to take them out. That last `else` statement is pretty far from its matching `if` statement, so I'm going to add a comment down there. And I found the Ding! message still not quite prominent enough, so I'm going to use a variant on `player:tell` to put it between blank lines. Here is my final version:

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration = $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  message = $command_utils:read("an optional reminder
message");
  if (!message)
    be = (iobjstr[$] == "s" ? "are" | "is");
    message = $string_utils:capitalize(iobjstr + " " +
      be + " up.");
  endif
  message = "  Ding!  " + message;
  fork (duration)
    player:tell_lines({"", message, ""});
  endfork
  player:tell("The timer starts ticking.");
else
  "Didn't get a good value for duration.";
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

Last but not least, we'll add help text and examine verbs. We've done this before:

```
@prop timer.help_msg {} rc

@edit timer.help_msg
enter
This is a simple timer.  To use it, type 'set timer for
<duration>'.
Here are a few examples:

  set timer for 3 minutes
  set timer for 45 seconds
  set timer for an hour

The timer will prompt you for an optional reminder message.

You can time more than one thing at once.  That is, you
don't have to wait until the first time is up before setting
the timer for another thing.  Reminder messages are
especially helpful when you are timing several things
simultaneously.
.
save
done

@prop timer.obvious_verbs {} rc

@edit timer.obvious_verbs
enter
  set %<what> for <duration>
  give/hand %<what> to <anyone>
  get/take %<what>
  drop/throw %<what>
  help %<what>
.
save
done


@verb timer:examine_verbs tnt rxd

@program timer:examine_verbs
what = dobjstr;
vrbs = {};
for vrb in (this.obvious_verbs)
  vrbs = {@vrbs, $string_utils:substitute(vrb, {{"%<what>",
what}})};
```

```
endfor
return {"Obvious verbs:", @vrbs};
.
```

Voilà, a finished product that does something useful! As you can see, objects and verbs evolve, from gleams in their creators' eyes to simple proof-of-concept prototypes, to fancy versions with whistles and bells, to finished products with nice exam verbs and help text.

What's Big and Red and Eats Rocks?

When I first started planning this tutorial, I wanted to make a small pet. Something that one could interact with, something that could respond in a limited way to things happening around it, something that would seem to take on "a life of its own", which is part of the true magic of programming on a MOO. So for our final project, we're going to make a big red rock eater.

First, the brainstorming. Petting the big red rock eater should generate more interesting results than petting a rock. There should be a command of the form, `feed <anything> to Red`. Hmm. What should happen to things that get fed to Red? Should they disappear? Should we make a special place called "`RedBelly`"? Should it be possible to feed a *player* to Red? That might be an interesting experience, but not all players are gracious about being moved. If it eats, then it would be nice to have a good VR way to get things back or have them re-appear. Should it excrete? Maybe it could go hunting and find some sort of treasure, like a cat finding a mouse. Or maybe it could be like a cartoon character and reach into its own innards and produce a previously-eaten object. I'll have to think about that one. Should it talk? How about if it had variable color spots? I want it to have a gender, so we'll make it a kid of the generic gendered object. If it only eats rocks, then I'll need a way of deciding whether a thing is a rock. Or maybe it eats anything, but is especially fond of rocks. How about a verb to tickle the rock eater? Nah. It wouldn't effect any change in state, and emote works just as well. So let's pass on that one. (I know, the same could be said of the :pet and :feed verbs on the pet rock, but those were instructional exercises.) Maybe it would eat rocks in the vicinity spontaneously. Or maybe not. It should have a repertoire of spontaneous actions, things that it "just does" from time to time. I know what cats do, but will have to think of particulars for rock eaters. Still, the concept is good. Maybe it's like a Tamagotchi™, and if you don't take good enough care of it, it dies! Or maybe if it gets hungry enough, it eats its owner and the owner gets booted! (Fun thoughts, but maybe that's taking things a bit too far.) Maybe, though, its description could change depending on how hungry it is -- like the rock gathering moss. That would be a good compromise.

That's how I typically start a project. I write down everything I can possibly think of, adding to the list over the course of several days (sometimes weeks, or even months). I consider possibilities, no matter how far-out, and pose myself problems. Some things I already know how to do, because I've done the same or similar things before. Others I may have to research.

The next step, then, is to create an object and start fleshing it out, making it more complex as I go along:

```
@create $thing named "a big red rock eater", "big red rock
eater", "red rock eater", "rock eater", "eater", "brre",
"Red"
```

I decided at the last minute to name mine "Red". Feel free to name yours something else. Naming is actually hard, sometimes. You have to decide what you want people to see when they enter a room (for example): `"You see a big red rock eater here,"` or `"You see Red here."` I may yet change my mind and rename him so that the name is `Red`, and `"big red rock eater"` appears in the description, instead. (The system ignores capitalization – so "red" and "Red" would both refer to our big red rock eater – except that it preserves capitalization when displaying text.)

```
@rename big red rock eater to "Red"
```

or how about:

```
@rename red to "Red (a big red rock eater)", "a big red rock
eater", "red rock eater", "rock eater", "eater", "Red"
```

There are some other options that I may explore later, in particular a `:title` verb, which usually but doesn't always return an item's name. I might want it to be `"Red (a big red rock eater)"` sometimes, and sometimes just, `"Red"`. I'll defer that for now, but it's an option to keep in mind.

```
@describe <your rock eater> as "<your idea of what a big red
rock eater looks like>"
```

Aha! Food will go where any other LambdaMOO food goes, that is, to its home if it has a `.home` property defined on it, otherwise to its owner's home. Whew, I'm glad to get that off my mind. (Creative development rarely follows a logical, linear sequence.)

Now that I've resolved the food issue to my satisfaction, let me show you a fun (optional) thing you can do with the description, which will further enrich your pronoun substitution skills. My rock eater is red, but he has spots, and I want the spots to be a different color each time someone looks. For variety.

Recall that we wrote a `:description` verb to enhance the pet rock's description with the addition of some moss. Our strategy here will be similar, with a custom description verb. In that verb, we will determine the color of the rock eater's spots. But we'll refer to the color in the description itself. Stay with me, here:

```
@property red.color "blue" r
```

Notice, again, that the property is `r` and not `rc`. That's because I intend to change it from within a verb, later, and not from the command line.

Start simple, then get fancy. For now, he'll have blue spots. Then we'll work on variable-colored spots. In my examples, I'm going to give the rock eater a fairly generic description, so as not to meddle with your own idea of the critter's physiognomy:

```
@describe red as "You see a big red rock eater with
%[tcolor] spots."

@verb red:description this none this rxd

@program red:description
base_description = pass(@args);
return $string_utils:pronoun_sub(base_description);
.
```

Here's what's going on with the new fancy footwork. Look first at the `.description` property. Recall that the "`%`" sign is a special symbol used by `$string_utils:pronoun_sub`. The square brackets delimit a unit of text that `$string_utils:pronoun_sub` will work on. The initial "`t`" inside the square brackets refers to `this`, a special variable in MOOcode that means the object on which the current verb is defined – here, the big red rock eater. The rest of what's in the square brackets is the name of a property on the object, in this case, `color`, which we just added. So in essence we're telling `$string_utils:pronoun_sub` to substitute `this.color` for `%[tcolor]`, and it does. Take a look at your rock eater now!

If my rock eater were always going to be red with blue spots, I would have just written that into the description and not gone to all the trouble. But I'm laying a foundation.

Next, I want the color of the spots to change. I'll start with a list of colors to choose from:

```
@prop red.color_list {} rc

@edit red.color_list
enter
blue
green
yellow
purple
orange
brown
tan
black
white
.
save
done
```

Make up your own list of colors. It can be any length. Now, in the description verb, we're going to select one of these colors at random and put that color into the property `this.color`. Then we'll do the substitution:

```
@program red:description
base_description = pass(@args);
"Pick a random color for the spots.";
this.color = this.color_list[random($)];
return $string_utils:pronoun_sub(base_description);
.
```

The only new thing here is `this.color_list[random($)]`, and even that isn't completely new, because of our work with the `.moss_list` property on the pet rock. Working from the inside out, then. "`$`", when used inside the square brackets, refers to the number of elements in the list. That's why it doesn't matter how long your `.color_list` is. If you add more colors later, or remove some, changing the length of the list, the code will still work. `random($)` (again, when within square brackets) gives a random number between 1 and the length of the list. So, `this.color = this.color_list[random($)]` sets the property `this.color` equal to a random element of the list `this.color_list`.

Here's my whiz-bang fancy version – see if you can figure out what's going on here:

```
@rmprop red.color
@prop red.color1 "blue" r
@prop red.color2 "tan" r

@describe red as "You see a big red rock eater with
%[tcolor1] and %[tcolor2] spots."

@program red:description
base_description = pass(@args);
"Fancy version!  Two *different* colors of spots!";
index = random(length(this.color_list));
this.color1 = this.color_list[index];
this.color2 = (listdelete(this.color_list,
index))[random($)];
return $string_utils:pronoun_sub(base_description);
.
```

And so you see, unlike leopards, big red rock eaters *can* change their spots. And hopefully you have at least a glimmer of how this technique could be adapted to other situations. You could give your rock eater a variable number of heads, for example.

Is It a Boy or a Girl?

Being a critter, let's suppose that it has a gender. This step isn't strictly necessary, except that it allows me to use pronoun substitutions when typing in

template messages for you to copy, and will give you some additional practice with pronouns as well. (Practice those pronoun substitutions!)

```
@chparent red to $gendered_object
```

What you get, by using this generic, is the verb for setting the gender, and a bunch of properties that are used for the various pronouns. Now you can set its gender, for example:

```
@gender red is male
```

Try typing:

```
@display red,
```

Note the comma, which means you want to display all inherited properties.

Pet the Nice Rock Eater...

We'll put a `:pet` verb on the rock eater, but this time we'll get a more gratifying response than we did from our rock. As always, we'll start simple and work up. Our initial goal will be to set up the `:pet` verb, and add message properties and their corresponding verbs:

```
@prop red.pet_msg "You pet %t."
@prop red.opet_msg "%N %<pets> %t."

@verb red:pet_msg tnt rxd
@addalias opet_msg to red:pet_msg

@program red:pet_msg
return $string_utils:pronoun_sub(this.(verb));
.

@verb red:pet this none none rxd

@program red:pet
player:tell(this:pet_msg());
player.location:announce(this:opet_msg());
.
```

This level of programming is so fundamental that I can almost type it in wholesale and have it work on the first try. (Though not quite -- I had a small typo in one of the verbs, and had to go back and fix it. Always test everything.) I usually set up the messages first, then the verb to do the pronoun substitution, then the verb that uses the messages.

So far, so good. But unlike a rock, a rock eater ought to react. So lets liven things up some. Here's a design decision: When it reacts, will the player and others in the room see the same thing, or different things? If I pet the rock eater, should I see, "The rock eater looks at you adoringly", and others see, "The rock eater looks at

Yib adoringly"? Or is it okay if everyone (including me) sees, "The rock eater looks at Yib adoringly"? The second choice is easier. In the spirit of starting easy and getting fancy, we'll do that. If the results aren't satisfying, we can gussy things up some more, later.

But now I have a dilemma. My short-range plan is to add a message such as, "`%T thumps his tail happily.`" But my long-range plan is to have an optional list of possible responses, and maybe I could use that list for more than one thing (after he's fed, for example). I want to name the message property in a way that is specific enough to be instructive to someone reading the code later, but general enough that I don't have to limit myself to the `:pet` verb. I think I'm going to make it a happy response, and later, if the occasion arises, I can add unhappy responses and/or neutral responses.

```
@prop red.happy_response_msg "%T looks at %n adoringly." rc
@addalias "happy_response_msg" to red:pet_msg

@program red:pet
player:tell(this:pet_msg());
player.location:announce(this:opet_msg());
player.location:announce_all(this:happy_response_msg());
.
```

Test this. Notice the call to `player.location:announce_all`. There are three forms of the `:announce` verb on the generic room. `:announce` announces text to everyone except the player who typed the command. `:announce_all` announces text to everyone, *including* the player who typed the command. `:announce_all_but` takes an additional argument that specifies a list of objects *not* to see the text. We'll use the third form later.

Now to diversify. Just as I've typed in a list of colors, I want to have a *list* of happy responses, and I want the message to select one of them. And, for compactness, I want to do it in the same message verb that I'm already using.

Here is the strategy: The `:pet_msg` verb is going to fetch `this.(verb)`, i.e. its corresponding message. If it's a quoted string (that's all we have, so far), then just do a pronoun substitution on that string. If it's a list (of strings), then select one of them at random, and *then* do the pronoun substitution:

```
@program red:pet_msg
"If it's a list, pick one at random.";
"Then do the pronoun substitution.";
msg = this.(verb);
if (typeof(msg) == LIST)
  msg = msg[random($)];
endif
return $string_utils:pronoun_sub(msg);
.
```

First, we'll test it to make sure all the old messages work fine. (Do that now.)

Then, edit red.happy_response_msg to be a list of strings:

```
@edit red.happy_response_msg
enter
%T thumps %[tpp] tail happily.
%T makes a rumbling noise in %[tpp] throat, reminiscent of a
cat's purring.
%T sighs contentedly.
%T does a happy little dance.
.
save
done
```

As with `%[tcolor]`, `$string_utils.pronoun_sub` translates `%[tpp]` into `this.pp`, which is a gendered object's *p*ossessive *p*ronoun.

Now, pet the nice rock eater to make sure that you get an appropriate variety of responses.

It Eats Rocks... Right?

We want to be able to feed rocks (and maybe other things) to the big red rock eater. Design decision: Shall it eat only rocks, or shall it eat anything, but especially like rocks? I choose to go for diversity on this one, so that you can feed the rock eater without having to hunt around for a rock. But either way, we'll need a strategy to tell whether an item *is* a rock or not, and there are some pitfalls there. Another design decision: Shall the rock eater eat players, or shall it be a domesticated rock eater that doesn't eat players? If it eats players, what happens to them then? I choose to sidestep that concern, and make a rock eater that eats rocks *and* other things, but doesn't eat players. (If I *were* going to have it eat players, I'd probably create a room that was the rock eater's tummy and move players there, where perhaps an adventure of some sort would await them. As always, I would start with something simple and make it progressively more complex.)

The syntax of the command will be, `feed <anything> to <rock eater>`. When the rock eater eats something, it will be moved to the rock eater itself. After a while, the food item will quietly go back to its home, or, if it doesn't have a home, to its owner's home. We have to account for the possibility that an item can't be moved to the rock eater (maybe its owner locked it down, for example), so we'll have messages to handle that case, and we have to account for someone *trying* to feed a player to the rock eater, and provide a suitable failure message. Accounting for every kind of misuse you can possibly think of is what makes for good, robust programming. We will have many opportunities to practice our pronoun substitution.

First, I'm going to tackle some behind-the-scenes stuff, in particular, filtering what things the rock eater can eat.

Baseline check – Do you still have your pet rock handy?

```
@move rock to red
```

You should get a message to the effect that either your rock doesn't want to go, or the big red rock eater didn't accept it.

```
@verb red:acceptable tnt rxd

@program red:acceptable
"This verb returns a truth value if an item may be moved to
the rock eater, and 0 if an item may not be moved to the
rock eater.";
{item} = args;
if (is_player(item))
  "No players!";
  result = 0;
else
  result = 1;
endif
return result;
.
```

`{item} = args;` This is a special kind of assignment statement. This verb won't be called from the command line. But it needs to receive some information (called *arguments*) so that it knows *what* is under consideration for acceptance. The built-in variable `args` is a list of arguments. We only expect one argument to this particular verb. The form `{item} = args`, is the preferred way of writing, `item = args[1]`. This is an idiom of the language; it's detailed in section 4.1.9 of the programmer's manual.

Here's one of my philosophies of writing code: Nobody writes bug-free code. All code will be maintained sooner or later. Even the person who writes the code sometimes forgets what e was thinking when e wrote it. It is better to write longer code that is easy to understand than to write compact code that is cryptic. If and only if you can compact the code without undue sacrifice of clarity, then more compact code is better.

Now, a shorter, more compact version:

```
@program red:acceptable
"This verb returns a truth value if an item may be moved to
the rock eater, and 0 if an item may not be moved to the
rock eater.";
"Players aren't accepted.";
{item} = args;
result = (is_player(item) ? 0 | 1);
return result;
.
```

And a shorter version still:

```
@program red:acceptable
"This verb returns a truth value if an item may be moved to
the rock eater, and 0 if an item may not be moved to the
```

```
rock eater.";
"Players aren't accepted.";
{item} = args;
return (is_player(item) ? 0 | 1);
.
```

Now try teleporting your rock to the rock eater. This should work. If you look at Red, you shouldn't see the rock, which is fine, since tummies are (usually) opaque. But is the rock really in there? Type:

```
@contents red
```

to see. This will also remind you of the object number of your rock, in case you want to teleport it back out again.

If you wanted to make a rock-shaped bulge in its tummy (say), you might alter its :description verb and/or add a verb called `:tell_contents`, which is what containers and rooms do. You would check the value of its `.contents` property and go from there. (The details are left as an exercise for the intrepid new programmer.)

Now, to the business of seeing to it that stomach contents get returned eventually.

The verb `:acceptable` is expected to return a truthful, silent answer, yes or no, to the question, "Will object A accept object B?" The verb `:accept` does the actual business of accepting (or rejecting), and may do any associated processing. For example, if you have ever tried to join someone who was in a locked room, you got a message saying that either you didn't want to go, or the room didn't accept you. That's done by the `:accept` verb. The `:accept` verb on the big red rock eater is going to fork a task to move the incoming item back to some appropriate place at a later time (if the item is acceptable). If the item is not acceptable (if it's a player, for example), then the `:accept` verb will just return a value of 0.

I had to tinker with the :accept verb quite a lot before it worked to my satisfaction, so I'll spare you the play-by-play development process. Don't worry if you don't understand every detail, but do try to follow along. Later you might want to write a custom `:accept` verb on some other object, and you'll know to revisit this example for a deeper understanding of it.

```
@prop red.digestion_duration 30 rc
```

This is the duration (in seconds) that a thing will stay in Red's tummy. While I'm testing, I'll set it to something short, like 30 seconds. When I'm satisfied that everything works, I'll change it to something like an hour, maybe.

```
@prop red.return_item_home_msg "The housekeeper arrives and
drops off %[titem]." rc
@prop red.item "This will be set to item.name by the :accept
verb." r
@addalias "return_item_home_msg" to red:pet_msg

@verb red:accept tnt rxd
```

```
@program red:accept
"Hold an item (while digesting), then try to send it home.";
{item} = args;
if (result = this:acceptable(@args))
  fork (this.digestion_duration)
    "Is it still there?";
    if (item.location == this)
      "Figure out where to send it, and try to send it
there.";
      place = ($object_utils:has_property(item, "home") ?
                item.home |
                item.owner.home);
      item:moveto(place);
      "Now see if it actually arrived.";
      if (item.location == place)
        if ($object_utils:has_verb(place, "announce_all"))
          "Set up item.name for appropriate pronoun
substitution.";
          this.item = item.name;
          place:announce_all(this:return_item_home_msg());
        endif
      else
        "We failed to get rid of it gracefully, just get rid
of it.";
        this:eject_basic(item);
      endif
    endif
  endfork
endif
return result;
.
```

There is one subtlety in particular to which I would like to call your attention. In the statement:

```
if (result = this:acceptable(@args))
```

I have done an assignment statement *within* the parenthetical conditional statement. This is perfectly legal and is often done. It's the same as:

```
if (this:acceptable(@args))
  <do stuff>
endif
return this:acceptable(@args);
```

except that I call the `:acceptable` verb once instead of twice, saving the result for later.

And now (at long last) we are ready to write the `:feed` verb itself.

Chow Time!

After all we've been through, this part will be quite easy. Here's the prototype:

```
@verb red:feed any to this rxd

@program red:feed
dobj:moveto(this);
if (dobj.location == this)
  player:tell("Red chomps hungrily.");
else
  player:tell("Red looks at you dubiously.");
endif
.
```

The item being fed to Red is the direct object of the command, and its object number is stored in the built-in variable `dobj`. Red is the indirect object of the command, and its object number will be stored in the built-in variable `iobj`, as well as the built-in variable `this` (the object on which the currently-executing verb is defined).

Here is the cleaned up, robust version:

```
@prop red.feed_msg "You feed %d to %t." rc
@prop red.ofeed_msg "%N %<feeds> %d to %t." rc
@prop red.no_feed_msg "You try to feed %d to %t." rc
@prop red.ono_feed_msg "%N %<tries> to feed %d to %t." rc
@prop red.ptui_msg "%T looks at %n dubiously. . o O ( Ptui!
)" rc

@addalias feed_msg to red:pet_msg
@addalias ofeed_msg to red:pet_msg
@addalias no_feed_msg to red:pet_msg
@addalias ono_feed_msg to red:pet_msg
@addalias ptui_msg to red:pet_msg

@program red:feed
"Try to feed it something.";
dobj:moveto(this);
if (dobj.location == this)
  "It ate the whole thing.";
  player:tell(this:feed_msg());
  player.location:announce(this:ofeed_msg());
  this.location:announce_all(
    this:happy_response_msg());
else
  "Ack!  Ptui!  Wouldn't accept it.";
  player:tell(this:no_feed_msg());
  player.location:announce(this:ono_feed_msg());
```

```
  this.location:announce_all(this:ptui_msg());
endif
.
```

Have you been testing all along? This works pretty darned well. Except that I tried to feed my pet rock to the rock eater before it had been returned again (I test a lot), and I got a traceback:

```
#26703:feed, line 2:  Invalid indirection
(End of traceback)
```

I can reproduce the error by trying to feed Red an object that doesn't exist:

```
feed mother-in-law to red

#26703:feed, line 2:  Invalid indirection
(End of traceback)
```

Looking at line 2, we're trying to move `dobj`. Aha! We need to add a check to make sure that the specified direct object is a *valid* object:

```
@program red:feed
"Try to feed it something.";
if (valid(dobj) && (dobj.location in {player,
player.location}) && (this.location in {player,
player.location}))
  dobj:moveto(this);
else
  "Either the thing being fed or the rock eater is not in
the vicinity, or the thing being fed isn't a valid object.";
  player:tell("I don't see that here.");
  "Just quit this verb right away.";
  return;
endif
if (dobj.location == this)
  "It ate the whole thing.";
  player:tell(this:feed_msg());
  player.location:announce(this:ofeed_msg());
  this.location:announce_all(this:happy_response_msg());
else
  "Ack!  Ptui!  Wouldn't accept it.";
  player:tell(this:no_feed_msg());
  player.location:announce(this:ono_feed_msg());
  this.location:announce_all(this:ptui_msg());
endif
.
```

I chose not to extract, "I don't see that here," into its own message. It isn't something I ever expect to change; I don't need to do any pronoun substitution; and it does double duty as a comment within the code. Also during play testing, I found out that someone could feed Red remotely, and I don't want that because the

messages display to the wrong place and seem incongruous, so I added a check to make sure that the thing being fed and the rock eater were either in the player's possession, or in the same location as the player. I should go back and add that same check to the :pet verb, too. I might not have found this bug on my own. Someone else noticed it. I like to invite friends to play-test my objects before I present them to the public, because that helps me find and fix the bugs I *didn't* think to look for.

```
@program red:pet
if (this.location in {player, player.location})
  player:tell(this:pet_msg());
  player.location:announce(this:opet_msg());
else
  player:tell("I don't see that here.");
endif
.
```

As a last little fillip, I offer the following: Since our pet_msg verb can handle a string *or* a list, I'm going to edit red.ptui_msg for greater variety:

```
@edit red.ptui_msg
enter
%T takes one taste of %d and promptly spits %[dpo] out
again. . o O ( Ptui! )
%T eats %d, but then, with a look of great consternation on
%[tpp] face, acks %[dpo] back up again. . o O ( Ptui! )
 .
save
done
```

(%[dpo] is the direct object's object pronoun.)

Whew, I'm hungry! I think I'll have a snack before going on to the next part.

The Breath of Life

The last step in making a pet is to enable it to respond spontaneously to things that happen around it, and thus seemingly take on a life of its own.

Whenever a verb calls a room's :announce verb (or one of its variants), the :announce verb calls the :tell verb on every object in the room that has one, and sends the specified text in as an argument. So by adding a :tell verb to our rock eater, it will automatically start "hearing" things going on around it. And then it can respond. We'll put in a random delay to make its actions seem even more independent:

```
@prop red.response_delay 20 rc
```

A delay (in seconds).

```
@prop red.action_msg {} rc
@edit red.action_msg
```

```
enter
%T chases %[tpp] tail in a slow, circling ballet.
%T leaps into the air and does a back flip, in a comic bid
for attention.
%T snuffles around, looking for rocks.
%T looks at you with big, sad, soulful eyes.
%T makes a wurfling sort of noise.
.
save
done

@addalias action_msg to red:pet_msg

@verb red:tell tnt rxd

@program red:tell
fork (random(this.response_delay))
    this.location:announce_all(this:action_msg());
endfork
.
```

Now say, "Boo," or something. You can check on your forked tasks by typing @forked.

Whoa! Down Boy!

```
@kill red:tell
```

Well! By now you've discovered that once started, Red just won't quit. This is because Red hears Red's own text, and responds to it! So what you get is a sort of chain reaction. Really, this is too much of a good thing, so now we have to work on toning things down some.

```
@program red:tell
fork (random(this.response_delay))
  this.location:announce_all_but({this},this:action_msg());
endfork
.
```

First, we'll use that third form of :announce that I mentioned earlier, :announce_all_but. This way you won't get an endless chain of actions. But you'll still get an action out of Red every single time there's a noise in the room. So for my next trick, I'm going to make it so that sometimes he responds, and sometimes he doesn't:

```
@prop red.action_odds 3 rc

@program red:tell
```

```
if (random(this.action_odds) == 1)
  fork (random(this.response_delay))
    "Odds of responding are 1 in this.action_odds.";
    this.location:announce_all_but({this},
this:action_msg());
  endfork
endif
.
```

This is better. But you might want a quieter pet still. (I did.) Whenever I would pet or feed Red, the messages would trigger an additional response, which still seemed like too much. So I can go back and edit the `:pet` and `:feed` verbs, *or* I can tinker with the `:tell` verb a bit more and prevent Red from hearing any of his own messages in the first place. Take a quick look at `help callers()`. This built-in function returns a list of all the object/verb pairs (tuples, actually -- there's additional information) that resulted in the current verb being called. So we're going to take a look at `callers()` from within `red:tell` and filter out any noise generated by Red himself. This is pretty advanced stuff, and even I do a bit of preliminary prototyping before using `callers()`, because I always forget just how it goes. Here's the prototype:

```
@prop red.callers 0 r
```

A temporary property to hold data that I want to look at.

```
@program red:tell
this.callers = callers();
if (random(this.action_odds) == 1)
  fork (random(this.response_delay))
    this.location:announce_all_but({this},
this:action_msg());
  endfork
endif
.
```

`Pet red`, to trigger the process. Now we will use a special form of the `eval` command, "`#`" to inspect the results:

```
#red
=> #26703 (Big Red Rock Eater)

#red.callers
=> {{#23920, "announce_all", #2, #3, #58337}, {#23920,
"announce_all", #24442, #17755, #58337}, {#23920,
"announce_all", #61050, #9805, #58337}, {#26703, "pet",
#58337, #26703, #58337}}
```

By scrutinizing it, I find the number of *my* rock eater (#26703 in this example). Your numbers will be different, but look anyway. This is a list of lists. I want to consider only the first elements, and see if Red's `:tell` verb was indirectly called by

Red. If it wasn't, then and only then will Red react. To do that, I'll use `$list_utils:slice` (there's help text – give it a try):

```
@program red:tell
"Respond to noises generated by anything  *except* this.";
if (random(this.action_odds) == 1)
  if (!(this in $list_utils:slice(callers())))
    fork (random(this.response_delay))
        this.location:announce_all_but({this},
          this:action_msg());
    endfork
  endif
else
  "This was the source of the noise, so do nothing.";
endif
.

@rmprop red.callers
```

Test this by saying things, petting your rock eater, feeding it, etc. Check `@forked` a lot.

This is almost perfect. (Wouldn't you know.) *If* you were to trigger Red's `:tell` verb and fork the task, then move Red to `#-1`[19], you would get a traceback, because `#-1` doesn't have an `:announce_all_but` verb. So we'll add a check for that:

```
@program red:tell
"Respond to noises generated by anything  *except* this.";
if (!(this in $list_utils:slice(callers())))
  if (random(this.action_odds) == 1)
    fork (random(this.response_delay))
      if ($object_utils:has_verb(this.location,
"announce_all_but"))
        this.location:announce_all_but({this},
this:action_msg());
      endif
    endfork
  endif
endif
.
```

Last, I take a step back to see if I can condense the code without sacrificing clarity, and I think I can. Those two nested `if` statements can be consolidated into one:

```
@program red:tell
"Respond to noises generated by anything  *except* this.";
if (!(this in $list_utils:slice(callers())) &&
```

[19] `#-1` is the null object. It has no properties or verbs.

```
    (random(this.action_odds) == 1))
  fork (random(this.response_delay))
    if ($object_utils:has_verb(this.location,
"announce_all_but"))
      this.location:announce_all_but({this},
this:action_msg());
    endif
  endfork
endif
.
```

Now all that's left is to tinker with `red.action_odds` and `red.response_delay` until you get the right feel for *your* pet. Some rock eaters might be placid, others might be frisky.

Care and Feeding of a Big Red Rock Eater

Just a last little bit of grooming to do, and we're done:

```
@prop red.obvious_verbs {} rc

@set red.obvious_verbs to {}
@edit red.obvious_verbs
enter
  pet %<what>
  feed <anything> to %<what>
.
save
done

@verb red:examine_verbs tnt rxd

@program red:examine_verbs
what = dobjstr;
vrbs = {};
for vrb in (this.obvious_verbs)
  vrbs = {@vrbs, $string_utils:substitute(vrb, {{"%<what>",
what}})};
endfor
return {"Obvious verbs:", @vrbs};
.
```

This is the usual stuff. I deliberately omitted `@gender %<what>` from the list of examine verbs, because that's an owner-only verb: You know it's there, and no one else needs to.

```
@prop red.help_msg {} rc

@edit red.help_msg
enter
This is %t.  Treat %[tpo] with love and kindness and %[tps]
will be your friend forever.

%[Tps] likes to eat rocks, but will eat just about anything
except players.  (Things which have been eaten will be moved
back to their homes after a while.)
.
save
done

@verb red:help_msg tnt rxd

@program red:help_msg
"This message has its own separate verb because the regular
message verb returns a random element of a list.  But if
this.help_msg is a list, we want the whole thing.";
return $string_utils:pronoun_sub(this.(verb));
.
```

This is your big red rock eater. Treat it with love and kindness, and it will be your friend forever.

Afterword

That's about it.

We started with the very rudiments of adding a verb to an object, and have worked our way through some fairly sophisticated stuff. You may or may not feel that you've grasped it all, but my primary goal was to give you some exposure to how MOOcode works, and to make it all less intimidating, in hopes that you will be inspired to explore further.

Be bold! Experiment. Try things. Don't be afraid of breaking something. There is very little on the MOO than can be harmed by accident, and even less that can't be fixed. Go for it!

MOO Programming Reference

This section is geared toward people who are at least somewhat comfortable with programming. For the fine points I refer you to the programmer's manual itself. Here, I have tried to present an overview of the MOO programming language in a format that is biased towards ease of reference rather than exhaustive explication. I have provided brief explanations of some points where I think clarification might be helpful.

Data Types

Variables are not of fixed data type; they become the data type of the value assigned to them. Use typeof(<variable>) to see what you've got.

INT	Integer: 29, 0, -0, 5
FLOAT	Floating point number: 29.0, 0.0, 5.0
NUM	Historical, same as INT. (FLOAT was a later addition.) FLOATs and INTs don't mix and match. Use toint() or tofloat() to force one to be the other. Note that (1 == 1.0) evaluates to false.
STR	A quoted string: "pickles". To include the double quote mark itself in a string, precede it with the backslash character: "Oliver shouts, \"Yow!\"" Strings are case-insensitive: ("carrot" == "CarROT") evaluates to true. Use equal("carrot", "CarROT") if you need to differentiate between the two.
OBJ	An object, e.g: #1234. In conditional statements, an object by itself evaluates to false. Use valid(<object>) instead. Some special objects: • #-1 or $nothing. Not valid, but anything may be moved there. The canonical invalid object. • #-2 or $ambiguous_match. • #-3 or $failed_match • $garbage This object and kids of it are valid but not useful. They are all owned by the special system player Hacker. When an object is recycled, it becomes a kid of $garbage.

LIST	{1, 2.5, "a string", {"a", "sublist"}, #321, E_RANGE, 2.5}
	
	LISTs are designated with curly braces ({}), may nest to an arbitrary number of levels, and their elements need not be of the same data type. Their elements are preserved in order and may include duplicates (i.e. they are not like mathematical sets).
	
	The @ operator yields the elements of the list as separate elements. Another way to phrase this is that it is the inverse of putting curly braces around some elements. Two canonical uses:
	
	<some-list> = {@<some-list>, <new_element>}; This is the usual way to add an element onto the end of a list. If a is {1, 2, 3}, and b is 4, then {@a, b} gives the four-element list {1, 2, 3, 4}, whereas {a, b} would give the two-element list {{1, 2, 3}, 4}.
	
	pass(@args); This causes the identically-named verb on the current object's parent to be run with the same argument list. If you just did pass(args), then the arguments to the verb being called would have an extra set of braces around them, thus making the argument list {{a, b, c}}, for example, instead of {a, b, c}.
	
	<element> in <some-list> will return the (1-based) index of the first instance of <element> in <some-list> if it is present, 0 otherwise. A typical usage might be:
	
	if (item.location in {player,
	 player.location})
	  <exprs>;
	endif

ERR	E_NONE	no error
	E_TYPE	type mismatch
	E_DIV	division by zero
	E_PERM	permission denied
	E_PROPNF	property not found
	E_VERBNF	verb not found
	E_VARNF	variable not found
	E_INVIND	invalid indirection
	E_RECMOVE	recursive move
	E_MAXREC	too many verb calls (max recursion)
	E_RANGE	range error (subscript too large, or zero, or negative)
	E_ARGS	incorrect number of arguments
	E_NACC	move refused by destination (i.e. object not acceptable)
	E_INVARG	invalid argument
	E_QUOTA	resource limit exceeded
	E_FLOAT	floating-point arithmetic error

Errors can be raised (yielding a traceback) or caught (then handled or ignored). The following two constructs are used to trap errors and deal with them:

```
`<expr1> ! ANY => <expr2>'
```

See section 4.1.12 of the programmer's manual. The single quotes are part of the expression, and are specifically back single quote and forward single quote.

```
try
  <exprs>;
except ANY
  <alternate exprs>;
endtry
```

See sections 4.2.7 and 4.2.8 of the programmer's manual.

Subscripting

Everything is 1-based.

You can subscript lists or strings.

<list-or-string>[<expr1>..<expr2>] gives slices (sub-list or sub-string). <expr1> and <expr2> must be in range; <list-or-string>[$] is the last element or character.

The following are well-formed:

```
<variable> = <list-or-string>[<expr>];
<variable> = <list-or-string>[<expr1..expr2>];
<list-or-string>[<expr1>] = <expr2>;
<some-list>[<expr1>..<expr2>] = <expr3>;
<some-string>[<expr1>..<expr2>] = <expr4>;
```

Note that in the above example, <expr3> must evaluate to a list and <expr4> must evaluate to a string.

Accessing Properties and Verbs on Objects

$<something> is the same as #0.<something>.

You can use parentheses to access property names and verbs dynamically:

```
<object>.(<calculated-property-name>)
<object>:(<calculated-verb-name>)
```

Variables

Variables are local and dynamic, coming into existence when assigned a value. For global variables, define and use a property.

In addition to the data types themselves (which evaluate to integers), the following built-in variables are provided:

this	The object on which the currently-running verb is defined.
player	The object number of the player who typed in the command.
caller	The objnum of the object on which the calling verb is defined, or player, if the verb was called from the command line.
verb	The name by which the currently-running verb was invoked. Verbs may have aliases.
args	The list of arguments with which a subroutine was called or, if a command-line verb, with which the command was invoked.
argstr	Everything that was typed in after the verb name on the command line.
dobj	The direct object as parsed from the command line.
dobjstr	The string from which dobj was matched.

prepstr	The string that was parsed as the preposition.
iobj	The indirect object as parsed from the command line.
iobjstr	The string from which iobj was matched.

Any of these may be reassigned within a verb and their values will persist into the next verb call except that `caller` will change to the current object, and a changed value of `player` will not persist unless the verb's owner is a wizard.

Scattering Assignment

Several variables may be assigned values in a single line, and this is often done to assign incoming arguments to named variables. A typical example might be:

```
{who, what, ?where = player.location, ?when=time()} = args;
```

See section 4.1.9 of the programmer's manual for a detailed explanation.

Operators

The following operators apply, in order of precedence:

Operator	Meaning
!	not
-	arithmetic negation (without a left operand)
^	exponentiation
*	multiplication
/	division
%	modulo
+	addition (note, + also concatenates two strings)
-	subtraction
==	is equal to (note, easy to confuse with the assignment operator – nasty!)
!=	is not equal to
<	less than
<=	less than or equal to (note, =< doesn't work)
>	greater than
>=	greater than or equal to (note, => doesn't work)
in	element position in a list
&&	logical "and"

<table>
<tr><td>||</td><td>logical “or”</td></tr>
<tr><td>… ? … | …</td><td>the conditional operator.
<expr1> ? <expr2> | <expr3>
is equivalent to:
if (<expr1)
  <expr2>;
else
  <expr3>;
endif</td></tr>
<tr><td>=</td><td>assignment (note, easy to confuse with a test for equality – nasty!)</td></tr>
</table>

Assignments may appear within expressions. Use parentheses liberally to avoid mistakes and confusion.

Truth Values

`0`, `-0`, `0.0`, `-0.0`, `""`, `{}`, errors, and objects all evaluate to false. Anything else evaluates to true.

Compound Statements

Use a semicolon after expressions within the body of a compound statement, but not after lines of the compound statement itself, thus:

```
if (<expr1>)
  "This is a comment.";
  <expr2>;
  <expr3>;
elseif (<expr4>)
  <more-exprs>;
elseif (<expr5>)
  <something else entirely>;
else
  "None of the above.";
  <final-exprs>;
endif
```

Looping

```
for <variable> in (<some-list>)
  <exprs>;
endfor

for <index> in [<int1>..<int2>]
  <exprs>;
endfor

while (<condition>)
  <exprs>;
endwhile

break;

break <name>;

continue;

continue <name>;
```

(See section 4.2.5 of the programmer's manual for the fine points of `break` and `continue`.)

Background Tasks

```
fork (<delay-in-seconds>)
  <exprs>;
endfork

fork <variable> (<delay-in-seconds>)
  <exprs>;
endfork
```

In the second example, `<variable>` receives the number (task_id) of the forked task.

Time Management

A task is the execution of a command from start to finish, or the execution of the statements within a `fork/endfork` statement from start to finish. Tasks are identified with numbers, and are allotted a fixed number of ticks and a fixed number of seconds for execution. The LambdaCore default is 30,000 ticks for a foreground

task and 15,000 ticks for a background task. Properties to override these numbers may be added to the object `$server_options` by a wizard and inspected (if present) by programmers.

If a task runs out of ticks, it is unceremoniously terminated by the system. If a task is in danger of running out of ticks, a programmer may get a new allotment by suspending the task briefly (note, `suspend ($login.current_lag)` is considered polite). When a task suspends, it obtains an additional allotment of ticks, so it is not uncommon to find or place a `suspend()` statement either right before or inside of a loop.

The utility verb `$command_utils:suspend_if_needed()` might suggest itself, but in fact it uses up a fair number of ticks, itself. Current fashion is to use a line of the form:

```
((ticks_left() < 3000) && suspend($login.current_lag));
```

See also sections 4.4 and 5.2.8 of the programmer's manual.

Argument Specifiers

When defining a verb on an object (with the `@verb` command), you must provide specifiers for the arguments with which the verb will be invoked.

The allowable specifiers are:

- direct object specifiers

```
this
any
none
```

- prepositions

```
none
any
with/using
at/to
in front of
in/inside/into
on/onto/upon/on top of
from/from inside/out of
over
through
under/underneath/beneath
behind
beside
for/about
is
```

```
as
of/off of
```

- indirect object specifiers

```
this
any
none
```

Definite and indefinite articles are omitted. When deciding which argument specifiers to use, it is helpful to imagine what a user would actually type when invoking the command, then generalize from that. When writing subroutines that aren't intended to be invoked from the command line, specify the arguments as "`this none this`".

Eval

The `eval` command evaluates a string as MOOcode. Like `say` and `emote`, it can be abbreviated to a single character command, `;`. A second form, `;;` evaluates a sequence of expressions, each terminated by a semicolon (as in a verb). Compound statements don't end in a semicolon. The form using two semicolons prints out 0 as its value; if you want to see results you should include a call to `player:tell();` at the end:

```
;;"Count players who have more than ten aliases"; total = 0;
for dude in (players()) if (length(dude.aliases) > 10) total
= total + 1; endif endfor player:tell("Total:  " +
tostr(total));
```

A second form of `eval`, "`#`" matches an object by name if it's in your vicinity, and is useful for looking at properties or just quickly finding out the object number of something close by. Property names can be chained:

```
#rock
#rock.moss_list
#yib p
#yib.aliases p
#yib.location.owner.name p
```

The last form, terminated by "` p`" matches the name of a player even if you're not in eir vicinity, so that you don't have to know eir number to look at a (readable) property on em. "`#`" can also be used with object numbers directly:

```
#58337.location.contents
```

Use `@setenv` to set up some commonly-used variable settings in advance:

```
@setenv me = player; here = player.location;
```

Inspect the result with

```
#me.eval_env
```

See also `help eval` and `help #`.

Ownership and Permissions

Every object has an owner. Every property on every object has an owner, but it doesn't have to be the same as the owner of the object. Every verb on every object has an owner, but it doesn't have to be the same as the owner of the object.

Task permissions are expressed as an object number, that of the player who owns the verb currently being executed.

The function `caller_perms()` returns the task permissions of the calling verb, or `#-1` (an invalid object) if the currently running verb was called from the command line.

Inherited verbs always have the same owner as the owner of the corresponding verb on the parent or ancestor object. They run with that owner's permissions, except that wizard-owned verbs can set the task permissions to another (usually non-wizardly) player.

To `+c` Or To `-c`, That Is The Question

Ownership of an inherited property *depends* on whether the property was initially defined as `+c` or `-c`. If it was defined as `+c` (think "may be *changed* by the owner of the child/descendent object"), then the property is *owned by* the owner of the child/descendent object. If the property was initially defined as `-c` then the property on all children and descendents is owned by the player who defined the property on the parent/ancestor object, *and its value can be changed by verbs running with that player's permissions*. This becomes relevant when one is making a generic object. If the owner of a child or descendent object will need to `@set` or otherwise change the property, then define it as `+c`. This is typically done for messages, and also for other parameters, for example the number of times one must turn the crank before the jack in the box pops out. If, on the other hand, one of the verbs you write on the generic will need to change the value of a property, then it should be defined as -c so that the property on all descendent objects will still be owned by you. Then your verbs, running with your permissions, can change it (for example, the number of times the crank on the jack in the box has been turned so far).

When you make an object strictly for your own use, it really doesn't matter whether the properties are `+c` or `-c`. It becomes an issue when other people make kids of your object. Then if a property that one of your verbs needs and tries to change is mistakenly `+c`, the verb will encounter a permissions error. If you `@chmod` the property to `-c`, then all *new* kids of the object will have that property owned by you, but it isn't changed retroactively for existing kids. If you make a property +c and find out later that it should have been -c, you can change it on all descendents by evaluating the following:

```
;$wiz_utils:set_property_flags(<object>, <property_name>,
<property_flags>)
```

You don't have to be a wizard to use this verb – just the owner of the object. Here's an illustration, supposing that a generic conker is object #1234:

```
;$wiz_utils:set_property_flags(#1234, "thwaps", "r")
```

This would have the effect of making the `.thwaps` property on all descendents of the generic conker readable but neither writable nor changeable (by owners of kid objects), and the property would be owned by the author of the generic conker in all cases (and could be changed by that player's verb(s)).

Hidden Treasures

Some verbs are called automatically, seemingly invisibly. Here are some of them:

`<object>:look_self`	Called when you look at `<object>`
`<object>:description`	Called (if it exists) by `:look_self`
`<object>:tell_contents`	Called (if it exists) by `:look_self`
`<object>:enterfunc`	Called when something is moved to `<object>`
`<object>:exitfunc`	Called when something is removed from `<object>`'s `.contents`.
`<room>:confunc`	Called when someone connects inside a room.
`<room>:disfunc`	Called when someone disconnects inside a room.
`<player>:confunc`	Called when a player connects
`<player>:disfunc`	Called when a player disconnects
`<object>:initialize`	Called when an object is created. Use, for example to initialize parameters on the kid of a generic object.
`<object>:recycle`	Called right before an object is recycled.
`<player-or-room>:huh`	Called if the parser can't find an object with the appropriate verbspec. This is how exits in rooms are invoked without the exit objects' having to be *in* rooms, for example.

A Couple "Tricks of the Trade"

Sending mail messages from within a verb: The relevant verb is `$mail_agent:send_message`. Personally, I always find the help text hard to read, so I am providing this illustrative example, which I hope may be helpful:

```
;$mail_agent:send_message(me, {me},
  "This is the subject heading", {"Line1", "Line2", "",
  "Oooga boooga!"})
```

Creating objects on the fly is fun, and possible if you are not over quota. Here is an example of how it's done.

```
@verb me:test none none none rd

@program me:test
"Sample verb to demonstrate creating an object on the fly.";
thing1 = `$recycler:_create($thing) ! ANY =>
  $nothing';
if (valid(thing1))
  thing1:set_name("thing1");
  thing1:moveto(player.location);
  "If you create a lot of things, then you need to measure
them as you go to avoid a 'resource limit exceeded' error.";
  $quota_utils:object_bytes(thing1);
  player:tell("You now have something that you didn't have
before!");
else
  player:tell(
    "Couldn't create thing1.  Don't know why.");
endif
.

test
```

To recycle an object (that you own) from within a verb:

```
$recycler:_recycle(<object>);
```

Programming Feature Objects

This is a very brief summary of the steps involved in creating a feature object. It isn't a tutorial on programming in general, but highlights a couple of quirks associated with programming this particular kind of object.

First, create a kid of the generic feature object:

```
@create $feature named <your-FO-name>
```

Then describe it.

Then program some verbs on it. Note that the verbs have to have the "x" permission flag set, so that they can be called from other verbs.

Then add help text. This can be done in either of two different ways. The first is to edit your feature object's `.help_msg` property. You should present each of the verbs on your FO that are intended for public use (as opposed to internal subroutines), give the syntax for the verb's usage, and a brief explanation of what the verb does. The other way is to put the documentation for each verb intended for public use as a set of comments at the top of the verb. The second is the officially preferred method (as per `help $feature`), but both will work.

THEN: In either case you must edit your FO's `.feature_verbs` property. If you put all the documentation in the help_msg property, then type:

```
;<your-FO>:set_feature_verbs({})
```

If you put the documentation for each public-use verb at the top of each verb, then type:

```
;<your-FO>:set_feature_verbs({"<verb1>", "<verb2>", ... ,
"<last-verb>"})
```

If you wish to restrict who may add your feature object, write a custom `:feature_ok` verb on it. This verb should return 0 if for whatever reason the person may not add the feature, or a truth value otherwise. An example of when this might come in handy might be a feature only for use by wizards.

See also `help features` and `help $feature`.

Built-In Functions

See the online help text or the programmer's manual for the specifics of each individual function – here's what's there:

The quintessential object-oriented function:

```
pass()
```

General operations applicable to all values:

```
typeof()        toobj()
tostr()         tofloat()
toliteral()     equal()
toint()         value_bytes()
tonum()         value_hash()
```

Operations on Numbers:

```
random()        max()
min()           abs()
```

```
floatstr()
sqrt()
sin()
cos()
tan()
asin()
acos()
atan()
sinh()
cosh()
tanh()
exp()
log()
log10()
ceil()
floor()
trunc()
```

Operations on Strings:

```
length()
strsub()
index()
rindex()
strcmp()
decode_binary()
encode_binary()
match()
rmatch()
substitute()
crypt()
string_hash()
binary_hash()
```

Operations on Lists:

```
length()
is_member()
listinsert()
listappend()
listdelete()
listset()
setadd()
setremove()
```

Manipulating Objects:

```
chparent()
valid()
parent()
children()
object_bytes()
max_object()
move()
properties()
property_info()
set_property_info()
add_property()
delete_property()
is_clear_property()
clear_property()
verbs()
verb_info()
set_verb_info()
verb_args()
set_verb_args()
add_verb()
delete_verb()
verb_code()
set_verb_code()
disassemble()
players()
is_player()
set_player_flag()
connected_players()
connected_seconds()
idle_seconds()
notify()
buffered_output_length()
read()
force_input()
```

```
flush_input()                    connection_option()
output_delimiters()              open_network_connection()
boot_player()                    listen()
connection_name()                unlisten()
set_connection_option()          listeners()
connection_options()
```

Operations Involving Times and Dates:

```
time()
ctime()
```

MOO-Code Evaluation and Task Manipulation:

```
raise()                          task_id()
call_function()                  suspend()
function_info()                  resume()
eval()                           queue_info()
set_task_perms()                 queued_tasks()
caller_perms()                   kill_task()
ticks_left()                     callers()
seconds_left()                   task_stack
```

Administrative Operations:

```
server_log()                     dump_database()
renumber()                       db_disk_size()
reset_max_object()               shutdown()
memory_usage()
```

$Utils

Some of the built-in functions are used frequently in everyday programming, some are used rarely, or only by wizards, or both. The MOO also provides a collection of utilities packages. Each utilities package has its own top-level help text, and each verb has more detailed help text. This list is just the $utils packages available in LambdaCore. A reference list of all the verbs on each is provided in Appendix B.

```
$wiz_utils                       $lock_utils
$math_utils, $trig_utils         $list_utils
$set_utils                       $command_utils
$seq_utils                       $code_utils
$gender_utils                    $building_utils
$time_utils                      $string_utils
$match_utils                     $generic_utils
$object_utils                    $quota_utils
```

```
$byte_quota_utils              $matrix_utils
$object_quota_utils            $convert_utils
```

Part II

LambdaMOO

LambdaMOO first went online on October 31, 1990, and is the oldest and largest MOO going. Its theme of a large mansion and its grounds is something that almost everyone can relate to, and many people have done an enormous amount of wonderfully creative building there.

Part I provided information that is applicable to all or nearly all MOOs based on LambdaCore. Part II will detail a variety of things that are specific to LambdaMOO, though many of them (feature objects, in particular) are available in some form on other MOOs.

Chapter 7 – Yib's Guide To Interesting Places

This is a personal compendium of my favorite places in and around Lambda House. Each section is devoted to a particular room, and describes why it's interesting and how to get there. Walking directions to each room are given in terms of previously-described rooms. To get an overview of the layout of the main part of the house, type `help map`. Other maps are available in the map room, which is northeast from the library.

An effort such as this can only begin to scratch the surface of a place as rich and varied as LambdaMOO, but it is my salute to those who have given so generously of their time and creativity.

The Coat Closet (#11) and The Linen Closet (#47726)

The coat closet is the room where all guests start when they first connect. Its other salient feature is that it's dark in there: You can hear people (provided someone says something), but, unlike most other rooms, you can't see them.

The linen closet is like the coat closet except that in addition to being dark it is also quiet: You neither hear nor see others who might also be there. This can be useful, for example, if you want to read help texts or mailing lists undisturbed.

There is a lever in each closet that will move you to the other one.

As of May, 2003, guests' home is set to the linen closet. A petition to set guests' home to the coat closet (since they begin there) awaits vetting by the wizards.

The coat closet is northeast from the living room.

To get to the linen closet from the living room, go north to the entrance hall, east twice, up, then northeast.

The Living Room (#17)

After the two closets, the living room is where it all begins. Accessed either by going out the door of the coat closet, or by pressing the button in the linen closet, this is where most new MOOers first hang out and visit before they find their own circle of friends, and where many old MOOers hang out to converse with one another and meet new people.

The cockatoo has graced the living room with its charming conversation almost since time immemorial. It "listens" to the conversation, "learns" short phrases, and parrots them back at a later time. You can make it be quiet for a while by typing `gag bird`. If you type `@gag! bird` instead (note the "@" sign and the exclamation point), then you will stop hearing it at all until you `@ungag` it again. You can erase

its stored up set of learned phrases by scrubbing it. You can also poke it to make it squawk something, feed it, and release it.

The fireplace/mantel is also an old and venerable living room object. You can put things on the mantel, and take things from it. You can build a fire, and roast things on it. Advanced investigations will yield a way to customize what players see if someone tries to roast *you* in there! If there isn't a fire going, you can enter the fireplace, pull the chain in the sooty chamber, and wind up somewhere else – a fun way to explore. Note that the name you use to identify it makes a difference: `put <something> on mantel` is different from `put <something> in fireplace` (especially if there is a fire burning!).

The living room's description mentioned a couch (two sets of couches, actually) for the longest time. Then someone built an actual VR couch. You can sit on it, shove people off, stuff things into it, jostle it, reupholster it, search for things, and (occasionally) fall in. From under the couch cushions, you can shout, or return something that falls in (from someone *else's* pockets, to be sure).

The birthday machine lets you look up people's birthdays, and register your own, if you so choose.

The Helpful Person Finder (one of two, the other residing in the Library) can be used to find helpful persons. Or ask around.

Please straighten the welcome poster if it happens to be a bit crooked.

The Entrance Hall (#19)

This small foyer is the hub of the currently-occupied portion of the house.

The mirror at about head height has been in the entrance hall for ages. Be advised that if you look at it, you will be transported to another place, the Looking Glass Tavern Bar. Newer portals (as such things are commonly called) let you look at and examine them without committing yourself to being transported, but this one has always provided the rush of going somewhere without quite knowing what's happening yet. If you want to return, just look at the mirror in the bar, and you will find yourself back in the entrance hall again.

The globe is a very old artifact, authored by waffle (#9082). You can enter the globe, and can add and describe your home country/state/town. Except for the top level (the seven continents), anyone may add or remove a place, so it's pretty dynamic. If you return later and find your home town missing, just add it again.

Edgar the Footman is one of the household servants. He isn't so good at fetching things, but you can give something to him and ask him to deliver it to someone else for you.

To get to the entrance hall, go north from the living room.

The Dining Room (#28)

I don't know whether anyone has ever actually dined in The Dining Room. For as long as I can remember, the dining room has been a repository for various toys and games. (Scrabble has had a particularly enduring popularity.)

Different games have different syntaxes for starting play. Many are set up as portable rooms. Enter them to get started. Others work differently. Examine them to get initial information on how to play.

The 'nopoly bank contains the Red Hotel, which is one of several places on the MOO where one can set up a home.

There is a chest for the games, but few people seem to put games away. Or, if they do, the games are set to return to the dining room proper. Oh, well.

To get to the dining room, go west from the entrance hall.

The Kitchen (#24)

The Kitchen is one of the original rooms of the mansion, and has within it many quaint examples of an older style of programming, when objects did not routinely have help text associated with them.

The cookbook explains how to make MOO food. Food and food fights were a big thing in the early days. People threw food, and dodged food, and had transparent shields rendering them impervious to food fights. Most players, though they don't realize it, can still type `boring on` and `boring off` to activate and deactivate this. You needn't feel inhibited about throwing food around: Mr. Clean will come and clean it all up again.

The cuisinart turns ordinary things into a puree, and purees are a kind of food, to be eaten and thrown etc. (The original ingredients of a puree are temporarily removed from sight, but are not really gone.) The cuisinart doesn't have help text per se, but you can type `about cuisinart` to get more information.

You can cook food and other things (and people) in the microwave.

The dishwasher is one way to remove food from things (and people).

A carrot is one kind of food. You can eat it. You can also bonk people with it. This is the original precursor to bonking feature objects, which force other people to say things. Strictly speaking, bonkers are unmannerly, but they have nonetheless become a part of our MOO culture, and in fact were used on MUDs even before MOOs came into existence.

There's also a nice plate of chocolate chip cookies. Help yourself. Take them to the living room to share!

Then there's the kitchen sink. You can wash your hands in it, or anything else. Once upon a time, before we had `@go` and `@join`, people had to wear teleporting rings in order to zip from Point A to Point B. Rings had a nasty tendency to slip down the drain while you were washing your hands.

The piece of Saran Wrap(tm) is one of the MOO's oldest practical jokes. People in the know can ask it not to cling to them. People not in the know attract it like, well, a clingy piece of Saran Wrap(tm). It flutters just at the edge of their vision until they peel it off, and people who look at them notice that they look silly wearing it. It has been set permanently not to pester guests, which got mixed reviews at the time. At one point it was fertile, and other people could make their own instances of it, but things started to get out of hand, so now there's only one again. The piece of Saran Wrap(tm) has had several owners over the years.

You can read and post notes on the refrigerator. Some notes have fallen into the pile of scraps of paper.

There is a vent, which you can enter. Many of the original rooms in the house are connected by the vent system, and this is an alternate way to get around.

There used to be a blender, which was a colorful and definitive way to leave the MOO. Lambda decided that it was just too gross, and banned it, so then you could off yourself by walking off the Edge of the World, at the westernmost end of the street in front of the house. Later, a ballot was passed that made it so that walking off the Edge only resulted in a long `@newting`, and maybe you stayed away, and maybe you came back. Then there was another ballot to bring back the blender and reduce the `@newting` time at the Edge of the World. (That ballot is in the throes of being implemented at the time of this writing.)

To get to the kitchen, go northwest from the living room. To get back to the living room, you must first go northeast to the entrance hall, then south to the living room. In spite of the protocol specified in `help theme`, which says that exits are supposed to be symmetric, this connection remains as an artifact of the original house on which Lambda House was modeled. There, when going from the living room to the kitchen, it feels like a one-step process, but going from the kitchen to the living room feels like going into the entrance hall, then turning a corner and going south into the living room.

The Family Room (#33)

The family room is "comfortably crowded with plush couches and easy chairs." Though not an especially popular venue today, it seems to have been quite the playground for early MOO technologists.

Of particular historical interest is the postcard from France. Reading the postcard will transport you to Un Cafe Parisien. (Type `out` to get back.) This is an early example of what has come to be called *themely transition*, whereby one is transported to a place that has nothing to do with the mansion and grounds proper, but the connection is nonetheless accomplished in a logical (if magical) way.

Blob's Apple][e computer seems to be a prime example of early experiments in MOO coding. I haven't figured out how to use it outside the following sequence of commands:

```
watch computer
put diskette in computer
turn computer on
```

<wait>

```
type "connect guest" on computer
type "look" on computer
type "out" on computer
```

<the computer crashes>

```
type "z" on computer
```

<the program restarts>

```
turn computer off
ignore computer
take diskette from computer
turn computer off
```

There is every reason to think that other programs could be written for it, if one wanted to do so.

The bookcase itself is of a fairly old vintage, before we had (various) generic open containers like tables and shelves and bowls and vases. So the bookcase is a regular `$container`. The description does not mention its having doors, but if you examine it, you will see that one of the obvious verbs is to open it, and when you do, the messages refer to opening doors. If the doors aren't open and you merely look at it, rather than examining it, then it isn't obvious that there's stuff in it.

You can `lookup` a word in the dictionary.

The VCR works and is fun. A camcorder and several tapes can be found in the bookcase.

The wind-up sushi works, and does about as much as you might expect a wind-up sushi to do.

You can rub Aladdin's lamp to get some food. (See the entry on The Kitchen for more information about food.)

The display case hanging on the wall seems to be intended as a haven for portable rooms, but I have not been able to confirm this for sure.

To get to the family room, go west from the kitchen.

The Laundry Room (#36)

As with any household, a variety of items come and go from the laundry room, but there always seems to be a pile of dirty laundry here. You can climb onto the pile, and jump off again.

There is a laundry chute here, which is one of the most delightful examples of interconnectedness among MOO objects that I know of. Easy puzzle: It is possible to enter the laundry chute from below.

From above, you can drop things into the chute (they wind up in the laundry room, as you would expect), and you can also slide down it yourself (woo-hoo!).

To get to the laundry room, go north from the family room.

Housekeeper's Quarters (#16563)

As you might expect, the housekeeper's quarters are neat and tidy.

By convention, the housekeeper's gender is not assumed, and no pronouns are used. Thus:

```
The housekeeper does the housekeeper's job efficiently and
indefatigably.
```

Since two of the most basic things one can do with objects is take them and drop them, one can easily imagine that dealing with clutter became an issue early on, and enlisting the services of the housekeeper is one way of managing said clutter. The housekeeper's mission is to return objects to their places, and there is a long note in the housekeeper's quarters which explains how to request that an object be returned periodically to a particular location. The housekeeper tries not to clean a thing up which the housekeeper believes to be in use, either because it is held by a connected player, or in the room with a connected player, or otherwise claims to be in use via a property or verb on the object in question.

There arises, then, the prospect of the owner of the location where someone wishes to store an object not wishing em to store it there. On LambdaMOO it is generally recognized that the rights of the former supercede those of the latter. There also is the quite separate question of how a room owner may consistently keep a room free of any and all unwanted objects. One way this is done is with the generic self-cleaning room, #27777. The housekeeper is savvy enough not to accept instructions to clean an item to a room whose owner has set the room to be self-cleaning, *unless* the room's owner has first indicated the item is welcome there by adding the item to the room's list of residents.

To get to the housekeeper's quarters, go west from the laundry room.

The Powder Room (#116)

The powder room has exactly the amenities that you would expect a powder room to have. Our plumbing is in good working order, in spite of the wide variety of objects that people flush down our loo.

The mirror hanging on the west wall of the powder room is kin to the mirror hanging on the east wall of the entrance hall: looking at it will transport you.

To get to the powder room, go east from the entrance hall, then north.

Ground Floor Stairwell (#6182)

The stairwell provides access to the original three floors of the mansion, and to the original set of sub-basements. Floors 1 and 2 generally contain various private suites; above those is access to the roof and the observatory. Below are an uncounted number of basements and sub-basements. The policy for getting a room connected to the lower stairwell is generous, so there's no telling what you might find in your explorations.

The house has an elevator. On the upper floors, you must exit the stairwell to access the elevator. In the basements, elevator access is from the stairwell itself. You can take the elevator to China!

<from Ground Floor Stairwell>

```
press down
```

<the elevator arrives, the doors open>

```
east
read directory
press B4251998
```

<wait>

```
out
```

To get to the ground floor stairwell, go east from the entrance hall, then south.

Master Bedroom (#6179)

Many people are put off when they first enter the Master Bedroom because of that PESKY ALARM! You walk in, the alarm goes off, and all of a sudden you feel like an unwelcome intruder instead of a welcome guest. The effect is quite impressive. But don't worry, you aren't irritating anyone; it's a puzzle. This puzzle is hard until you figure out that you need to use objects from other rooms besides the Master Bedroom, medium hard until you identify the objects needed to solve it (hint, everything you need is in the suite of rooms connected to the master bedroom), and fairly easy once you actually get going on it. Once you solve the puzzle, the alarm will no longer go off when you enter the room.

To get to the master bedroom, go east from the entrance hall twice, then south.

The Deck (#349)

The deck connects several rooms at the back of the main part of the house.

The Rube Goldberg contraption, which resides on the deck, is a fine bit of building, in that any member of the community is free to add to it, and many have.

At the easiest level, you can pull the lever to see a very wide variety of actions. Or you can enter it and learn how to add your own bit of genius to it.

The instructions are a bit hard to follow, but it helps to think of a "motion" not as a motion so much as a state the machine is in. Interactions take the machine from one state to another. Motions have names (and sometimes descriptions), interactions have names and descriptions. Only the descriptions of interactions are actually presented when someone pulls the lever.

One way to get your bearings before adding a motion or interaction is to enter the machine and type `motions`. Select a motion from the list (`ball in passages` for example), and type `show <motion>`. Look at the list of preceding and following interactions, choose a following interaction, and type `show <following interaction>`. Look at the "`from motion`" and "`to motion`", and show the "`to motion`". By repeating this sequence, you can follow one possible output path of the machine.

Then, of course, you'll want to add your own. Simplest is to skip creating your own motion, and just create an interaction connecting two existing motions in a new way. Note that the contraption's parsing code doesn't seem to like quotation marks, which is contrary to the way coding conventions have evolved since the contraption was made. Here's a simple sequence for adding an interaction connecting two existing motions:

```
create interaction named quickstop from ball in passages to
finish

describe quickstop as The ball shunts over into a
catapult.*The catapult sends the ball flying across the
contraption where it hits a large red button labeled
'EMERGENCY STOP'.
```

(The asterisk, " `*`" is part of the syntax, and makes things appear on separate lines.)

Then you'll want to see your interaction actually happen, so type `exit` and start pulling the lever on the machine relentlessly. The charm of the contraption is that it has so many motions and interactions. The frustration is that because there are so many, yours may take forever to come up. The contraption's documentation says that it tries to show each interaction with equal probability, given the limitations of the connections. In reality, there are some that seem to come up over and over again, and it starts to get tedious. But just when you get exasperated, some never-before-seen action occurs which is really funny. That's life, I guess.

(Author's note: I try to test everything that I document. I haven't yet seen my interaction, and I've pulled the lever a *lot* of times. Your mileage may vary.)

To get to the deck, go south from the master bedroom, south from the half-bath in the master bedroom suite, or southeast from the living room.

The Hot Tub Deck (#385)

The Hot Tub Deck connects the deck to the hot tub, and provides access to the tub control room and to the roof (via the rose trellis).

Nowadays, a room such as the hot tub would probably be implemented as a portable room (with exterior and interior descriptions), albeit one that would be locked in place. The technology of portable rooms probably didn't exist at the time that the hot tub deck and hot tub were built, and access to the hot tub was implemented using a combination of verbs on the room and regular exits, in this case one with several aliases. You can enter the hot tub by typing any of the following: `enter, enter tub, enter hot tub, enter hottub, swim, dive, tub, hottub, hot-tub`.

You can cover and uncover the hot tub, and can push people into it (though this is considered nasty by some).

The rose trellis is one of the neatest exits that I know of. You can ascend either by typing `climb trellis` or `up`. The exit up connects, eventually, to the roof of Lambda's Den. En route, you'll pass by the window of Lambda's Den, and you can `peer in window` and `tap on window`. Although the occupants of Lambda's Den are rarely logged on, when they are they can and do notice when you peer and/or tap on the window. When you do get to the roof, you can try your hand at bungee jumping (The folks in the hot tub below will get to see your performance.)

To get to the hot tub deck, go east from the deck.

The Hot Tub (#388)

The Hot Tub has long been a popular alternative to the living room as a gathering place. Boring conversation in the living room? Try the hot tub. In earlier days, one might have expected conversation in the hot tub to tend a bit more to the erotic than that of the living room, though living room habitues have never exactly been thought of as prudish. In more recent times, however, venues with more overt names (e.g. "Sensual Respites" and "The Sex Room") have tried to take over that function, with some success. This author still raises an eyebrow or two if spotted in the hot tub, however, so perhaps the magic hasn't totally worn off.

You can check the current temperature of the water by looking at the thermometer. (If it isn't to your satisfaction, find the tub control room and adjust the thermostat.)

You can splash another person (or thing), and you can dunk another person (or thing). You can `push left button` to turn the light off and on, and you can `push right button` to turn the bubbling jets on or off. Surprisingly, perhaps, you can cover and uncover the tub from within, as well.

The hot tub bar was a later addition, and is another example of food and drink, different from those obtainable in the kitchen. Typing `drink <anything> from bar` will get you a drink (of sorts). You will be prompted for alcoholic vs. non-

alcoholic. You can also type `eat <anything> from bar` to obtain more substantial refreshment.

To get to the hot tub, type `enter tub` from the hot tub deck.

The Tub Control Room (#491)

The most obvious object in this room is the thermostat for the hot tub. The minimum setting is 80, and the maximum is 120. As with real hot tubs, there is a significant delay between resetting the thermostat and the water actually reaching the desired temperature. Just as in the offline world, players here will reliably try to over-boost or under-boost the temperature setting in an effort to make the tub heat up or cool down faster. This routinely backfires, of course, in that the tub doesn't heat up or cool down any faster, and, hours later, the tub reaches an undesirably warm or cool temperature. But people will be people, after all, and this great bit of MOO-coding catches us at it red-handed.

I can't imagine why anyone would want to do such a thing, but you can pull the plug (type `pull plug`) and drain the tub. If it's already unplugged, you can plug it back up again (type `push plug` or `plug hole`).

The description of this room changes, depending on whether the tub light is on or off, whether the bubbling jets are on or off, and whether the tub is empty or full.

The radio transmitter that is there appears to be a relic of a long ago programming project which never really came to much. To use it, you have to find one of its counterparts and a partner. Each of you has to hold the radio, and then you can talk back and forth on it. The LambdaMOO Programmers Reference Manual makes a references to this radio.

There is an unmarked exit south from this room, which leads you to a dark, damp tunnel. From there you can get to the pool deck, or explore the tunnel further on to the west, which leads to an RPG area. (LambdaMOO has its roots in the Dungeons and Dragons tradition, and the RPG (Role Playing Game) is our version of that. You get initiated, train, look for treasure, and fight monsters and such. I am not an RPG player myself, and will not be documenting the RPG. For those who are interested: From the entrance hall, go east, east, up, east, east, south, east, and then north. You should be in the initiation chamber (#88). After initiation, go back (south) to the atrium and read the Tome of Lore that is there.)

To get to the tub control room, proceed down from the hot tub deck.

The Pool Deck (#1425)

This is another bit of quite old building, done in an older style. The description refers to some screws, but `exam screws`, the logical inquiry, nets the response, `I see no "screws" here.` How frustrating. You can, however, try `unscrew screws` or `screw screws`, depending on their state, and you will get a response that

makes a bit more sense. You guessed it, it's a puzzle. The crux of the puzzle is this: You need to find the appropriate tool (hint, it's a screwdriver). If you are holding it, then unscrewing the screws will reveal an underground tunnel that leads to an RPG area. It's a hard puzzle in that you have to go pretty far afield to find the screwdriver, but not an impossible puzzle in that the screwdriver is in a place where you might expect to find one.

There is a sign, with a glass bottle attached. There is nothing special about the bottle (that I can find). It's just there for illustrative purposes. The sign, however, gives us another glimpse of human nature at its best. Those actions it prohibits are actions the room knows about, and I'll leave it at that.

To enter the pool, type `enter pool`. Better: Go up, and try a dive from the diving board. Not to be missed: Go up twice, and try the high dive.

To get to the pool deck, go south from the deck.

The Pool (#1428)

The pool is bigger than it looks, and allows for all sorts of shenanigans. Go there with several friends.

As with the hot tub, you can splash and dunk people.

There are a variety of pool toys, including a lily pad, an air mattress, a shark cage, a hypercube tub toy, a green plastic snake, a beach ball, and a fishing boat. In general, the following apply:

```
float on <pool toy>
leave <pool toy>
push <anyone> from pool toy
deflate <pool toy>
inflate <pool toy>
```

(`deflate <pool toy>`: [with or without someone on it])

Different pool toys have different capacities, so different things happen if more than one person tries to float on one. One of them has a special deflate verb. The fishing boat has extra programming: You can fish from it. Surprising, in a chlorinated pool, no? Reminder: Sometimes you have to be patient when fishing.

The pool sweep is a fine contraption. Its messages are triggered by noises in the pool, so if you just hang out there by yourself, it won't do much, but if there's a pool party going on, you'll hear more from it. If you don't want to hear from it, you can `gag` it (different from `@gag`). If you try to take it, there are two different results, depending on whether it is gagged or not.

There is an interesting variable exit from the pool. Typing `dive` will take you down to the bottom, and from there, who knows?

To get to the pool, type `swim` or `jump` from the pool deck.

Hedge Maze (#17682)

For the longest time, I thought the hedge maze was just another bit of atmospheric tiny scenery: all that hedgework, and nothing else. For all I knew, it was a clever bit of programming to take you around in convoluted and never-ending loops, end of story. Later on someone pointed out that it's a puzzle. One of the things that I think holds people back from tackling puzzles is not knowing what they're letting themselves in for. So, without giving away the game, here are a few facts about the hedge maze that might help you to decide whether to tackle it and when.

It's finite, and fixed, which is to say that its layout is constant: The arrangement of rooms and exits will be the same tomorrow as it is today.

All exits are symmetric: If you go southeast, then northwest will take you back to where you just were.

There's a goal: When you've reached it, you'll know.

Scope: When I was working on it, the questions that kept coming to mind were, "How big *is* this thing, anyway?" and, later, "Is there an end?" It's pretty big. When I finally decided to solve it, it took me about two hours, spread over two sessions. Compulsive person that I am, I then mapped out the whole thing on a blotter-sized piece of graph paper. There are 199 locations within it. The shortest path to the middle is 83 steps. The maze entrance is its northwestern-most point.

It's clever. If you teleport out, then teleport back in, it takes you back to the place where you left off. This may or may not be to your liking. If not, you can go back to the pool deck and start at the entrance again.

Historical note: It's one room, cleverly programmed to seem like lots of separate rooms. This sort of thing later became known as a *multi-room*. At one time, players were limited to a finite number of objects, rather than a finite amount of byte-based quota, and a multi-room was one way to simulate having more objects (for rooms and exits connecting them) than you actually had. After the transition to byte-based quota, multi-rooms were strongly discouraged – better to have a discrete number of actual rooms with real exits between them.

To get to the hedge maze, go southeast from the pool deck.

The Kitchen Patio (#1467)

The Kitchen Patio is mostly a linking room between the kitchen and the yard, but it does have a couple of interactive objects such as you might find, well, on a kitchen patio. These are Mazer's cricket ball, tennis ball, and Ball-buddy. Here's a short primer on playing a simple game of catch.

For the sake of discussion, let's use the tennis ball:

```
take tennis
throw tennis to buddy
```

<buddy catches the ball>

<buddy throws the ball back to you>

```
catch tennis
```

The fancy part: You can embellish the messages that are displayed when you throw or catch a ball. The syntax is laid out fairly clearly (if not in a logical sequence) when you examine the ball, but here are a couple of examples. Note, the message to others prepends your name, so you have to omit that. Another note: The original author (not the current owner) tried to be clever with pronoun substitution, but in this reviewer's opinion wound up making it somewhat more fussy and complicated. For "throw" messages, indirect object pronouns refer to the catcher. For "catch" messages, indirect object pronouns refer to the thrower:

```
teach tennis to throw with "You lob %t gently." and "gently
lobs %t toward %i."
teach tennis to catch with "You twirl, then catch." and
"twirls around gracefully before catching it.  %I cheers
wildly."
show tennis
unteach tennis to throw with 1
unteach tennis to catch with 1
```

Ball-buddy is very handy for testing your messages. You have to put a "%i" somewhere in your throw message (the one that others see) in order for Ball-buddy to know it's for him, though.

To get to the kitchen patio, go south from the kitchen or northwest from the pool deck.

The Yard (#2883)

The yard is showing signs of neglect. It used to look like this:

```
The yard has carefully manicured grass that snuggles up to
the rosebushes to the east, and extends southward a ways
toward what appears to be a gazebo.  Off to the west, the
yard becomes less well tended.  To the north is a sliding
glass door into the house.
```

Now it looks like this:

```
The yard has ankle-high grass that turns to weeds next to
the rosebushes to the east, and extends southward a ways
toward what appears to be an old gazebo.  Off to the west,
the yard becomes even less well tended.  To the north is a
crooked sliding glass door into the house.
```

Kilik and I were bored one day, and we decided between us that the house as a whole seemed too static. We looked around for small ways for the house to evolve, and we decided that maybe things would get dusty and fall into disrepair somewhat. We hardly went on a rampage, but here and there around the house, in areas that

one of us owns, or that are owned by someone whose stuff we have permission to tweak, there are slight changes. The welcome poster in the living room becoming slightly crooked from time to time is one example. Another is the condition of the yard. Original descriptions of things that have changed are kept in the property .description_OEM, so no original work has been destroyed.

So now the yard has ankle high grass. Unless you mow it. Now, our lawn mower is quite a piece of work, too. It was programmed by Wintermute, who always had very dark sense of humor (and view of life). If you start the lawnmower in the "run" position instead of the "idle" position, it will come straight at you and chop you to bits. Being Wintermute's creation, it does the same thing even if you start it in the "idle" position and then set it to "run", though I could have sworn that once upon a time there was a way to mow the lawn safely. Either way, if you subsequently return to the yard, you'll find a ragged swath cut through the grass.

The frisbee is the original that inspired the Generic throwable and catchable object with variable flight time, which in turn is the parent of Mazer's cricket ball, the tennis ball, and many others.

Kilik's Patented Fireworks Show was used to great effect on July 4, 1994. You can add fireworks of your own by making a kid of the generic fireworks (#7618) and putting appropriate messages on it, then contributing it to the show. (A look at the code suggests that an actual fireworks show would bomb, because the old sky-watching FO no longer exists. So much for Kilik's and my plans for tame obsolescence – things go to pot on their own. An enterprising programmer might want to look into re-engineering it.)

To get to the yard, go west from the kitchen patio or south from the family room.

Driveway (#6193)

The driveway is rather more littered than its owner would prefer, but is less littered than it used to be, and therein lies its story, of history and technology.

Once upon a time, the driveway was owned by gru. Being a busy wizard at the time, gru had gotten out of the habit of visiting the driveway, and, in the interim, many others had gotten into the habit of leaving stuff there, and sometimes locking stuff there. For in those days, O Best Beloved, the self-cleaning room was not in the driveway's ancestral hierarchy.

One day gru and Yib (a young, upstart programmer) chanced to meet in the driveway, and noticed that ALL KINDS OF CRUFT had accumulated there, some of which gru had given permission to stay, but most of which he had not. gru and Yib set to moving all the unwanted stuff out into the ether, unceremoniously stepping on toes in the process. Once again the issue of object owners' rights (to keep their stuff where they wanted) vs. a room owner's right to determine what stayed and what went reared its ugly head, though it wasn't exactly an issue over which the entire MOO got up in arms. But those affected at the time cared very much, and became cross.

gru realized that he wanted the driveway to be kept cleared of junk, and that he didn't have time to do the job himself. So he offered ownership of the driveway to Yib, with the understanding that she would evict things and people from time to time, and thus began Yib's career as a young bugbear. (Today she is well known as an old bugbear.)

Yib immediately turned her attention to the Foodmart shopping cart.

"Wait," said gru, "that's genna's, and I promised her she could keep it here."

"Oh," said Yib. And it has been there ever since.

One day, Bartlebooth suggested to Yib that she create a magpie that would fly around the MOO snitching things.

"Brilliant!" said Yib, "Now I'll be able to keep the driveway clean without having to go there all the time." And so she did.

And that is the story of how Yib came to own the driveway, why it is littered with a Foodmart shopping cart despite Yib's obsessive neatness, and how the black magpie came to be.

The springboard is aimed at a second story window, and, if you start from the driveway, it doesn't do much. The key is to find Xythian's apartment in the second floor corridor, and jump *out.*

The MidNite-MOO is a tabloid newspaper. The copy in the driveway is a VR interface to the `*MidNite-MOO` mailing list. If you try to read it but all the articles have expired, you should make up an outrageously sleazy, slanderous story about someone you know (or someone you don't) and post it to *MM.

From time to time a sleek black limousine parks in the driveway. It will take you to an area of the MOO known as Singles, and is an example of themely transition to an area that is not part of the mansion and grounds proper. Frowned-upon in its early days by the Establishment, Singles is a link to some well-known and visited parts of the MOO, generally frequented by what the Old Guard think of as a Younger Crowd.

The sometimes-shiny penny is another VR interface to another mailing list, *penny.

The information center was added as an alternative for people who don't or can't find their way to the library and MOOseum, by someone who considers those locations to be too far away or too difficult to use, or both.

To get to the driveway, go north from the entrance hall.

The Library (#1670)

The library is a place that every player should know about. It is a place where any player may place anything e considers to be of general interest.

One hallmark of the library is that it lists its contents in two columns. At the beginning, items that players wished to park there were just listed individually, in any order, higgledy-piggledy. A few years ago, ownership of the library was

transferred to this author so that library materials might be presented in a somewhat more organized way. It was an interesting endeavor.

My goals were two-fold: to make shelves, so that materials could be sorted into categories, and to persuade all the owners of materials in the library to change their items' home from the library proper to the appropriate shelf. The diplomatic part took far more effort than the programmatic.

The library now contains several shelves. To access an item, first look at a shelf that interests you. Then take an item from that shelf. For example:

```
take Yib's Guide from Geography Shelf
```

Most items (though not all) have a "`read`" verb on them. You should `examine` an item to see what obvious verbs are available. Some are portable rooms which you can enter, for example.

You needn't worry about returning an item to a shelf – all are programmed, one way or another, to return to the appropriate shelf eventually. Some will stay with you for as long as you are connected, others will return after a preset amount of time.

The library continues its original policy of inclusion – anyone may place something there. New items should be placed on the New Submissions Shelf. The library's owner, any wizard, or any current member of the Architecture Review Board may then catalog an item to an appropriate shelf. For more detailed information on how that process works, copies of the library policy are on the New Submissions Shelf and the Reference Shelf.

Other items of interest:

- If you stare at the strange painting, you will be transported somewhere else.
- You can write a note for the bulletin board. Type `list bulletin board` to see the current notes that are on it.
- To access the ownership transfer station, type `enter ots`. Its purpose is to transfer ownership of an item from one player to another.
- The helpful person finder is a twin of the one in the living room.

There are a variety of other rooms off the library that are also of interest: North is the LambdaMOO academy, which has a robot that may teach you something about programming. Northwest is the Law Section, which contains information about ballots. Northeast is a map room – there you will find maps of various parts of the MOO. West is the more informal library alcove, and up from there is the library turret, one of several places where one may connect a room, if desired.

To get to the library, go south from the linen closet, then east, then north.

The Map Room (#3002)

The Map Room was added on to the library when a bunch of us decided that the old atlas that used to be on top of the mantel was just too out-of-date. At that time, too, the old atlas was retired to the geography shelf in the library, and a new atlas was put on the mantel in its stead.

The map room has a nifty rooms database object, which you can use to look up a room by its name. For example:

```
find mud wrestling pit in db
```

Feel free to borrow any map. The housekeeper will return it after you log off.

To get to the map room, go northeast from the library.

Library Alcove (#95512)

The library alcove was created when the library was reorganized, and is a less formal space. One might repair to the alcove, for example, if one wished to converse with a friend or acquaintance while according peace and quiet to those in the library proper.

It is also a place for a more relaxed sort of reading. The Daily Whale is a collection of cartoons written by Frand, from 13 March 1991 to 19 April 1994. There is a stack of Lambda MoosPaper back issues, and the generic hard-core porn rag, which will make your eyes bug out. "Walking Tours of LambdaMOO" is a pamphlet rack, with the interesting feature that you can take a pamphlet from it and carry it with you as you explore.

Sounding the enormous Chinese Gong will summon the Butler, whom you can ask for a cup of tea.

To get to the library alcove, go west from the library.

Top of the Library Turret and Black Hole (#69651)

From time to time people decide they need a break from the MOO, but nonetheless find themselves unable to stay away. Some ask a wizard to `@newt` them. Some use the Russian Roulette pistol. A few jump off the Edge of the World. While it is possible to write a verb to self-newt for any arbitrary duration of time (and there is a note on the reference shelf of the library explaining how to do this), it is not particularly convenient to do so, especially if one is not comfortable with programming. At the top of the library turret, then, is a black hole. The black hole is an interface to the self-newt process that is easy and convenient to use.

You can enter the black hole and leave it at will. You can enter the black hole for a specified duration (e.g. `enter black hole for 1 week`). If you then disconnect while still inside the black hole, you will be unable to log back on to the MOO until the week (or whatever duration you specified) has passed. The duration of one's self-newting is private, and players may also set their home to the black hole and come and go as they please, obscuring who has self-newted and who simply chooses to set eir home there. (Those who wish greater privacy in their self-newting should set their home to the black hole, also, before disconnecting.)

To get to the top of the library turret, go up from the library alcove until you get there.

LambdaMOO Museum (#50827)

Like the library, the LambdaMOO Museum is another of our venerated institutions. Inaugurated in July of 1992, for years it has been the resource of choice for researching existing generics, player classes, feature objects, and room types.

Several years ago, the museum's founder, thinking he was soon to lose net access, offered ownership of the museum to a trusted friend, who, alas, declined, citing other competing obligations. In the years since, the museum declined somewhat, suffering from a combination of fervent protectionism and benign neglect.

In the spring of 1999, a new curator volunteered eir services, and, after some negotiation was approved by the original curator. There are some new exhibits, enhanced searching capabilities, and players may now catalog their own items for display in the appropriate section. An equestrian statue of the original curator is said to be in the works.

To get to the LambdaMOO Museum, go south from the library, east three times, then north.

Negative Museum (#68493)

Tucked into a nook, far from the conventional museum, is a little gem of an exhibit dedicated to the negative objects on the MOO. It's a long walk, but well worth the trip. Starting from the entrance hall, proceed:

east to Corridor
south to Ground Floor Stairwell
up to First Floor Stairwell
west to First Floor Corridor
south to First Floor Corridor
east to First Floor Corridor
north to First Floor Corridor
north to First Floor Corridor (Boardroom)
east to East Corridor
east to East Corridor
east to Corner
north to North Corridor
north to North Corridor
north to North Corridor
east to the Negative Museum

Underground Arcade (#16471)

This room is the hub of an underground complex of shops. Here you will find Sammy the clown, who will give you a balloon if you ask, and a directory of the various shops. The directory itself is something of a historical document – some of the shops listed have long since closed, their entries in the directory the only record of their bygone existence.

The arcade itself has three wings, east, north, and west.

To get to the underground arcade, go east from the entrance hall, south into the stairwell, down two levels, then north.

Xorbon's Floral Shoppe (#33958)

Xorbon's is *the* place to get flowers for your sweetie, or someone you just happen to like. The shop itself is a delightful venue – ordering flowers is easy, and delivery is prompt and polite. Be sure to get there early on Valentines Day, though, as they've been known to sell out.

If you are lucky enough to receive a bouquet, be sure to water your flowers from time to time to keep them fresh and healthy.

To get to Xorbon's, go east from the underground arcade lobby, then north.

The Pizza Parlor (#15229)

Mama Bungweisi's is a great place to go and party with friends. You can order pizza, share it with your friends, get a soda, and, if you're clever, get a quarter from the change machine and play your favorite tunes on the antiquated jukebox that is there.

The parlor is animated with a variety of themely messages that give it a certain ambiance all its own. Plus, they deliver!

To get to the pizza parlor, go east from the underground arcade lobby, then northeast.

Tasks 'n Frobbin's (#17499)

After your visit to Mama Bungweisi's, be sure to stop in next door at Tasks 'n Frobbin's for an ice cream cone. Order your cone at the counter, then top it with any of the twenty-five million eight hundred seventy-four thousand one hundred thirty-three flavors of ice cream that they have there. Read the menu to see the price list.

(For those who may not know, the name of this establishment is a pun on an American ice cream store franchise named “Baskin Robbins”.)

To get to Tasks ’n Frobbin’s, go southwest from the pizza parlor, then southeast.

Mud Wrestling Pit (#36017)

The mud wrestling pit is best enjoyed with a party of two or more people, and is a great place to let loose and let fly. You can throw mud, muck, goo, or just about anything else at someone – the clever part is that whatever you throw is then integrated into your target’s description. You can also `wrestle <someone>`. You can take a refreshing shower as often as you like, too (which shower attendant you get depends on your gender).

The mud wrestling pit was originally authored by gru. Later, gru went into a period of seclusion, recycling much of what he owned, including the mud wrestling pit and Buddy (a robot, beloved by some). gru later graciously made the code available again, at this author’s request, and I was able to reconstruct it.

To get to the mud wrestling pit, go north from the lobby of the underground arcade, then northeast.

The Garage (#39)

The main thing about a garage is, you go there and rummage around for things, and so it is with ours.

A room owner can designate certain objects or players as *residents* of a room, and can write programming on a room that treats residents and non-residents differently.

In this case, if an object has been designated as a resident, it will be visible, whereas if you drop an object that is not a resident, it will not be visible. Furthermore (and this is the good part) you can search for items that are residents.

You might find (or search for) a shipping crate. The shipping crate contains a bunch of motorcycle parts, and this is a puzzle by yduJ. I found it difficult, but then, I know very little about motorcycles. (Hint: Several parts need to be connected to more than one thing.) Another useful thing you might search for in the garage is a tool box and/or certain tools.

To get to the garage, go north from the housekeeper’s quarters.

A Decent Piece of Turf (#24641)

A Decent Piece of Turf is the hub of and gateway to the older part of the mansion grounds. It was created and landscaped by The Great Bartlebooth. The hallmark of Bartlebooth’s work is that each object is characterized by a deliberate, defining, idiosyncratic flaw. Nothing of his suffers from bland perfection. In the case

of a decent piece of turf, "`...your eyes feast on the thick grass, worthy of the gardens at Kew, except for an ugly weed stump, which is an unsightly reminder of everything the turf denies.`" The stump itself is a wonderful exercise for would-be gardeners. You can dig the stump, but that leaves a divot, which must then be filled, which leaves a bald patch, which must then be watered. To your horror, a weed sprouts!

The topiary bush is another excellent example of Bartlebooth's work. You can clip the top, middle, and bottom to resemble a human, bird, or animal, and each combination yields a new creature, each more fantastic than the last. (The centaur is my personal favorite.)

The reflecting pool serves as a sundial; the number of fat happy goldfish is the number of connected players.

The rose tree is an example of Carrot's work at its least baroque, and that is still pretty baroque. You can pick a rose from the tree and take it with you. The colors are most unusual. Each rose lasts 48 hours.

The flag is a wind sock (in disguise), a requirement for helicopters to land.

To get to a decent piece of turf, go south from the yard.

Gazebo (#52061)

The gazebo is not, in and of itself, a heavily programmed or interactive place, yet over the years players have gone there to interact. In its heyday, it was a favored trysting place. (Its owner used to find abandoned articles of clothing there on a fairly regular basis.) Delightfully described, it was one of the first publicly connected rooms with integrated seats, and it has a charming array of them.

The small table there has a conch shell on it. It is sometimes said that listening to one will transport you to faraway places.

To get to the gazebo, go to a decent piece of turf, then type `enter gazebo`.

Gazebo Roof (#70379)

From the top of the gazebo roof, you can see in each of the four cardinal compass-point directions. The weather vane is a VR way of seeing what the current lag is.

The gazebo roof is also where the black magpie makes its nest, hoarding all the various things it has scavenged from around the MOO. There's all sorts of random stuff in there.

To get to the gazebo roof, go to a decent piece of turf, then type `climb trellis`.

Makeshift Cafe (#39999)

In the early middle days of LambdaMOO, when the living room, according to some, had started to change its makeup, the so-called Power Elite chose to gather at the Makeshift Cafe on the grounds behind the house. There was a partial overlap between this crowd and the gang at JaysHouseMOO, which is how the JAYSHOUSE picturephone came to be there. It is said that it can sometimes allow communication between players on the two MOOs, but that it depends on a switch at JaysHouseMOO being in the "on" position. This author has never seen the picturephone actually work.

The RL is owned by Doug. Doug wrote to me, and told me that it's called "a RL" rather than "an RL" because he has always asserted that the proper pronunciation of "RL" is "real life". Legend has it that on occasion it would somehow boot wizards. It has an interesting message if you try to take it.

The human cannon is a delightful piece of work, and behaves slightly differently depending on whether zero, one, or more people are inside when it's fired.

The cheap number puzzle works just about exactly as you might expect.

The Lambda MOOsPaper is a vestige of a bygone era. At one time it was owned by waffle, and was taken seriously as a local compendium of online and offline events. There have been occasional efforts to revive it, but none has succeeded to date.

To get to the makeshift cafe, go west from a decent piece of turf.

The Underground Waterfall (#15413)

The Underground Waterfall is one of the loveliest places on the MOO that I know of, and a wonderful example of how much can be done with a room without doing any actual programming. Though it has been criticized for its lack of verbs, it is one of the very best pieces of Tiny Scenery extant.

The room is rich with details: You can look at any of the trees, the undergrowth, the clearing, the pool, the edge (of the pool), the water, the moss-covered rocks, the waterfall, the bank, the pond, the stream, the foliage, the small wooden sign, the plants, down (at the lush grass), at the break in the foliage to the south. (Note, if you have Carrot's Viewing Feature (#46278), you can type `@dview` and words in the description for which there is a corresponding detail will appear in upper case.)

The available seats are the pool, the mossy rocks, edge of the waterfall, the bank of the pool, the stepping stones, and the grass.

The surrounding area has long been known as "The Land Down Under". There is a map of the land down under in the atlas on the mantel in the living room.

To get to the underground waterfall, go to a decent piece of turf, enter the gazebo, then go down through the trap door.

West of the Yard (#3942)

West of the Yard used to be called "West of Gardens", but that was back when the gardens existed only as a mere mention in the description of the living room. This location's owner, Jon_BonJarleycorn, is something of a historical figure at LambdaMOO, but was nontheless reaped in his time and his posessions recycled. A hue and cry ensued, and the wizards restored not only the player character, but nearly all his original objects (many of them rooms that were part of the grounds) with their original object numbers. In the course of this fiasco and its subsequent resolution, the chapel, which used to be found in West of Gardens (now West of the Yard), got lost in the shuffle. It has since been found and restored to its former location.

The chapel was originally programmed by gru, who later bequeathed it to Jool, who loaned it to Yib, who spruced it up a bit and then gave it back to Jool. It was once a popular venue for MOOers who wanted to get married in VR. Inside the chapel, you can sit, stand, kneel, prostrate, levitate, pray, meditate, shout, whine, flame, chant, cower, pout, contemplate, sigh, confess, float, plead, relax, officiate, sermonize, listen, recline, rant, wait, idle, sleep, lag, respond, nap, or genuflect (as your inclinations dictate). These are not mere emotes: The chapel's description is cleverly programmed to show its occupants sitting and praying, standing and shouting, chanting, floating, etc. (as their inclinations dictate). The chaplain is a delightful instance of an old bit of MOO technology called a monitor. Akin to the globe in the entrance hall (where anyone may add or delete a location), anyone may add or erase a message from the chaplain's repertoire. The nice thing is that this chaplain is quiet while others are speaking, but offers words of comfort and wisdom during moments that would otherwise be silent.

To get to west of the yard, go west from the yard, or north from the makeshift cafe.

Base of Large Oak Tree (#2834)

Lambda house is graced with a magnificent old oak tree, which you can climb. Climbing trees is inherently fun, and there is a good bit of casual exploring to do near the base of the tree and up among its branches. There is a tree house, whose rope ladder you can raise and lower. Further up, at the top of the tree, there is an acorn. You can enter the acorn, and there is a puzzle. Figuring out the nature of the puzzle is part of the puzzle (hint, examine *everything* to get started). I haven't solved this puzzle yet, but it has gotten good reviews from others.

If you get a rash of poison ivy, calamine lotion is said to help. (Try the first aid kit in the half bath near the master bathroom.)

To get to the base of the large oak tree, go south from the patio, or east from the decent piece of turf.

Tree Root System and Stone God Puzzle (#18691)

The Stone God Puzzle is a fine one. Here are a few hints, just to help you get started:

- Use a pencil and paper.
- Get some coins or buttons or jelly beans to use as tokens, to place and move on the paper.
- Most sandstone walls can be pushed. The syntax is `push <direction>`, e.g. `push east`.
- No granite walls can be pushed, but you can try.
- You can tap on a wall (e.g. `tap on north`) to see if a sandstone wall can be pushed.

To get to the tree root system, go southeast from the base of the large oak tree (yes, you have to endure the brambles), then enter tree, then down, then north.

Landing Site (#5468)

The landing site was LambdaMOO's early concession to those who insisted on a space theme, and over the years it has served a variety of UFOs and other vessels. Today it is home to helicopter N001LM, which was my first programming project there.

From outside, you can preflight it (always a prudent idea), and enter it. From inside, there are a variety of options:

```
start
hover
fly
faster
slower
higher
lower
look placard
look out
look down
look hobbs
look chart
scrutinize chart
land
land <location>
overfly <location>
wave
stop
shutdown
exit/jump/disem*bark
```

The helicopter's description and behavior change depending on whether it's running or stopped, parked, hovering, or in flight.

If you land on the asphalt roof, they'll hear the rumble of the engine in the living room. Some other locations also broadcast helicopter arrivals and departures to neighboring rooms.

There are appropriate messages for spectators outside the helicopter when it takes off, flies overhead, and lands. These messages also differ depending on how

high the helicopter is flying. If you are at the landing site when someone crashes it, you get to see a team of engineers truck it in and put it back together.

It knows the difference between a low-altitude hover (in the location you took off from) and a high-altitude-hover (in the sky, over a location).

If you fly a MOO helicopter, you get a souvenir pair of pilot's wings. If you jump out of the helicopter while it's airborne, the pin turns into a little anvil with wings. If you crash, it also changes, depending on where you end up. The pins last 24 hours. (What do you expect from a cheap souvenir?)

As you overfly locations, people on the ground are notified. (In the early testing stages, not knowing any better, I flew back and forth, constantly, over the Makeshift Cafe, where WhiteRabbit was hanging out, and really got on his Bad List. It took me years to redeem myself.) Aircraft cast shadows, which are actual objects that are moved to the various locations the aircraft overfly. This enables people on the ground to wave to people in the aircraft (type `wave at helicopter` for example).

Helicopters can only be landed at catalogued outdoor rooms that have wind socks. This is to prevent people from landing helicopters in the living room and other indoor locations.

To get to the landing site, go west from the yard through West of the Yard to the meadow, then north.

Hell (#19232)

Hell, nowadays, just isn't the place it used to be, and it is included here mostly for historical interest. Owned by Wintermute, it has been neglected for years.

Once upon a time, one was pelted with a torrent of particularly offensive spam immediately upon arrival. Now, it is a much quieter place. Once upon a time, `trapdoor` was the verb of choice to be used on the hapless guest: A trap door opened up underneath em, and Boom! e was sent straight to hell. Now, guests are sent to the kitchen to fetch cookies or tea.

When I was building the MOO helicopter, "Ooh! Can I crash it?" was perhaps the most frequently asked question. I didn't want to do it. (In point of fact, helicopters are quite capable of gliding to the ground and landing safely in the event of an engine failure, and having worked so hard to make it realistic in other ways, I was not pleased at this request. Furthermore, I was stymied as to how I was going to depict a helicopter crash in a meaningful way.) Finally I caved in, though, and added the code. And I sent all the helicopter's occupants straight to hell, without giving them the satisfaction of actually seeing the helicopter's mangled remains, and got the last laugh.

The road to hell is paved with good intentions. To get there on LambdaMOO, however, you must either crash a helicopter or take the elevator.

Heaven (#59714)

Heaven exists because Hell exists. After sending countless players to hell, I got bored, and wanted to add a bit of variety to the “crash experience”. So I created heaven as an alternative, and randomized crash victims’ fates. There’s nothing inherently interactive about the place – just a variety of messages that represent all the best things I can imagine.

Then it seemed only right to connect it, so that people wouldn’t have to break the VR and teleport out, so I did that, too.

Bartlebooth said, “Heaven should be hard to get into: Make it a puzzle!” So I did. In order to walk in through the Pearly Gates, you have to have the key, which is “hidden in plain sight” somewhere in the main house.

It’s hard to get into Heaven.

Open Field (#58923)

The open field is a very early example of Klaatu’s difficult-to-use but nonetheless brilliantly-conceived seasonal animated room. The room’s description and animation messages change from day to night and in accordance with the seasons. It’s a wonderful place for bird watching and star gazing, with different birds during the day and different constellations during the night each month of the year.

It is also the mooring place for three majestic hot air balloons, scarlet red, royal blue, and emerald green.

To get to the open field, go south from the decent piece of turf.

Formal Gardens (#59102)

Shortly after my first substantial quota grant, that rascal Carrot paged me that I should make a giant piece of Tiny Scenery, “… because you can.” This was really a very startling suggestion, because Carrot was an ARB member at the time (and this was back when ARB members were appointed rather than elected), and Tiny Scenery was highly frowned upon. But I was new, and eager to please, and so I set out to make the *very best* piece of Tiny Scenery I possibly could.

Very little of what now lies behind the house existed at that point; in fact, yduJ had posted to a public list that there was nothing south of the pool yet and that new building there would be welcome. Since the description of the living room mentioned a view of “the gardens” through the windows to the south, and since I had gone looking for them in vain, I decided that that would be my project. In addition, my programming buddy Klaatu had recently finished his generic seasonal animated room. I was the only one who really knew about it, and I resolved to showcase it. The formal gardens have a base description, plus a coda that changes each month and between day and night, plus animation messages that change each

month and between day and night. (The prose is about as flowery as you can get. I'm somewhat self-conscious about it today, but that's the way it goes.)

Meanwhile, Greene, one of the more flamboyant MOOers of the day, had made a jungle, connected a room named "Brazil" to that, and asked that it be connected south of the pool deck. Without a finished project to present, I was pre-empted, to my great consternation.

I continued working on it anyway, elevating it from "mere" Tiny Scenery to making it so that you could pick flowers there and actually have an object to take away – creating objects on the fly was my great craze at the time. It's a little-known fact, by the way, that you don't have to take pot luck with `pick flower`. If you want a single white rose, you can type `select a single white rose` and get one.

Then came the problem of where to connect it, since south of the pool was TAKEN (!#$%&*). Bartlebooth and I discussed it, and decided to connect it to the open field with a bit of woods in between as a buffer, and so the formal gardens were a distant treasure (relatively speaking) for quite a while.

Eventually, Greene stopped connecting. I waited until the *day* she was eligible for reaping, and then asked to connect the formal gardens south of the pool when Greene's jungle was eventually recycled. Thus the formal gardens came to be in their current location (with the arbor in between as a sort of transition area). Then I had to reconcile the geography at the formal gardens' original location, which is how the ruined garden came to be in that spot.

Much later, I was telling this story to a friend, who asked me, "Do you know what's actually behind the house IRL?"

"What?" I asked.

"A jungle," he said.

Go figure.

To get to the formal gardens, go south from the pool deck to the arbor, then south again.

Caretaker's Cottage (#72097)

The caretaker's cottage, nestled snugly in the formal gardens, is a splendid example of Bartlebooth's genius for interior design. The fireplace and the antler chair are especially well described.

As work on this room evolved, it came to be something of a *nexus room*, meaning that it is a room from which you can connect to all sorts of other places. Historically, the ARB has frowned on nexus rooms per se, calling them unthemely. In this room's defense, however, I would point out that the objective was less to provide a jumping off point to so many places than to showcase various *ways* of getting around.

The fireplace is a child of the fireplace that's in the living room. To use it, type:

```
enter fireplace
```

then:

```
pull chain
```

There will be some grinding of gears, and then the fireplace will turn to face a new room. Any room containing a child of the living room fireplace (#43212) is automatically added to the rooms you can explore this way. You can open and close the fireplace in any room. If the fireplace is closed, you can't enter it, and people can't come out. (Words to the wise, if you happen to have one in a room that's intended to be private.)

`On the north wall hangs a painting.` This is not just any painting, but a kid of the generic "Trump" portal. (This is an allusion to the "Amber" series by Roger Zelazny.) If you stare at it, you will be transported to the location of another portal to which this one is linked:

`On the south wall is a brass panel, with a row of buttons on it.` To see the places to which the panel can transport you, type:

```
press list on panel
```

On the antler table are a snow globe and a conch shell. The shell will take you to the beach if you first take it from the table and then listen to it. The snow globe will take you to a variety of places if you take it from the table and then shake it.

To get to the caretaker's cottage, go southwest from the formal gardens.

Petting Zoo (#74291)

If you look east from the formal gardens, you will catch sight of an elephant! The Petting Zoo was originally authored by my friend Lamont_Cranston (gone but not forgotten). There you will find a variety of animals, some of which can be petted and some of which can't. You'll also find a machine with birdseed in it, and you can feed that to the parrot, or try to feed it to the other animals. To the southeast is a storage shed with other kinds of food for the other animals.

The timing of several of the messages seems to be a bit off – the parrot greets you upon arrival, but there's a delay, and likewise after you depart – but the charm of the rest of the programming makes up for this.

Each animal has its favorite food, but most of the animals also have fun messages if you try to feed them the wrong kind of food. You can also try to feed players to the different animals, with varying results.

The parrot is a parody of the living room's cockatoo. I dimly recall that there used to be an ongoing repartee between the parrot and the lion, but maybe the room's parent has been changed by the new owner, or something, because unfortunately those messages that I remember seem to be gone. Going to the zoo is a fun outing, even so.

To get to the petting zoo, go east from the formal gardens.

Forest (near the Open Field) (#52061)

When I wasn't able to get my first choice connection for the formal gardens, south of the pool, I looked around for a place to put them, and decided to put them further out on the vast grounds of the mansion. Putting the gardens immediately adjacent to the open field just didn't feel quite right, so I added a patch of woods between the two, with an idyllic description and a few animated messages for atmosphere. Later, Bartlebooth added an oak tree, on which people can (and do) carve their initials and short messages, "Y LOVES B" being a typical example.

The several parts that make up this particular forest work as expansion pieces, so that different locations don't have to exist cheek by jowl, but rather can have the feeling of a bit more space between them. The general layout is that each segment of the woods has a path that leads either back towards the house or further into the woods, plus one other connection, so that it's difficult to get lost. The first segment of woods (near the open field), being the head of the trail, is an exception, having the open field to its west, a dense thicket to the east, and base of large oak tree to the northeast. The path through the woods continues to the southeast.

There's a rabbit that hops through the woods and environs, as well. A certain dog used to love to chase rabbits, and this one was made for him to chase on the MOO. Anyone can chase the rabbit. It's shy, so trying to pet it or pick it up may or may not result in success. (It's not a puzzle, by the way. Just a shy rabbit.)

To get to the forest near the open field, go east from the open field, or southwest from the base of large oak tree.

Ruined Garden (#66623)

When the person who owned the jungle south of the pool was finally reaped and I was permitted to place the formal gardens there (at long last!) the question arose as to what to do about the gardens' previous site. Bartlebooth and I brainstormed a bit and decided that what should be in that place was a formal garden which had been allowed to go to ruin. There are many parallels to the original formal gardens – the description and animation messages change with the seasons and between day and night. The mood is rather sinister by day, but pleasant and cheerful at night. As in the formal gardens, you can pick flowers, but these are wild flowers with names like "nettleleaf horsemint", "creeping charlie", and "orange sneezeweed".

There is an urn in the ruined garden, repository for a couple of mementos – a handkerchief embroidered with the initial of the beholder, and a pistol, with which players have been known to play Russian Roulette. The loser is `@newted` for an interval of between two and six days.

Around the time that the ruined garden was being, well, ruined, there was also agitation afoot for the wizards to document in some way players who were reaped. What had been happening was that the only notice one got that someone was no longer with us was a message from the login watcher that some object number was no longer a valid player, but no good way to tell who it had been. Thus began what I

came to think of as the Great Death Project. People had requested a graveyard with a headstone giving information about every reaped player. This turns out not to be practical, because there are many, many people who request a character, log on just once or only a few times, and then never return. They are reaped to save database space, and erecting stones in their memory would defeat the purpose. It was finally agreed that gravestones would be erected for some players, and the ruined garden is where those stones repose. (The Tomb of the Unknown MOOers is for those who do not get a stone of their own.) The gravestones come in a variety of styles, and age (very slowly) over the course of time. Each has the name, object number, and first and last connection dates of a player, plus an epitaph. One can also place objects on the stones – a bouquet of wild flowers, for example.

If you are in the garden at just the right moment, you can actually witness the undertaker and his assistant carrying out their dolorous task.

To get to the ruined garden, go southeast from the forest near the open field, then south.

Undertaker's Cottage (#101792)

The undertaker's cottage and its contents make up the second part of The Great Death Project. The problem at hand was to find a way to record the departures of reaped and MOOicided players without clogging up the database unduly. This was accomplished in a couple of ways. First, the mailing list `*obituaries` was created, where reaped players' names, aliases, and numbers are posted. Unlike most mailing lists which expire after 30 days, this one expires after three days, which approximates how long obituaries run in many newspapers. Second, there is a family bible, in which "deaths" are recorded. The family bible is on the mantel of the fireplace in the undertaker's cottage (type `take bible from mantel`). One can look up recently departed players by name (though not by alias) or by object number. This information is kept for 40 days and 40 nights, so that if someone doesn't log on soon enough to catch a posting to `*obits`, one can look someone up in the family bible and find out who the "missing" object number used to be. Last, there is an epitaph registry, in which one may record an epitaph for someone. (Note, this must be done before the subject's demise.) The presence of an epitaph in the registry is what causes a player to get a headstone in the ruined garden if that player is ever reaped, the idea being that an epitaph signals that someone cared enough about a player to want to commemorate eir departure from the MOO. You can browse entries in the epitaph registry as well as write them.

One can give one's name to the undertaker. This is a relic of the fact that one used to have to rename oneself to something like "Toad13" before walking off the edge of the world to commit MOOicide. Giving one's name to the undertaker before renaming oneself caused one's actual name and aliases to appear in the appropriate post on `*obituaries`. Giving one's name to the undertaker is no longer meaningful, but might be, again, when *Ballot:Bring_Back_the_Blender is implemented.

The undertaker's cottage doubles as a funeral parlor (with a mostly-working pump organ), for those who wish to gather and mourn the dear departed.

Geographically, the undertaker's cottage corresponds exactly to the caretaker's cottage in the formal gardens, and its floor plan is the same. To get to the undertaker's cottage, go southwest from the ruined garden.

Crypt (#5502)

It seems reasonable to think that a mansion as large and as old as this one would have a family crypt where the bodies of the recently departed would be laid out. When the Russian Roulette pistol "kills" a person, eir "corpse" is moved to the crypt and laid out on the granite slab.

To get to the crypt, go down from the undertaker's cottage. Linger a while to let your eyes get used to the light, and to get a feel for the place.

Catacombs (#10328)

The catacombs are a classic "maze of twisty little passages" puzzle. Each of the nineteen chambers holds a different treasure, and this is part of how one can tell one chamber from another (as well as the listed available exits being different). The object of the puzzle is to collect all the treasures and deposit them in the sarcophagus that's in one of the chambers. The prize for solving the puzzle is that you get an amulet with your name inscribed on it (presented in a ceremony that includes a skeleton ballet). The amulet is a rotating trophy, and is yours to wear until someone else solves the puzzle.

Anyone may adopt a catacomb chamber as eir home.

To get to the first catacomb chamber, go south from the crypt.

The Green Cathedral (#83618)

The Green Cathedral's purpose is just to be a quiet, restful spot in the woods. The description changes with the time of day. In the interest of solitude, it will accommodate no more than two connected players at any given time.

To get to the green cathedral, start in the open field. Go east into the forest (near the open field), southeast (near the ruined garden), southwest (near the green cathedral), then north.

Japanese Garden (#38351)

The Japanese Garden is a tranquil spot deep in the forest. You can feed the fish, whose level of hunger varies depending on how recently they've been fed. You can also sit and meditate, which will enable you to learn about your Karma. (This is the original code on which the popular `Karma FO (#1283)` is based.)

To get to the Japanese Garden, go southwest from the forest near the green cathedral, then southeast.

A Dense Thicket (#83100)

It's difficult to balance trying to keep an area within the stated LambdaMOO theme, and wanting to say yes to people who have put creative effort into building and describing an area that needs to be outdoors but which might not actually be found on the grounds of a large mansion. The dense thicket is one themely transition to such areas, the idea being that if you wander into a dense thicket on the grounds of a mansion, you might get lost and then not actually be *on* the grounds of the mansion anymore. The dense thicket has a variable exit. At any given time, the description will mention a place that lies in some specific direction. If you type `wander` while in the dense thicket, you will seem to wander about for a bit, and then catch a glimpse of a different destination. These destinations include parts known and unknown, and are left for the intrepid explorer to discover on eir own.

To get to the dense thicket go east from the forest near the open field.

Fishing Pond (#66099)

The fishing pond is a fine place to spend a leisurely afternoon.

I inherited the fishing pond from a friend who knew he was going to lose MOO access, and it is to my great chagrin that I never finished it as I promised I would, and to my greater chagrin that I can no longer remember the original author's name. When I finally decided to face facts and admit I just wasn't going to get around to doing right by this project, I looked for a volunteer who would, and that volunteer was Road_Dog (#96779).

Now, my dad used to take me fishing in my youth, and I know that if you bait a hook with a worm, you hang it over the side of the boat (or a bridge) and wait for a fish to bite; and if you're going to cast for fish, then you use a lure (or maybe a fly). I have a standing disagreement with Road_Dog about his implementation of having to bait one's hook before casting. Be that as it may, Roadie has put a lot of effort into making the fishing pond a fun place, with many different ways to interact. Roadie's emphasis was more on making things intuitively usable than on strict object-oriented programming. The down side is that using the `examine` verb with various objects doesn't always tell you the whole story of what you can do, but the up side is that if you just try things that seem natural, they often work.

Bert Burnsmythe is our resident fishing guide, and when he arrives on the scene you can ask him for a pole (`ask Bert for pole`) to get started on your great fishing adventure. Good luck. Ladies: Be sure to kiss the frog; you never know when you might get lucky.

To get to the fishing pond, go southeast from the open field.

Guest Room (#864)

The main feature of the upstairs guest room is that it is home to the enormous model railroad layout. This marvel of miniaturization, created by waffle, is the epitome of themely transition, in this author's humble opinion. Lots of things can go there that might not otherwise make sense as part of a mansion and its grounds, and lots of things have. Places geared to a more urban theme, for example, often connect in Tiny Town and its environs. In the beginning, this was blessed by the ARB as an officially sanctioned themely transition. Later boards would dismiss Tiny Town connections as "boring" and exhort builders to find a connection that was "more imaginative". I don't know of any recent builders who have built on the train layout and applied for quota, so it's hard to know the current ARB members' take on this.

If you look at the layout, you can see where the train is, and also tiny figures moving about, if there are any connected players there at the time.

If you type:

```
enter layout
```

you will be transported to a randomly-selected location on it. Many of the locations there have train service, though some do not. From those locations you can see (far above and greatly magnified) the guest room and its contents. To exit the layout, type:

```
jump
```

You will be transported back to the guest room.

Other rooms may branch off from the rooms accessible in this way, from which you may or may not be able to see the guest room and jump back to it. The idea is that at the first point of transition, the fact that you have just been miniaturized is quite obvious. As you explore the environs, you get further and further "into" a particular scene and the fact that you're tiny is less obvious, perhaps even forgotten.

Also in the guest room (usually) is a wind-up duck. This is a full implementation of the project described in yduJ's programming tutorial (found in the library). You have to take the duck to wind it up, then if you drop it, it does its thing. In particular, its description changes depending on its state, and it stops quacking if you pick it up, so it is worth playing with and looking at more than once.

To get to the guest room, go south from the library into the corridor, then south again.

Main Street Station (#31821)

The TinyScenery Express runs in a loop. A logical embarkation point is Main Street Station, in Tiny Town.

You can read the schedule to see the list of stops.

The loudspeaker keeps you informed of the train's progress. If you don't like the noise, you can switch it off.

When the train pulls into the station, you can board it by typing `enter train`. To get off the train at any point, type `exit`. While on board, you can type `look out` to see the train's surroundings.

Tiny Town, to the south of Main Street Station, is the most elaborately developed part of the train layout as of this writing. Some of the other stops are mere Tiny Scenery at present, but some have been or are being developed into more, and are fun to explore. West of House replicates the entry point to the Zork game. (The game has not been ported to LambdaMOO in its entirety, however.) South of Reservoir has been developed: To the north is Swine Lake, to the south, the Happy Trails Camping Area, where portable homes are welcome to park. Grand Central Station provides access to several different areas.

The owner of the enormous model railroad layout specifically welcomes connection requests.

There is no direct way to get to Main Street Station from the guest room, as entering the enormous model railroad layout puts you down on a random spot. You can, however, go to the library, take Modern Model Railroading from the Geography shelf, and read it. You will be transported to Main Street Station directly.

The Corner of Main Street and Queens Boulevard (#31889)

The Corner of Main Street and Queens Boulevard is not an interactive area in and of itself, but as the virtual hub of Tiny Town, it is an important bit of infrastructure.

To get to the corner of Main and Queens, go to Main Street Station, then south to Main Street, then south once more.

Club Dred (#50590)

Club Dred, party venue extraordinaire, is one of the most richly programmed areas I know about. It comprises several rooms, including the main part of the club itself, the back hallway, ladies' and men's rooms, the balcony, the kitchen, a video room, the lower hallway and a band hall with a stage. It's an excellent place to gather with friends for an evening of revelry.

In each room, you can type `about here` for details and instructions for doing things. Some highlights:

In the main part of the Club, if you want to order a drink, you should type `sit at table`. Then, when Abigail the waitress comes over, you can type `order <drink> from Abigail` or `order <drink> from Abigail for <player>`. Presently she will deliver a drink object, which you can drink, sip, taste, etc. Abigail is quite a character; if you are feeling bold, you might flirt with her. If you are really feeling rowdy, you might even goose her! The bar is usually too crowded to find a seat, but you can stand and `order <drink> from bar` if you are not in the mood to sit down. There are some regular characters who only show up if six or more people are present.

From the main part of the club, go west to the back hallway. Here you will find three collages of photographs. You can type `look first`, `look second`, or `look third`, OR you can type `look for <player>` if you have someone particular in mind. The pay phone is here, and it's fun and easy to use. `About here` will give you the basic instructions you need to use it. If for some reason the person you're calling has difficulty anyway, you may need to prompt em via page to type `examine phone`, and things should go fairly smoothly from there.

From the back hallway, exits north and south lead to the men's and ladies' rooms respectively. The basic commands in those rooms are what you would expect, plus you can `look wall`, `read <message-number> on wall` and `write <message> on wall` to participate in Club Dred's long tradition of graffiti.

The exit `up` from the back hallway leads to the balcony. As with all the rooms in Club Dred, typing `about here` yields helpful information, in this case about how to look at the play list and request and dedicate a song. You can `jump` from the balcony only if heavy metal is playing.

From the main part of the club again, you can sometimes sneak back into the kitchen (it depends on whether or not the bartender is paying close attention). There you'll find the Club Dred freezer, which is a cross between a container and a room. You can put things or people into it. People turn progressively bluer the longer they stay in there.

Again from the main part of the club, you can go down to the video room. The main feature of this room is that the tables are for two, and it's possible to have more intimate conversations as no one except the person sitting at the table with you can overhear the conversation. West from the video room is the lower hallway. You can buy a ticket from Ian if you wish to proceed (west) into the band hall. The way south leads to an RPG area; non-rpg players are denied entry. Ladies: Do not miss a chance to flirt with Ian. (He's shy, so you may have to persist, but his attentions are worth it!)

To get to Club Dred, go northwest from the corner of Main Street and Queens Boulevard.

The Morgue (#70385)

The Morgue is the single creepiest place I have ever been to on LambdaMOO, and I even remember the consternation of the ARB members at the time of its

creator's quota application, because the goings on there are so unspeakable. Yet they met the criteria and the request was granted.

The commands you'll need to do your grisly work there are:

```
get body from body bag

get instrument from table
```

or:

```
get tool from cabinet

sit on <body>
cut <body> with <tool-or-instrument>
stand from <body>
toss <body> in bag
```

As always, `exam <object>` will give you more information about its, er, uses. The body will be that of a real MOOer who has been absent for some while. The tools are many and varied. The sound effects will be with me for a long time.

To get to the morgue, go southeast from the corner of Main Street and Queens Boulevard.

Sensual Respites (#72239)

I am not a habituée of Sensual Respites, and I fear that I will not be able to do justice to the fine programming that I know has been done there and in the local environs, but it would be a crime to omit it, and so I am forging ahead.

Sexually explicit material is appropriate and encouraged in Sensual Respites, and it is probably second only to the living room as a popular hangout. Neighboring rooms offer a wide variety of variations on sexual themes. There is no directory, however – one is expected to explore on one's own.

Highlights include that the genders of the room's occupants are displayed when you enter, idlers are moved to adjacent rooms depending on how busy Sensual Respites itself becomes, and there is a verb with which enables players to `@bounce` other players who are being obnoxious.

Sensual Respites began as an adjunct to The Sex Room (#53011). Over the years of its existence it has expanded into a suite of many rooms, and its programming has become ever richer. The Sensual Suites Hotel was relocated from the Singles complex to Tiny Town around the start of the new millennium (in honor of all the tiny sex that takes place there).

To get to Sensual Respites, go west from the corner of Main Street and Queens Boulevard to West Queens Boulevard, thence north into the Sensual Suites Hotel. From there, go northwest into the office, down to Members Only, then southwest.

Yib's Palace! (#93665)

Yib's Palace! is a gambling casino in Tiny Town.

Even after I became a wizard, I avoided using wizperms in most of my building – partly to minimize unintended security holes, but mostly because I enjoyed the challenge of making toys that were fun that "anyone" could make – I liked showing how much could be done *without* needing a wizbit. Then one day the question occurred to me, "What are you saving it for?" and Yib's Palace! is the result of that. I had bandied about the notion of one's occasionally being able to find a few bytes of quota when searching the living room couch for lost items, but the bookkeeping necessary to keep people from trying to cheat seemed to be more trouble than a few bytes of quota were worth. Eventually someone in my circle of brainstorming pals suggested a slot machine that would take in and give out quota, and I went with that. I bought a few books about how slot machines work, and spent several hours with a spreadsheet program trying to get the reels just right so that the payouts would work out. The consensus among people who gave early feedback was that the house should take a cut, as is the case with real slot machines, so that's factored in. Once that was done, I made several particular slot machines and a place to put them. In the spirit of "something for everyone", there is a slot machine for every budget, from 5 bytes up to 100 bytes at a time. (There used to be two other, more expensive machines, but someone tried to cheat, and another wizard looked at the possible payout (low odds, but possible) and decided e was uncomfortable.) cArrOT got into the act, and made some additional, fancier machines later, and those are a lot of fun, too.

You have to give explicit consent before you can win or lose quota. After you've played a while, you can ask the pit boss for the tally to see whether you're ahead or behind, and for the scoop, to see everyone's current standings.

You can order a drink from our sexy waitress (sister to Abigail, who works across the street at the more respectable Club Dred), or, if it gets too noisy in the casino itself, you can repair to the High Rollers' Club (just to the north) and have a drink with your friends there.

To get to Yib's Palace! go northeast from the corner of Main Street and Queens Boulevard.

Skid Row (#73624)

Not long after Yib's Palace! opened, someone said to me, "Let me know when you've built Skid Row," and that's typical of how I get my best inspiration for building places. The challenge (as always) was to make it interesting and not just tiny scenery.

Its main purpose is to serve as a home for MOObums who have no other place to go: anyone may set eir home there. You can sit on the curb; disconnected players are depicted as sleeping in the gutter. The dumpster has a rotating inventory. You can toss things in, and even go dumpster diving if you're of a mind to. If you're feeling

really desperate, you can set your home inside the dumpster itself. You can also deface the dumpster with graffiti and read what others have written.

To get to Skid Row, go east from the corner of Main Street and Queens Boulevard.

The Drawing Room (#56651)

The drawing room was created to provide a venue that was close to the living room but for quieter conversations. It is one of three rooms served by James the Butler. (The other two are the smoking room and the library alcove.) James is the result of my second major project after the MOO helicopters. He is a 'bot who serves drinks, and was built in collaboration with my friend The_Walrus (I implemented the 'bot, The_Walrus implemented the drinks.) Butlers and bartenders are derived from the same generic drink-serving puppet; the difference is that a bartender will serve up absolutely anything, while a butler has a fixed selection of drinks to offer. In the early days of the drawing room, James's repertoire included only brandy, sherry, port, plus tea and coffee. His repertoire is still fixed, but the selection has expanded greatly over the years and for a few years, at least, James has also been authorized to deliver a programmer bit, which non-programmers may then install. James visits the drawing room periodically, offering refreshment; he can also be summoned by ringing the small bell on the curio table.

At one end of the room is a large aquarium filled with fish. There is fish food nearby, and feeding the fish is always a pleasant way to pass a few moments. This is one of a very few cases I know of where looking at an object gives more information than examining it – you must feed the fish by name, but only looking at the tank will list the names of the fish. The tank is also a gateway to the RPG system – stare at it to enter.

The curio table normally holds three items: the bell to summon James, a cherry puzzle box, and a stamp album. The cherry puzzle box is pleasant and fairly easy to solve. It is implemented as a portable room; to begin, you must remove the box from the table, drop it, and then enter it.

The stamp album, created by Bartlebooth, is one of my all-time favorite objects on the MOO. Each page in the album shows a selection of stamps; each stamp is based on an actual room in the MOO. You can study a particular stamp to see the room's full description and number. The stamps' descriptions are derived from a number of factors. The shape of each stamp, the "country of origin", what or whom is pictured on it, its value (in MooCents), and whether it is cancelled or not all have meaning. Type `about album` for those details.

To get to the drawing room, go northwest from the dining room.

The Smoking Room (#51556)

The smoking room was created at the same time as and in concert with the drawing room, expressing its authors' more exuberant side, and of the two rooms it is consistently the more popular. (Go figure.)

For swashbuckling fun, find a worthy opponent, take a fencing sabre each from the pair of hooks above the fireplace, and have at it.

For your smoking pleasure, there is a rack of pipes and a humidor of fine cigars. Each pipe is different, and blowing smoke rings is considered de rigueur. To my own surprise the exploding cigars are perhaps my favorite of all the things I've made on LambdaMOO. Best to smell one *before* you light it, however.

The bookcase offers a choice selection of classic literature, for your reading enjoyment.

Pull on the bell cord to summon the butler.

To get to the smoking room, go northeast from the dining room or east from the drawing room.

yduJ's Hair Salon (#3443)

This venerable establishment has been in existence for longer than I can remember, and it is a gem. LOVE your hair! Luigi's talent for giving one's hair new life is unsurpassed. Just chat with him and say what you'd like, and it's yours.

In Real Life, a haircut of great or questionable merit only lasts a few weeks. On LambdaMOO, it's gone within twenty-four hours.

To get to yduJ's Hair Salon, go east from the entrance hall, south into the stairwell, up one flight, west four times, then north. Or go up from the family room, east twice, and then north.

Hacker's Heaven (#4747)

This used to be the place where all the "in" people hung out, at least for a while – as is well known, places wax and wane in popularity over the course of time.

The magic number repository is a child of the recycling center. You can look at it to see what appealing object numbers are available, and request one if you wish. The magic number extractor can be used to search for additional magic numbers.

The LambdaMOO Official Helpful Person Badge Dispenser is here, and if you ever wondered where those Official Helpful Person badges came from, this is the place.

The number (it varies) is a special object that can be mathematically manipulated in a variety of ways and is not limited to 32 bits.

The other items include intravenous caffeine machine, the Gary_Severn Memorial Fission Reactor and Power Supply, thermometer, harmless geusting simulator, Capitalization Police, Anarchist, and Political Bumpersticker. They are artifacts of long-ago good times, created and programmed for the joy of it. Or else I'm not enough of a hacker to appreciate their deeper worth.

To get to Hacker's Heaven go north from yduJ's Hair Salon.

The Blue Iguana (#71896)

This very fine drinking establishment first opened its doors circa 1994. Sydney, the bartender, will be glad to serve you a drink.

There is a pool table here, which, while it takes a bit of focus to master, is well worth the effort and terrific fun. Beware of pool sharks, however – one's skill improves with practice. Toss the coin and have your opponent call it in the air to see who begins.

This would be an excellent venue for a good-sized party if one wanted a change from some of the more usual haunts. Best enjoyed by two or more people at a time.

To get to The Blue Iguana, go to the gazebo, `take shell from table`, then `listen to shell`. Then from KokoMOO beach, proceed north.

The Middle of the Ocean (#6404)

The ocean connects many of the beaches that people have built here over the years. Like the thicket, it utilizes a variable exit. At any particular moment, the tide will be flowing in the direction of some particular place. This changes with time, or, if you're tired of waiting, type `wait`, and the tide will change direction sooner.

To get to the middle of the ocean, type `swim` from any of several beaches. One way to get there is to listen to the shell on the table in the gazebo, then swim out from KokoMOO beach.

Sandcastle Beach (#7542)

Of the various beaches on LambdaMOO, Sandcastle beach is perhaps my favorite. You can bury things (and people!) in the sand, and build arbitrarily elaborate castles, too. To get started, type `about here`.

To get to Sandcastle Beach, swim to the middle of the ocean, wait for the tide to turn, then swim for shore.

Deep Thought's Foyer (#34650)

Deep Thought is the hardest puzzle on the MOO that I've solved so far (though I know of harder ones), and it is justifiably famous. It's a programming puzzle. The object is to get from Deep Thought's Foyer to Deep Thought's Lair. Good luck. More skill.

To get to Deep Thought's Foyer, proceed as follows, starting from the entrance hall:

east to Corridor
south to Ground Floor Stairwell
up to First Floor Stairwell
west to First Floor Corridor
south to First Floor Corridor
east to First Floor Corridor
north to First Floor Corridor
north to First Floor Corridor (Boardroom)
east to East Corridor
east to East Corridor
south to Hag's
south to Deep Thought's Foyer

Vent System (#23032)

The vent system is just vast. It connects many, many of the older public and private rooms on the MOO, and might even be used by MOO archaeologists to identify which rooms are among the oldest here.

I'm told that long ago the vent system used to rattle and clank mightily, but have never heard this myself. The naming convention is that the part before the hyphen in a vent room's name represents the floor (GW is ground floor west, 2W is second floor west, 15E is floor 1.5 east (because of short flights of stairs)). The part of a duct's name after the hyphen sometimes seems to give a hint about a connected room, if any, for example. "L" for laundry, "K" for kitchen, etc., but I have not broken all of this part of the code.

I haven't explored every bit of it, but the descriptions do vary: sometimes new, sometimes old, sometimes dusty, seemingly corresponding to the parts of the house that they run through.

There are many ways to enter the vent system, and there is a map in the map room (northeast from the library). Here is a quick tour of a part of it:

Starting from the entrance hall, go east to the corridor.
Then south into the stairwell.

Then up three flights to the top.

Then west to the utility roof.

Enter the air conditioner which is usually there (it's possible someone may have moved it; the housekeeper returns it periodically).

Go east.

Go down to the bottom. While you're there, jump on the springboard, and when you've had enough fun with that, go back to the bottom and proceed west twice.

Go up, and then out. You will be in a familiar place.

To get to the vent system, you can type `vent` from almost any room that's connected (the laundry room is one), and to exit the vent system to a connected room, you can generally type `out`. A few chambers use compass directions, instead.

The Edge of the World (#40309)

The Edge of the World used to be the place to go if you wanted to leave the MOO for good. In days of yore, one could divest oneself of various ties to the MOO (recycle all objects, get rid of all morphs, even give up one's name), then walk off the edge of the world and one's character would be recycled. MOOicide was discouraged by the wizards, and if one chose to leave the MOO in this fashion, and later wanted to come back, e would have to write a letter to the wizards saying why e MOOicided and how e intended to avoid similar situations in the future.

In 1997, *Ballot:MOOicide_Reform passed, which changed the behavior of the Edge of the World to `@newting` a player for one month less than the reap period. If the person continued to stay away and was subsequently reaped, e could get a new character and come back for the asking without having to explain emself to a wizard – or anyone else – unless e wanted to.

In 1998, *Ballot:Bring_Back_the_Blender passed, which called for the cuisinart in the kitchen to be modified so that it would `@toad` a player, and for jumping off the edge of the world to result in a newting similar to that associated with the Russian Roulette pistol, i.e. between two and six days. Implementation of this ballot is not yet complete. In the interim, the wizard TheCat (now Retired-Wizard-4) decided that while the blender part of *B:BBB was being implemented, the wizards could fudge the edge of the world part by manually un-newting anyone who had been a newt for more than a week.

In 2000, *Petition:Drama_Queens was written to request that the wizards cease this behavior. The ballot passed.

The departure log lets you read a list of people who have walked off the edge.

To get to the edge of the world, go north from the driveway, then west as far as you can go.

```
You see a banana peel here.
```

Chapter 8 – LambdaMOO-Specific Reference Information

LambdaMOO, the original and largest MOO in existence, has been extended by its users in a marvelous variety of ways. This chapter attempts to document some of the highlights. It is divided into three sections: Feature Objects, Player Classes, and a detailed description of LambdaMOO's political system and how to use it.

A Short Compendium of LambdaMOO Feature Objects

There are around a thousand feature objects on LambdaMOO, which makes a thorough survey impractical. There are some, however, that I've come to think of as "standards of the jazz repertoire", in a manner of speaking, either because of their popularity or usefulness or both, and it is those that I choose to present here.

It's not uncommon for a feature object to have a variety of unrelated verbs, the theme between them being merely the author who wrote them. I have written about those verbs that inspired me to choose these feature objects as examples; you should read the help text for a particular feature object (FO) to see what other commands it offers.

The section is divided into three main categories: popular social feature objects, informational feature objects, and utilities.

Popular Social Feature Objects

#30203 (Stage-Talk Feature)

Stage talk is also known as *directed say*. This FO lets you direct speech to a particular person. To use it, precede a player's name with a dash. If I type:

```
-Plaid_Guest Hi.  Welcome to LambdaMOO.
```

then everyone in the room will see:

```
Yib [to Plaid_Guest]: Hi.  Welcome to LambdaMOO.
```

This FO also provides a way to simulate pointing to yourself and saying something. If I type:

```
<In a silly mood, today.
```

then everyone in the room will see:

```
Yib <- In a silly mood, today.
```

The Stage-Talk feature is provided as part of LambdaCore.

#40842 (Social Verb Core and Feature Object)

This is a very popular set of "short cut" verbs for frequently used gestures. The verbs can be used either alone or directed at a player or other object in the room. If I type:

```
wave eep
```

then I see:

```
You wave to eep.
```

eep sees:

```
Yib waves to you.
```

The other people in the room see:

```
Yib waves to eep.
```

The social verbs provided by this FO are:

comfort	cry	nod	hug
wink	poke	grin	kiss
yawn	shrug	smile	french
wave	blush	laugh	bow
cackle	cringe	sigh	
giggle	smirk	chuckle	

You can type `social` to see a list of these verbs online. Several of the verbs have text which modifies them, for example `nod` depicts you as nodding solemnly.

The help text explains how to customize these verbs if you don't like the modifiers provided by the FO itself.

Versions of this feature object are frequently found on other MOOs, but are not standardized.

#21132 (Antisocial Feature)

This verb is a natural follow-on to the social verb core. The syntax is basically the same, except that you can't use the verbs by themselves – you have to designate a target. On the other hand, you can specify more than one target, for example:

```
eye Klaatu Kirlan Boo
You eye Klaatu, Kirlan, and Boo warily.
```

The available anti-social verbs are:

eye	pat	eyeball
grump	paperwork	mess
feh	poke	wake

```
rtfm            pout            sycophant
ignore          pave            smoke
waggle          BillyBragg      salad
Blake           bop             report
glare           EPOXY           Snideley
toy             waffle          sledge
pound           postage         tile
silly           mmt             wither
gag             rael            alpo
frown           cough           crush
prod            ice             deconstruct
unplease        flame           smell
WhiteRabbit     roll            stern
unamused        divine          shoebox
peer            disembowel      twist
snooty          Buddy           thwap
face            defenestrate    whuggle
growl           neuter          Wholeflaffer
tongue          handcuff        Sapphos
ridicule        hoop            value
bury            blame           reverse-@eject
sigh            dist            boot
```

I recommend trying them out with a good-natured friend.

#4572 (APHiD's Socializing Feature Object)

This verb is an extension of `emote`. It enables you to direct any gesture to a particular person (or to everyone in the room) rather than having to rely on those provided by other feature objects or having to customize a message in advance. The command begins with a period (`.`) followed by the first person singular of any verb (no space in between) followed by the rest of your text. If I type:

```
.hit Nim up for some chocolate.
```

I will see:

```
You hit Nim up for some chocolate.
```

Nim will see:

```
Yib hits you up for some chocolate.
```

Everyone else will see:

```
Yib hits Nim up for some chocolate.
```

If you have this feature and type `@social-option +all`, then you can use the syntax:

```
.wave to everyone
```

Each person will see a message listing everyone in the room except that the word "you" will be substituted for eir name in the message.

There are other Feature Objects that offer variations of this verb, which is sometimes referred to as *posing*.

#5490 (Dancing Feature Object)

I've included this feature object in my compendium because it's fun and because it's one that people frequently ask about. It provides a selection of dances which you can perform either solo or with a partner. It provides a verb, `@polite`, which lets you be polite (ask, first) or more impulsive when you dance with someone. You can type `@dances` to see the choices available.

```
Werebull hands Yib a rose which she places between her
teeth.  Then Werebull leads Yib through a rhythmic tango,
stepping across the floor and ending with Werebull holding
Yib in a low dip.
```

This feature object was originally programmed by APHiD.

Informational Feature Objects

People are naturally curious about the world around them. Or nosy, depending on your point of view. The properties and verbs on just about everything are readable, and I am continually amazed at the number of ways that programmers find to combine disparate data and draw unexpected conclusions. There is a slowly evolving tug of war between the snoopers and the snooped-on, as well. As verbs query ever more extensively, counter-verbs are written to detect and/or deflect various queries. As detection verbs become known and come into more widespread use, snooping verbs become more stealthy in their methods, and on it goes.

This section is divided into three parts, based on the kinds of things the FO's provide information about: People (players), places, and things.

Feature Objects that Primarily Provide Information About People

#24222 (login watcher)

The login watcher informs you when other players connect to or disconnect from the MOO. It provides a way for you to designate individual players as "interesting", and then you will see messages like the following:

```
< connected: Tartan_Guest.  Total: 192 >
< disconnected: Elephant_Ears.  Total: 186 >
```

The `@wwho` verb on this feature object shows you an `@who` listing of only those players you consider interesting (instead of everyone on the MOO).

Your @interesing list is technically private (the property where it's stored is unreadable), but the *act* of adding someone to or removing someone from your @interesting list can be detected, as can the use of @wwho. (See also help #14141:@sint.)

#24262 (Fast & Dangerous Info FO)

This is the most comprehensive informational FO that I know about, though it gets occasional criticism for not being stealthy enough.

```
@fbi
```

will give you a detailed report with lots of information about a player.

```
@kgb
```

will do essentially the same thing but with less chance of triggering someone's detection verbs, if e has any.

Both of these verbs are resource hogs, and the FO provides a large suite of more specialized verbs that you can use if you only want to know a particular thing about someone and not eir entire life story. These are:

@@ <player or players>	Gives quick information about one or more players' location(s).
@aka <player>	Lists someone's aliases.
@morphs <player>	Lists someone's morphs.
@nohelp <object>	If an object doesn't have formal help text per se, this verb will list all the help for the individual verbs.
@xref <player>	Shows some relational information about <player>, for example whether you own kids of eir generics or use feature objects e owns, etc.
@bdays	Shows you the names and ages of players whose birthday is today. (Uses info from the birthday machine in the Living Room.)
@cwo	Lists @interesting players who **c**onnected and disconnected **w**hile you were **o**ut.
@enemies <player>	Prints a list of people with whom <player> "might have strained diplomatic relations". This might include persons that <player> has @gagged, @refused or @banned, for example.
@fields <topic>	Prints a list of specific fields known to the helpful person finder, about which you may then make further inquiry to identify a helpful person. Example: @fields newbie

@fusers <feature object>	Prints a list of players who use the specified feature object.
@holding <player>	Prints an easy-to-read list of the items the specified player is carrying in eir inventory.
@horniness <player>	Offers information about the kind of sex that <player> seems to be horny for, if any. This is determined by <player>'s .horny property (if present), which can be either a string giving the object number of a player (e.g. "#50222"), any string (e.g. "vanilla"), or a one-letter code. The codes are as follows: "S"(traight), "G"(ay), "B"(I), "F"(arm, i.e. animals), or "X" (bondage, discipline, sadism or masochism). A person will only be reported by this verb as being horny if e is registered with the birthday machine as being 18 years of age or older.
@intruders @intruders <player>	Lists people who are in rooms you own or in rooms that <player> owns.
@kusers <object>	Shows all kids of the specified object and their owners.
@spy <player>	Shows a description of the room <player> is in, plus who's with em.
@log @log <player>	Shows recent login and logout activity for the specified player.
@oc	Gives an **o**nline **c**ount of how many players are connected and maximum new logins allowed.
@on	Shows a list of everyone online. Uses shortest aliases and tries to fit it all on one screen.
@opals @opals <player>	Gives a list of **o**nline **p**als of the specified player. (This verb is useful because different player classes have different ways of designating pals.)
@reapable	Shows you whether any of your pals or anyone you have designated as @interesting is up for reaping soon.
@recon <room>	Prints information about a specified room, including its description, the names and genders of connected players in the room, and some information about the room's security setting, if available.
@sen <player or players>	Provides information about players' seniority based on MOO ages.

@status <player>	Shows abbreviated information about a player's status on the MOO. See help #24262:@status for an explanation of the various fields and abbreviations.
@subscribers	Shows a list of players who subscribe to a list with notification. (The identities of players who subscribe to a list without notification are not available to this FO.)
@using #24262	Gives online information about all the verbs on this FO, including the syntax for each verb.
@wf @wf <location>	Think ("**w**here **f**rom") Lists players from a specified location. Uses #5365 (Real Life Registry). (See help #5365.)
@xpr	Lists the "e**xp**e**r**ience levels" of all connected non-guest players sorted into five MOO age categories, "dinos", "fogies", "oldies", "middies", and "newbies". See help #24262:@xpr for definitions of these terms.
@youths @youths!	Prints a list of players believed to be 16 years old or younger (according to the birthday machine). The second form lists both connected and unconnected players.
@geezers @geezers!	Prints a list of players believed to be as old as or older than Haakon (#2) (according to the birthday machine). The second form lists both connected and unconnected players.
ii <player>	Prints a player's inventory, with object numbers.
wias with <string>	**W**h**i**ch (player) **a**liases **s**tart with a specified string? Useful for screening possible names you might want to add.
woas with <string>	**W**h**o**se **a**liases **s**tart with a specified string? This version is much spammier than wias, giving each player's name and number and the alias in question that e goes by.

#36714 (Carrot's Social Interaction Feature)

This feature object shows information about players' MOO ages, offline ages (as provided to and by the birthday machine), and also provides some verbs that list players' friends and cliques.

#1283 (Karma FO)

This FO is strictly for fun, but is popular. It gives you the same information about yourself that you can get by sitting and meditating in the Japanese Garden, and also will give you similar information about other players. There is a note on the reference shelf in the library that explains how one's karma is determined.

Feature Objects that Primarily Provide Information About Places

#23824 (Compass Rosette Feature Object)

This is an *extremely* useful FO for exploring – a very good adjunct or alternative to the `@ways` command. The `@rose` command can be customized in a variety of ways. This feature object has been ported to many MOOs, with the author's permission.

#46278 (Carrot's Viewing Feature)

This FO provides the `@dview` command, which capitalizes the names of details in a room's description or a detail itself. One might argue that this is cheating, but on the other hand, it might lead people to look at some details that they might otherwise not have bothered even to look for.

#41975 (Obvious Features Object)

This feature object offers a potpourri of verbs. One that is very useful for exploring is `@lrs` (also `@lrs <room>`) which stands for "long range scan". It presents a tiered list (three levels) of exits and destinations accessible from the room you're in or the room you specify.

Feature Objects that Primarily Provide Information About Objects

#10218 (Mazer's Object Utilities Feature Object)

This FO is a nice collection of commands for investigating the object hierarchy. It includes `@ancestors`, `@descendents`, `@descendents_suspended`, `@branches`, `@branches_suspended`, `@leaves`, and `@leaves_suspended`.

#113366 (LambdaMOO Museum Search FO)

The number of generic objects on LambdaMOO has become almost mind-bogglingly large. The `@msearch` command helps you narrow your search. You could, for example, type `@msearch puppet`, and after the search was complete you

would be directed to specific displays in the museum showing objects that have the word “puppet” in their name.

Utility Feature Objects

#35353 (Pasting Feature)

This feature object is included with the LambdaCore. You can use the `@paste` command to display multiple lines of text to everyone in the room you're in, or `@pasteto <player>` to display multiple lines of text to one person. In both cases, the system prompts you to type in lines of text, followed by a period on a line by itself. In lieu of typing, of course, you can paste in text from another window. Note, `@paste` is sensitive to lag and sometimes the text doesn't appear immediately. Output from the pasting feature object looks like this:

```
----------------------Private message from Frebblebit----------------------
This text was generated with the
@pasteto command.
--------------------------------end message--------------------------------
```

#25552 (Multi-communications feature)

This is the feature object that enables players to talk on channels. It has extensive help text. The basic commands are:

`@channels`	Lists all channels.
`@xcon <channel>`	Quit one channel and start listening to another.
`@xsw <channel>`	Continue listening to your current channel, but talk on and listen to a new channel.
`@xsilence <channel>`	Quit listening to and talking on a channel.
`@xm <text>` `xm <text>`	Broadcast `<text>` on your current channel.
`@xm: <text>` `xm:<text>`	Emote `<text>` on your current channel.

`@xwho`
`@xwho <channel>`
`@xwho all`
	Shows who is listening on your current/designated channel, or all channels.

#65000 (Quota-Transferral Feature)

In 1994, `*Ballot:Quota_Transfers` passed, enabling players to transfer quota to one another. This feature object is the mechanism by which quota transfer is accomplished. The syntax is:

```
@qt <amount> to <recipient> <optional reason>
```

You must be at least one month old to transfer quota. Transfers are posted to the public mailing list `*Quota-Transfer-Log`.

Popular LambdaMOO Player Classes

The section on player classes in Part I summarized all the commands on all the player classes that are provided as part of LambdaCore. This section outlines the commands available on the most popular LambdaMOO player classes.

LambdaMOO has over sixty player classes to choose from. For this section, I have chosen classes from the player class hierarchy that are used by about 70% of the LambdaMOO population.

Details of the syntax and usage of each command are given in Appendix A. Simple summaries are presented here to give a sense of what each of these player classes has to offer.

Generic LambdaMOO Citizen

All players on LambdaMOO are members of the Generic LambdaMOO Citizen player class. People who run other MOOs might want to consider putting a citizen player class between Frand's player class and generic builder.

Some of these commands were mandated by ballot, others are utility commands relating to structures that are specific to LambdaMOO, (e.g. the RPG system).

`@tutorial` – Transports you to the tutorial, where you can practice a variety of basic commands.

`@ban, @ban!` – Universally bans an object from all rooms that you own. The second form also bans any children of the banned object.

`@unban` – Stop banning a person or thing from rooms you own.

`@banned` – List those players and things you have `@banned`.

`@gms` – Prints a list of RPG Game-Masters

`@make-petition` – Create a petition object.

`@petitions` – List existing petitions.

`@ballots` – List ballots. With no arguments, lists open ballots. Can also take any of "`passed`", "`failed`" or "`all`" as an argument.

`@petition-options` – Displays and/or sets various options relating to petitions, including whether or not you want to be automatically informed about open ballots.

@nominate – A command to nominate a player to elected office. The elected boards are the Architectural Review Board (ARB), the reapers, and the registrars.

@arb-petitions, @reaper-petitions, @registrar-petitions – List petitions of people currently nominated for the Architectural Review Board, reapers, and registrars, respectively.

@arb-ballots, @reaper-ballots, @registrar-ballots – List the specified open ballots.

@boot – A command which permits players to boot a guest off the system. Two players must act in concert, and a reason must be given. The offending guest's site is blocked for 1 hour.

@witness – A set of commands (see `help @witness`) to reliably log an exchange between players, view such logs, and publish such logs to a mailing list.

@age – Displays a person's age, with deductions for system down time. (The deductions are an enhancement to the generic player's `@age` command.)

@arb, @reapers, @registrars – Display a list of current Architecture Review Board members, current reapers, or current registrars, respectively.

@flush-cache – Delete one's feature cache. Feature caching was mandated by ballot #71687, although an inspection of the code suggests it may only have been partially implemented.

@will – A command to specify how you would want various objects you own to be distributed, if you were to leave and never come back.

@email – LambdaMOO has an email policy which prohibits the use of solely free anonymous email addresses. Existing users with such accounts must provide an *identified* email address (one whose owner's name is known or knowable), either their own or a relative's, in order to gain full access to LambdaMOO. `@email` is the command by which such users can provide a valid email address. (See `help @email` for more information.)

@gag-site <guest> for <duration> – Gag a guest and all guests from the same site as the specified guest.

@ungag-site – Stop gagging all guests from a particular site

@gag-sites – List the guests whose sites you have gagged.

Generic Player Class With Additional Features of Dubious Utility(#7069)

This player class was programmed by Gary_Severn (#15), who was also the wizard Dukas. There is no help text for it, per se. I presume it was programmed before adding help text to player classes and generic objects was as established a convention as it is today.

seek – This command tries to combine the convenience of teleporting with the appearance of walking. If you `seek` a person, it tries to find an entrance into the

room e's in and move you through it. It only sort of works, but the concept is good. (There is an updated version of this on FO #27325:@seek.)

Another feature of #7069 that isn't a command per se:
The title verb affixes an idle status to your name when someone looks at a room you're in.

Experimental Guinea Pig Class with Even More Features of Dubious Utility (#5803)

(See also help 5803-index.)

@ss/@sshow – A short version of `@show` that doesn't list properties or verbs.

ways – Very like `@ways`. Shows a short list of obvious exits. Unlike `@ways`, it doesn't let you specify a location, but only shows you exits from the room you are in at the time. My speculation is this: This command existed first, then the more flexible `@ways` command was added to Frand's player class at a later time and `#5803:ways` was rendered obsolete but no-one thought to remove it.

! – A polite spoof verb. `!<string>` announces an arbitrary string to your current location, except that if the string doesn't contain the your name somewhere as a distinct word, your `.spoof_attribution` property (with the substitutions of player.name for `%n` and `%` for `%%`) is appended and if the resulting string *still* doesn't contain your name, then your name is appended anyway.

@lastlog – This verb takes two forms:

```
@lastlog <player> <player> <player>
```

is just like `@who`, except that it shows disconnected players before connected players.

```
@lastlog for <number> <day/week/month>[s]
```

(e.g. `@lastlog for 2 days`) lists all players who have logged on during the specified interval.

boring – `boring on` makes you immune to food fights; `boring off` allows you to participate again. Subsequent to the creation of this command, at least one other object was modified to respect the `.boring` property: If you are `.boring`, then things will not fall out of your pockets down between the Living Room couch cushions.

@pedit – A property editor for properties that are other than strings or lists of strings. The verb documentation describes it as "experimental".

party – Looks for various rooms with more than one player, tells you who's there, how long they've been idle, and asks if you'd like to go there.

heartbeat – Starts up a task to print out the time every `<n>` minutes from now on unless you've been active during that minute. Good for people who like to stay

idle for long periods of time and who want to have some means of time stamping occurrences that show up in their client buffers. (This verb is actually not callable from the command line; one has to write a separate verb to call it, or use `eval`.)

`+` – This is the remote-emote command. It is kind of like a regular emote in that you specify your actions in the third person, but you specify the player who will see it, and you don't have to be in the same room as em. If I am in Yib's Study, and I type `+boo beans you with a water balloon`, then Boo would see: `(from Yib's Study) Yib beans you with a water balloon.`

`eprint` – This is a command that will mostly be of interest to programmers. Type `eprint <long-horrible-complicated-moocode-expression>`, and it will format said expression to fit within `.linelen` columns in a way that's (usually) more readable.

`@prettylist` – This command lists a verb with indentations that are intended to make it easier to read. Its author (Rog) acknowledges that it is slow.

`@nprop` – The author of this verb (Rog) writes:

The only difference between `@nprop` and `@prop` is that `@nprop` does a full eval of its value argument, whereas `@prop` only accepts literals; e.g.,

```
@nprop foo.bar x:contents() rc Wizard
```

It may also be that I never moved this to #217 because I decided that doing a full eval wasn't a good idea anyway (if you need to do it, then you may as well just do `;foo.bar = <whatever>`).

`@nlist` – The author of this verb (Rog) writes:

`@nlist` seems to differ from `@list` in accepting an `upload` argument, which causes the listing to appear in a form that can then be edited and then blasted back by a client. This has been superceded by local editing (i.e., `@edito +local`, `@edit foo:bar`) and so is likewise not terribly useful anymore.

`page` – This is a fancier version of the original page command that enables you to page more than one player at a time. If you do, you must enclose the list of player names in quotes:

```
page "Klaatu Nim Bartlebooth" Party at Club Dred!
```

Player Class hacked with eval that does substitutions and assorted stuff (#8855)

This player class has a hodgepodge of commands, some of which are strictly of interest to programmers, but others of which are of a more general interest. See also `help pcs-index`.

`follow` – This command is not to enable you to follow someone, but to allow someone else to follow you, although since most LambdaMOO players are descended from this player class, pretty much anyone can follow anyone.

stop-following – A person would use this command to stop following you.

@list-followers – List the people who are following you.

lose – lose <name> will cause <name> to cease following you. lose everyone will cause everyone to cease following you.

'<player> <text> – A shortcut for paging someone. Append the name of the player you want to page directly after the ' character:

```
'Klaatu Hi!  It's been ages!  How have you been?
```

@parent – Tells you an object's direct parent (as opposed to a list of its ancestors, which is what @parents does).

@define, @undefine, @listdefs – For those who program, the @define command lets you pre-define one or more variables, which are then in place when you run the eval command (i.e. globals). @undefine lets you remove definitions, and @listdefs lists what you currently have pre-defined.

Politically Correct Featureful Player Class Created Because Nobody Would @Copy Verbs To 8855 (#33337)

This player class enables you to refuse a couple of additional actions from players. **@refuse flames from <person>** inhibits your reading posts from said person on public mailing lists. Instead of the body of the message, you will see something like:

```
[ Flame from ReaDinG (#61050) skipped: 133 lines. ]
```

If curiosity gets the better of you, you can either @unrefuse flames from the person, OR use @peek to look at the message and see what it actually says.

mu – This stands for "murmur". It's an alternate form of whisper, whose syntax is like page. mu Yib Shall we withdraw to the drawing room? is the same as whisper "Shall we withdraw to the drawing room?" to Yib. The advantage of the first over the second is that it's shorter to type, and you don't have to put the text to be whispered in quotes.

@watch – This command tells you when an idle player becomes active, again. When the player does become active, the system prints eir name enclosed in square brackets on a line by itself:

```
[Yib's_Assistant]
```

It is possible for a player to detect when you start watching em (by modifying eir :title verb).

goto – This command has been disabled, but is documented here as an item of historical interest. One could type goto <location> and be caused (if possible) to walk to the location using exits rather than by teleporting. The verb has a comment indicating that it was disabled because it was "too spammy".

'<player> <text> – This is an enhanced version of the shortcut page verb. It records the last person you paged, so that subsequently you can omit eir name:

```
'Kirlan Fee, Fie, Fo
' Fum!
```

Kirlan would see both lines as a regular page from you.

@pc-news – This was a way for the author of this player class to simulate a new news item, but only for members of this player class.

@set-tell-filter, @unset-tell-filter, @tell-filter – A tell filter is an object which can intercept text that you are about to see and modify it in some way before you see it. One might use one, for example to prepend the name of the person typing, if it doesn't appear in the text itself, or to add a time stamp to every line. `@set-tell-filter` designates such an object. `@unset-tell-filter` ceases to use your current tell filter. `@tell-filter` tells you what tell filter you are currently using.

@pc-options – This player class uses an options package similar to the mail-options package. The `@pc-options` command shows you what the options are and how you have them set. In particular, you can choose to set a crosspost limit (ignore messages on mailing lists if they've been cross-posted to more than a specified number of lists). You can also have the system ask you for confirmation before displaying a long mail message.

Here's an odd note: The Schmoo class, #4803, has a shout verb. #33337 isn't Schmoo class, but was able to intercept Schmoo shouts. The verb `:who's_infidel_slime` fingered those non-Schmoos who intercepted Schmoo shouts. See also `help pc-index`. I don't know why it's called "Politically Correct".

Generic Super_Huh Player (#26026)

This player class offers the very helpful facility to "note down" object numbers for future reference. In *some* cases you can refer to these objects by name instead of number. For example, if you type `@remember #2031`, which is Yib's Guide to Interesting Places, you can type `read guide` at any time and read the current page, even if the guide is on its shelf in the library. Unfortunately, this only works some of the time (because of how the parser makes decisions about what object you really mean when you type a command).

@remember – This is how you "note down" an object's number for future reference.

@known – Lists those objects that you have opted to remember

@forget – Remove an object from your list of known objects.

Detailed Player Class (#6669)

The detailed player class lets you define a detail on yourself, so that if your description mentions a snazzy belt buckle, you can make it so that the command `look buckle on <you>` tells the person looking some details about the belt buckle.

`@detail` – This command lets one add, change, or remove a detail. Type `help #6669:@detail` for the various syntaxes.

`@details` – This command lists the details you have defined on yourself.

Other Player Classes

Here we come to the major branch between general player classes. Most people describe the division as being between `SSPC (#49900)` (a descendent of `Sick's Sick of Spam player class (#59900)`) and `Super-Schmoo (#4803)` (a descendent of the `Piping Player Class (#20781)`). Different players find different advantages in each. (The reason why an object can be a descendent of an object with a higher number than itself is because object numbers are recycled and re-used.)

Feature objects were a later invention than player classes, so in the early days the commands available to you were mostly a function of what player class you chose. This all happened before my time, but by many accounts there was some strong competition between player class authors to garner users. The two branches provide several of the same functions but do so in different ways. These include morphing, keeping a list of pals so that you can see which of your pals are logged on, etc., and answering machines (a much later addition). The Schmoo player class also provided the facility to define various descriptions associated with various states of undress, and has what is called "Schmoo shouting", enabling users of the Schmoo player class to broadcast remarks to one another.

The major drawback to fancy player classes, but particularly ones that provide morphing, is that they take up a lot of quota. Aside from that, it's probably just a matter of taste regarding the syntax of the commands provided. Most of the functionality is the same, now, between the two branches, and with few exceptions most new commands are provided via feature objects, instead.

An Overview of LambdaMOO's Political System, and How to Use It

This section outlines the history of how LambdaMOO has been governed from its inception to the present, and gives a detailed description of how to use the mechanisms of the ballot system that we currently have in place. (For those who are

curious about what kinds of issues arise to vote on, a compendium of all closed ballots (as of May 10, 2003) is provided in Appendix E.)

Historical Overview

The birth of LambdaMOO is generally recognized as the date on which Haakon (#2), the ArchWizard, first connected, which was October 31, 1990. At the very beginning, LambdaMOO consisted of just a few friends, and the differences that arose were simply worked out informally. As the MOO grew in size, it ceased to be a place where everyone knew everyone else. When differences arose, people tended to turn to the wizards to help them sort out their squabbles and/or make an arbitrary decision, and the wizards tended to try to assist. Eventually, the wizards asked Haakon to get them out of the "discipline/manners/arbitration" business, and in December of 1992, Haakon posted "LambdaMOO Takes A New Direction", (see Appendix C) in which he decreed that the wizards would henceforth serve strictly as systems programmers and that players at large were now free (whether they wanted to be or not) to sort out their own differences.

Some time later a character going by the name of Mr_Bungle depicted some other characters as brutally sodomizing themselves in the Living Room. This was decried by many as abusive and intolerable, and there was much discussion on public mailing lists (mostly `*Social-Issues`) of what to do. In this particular instance, a wizard eventually took it upon emself to `@toad` Mr_Bungle, but the fact that the populace had no *mechanism* with which to govern themselves suddenly became clear. Haakon then created a petition and ballot system, intended to provide a way for the citizenry to express their collective views to the wizards and compel them to take specific technical actions intended to have social consequences. (See `help petitions-motivation` for some additional details about the particulars of how the petition system was implemented.) Haakon's original vision was that the petition/ballot system as given would be used by the populace to bootstrap a more sophisticated system. This has not yet come to pass.

The first petition to pass as a ballot was `*Ballot:Arbitration`. It called for the creation of a system that players could use to resolve disputes among themselves, including a system for volunteer arbitrators and a system for making "minor" changes to the arbitration system itself without having to go through the larger petition/ballot process. Every petition is subject to vetting by the wizards as one of the requirements it must meet before being promoted to a ballot, and today that petition would probably be denied vetting for being "insufficiently precise for the wizards to know how to implement it." `*B:Arbitration` was authored by a wizard, though, and everyone was optimistic, and no one foresaw the pitfalls that were in store. As it turned out, the arbitration system was fraught with difficulties, and often used for "playing games with the system" rather than in a sincere effort by players to resolve their differences with one another. `*B:Arbitration` was repealed in February of 1999; as of May 10, 2003, no conflict resolution system has yet been legislated to take its place.

The wizards tried very hard to restrict themselves to purely technical actions and not take any actions that would have social consequences, but with LTAND Haakon had, in fact, made a promise on the wizards' behalf that was impossible to keep. Many technical decisions have social consequences, and a choice not to execute some technical action can also have social consequences. Too often the wizards found themselves in a position of "damned if they did and damned if they didn't," and the damnations finally became more than the wizards were willing to bear. In May of 1996, the LambdaMOO wizards collectively authored a document titled, "LambdaMOO Takes Another Direction", commonly known as LTAD (see Appendix D for the full text). This document acknowledged that the line between technical and social decisions often is not a clear one, acknowledged that the wizards had (of necessity) made decisions in the past that had social consequences, and acknowledged that they would continue to do so. LTAD formally reintroduced wizardly fiat. In an effort to counterbalance the reintroduction of wizardly fiat, the wizards also announced the creation of a special "standing" petition, `*P:Shutdown`. Should it pass as a ballot, the wizards pledge to shut LambdaMOO down for good. `*P:Shutdown` has come to ballot (and failed) on two occasions. This special petition is pre-vetted, and furthermore requires only a simple majority to pass. The reason for requiring only a simple majority is that most of the wizards felt that if more than half didn't want LambdaMOO to keep going, they'd just as soon pack it in, themselves. (The author of this book was a LambdaMOO wizard at the time and participated in the crafting of LTAD.)

A Citizen's Guide to the LambdaMOO Petition System

But First, For Those Who Just Don't Want to be Bothered

LambdaMOO's petition system has a variety of mechanisms in it to ensure that players are adequately notified of open ballots and current elections. For some, this is too much of a good thing. Those who do not wish to be notified or lobbied should type the following commands:

```
@petition-option +noannounce
@petition-option +noannounce_ARB
```

New ballots and ARB elections are announced in *news. In addition, each time a player logs in, e is notified of ballots and/or ARB ballots on which e has not yet voted. These two petition options suppress these notifications. (You will still see the *news entry.)

```
@refuse politics for <duration, e.g. 100 years>
```

Refusing politics is an advisory act rather than a prohibitive act in that people are supposed to refrain from lobbying you if you have `@refused` politics, but this is not programmatically enforced. An older system depended on players adding a `.apolitical` property or `:apolitical` verb to themselves, and some players still utilize this method. See `help apolitical`.

To see if someone else has refused politics, type:

```
@refusals for <player>
```

The Particulars

Only primary, non-guest, non wizard characters may participate in the petition system. The voting age is 30 days from the time of first connection.

A player must be at least one year old to run for the Architecture Review Board.

A player must be at least four months old to run for the office of reaper.

A petition must acquire at least 10 signatures before it may be submitted for vetting.

The number of signatures required for a petition to be promoted to a ballot is 10% of the average of all votes for and against all previous closed ballots, not to be fewer than 50. The number of signatures required to promote a nomination petition to a ballot is 50. For a nomination petition to be promoted to a ballot, the nominee must be one of the signatories (indicating acceptance of the nomination).

A regular ballot requires a 2/3 majority of yes/no votes to pass. There is no minimum number of votes, i.e. no quorum.

For elected offices, those players who receive more "yes" votes than "no" votes are eligible, and ballots are ranked, in decreasing order, by the difference between the number of "yes" votes and the number of "no" votes. Of those, the top `<n>` are selected, where `<n>` is the number of vacancies on the board for which elections are being held. (See `help @ARB-ballots`.)

Votes are secret. Upon closure of a ballot, the object numbers of those who voted yes or no are stored on a randomized list, though which way a person voted is not stored. The list of those who voted is readable only by wizards, and is kept only for reference should there be a case involving accusations of multiple-character voting.

Time limits for petitions and ballots:

From first creation to author's signature	No expiration.
From author's signature until submission for vetting (10 or more signatures required)	14 days.
During vetting	No expiration. Signatures may not accrue while a petition is waiting to be vetted.
From vetting to acquiring enough signatures to promote a petition to a ballot	90 days.
From promotion to a ballot to ballot closure	14 days.

Should the system crash, the clock is considered to be stopped. Petitions and ballots have the duration of the down time added to their expiration times; players have the down time subtracted from their age for purposes of participating in the political system. (This is why the `@age` command gives a second (younger) age "for official purposes".)

Vetting Criteria

Before a petition may become a ballot, it must be vetted by the wizards. The vetting criteria are as follows:

A petition must be:

- Appropriate according to the guidelines given in `help petitions` (see below).
- Sufficiently precise and detailed in its description of the desired effect or facility that the wizards can understand how to implement it. (It is best, however, if you do not specify a particular implementation.)
- Technically feasible for the wizard to implement.
- Not likely, in the wizards' opinion, to jeopardize the functional integrity of the MOO.
- Not likely, in the wizards' opinion, to bring the wizards, Pavel Curtis, Stanford University, or Placeware, Inc. into conflict with any real-world laws or regulations.
- Consistent with passed ballot #55018, which states, "No petition may call for any change which results in differential treatment between those who sign it and those who do not."

The following is set forth in `help petitions`:

> Petitions and ballots are for proposals that the wizards perform some set of purely technical actions that only wizards can perform, where those actions are intended to address some social goal. For example, the actions might be intended
>
> - To directly achieve some social goal (such as banishing some individual from the MOO or removing some person from the list of wizards).
> - To grant some piece of wizardly power to the general MOO in some form that's intended to be used to address social problems in the future (such as creating a truly escape-proof jail or giving players a way to temporarily banish each other from the MOO).
> - To restrict some or all players in some way that is believed will help to achieve some social goal (such as keeping guests from logging in anonymously or making it impossible for players to print out messages containing certain words.
> - To modify the petitions and ballots mechanism itself in some qualitative way (such as changing it to require fewer signatures on petitions or only a simple majority on ballots).

Petitions and ballots are not for simple requests that the wizards fix some bug in the core nor, at the other end of the spectrum, are they for proposals that involve the wizards' having to exercise some kind of social judgement role (such as demanding that the wizards banish anyone who's being abusive).

Participating as a Voting Member of the Populace

Petitions and ballots are objects, owned by the system character, `Petitioner (#4)`. As such, each has its own object number, by which it can always be referenced, but one may also reference a petition or ballot as follows:

```
*Petition:<name or alias>
*P:<name or alias>
*Ballot:<name or alias>
*B:<name or alias>
*ARB-Petition:<player name>
*ARB-P:<player name>
*Reaper-Petition:<player name>
*Reaper-P:<player name>
```

The generic ballot is a child of the generic petition, and so all commands that can be used on or with petitions can also be used on or with ballots, though some may not apply and are disabled as appropriate. (You can't sign a ballot, for example, but you can list signatures on petitions *or* ballots.)

A petition's or ballot's status is recorded in its description. This includes the number of signatures acquired and required, whether it is up for vetting, expiration date, etc. As per the naming conventions listed above, you can type `look <petition or ballot>` to get some basic information about it.

To see a list of petitions or ballots, type one of the following:

```
@petitions [all | public | signed | vetted]
@ballots [all | open | closed | passed | defeated]
@arb-petitions
@arb-ballots
@reaper-petitions
@reaper-ballots
```

Use the commands `decline <petition>` and `undecline <petition>` to control which petitions display when you use the `@petitions` command.

You can type `read <petition or ballot>` to view its text. You can mail a petition or ballot's text to yourself by typing `mailme <petition or ballot>`. Wizards sometimes add implementation notes to petitions they have vetted. You can read these notes (if present) by typing `impl <petition or ballot>`.

Petitions and ballots are also mailing lists. You can read mail on them and send mail to them just as you would any other mail recipient.

To sign a petition, type `sign <petition>`. If you have a feature object that depicts you holding up a sign with text on it, you may need to use one of the alternate forms of this command, `@sign <petition>` or `psign <petition>`. You can remove your signature from a petition at any time by typing `unsign <petition>`.

If you are one of the first ten people to sign a petition, i.e., if it has not yet been submitted for vetting, then ideally you are signing to indicate not only that you think the measure deserves to be brought before the LambdaMOO populace as a ballot, but also that you have reviewed it and believe that it meets the vetting criteria.

To vote on a ballot, type one of:

```
vote yes on <ballot>
vote no on <ballot>
abstain on <ballot>
```

You may change your vote as many times as you wish, up until the time a ballot closes. Note, you must manually cast your vote even if you have signed the petition – a vote is not automatically cast for you.

There is an `@petition-options` package for petitions that works along the same lines as `@mail-options`, `@edit-options`, and `@display-options`:

Command	Description
`@petition-options`	Display all your current petition-option settings.
`@petition-option <option>`	Display the current setting for a particular petition option.

`@petition-option petition_order=created`
`@petition-option petition_order=written`
`@petition-option petition_order=vetted`
`@petition-option petition_order=written`
`@petition-option petition_order=stages`
	Controls the order in which petitions are displayed.

`@petition-option signature_order=signing` `@petition-option signature_order=name`	Controls the order in which petition or ballot signatures are listed.
`@petition-option +noannounce` `@petition-option –noannounce`	Controls whether or not you will see an announcement when you log in of open ballots on which you have not yet voted.
`@petition-option +no_announce_ARB` `@petition-option -no_announce_ARB`	Controls whether or not you will be notified when you log in of ARB ballots on which you have not yet voted.

`@petition-option subscribe=query`
`@petition-option subscribe=always`
`@petition-option subscribe=never`
	Controls whether you will always or never be subscribed to a petition's mailing list when you sign it, or asked if you wish to subscribe.

`@petition-option subscribe_ARB=query`
`@petition-option subscribe_ARB=always`
`@petition-option subscribe_ARB=never`
	Same as above, except for ARB nominating petitions.

The LambdaMOO mailing list `*committee` receives automated notifications of new petitions and changes to the status of existing petitions and ballots.

Creating Petitions

This section describes the sequence of steps for creating a petition and getting it promoted to a ballot.

The first step is to make a petition object using the `@make-petition` command. The syntax is similar to `@create`:

```
@make-petition <name>,<alias>,<alias>, ... , <alias>
```

This will create a petition object, owned by Petitioner (#4). The petition and its associated mailing list do not come out of your quota. You may `@rename` your petition at any time without losing signatures. (Note, this is not the same as giving your petition a new title (see below), which *does* erase all signatures.) I recommend giving your petition a short name (even an abbreviation) because the is the name by which people will reference it. You will have a chance to put a longer line of descriptive text in the title.

You may only have one petition at a time.

At any time up until the petition is promoted to a ballot, you may type `burn <petition>` and it will be recycled.

Next, give your petition a descriptive title. Use the retitle command even when you are giving your petition a title for the first time:

```
retitle <petition> as <descriptive title>
```

The first time you give your petition a title, it won't have any signatures on it. You can retitle it later, if you choose to, but all signatures on it will be erased.

Next, edit the text of your petition by typing `@notedit <petition>`. Editing the text will also erase all signatures. Many petition authors tend to sign their petition at this point and start collecting feedback *and* signatures (no one may sign a petition until its author has signed it). I recommend against this, because then every time someone makes a good suggestion and you change the petition, all the signatures are lost and you have to collect them all over again. One or two rounds of

this isn't bad, but more than that and people tend to lose enthusiasm. Instead, I recommend that you type:

```
post <petition>
```

This will cause it to be listed whenever someone invokes the @petitions command, even though you haven't signed it, yet. You can type unpost <petition> at any time if you change your mind.

After you have gotten as much feedback as you think you're going to at this early stage, and incorporated as many changes into the petition's text as you choose to, then is the time to sign it and start asking others to sign it, too. Once a petition's author has signed it, the clock starts: You must get at least 10 signatures in order to submit it for vetting, and you have fourteen days in which to do this. If you can't get ten signatures (including your own) within that time, the petition will expire and automatically be recycled.

Once you have ten signatures, you may then submit the petition for vetting by typing:

```
submit <petition>
```

A mail message indicating that you have requested vetting will automatically be sent to the petition mailing list and to *wizards. At this point, the clock stops: Petitions do not expire while awaiting vetting. Neither can they gather more signatures.

Once the petition is vetted, the clock starts again. The next thing to do is gather the remaining signatures needed to promote your petition to a ballot. The number usually hovers around 50. You can find out exactly how many more signatures it needs by looking at it. It's important to remember that not everyone is open to being lobbied to read and sign pending legislation. Before approaching someone, you should type @refusals for <player> and not approach em if e is refusing politics. If you are using a program to lobby people, it should include a call to:

```
$code_utils:verb_or_property(<player>, "apolitical");
```

and not bother em if you get a truth value. You have 90 days to obtain the requisite number of signatures to promote the petition to ballot status, or else it will expire and automatically be recycled. (As with the number of signatures, you can see how much time remains on the petition by looking at it.)

Once your petition is promoted to ballot status, you are then free to create a new petition.

In addition to the @petition-options outlined above, there are a couple of options that petition authors can set on petitions themselves:

@options on <petition>	Display current option settings.
@set-option notify on <petition> @unset-option notify on <petition>	If this is set, you will be notified whenever someone signs or unsigns the petition.

`@set-option autosubmit on <petition>` `@unset-option autosubmit on <petition>`	If this option is set, the petition will automatically be submitted for vetting when it acquires 10 signatures.

Nomination Petitions

Elections are announced on `*news`. To nominate someone, type:

```
@nominate <player> for <office>
```

The possible offices are ARB, Reaper, or Registrar.

Nomination petitions do not require vetting. They are promoted to ballot status automatically when they get 50 signatures, but only if the nominee emself has signed it, indicating that e accepts the nomination.

Keeping Up with it All

The mailing list `*committee` provides a daily update that includes (as applicable) new petitions, burned or expired petitions, changes to a petition's status (e.g. submitted for vetting), promotions to ballot, and signature changes.

Glossary of Terms

@ – By convention, the first character of a command that transcends the MOO's virtual reality. One of many descriptions of a MOO is "a text-based virtual reality". MOOs have themes, or general motifs, such as "a large mansion and its grounds". Within this text-based virtual reality, players can talk with one another, gesture (emote), move from room to room, pick up and drop various objects, and so on. These are sometimes referred to as *VR actions*. There are also several commands that one can issue that break with or transcend the virtual reality: `@who` lists all the players who are logged on, for example, and `@join` moves you to someone else's location on the MOO. I call these commands *meta-VR*.

– Every object on a MOO has a unique number, which is indicated by the "`#`" sign. There is much you can do without paying attention to object numbers, but an object can always be referred to by its number. If you are not holding an object or in the same room with it, then (with a few exceptions) you *must* refer to it by number for the system to figure out which object you mean.

ARB – The Architecture Review Board. In December, 1991 the LambdaMOO wizards created the Architecture Review Board to assist them in assessing who should receive more quota to build with. In July, 1993, a ballot passed that made the ARB an elected body.

alias – Every valid object in the MOO has a name. Objects may have additional aliases, which are other names (often shorter) that can also be used to refer to an object. An object named "`a big, black, hairy spider`" might have the alias "`spider`", for example. One way to see an object's aliases is to `examine` it. You can add an alias with the `@addalias` command; you can remove one with the `@rmalias` command. See also `help @rename`.

arguments – arguments are pieces of information that are provided to a command, program, verb or subroutine so that it can do its job. If you look at the room you are in by typing the word `look` by itself, we say that you have invoked the `look` verb with no arguments. If you type `look hat`, then the word "hat" is an argument to the `look` command. If you type `look rabbit in hat`, then the words "`rabbit`", "`in`" and "`hat`" are arguments to the `look` command. In such a case, "`rabbit`" is the direct object, "`in`" is the preposition, and "`hat`" is the indirect object. Some commands always take the same fixed number of arguments. For example, the command `@go` always takes one argument, the room to which you wish to travel. Other commands can take an arbitrary number of arguments. For example, `go` needs at least one argument, but is able to take several. `go north` will move you from your current location through the exit named "`north`", if there is one. If you are in the Living Room on LambdaMOO, typing `go north east east up east north` will move you through successive exits until you arrive at the library.

argument specifiers – When a programmer first creates a verb on an object, e must, as part of the command that creates the verb, specify what arguments the verb takes if any. This is done by typing a word that stands for the direct object of the

command, a word that stands for a preposition, and a word that stands for the indirect object. If I am creating a trophy, and want to write a verb to award it to someone, I might type the following:

```
@verb trophy:award this to any
```

"`Award`" is the name of the verb itself. "`This`" means that when someone types the award command, the trophy ("`this`") will be in the position of the direct object. "`To`" is the preposition, signifying that the trophy will be awarded *to* someone, not from or about em, for example. Last, "`any`" indicates that the trophy can be awarded to anyone or anything. The words "`this to any`" are the argument specifiers.

`background task` – A foreground task is a task that executes "while you wait". There are some things that the computer does independent of a person typing in a command and waiting to see the result. Suppose I have a MOO timer, and I type the command `set timer for 5 minutes`. The computer might print to my screen, "`Timer started`" and that exchange represents a completed foreground task. I proceed to chat with my friends on the MOO. Each invocation of `say` and `emote` also represents a (short) foreground task. Meanwhile, the timer program is counting off the seconds, up to five minutes, without tying up my screen. That is, I don't have to wait the five minutes before I can type something else. The counting off of the seconds is said to be accomplished "in the background", or is described as a *background task*. When the five minutes are up, the timer prints the line, `DING! 5 minutes are up!` to my screen, and the background task is ended. (See also **task**.)

`bash` – An offline party or other gathering of MOOers. Sometimes bashes are given a qualifying prefix. "NYE-bash" would be a New Year's Eve bash. "Sushi-bash" would suggest some MOOers going out for sushi together. Usually the term "bash" implies that any MOOer who is told the time and place is welcome to attend.

`'bot` – 'Bot is short for "robot", and it typically refers to something that acts like a player but isn't a player. There is a further refinement, which is whether the bot is an actual *player object* or not. Some people have written programs that log on to LambdaMOO (using a player name and password). These programs maneuver the player-object through the MOO, and are programmed to recognize and react to conversation and perhaps other text generated by human typists. Another kind of 'bot is a non-player object within LambdaMOO that is controlled to a greater or lesser extent by a human typist but which is not in fact a player object. These are sometimes also called *puppets*. This kind of 'bot is easier to identify, because if you examine it, you'll see that it is not its own owner. Real players own their player objects; puppets and other automata do not.

`built-in function` – A program is a collection of commands which are executed in a particular order. These commands can either be other programs (often called "subroutines"), or any of a subset of commands that are intrinsic to the system (in other words, commands that are provided by the server rather than written in MOO code). Some examples of built-in functions:

```
length(<item>)
players()
connected_players()
```

byte – A byte is a unit of computer storage space. Typically, one letter of the alphabet takes up one byte of space.

call – Suppose you are making chocolate soufflé. Recipes have several steps, and sometimes refer to other recipes, e.g., "Make a béchamel sauce (see page 257)." Similarly, commands (verbs) can – and usually do – consist of several steps, often referring to other verbs. These other verbs are sometimes called *subroutines*, and when a verb asks the computer to execute one, we say that the verb *calls* the subroutine.

channel – Many MOOs have a special communications facility called channels. A fair analogy might be to compare channels to Citizens' Band radio. One connects to a particular channel, and can then listen to everyone who is talking on the channel and can transmit on the channel without being co-present in the same room(s) as others who are also connected to the channel.

character – A *character* is another name for a *player*, which is an object that represents a typist within the context of the MOO. Sometimes the two terms are combined: *player-character*. The distinguishing feature of a player-character object, unlike every other kind of object, is that the built-in system function `is_player(<object>)` returns 1.

child – With the exception of the root object (`#1`), every valid object has all the same properties and verbs as another object, said to be its parent. An object is said to be a *child* of that other object. An object can have many children but only one parent..

class – When an object is intended solely to serve as the parent of other objects rather than being used itself, it is referred to as a *generic* or as a *class*. Things that have that object as a parent or ancestor are said to be of that object's class.

command – A command is a word, sometimes with accompanying arguments, that a typist enters with the intention of obtaining some result or causing some effect. All commands are verbs, but not all verbs are commands. This is because some verbs are only meant to be called from within other verbs and not directly by a typist. Examples of commands are:

```
look me
@who
put rabbit in hat
```

command line – When you type a command, you are said to type it *at the command line*. Some verbs (e.g. `say`, `emote`, `page`, `@join`) are intended to be invoked as commands, and these are sometimes called *command line verbs.* Other verbs are meant to be called only from within other verbs. These are called *subroutines.*

connected – A room is said to be *connected* if you can get to it without having to teleport.

contents – Every valid object in a MOO has a property that designates its location (given in terms of an object number), and another property that designates its contents: things whose location is in turn the object in question. If A contains B, then B appears in A's `.contents` property, and the value of B's `.location` property is A. A refinement to this concept is that the notion of containment extends to

contents of contents. Think of nested boxes. If A contains B and B contains C, then even though C doesn't appear directly in A's `.contents` property, C is said to be in A's *containment hierarchy*. Objects cannot contain themselves, nor can two objects contain each other simultaneously or otherwise violate the containment hierarchy. (The MOO has no formal way to designate sizes of things, so a tiny little jewel box can easily contain a hippopotamus!)

`core` – Also referred to as the *core database*. A brand new MOO that is just up and running consists of two major pieces, the server and the core database. The server is the actual program that runs on the host computer. The core database is that set of objects within the MOO that every MOO starts with. The core database includes several objects that are sets of utility programs, written in the MOO programming language, which are used in making more complex objects and MOO programs. The core also includes #2 (the player that is the ArchWizard), the system object, the generic room, the generic exit, and various other items. The core includes a verb, `#0:core_objects`, that defines the list of objects that comprise the core database. This book confines itself to MOOs based on LambdaCore. Other available cores include JHCore and EnCore.

`data types` – The MOO programming language recognizes six different kinds or *types* of data: integers, decimal numbers (also called *floating point numbers*), character strings, objects, error codes, and lists.

`database` – When used in the context of a MOO, the *database* refers to the collection of *all* the existing objects along with their associated properties and verbs. The server loads the database to run the MOO.

`defined` – Every verb is associated with an object, either a player, a player class, a place (room), or a concrete object. It is associated with this object when it is first programmed. (In the case of rooms or players or articles, the verb usually manipulates that object in some way.) We say that a verb is *defined* on the object to which it was attached at the time it was programmed. We care about which object a verb is defined on when we want to list it to see how it works, or perhaps fix a bug (if the verb is defined on an object that we own).

`descended from` – An object is said to be descended from another object if the other object is in its chain of parents.

`expression` – An expression is a combination of characters that, when evaluated as a piece of MOO code, generates a value.

`exit` – An exit is a special kind of object that doesn't exist in any particular place per se (its actual location is usually `#-1 ($nothing)`, but that is associated with a room, its *source*. When you are in a room, there are what are called *obvious exits*. When you type the name of an obvious exit, you are transported to that exit's destination, and that is referred to as *invoking* an exit.

`fertile` – Every valid object on LambdaMOO has a property that indicates which object is its parent. An object's initial parent is specified when it is created. You may create a child of an object you don't own if and only if the (potential) parent object is fertile. For an object to become fertile, its owner must make it fertile using the `@chmod` command.

flag – A flag is a kind of variable whose value is either TRUE or FALSE.

foreground task – A task that executes "while you wait", typically the result of typing in a command. See also **background task**.

forked task – This is another name for **background task**.

fork bomb – A fork bomb is a program that generates more and more forked tasks until the system is overloaded. The system becomes terribly lagged, then grinds to a halt. A fork bomb is usually viewed by a MOO's wizards as a "denial of service attack" and its perpetrator, when found, may have eir programming privileges revoked. On rare occasions, a new programmer will generate a huge number of forked tasks inadvertently. If the wizards believe this to be the case, there is usually no punishment beyond disabling the offending verb and asking the programmer to be more careful in the future.

gag – To gag someone is to indicate to the system that you do not wish to see any text that results from the gagged player's typing anything. Other people will still see the gagged player's output. See `help @gag`.

game master – In the LambdaMOO RPG, one who has programming privileges. (See also **Grand_Master**.)

generic –An object that exists solely to be a parent of other objects.

Grand_Master – A character on LambdaMOO who owns all RPG objects and has access to statistics otherwise unreadable to other players (except wizards). One or more typists may have and use this character's password. Grand_Master is used only for RPG administrative purposes and occasional debugging. (See also **game master**.)

gurst – In LambdaMOO parlance, a gurst is a guest whose typist has a registered player-character. Some players log on as guests in order to circumvent noise abatement measures. Some do so to rediscover the joys of truly anonymous MOOing or simply for a change of pace. Some do so because they have for some reason temporarily or permanently lost the ability to connect as their regular player-character. Opinions differ on whether unruly behavior or an intent to deceive are necessary in order for a guest to be considered a gurst.

idle – To remain connected without interacting. It is used as both an adjective and a verb: AcidHorse is idle. AcidHorse is idling. It is not unusual for a person to emote `:idles` before paying attention to another window or leaving eir keyboard while still connected to the MOO.

input – information that goes into something, usually a computer program. A program that sorts numbers, for example, might display the prompt, `Enter some numbers to sort:` The numbers that the user types in would be the input. (The sorted list of numbers would be the program's output.)

invalid – MOO objects are said to be either valid or invalid. An object is valid if someone has created it or recreated it (with the `@create` or `@recreate` commands respectively). An object is invalid if it has been recycled, or if it has never been created in the first place. (E.g. object #99999999999999999999 is not a valid object (as of this writing).) There are a few objects that exist but are not valid. These usually

have negative object numbers (e.g. #-3), and are used by the system to designate various error conditions.

`inventory` – Those objects that a player is holding or carrying.

`invoke` – To cause a command to be executed by typing its name. If there is an exit in your vicinity named "north" for example, you are said to invoke the exit when you type `north`.

`lag` – A (usually unexplained) delay in the system's response time. Normally when you type something in, especially if you're simply saying or emoting something, the associated text prints out on your screen right away, and if you're in a room full of active players, text appears at a fairly steady rate. There are a variety of reasons why someone else's text might be delayed in its appearance: e might have been called away from the keyboard unexpectedly, e might be thinking about eir response, or typing in a long line, or multi-tasking, or it might just be because of lag. A sure sign of a lag storm is when your *own* text is delayed in appearing. Sometimes lag isn't on the MOO itself, but is in the network somewhere between the MOO and a typist's computer. This is usually referred to as *net lag*. It is characterized by some MOOers experiencing lag and others not.

`LambdaCore` – A core database derived from LambdaMOO. (See also **core**.)

`$limbo` – The location to which a player is returned when e logs off if any of the following apply: eir home is `$player_start`, eir home won't accept em as a resident, or eir home is invalid.

`list` – A list is a particular way of representing a set of things in the MOO programming language. The elements of a list may be numbers, strings of characters, objects, other lists, or any combination. Lists are designated using curly braces "{}", and their elements are separated by commas. Example: `{"This", "is", "a", "list", "of", "strings", "."}` The empty list is a meaningful construct in the MOO programming language, and is designated with just the curly braces: `{}`.

`lurk` – To read a mailing list without posting, to stay in a room without saying, emoting or otherwise contributing anything, or to listen to a channel without speaking on it.

`matching` – Matching refers to the system associating a name you type in with an object's unique number. If you page a particular player (i.e. `page mockturtle Don't you ever sleep?`), the code for the "page" command matches `mockturtle` to mockturtle's object number, and forwards the message accordingly. If you are holding a rock and type `drop rock`, the system will match the word "rock" with the rock you are holding and move the rock from you to the room you are in. If you are holding more than one rock, the system will be unable to match the word "rock" to a unique object, and you would see, `I don't know which "rock" you mean.` If you weren't holding a rock at all, The system would display, `I see no "rock" here.` (The first case is referred to as an *ambiguous match* and the second is a *failed match*.) In general, in order for the system to match a name that you type with a unique object, you either have to be holding the object or in the same room as the object. Exceptions to this include many of the commands that refer to players (e.g.

page and @join) since player names are unique, and also the @go command, which can look for the name of a room in a player's .rooms database.

mav – According to the FAQ found at http://www.mudconnect.com/mudfaq/mudfaq-p1.html#q30, Mav was a TinyMUDder who would sometimes accidentally emote something to an entire room when he meant to whisper or page it. The word has come to mean any mix-up between say, emote, remote-emote, whisper, page, etc.

meta-VR – Actions or commands that "break" the virtual reality and call attention to the fact that you are using a computer program and not a real (i.e. tangible) mansion, castle, cave system, what-have-you. By convention, most non-VR commands on MOOs begin with the "@" sign, although there are exceptions to this. Some examples of non-VR commands would be @who, @join <person>, @send <person-or-mailing-list>.

morph – A morph is an alternate presentation that a player-character may adopt, while temporarily storing eir original name, gender, and description. An alter ego. Did you ever notice that you never see Super Man and Clark Kent together at the same time? They might be morphs of the same person. While a player's name, description, gender, and messages may change when e morphs, eir object number (and password) remain the same.

multiple characters – Many people choose to have more than one character on a MOO. Multiple characters are different from morphs in that there are two (or more) separate player objects with different object numbers; it is more difficult for players to determine that multiple characters are controlled by the same typist than it is with multiple morphs. LambdaMOO permits multiple characters, but they must be registered as such. Among other things, only one of a typist's multiple characters may participate in the political system there.

multi-tasking – This term refers to a computer that is executing more than one task at the same time. Informally, a person is said to be multi-tasking if eir attention is divided between two more simultaneous activities.

newt – The act of metaphorically turning a player into a newt or, a player who has had this done to em. The @newt command, available to wizards only, blocks a player's access to the system, either for a specified or indefinite period of time. This action is typically taken when the wizards believe a player might be a threat to the system. It is occasionally done for punitive reasons or for noise abatement. It is possible for a player to effectively newt emself, using the boot_player() built-in function within a custom :confunc verb. (See also **toad**.)

non-VR – See **meta-VR**.

object – Objects are the fundamental building blocks of an object-oriented system. On a MOO, every object has a unique number (prefixed by the "#" sign), a name, an owner, a location, and a property listing its contents.

options package – An options package is a set of commands and data values that enables you to customize some aspect of your MOOing experience. There are many options packages that govern how different things work for you, specifically. Examples of these include mail-options, to customize various aspects of sending

and receiving mail, `edit-options`, which customize certain aspects of how the editors work, `builder-options`, and `programmer-options`. Options packages can be associated with player classes, generic rooms, feature objects, or any other kind of object. There is no single definitive way to list all the options packages available to you, but they are usually referenced in other objects' help texts. `Help options` will give you a list of some of them.

`output` – The result of (usually) a computer program, which is displayed to one or more users. Sometimes one refers to a person's output, meaning that which the person produces, either manually or with the aid of a program.

`parameters` – Limits, usually numeric, that are set in advance. One speaks of "working within a set of parameters." Parameters are not the same as arguments: If one had a program to sort numbers, the arguments would be the numbers to sort. A parameter might be the maximum number of arguments the program could or would accept.

`parent` – See **child**.

`parser` – When you type a line of text, the system has to figure out which segment of what you typed constitutes the command, and identify the verb to run, the direct object, the preposition, and indirect object, if present. The part of the system that does this is called the parser.

`player, player-character, player object` – An object on a MOO that represents the VR embodiment of a human typist.

`player class` – An object on a MOO that serves as a repository for additional commands that a player might choose to use. A player adopts a player class by changing eir parent to the player class. Implicit in this act is adopting all ancestors of the selected player class as well. There is a calculated risk in adopting a player class which is that player class owners could theoretically intercept private communications, and are able to change some of a player's fundamental attributes. Such incidents are rare, but one should know the risks going in. (See the section on player classes beginning on page 43 for more about this.)

`$player_start` – The location where guests, new players, and players without an otherwise-valid home find themselves when they log on.

`port` – (Think "transport".) To record all the particulars of an object on one MOO and use the information to recreate it as exactly as possible on another MOO. One should ask permission of an object's author before porting it.

`primary character` – On some MOOs, human typists are permitted to have more than one player character. On LambdaMOO, only one of these has the rights of citizenship (authoring and signing petitions, voting, etc.). The one with the voting rights is referred to as one's primary character. Others are referred to as *secondary characters*. MOOs that support multiple characters sometimes have a registry to differentiate primary and secondary characters, which only wizards can access. By convention, it is the prerogative of the typist and no one else to disclose secondary character information. If someone shares such information, it behooves one to treat it as privileged.

program – A sequenced set of instructions that a computer follows slavishly. Programs can be very simple or quite complex or anything in between. On a MOO, the terms *program* and *verb* are frequently used interchangeably. (One programs a verb, but does not verb a program, however.)

programmer – A person who programs computers. In a MOO, wizards grant what is called a *programmer bit* which changes a player's `.programmer` property from 0 to 1. The system then recognizes the player as empowered to write MOO programs. When contrasted with the term *user*, this term more specifically refers to the person who wrote the program that the user is using.

property – A property is a named piece of data associated with an object. Within the system, properties are designated by the object number followed by a period (`.`) followed by the property's name. When speaking of a property independent of the object it might be associated with, the object number is sometimes omitted. For example, "Every valid object has the following properties: `.name`, `.location`, `.contents`, `.owner`." Players may change the values of many properties on themselves and on objects they own. Programmers may add new properties to objects they own. Properties are used to store data that are needed or wanted after a verb has finished executing, or for data that are needed by more than one verb. (See also **variable**.)

puppet – An object that may impersonate a player, whose "hearing" may be monitored by a player and whose responses may be controlled by a player, but which is not in fact a player. Butlers and bartenders at various venues are likely to be puppets.

queue – A queue is a list of tasks that are scheduled to run at a later time.

reap – To expunge a player-character from a MOO, typically because e has been inactive for a long time.

reaper – Traditionally, only MOO wizards have the power to reap a player. On LambdaMOO, certain non-wizard players are entrusted with this task. LambdaMOO reapers are elected.

response latency – The delay between a real time communication to a player and that player's response. Reasons for a lengthy response latency might be thinking before answering, composing and typing in a long response, being away from the keyboard, or system **lag** (which see).

return value – When a verb is called or an expression is evaluated, it always returns a value. The *return value* of the expression `2 + 2` is `4`, to give a simple example. The return value is not always the entirety of the result – sometimes a verb may also have a *side effect*. In many cases, the return value is either 1 or 0, signifying, "The operation was successful," or, "The operation was not successful," respectively. The operation in question is whatever the verb is supposed to do.

room – An object that is descended from `$room`. More loosely, an object that players can enter and which looks, sounds, smells, and feels like a room (i.e. it might as well be a room).

.rooms database – A property on a player that is a list of object numbers and names or abbreviations for rooms. One's `.rooms` database is used in conjunction

with the @go command, so that you can type @go library, for example, without having to remember the library's object number. You can type @rooms to see your own .rooms database. You can modify your .rooms database with the @addroom and @rmroom commands.

RPG – Role Playing Game. Some MOOs have within them a role playing game akin to the early adventure/dungeon games from which social MOOs arose. On LambdaMOO, this game is referred to as an "area", though various sets of rooms might be tightly or loosely connected. Briefly, one becomes "initiated", a process by which one acquires a surrogate (called a "doll" or "voodoo doll") on which are recorded one's victories and defeats over various and sundry RPG opponents. You train to increase skill and then venture forth to seek treasure, fight various monsters, and so forth.

secondary character – Some MOOs permit a typist to have more than one player character. Usually the first or oldest of these is designated as one's primary character, and others are referred to as secondary characters. On LambdaMOO it matters especially because only primary characters have the rights of citizenship (authoring petitions, voting, etc.).

server – The program, written in the C programming language (as it happens) that, when running, is the MOO. This program accepts connections from players logging on, reads what players type in, and responds accordingly. When you type something in, part of the program that's running (the parser) figures out what command you've typed and which object(s) you're trying to manipulate, and then causes the appropriate function or verb to be executed.

shouting – There are two usages of the term "shouting". One is to say or emote something in all capital letters. The other is to broadcast something to everyone who is logged on, even though they aren't in the same room. An appropriate use of the latter would be for a wizard to shout that e is about to reboot the system for some reason.

side effect – Some verbs just return a value: an arithmetic calculation, the name or location of an object, etc. But other verbs do things in addition to returning a value, such as announcing text to a room, or changing something in the database. These additional things are called *side effects.*

spam – Copious amounts of unwanted text whose volume is so great it renders its content useless or pointless.

spoof – To cause unattributed text to appear on other people's screens, or the unattributed text itself. There are three general forms. In one, no player's name is included: "The chandelier falls to the floor with a crash!" In another, the name of the player perpetrating the spoof does appear in the text, but not at the beginning, and another player's name might appear at the beginning instead. The classic form of this is, "Werebull causes Yib to fall down laughing," (with Yib causing this text to appear, not Werebull). Some players vehemently object to this form of spoof; others take it in stride. It is in fairly common use. The third form is particularly offensive and considered officially unmannerly on most MOOs, and this is text that depicts a player doing or saying something, which text the depicted player did not emself type in: "Yib produces a previously unseen puke green wiffle bat and proceeds to bash

herself several times over the head with it." (Where some unnamed stinker caused this text to appear and Yib didn't.) There is no programmatic way to prevent players from spoofing, but there are a few different ways to detect it, including tell-filters and the combined commands `@paranoid` and `@check-full`.

`string` – A sequence of letters, numerals, or punctuation marks or any combination thereof. When depicted, a string is enclosed between double quotation mark characters, e.g., `"chocolate souffle"`.

`subroutine` – Some verbs are called not from the command line but from within other verbs. These are called subroutines. Suppose you wanted to make a chocolate soufflé. The recipe might begin, "Make a béchamel sauce (see page 257)." You would turn to page 257, follow the directions for making a béchamel sauce, then return to your place in the soufflé recipe. Executing a subroutine is like making the béchamel sauce.

`syntax` – A generalized expression of the correct usage of a command or subroutine.

`system` – This is a general term that is used to refer to a program that is running or a set of programs working together. It can mean the MOO itself, as in, "What did the system respond when you typed `@parents me`?" or it can refer to the operating system on the machine on which the MOO is running, as in, "The system will be shut down in two hours for its ritual Saturday night bath."

`system character` – A player object that does not actually have a human typist associated with it. System characters are typically used to serve as the owner of record of objects associated with one or another project. On LambdaMOO, for example, the system character "Petitioner" owns all petitions, ballots, and related objects.

`task` – Many players can use a MOO at once. The system receives text that a player types, processes it in some way, and then (usually) prints text to the player's screen in response. Whenever a player types in a command and the system executes it, that is called a task, and more specifically, it is called a *foreground task*. Other tasks run "in the background", which is to say that the player who initiated this task is free to type in another command (thereby starting another task) before the *background task* is finished. Many objects with "delayed reaction" behavior utilize background tasks. Here's an example. In the LambdaMOO Living Room, there is a fireplace. You can pile logs in the fire place, and then light the fire. While the logs are burning, the fire hisses and crackles and pops, but meanwhile you are free to continue conversing with others present. It is a background task that causes the fireplace noises to appear periodically. Every background task has a unique numerical identification number called its `task_id`.

`teleport` – Moving to a room in a way that is inconsistent with the Virtual Reality, e.g. using the `@go` or `@join` command.

`tell-filter` – Every player has a verb on emself called "`:tell`". (Some non-player objects have `:tell` verbs, too.) This verb receives as input a string of text, and prints that text on the player's screen. A tell filter pre-processes text before it is displayed. For example, it might inspect the text and prepend name of the player who initiated it, maybe in angle brackets. So instead of seeing, "Jack causes Jill to fall down laughing," you might see, "<Jill> Jack causes Jill to fall down laughing."

tick – A tick is a unit of computation. Just as it takes most people less effort to add 2 + 2 than to multiply 13 by 8, different tasks take different amounts of computing power, and these amounts are measured in units called *ticks*. When you type something in, 30,000 ticks are allotted to the task. (This is the default. The actual number may vary from MOO to MOO). Programmers of commands can, if necessary, ask the system to "take a breath" (metaphorically speaking) and then resume with an additional allotment of ticks, though this means that a command will take longer to complete, both in terms of absolute time (seconds) and ticks. Why do we care about ticks? Because each task gets only its allotted number of ticks before the system switches to allow the next task some ticks to compute. The computer can only do so much at once before it starts to get bogged down. When the system bogs down, everyone experiences lag (increased response time). Good programmers try to write code that uses a minimum of ticks without sacrificing clarity for future readers or maintainers.

tiny scenery – Objects (especially rooms) that have descriptions only and are not in any way interactive, or items that are mentioned in a room's description for which there is no corresponding object.

toad – The act of metaphorically turning a player into a toad, or, a player who has had this done to em. The `@toad` command, available to wizards only, removes the flag by which the system recognizes the player-object as a player. This action is a natural part of the reaping process (but does not constitute all of the reaping process). On infrequent occasions it is done for punitive reasons. It is impossible for a player to toad emself. Contrary to popular belief, toading can be undone, as long as the player object has not yet been recycled.

toad scar – When a player is `@toaded`, one of the side effects is that eir object number is removed from the list of players returned by the built-in function `players()`. If a player is reinstated, eir player number is appended to the *end* of the list of players, thus appearing out of numerical sequence, and this appearing out of sequence is what is meant by the phrase *toad scar*. To quote Nosredna, a LambdaMOO wizard, "The difference between toading and newting is that toading leaves a scar and newting doesn't."

troll –Trolls are players who log on and make inflammatory remarks or send inflammatory posts to mailing lists, caring more about riling people up than the actual substance of their utterances. Contrary to what one might think, trolls are not universally reviled. Some players actively enjoy challenging trolls about their alleged views, and believe that both they and the trolls realize that there is a sort of "dance" going on. (N.B. The name comes not from unruly fairies, but rather the act of dragging bait through the water, hoping that fish will bite or chase it.)

typist – The human being who is sitting at the computer keyboard typing. A single typist may have one or more player-characters.

user – A person using a computer program. This term is sometimes contrasted with *programmer*, the person who wrote the program that the user is using.

valid – A *valid* object is one that can be used within the MOO in certain conventional ways. There are certain pieces of information that are attached to every valid object without exception. These pieces of information include the object's

owner (identified by object number), its location (identified by object number), its contents (a list of one or more object numbers or the empty list), and its parent (identified by object number). An object that has a number but doesn't have these pieces of information associated with it is not a valid object, by definition. Invalid objects exist, though, and are used in a number of ways. One of these is `$nothing` (`#-1`), which is where rooms are conventionally located. Other so-called invalid objects signal an error condition, specifically a failed or ambiguous match.

`variable` – A named piece of data that is used within a verb, but which does not exist before the verb runs or after the verb has finished executing. Variables are used to store intermediate results while a verb is in the process of running but which are needed neither at a later time nor by another verb. (Contrast **`property`**, which is used to store a result or state for later re-use.)

`verb` – A verb is a named, ordered sequence of commands that the server can interpret and execute. Whenever you type a command, for example `@who` or `put rock in box`, you are asking the computer to do something. You are using a verb. Some verbs are not intended to be used directly by someone typing at a terminal, but are intended to be called by other verbs. These verbs generally either produce some side effect – such as changing some data somewhere, or return some intermediate result to the verb that called it (such as 3, or an object number) – or both. The beauty of a MOO is that ordinary players can create new objects and write new verbs on them, thus extending the richness and variety of the environment.

`VR` – Virtual Reality. In particular, *VR* refers to actions that conform to the virtual reality of the MOO you are using. Examples would be saying things to people in the room with you, "walking" (i.e. using conventional exits and modes of transportation) as opposed to teleporting, etc. (See also **`meta-VR`**.)

`wheel` – An influential person. As in, "big wheel".

`wizard` – A wizard is a player on a MOO with special powers not available to ordinary players, among them the power to create new players, `@newt` and `@toad` existing players, read otherwise-unreadable properties on any object, read otherwise-unreadable verbs on any object, read any message on any mail recipient (including players' private MOOmail, though wizards generally do not exercise this power), view all background tasks, and kill any background task. Wizards are usually hand-picked by the person who owns or is responsible for the system as a whole (this person is referred to as the ArchWizard). In general, one cannot become a wizard simply by reaching a certain specified level of proficiency, although proficiency is usually one of the criteria for selecting wizards, along with trustworthiness. Wizards are expected to use their powers with discretion. If you do not trust the wizards on a particular MOO to do so, you should not participate on it.

Conversational Typing Abbreviations

addy – address
afaik – as far as I know
afk – away from keyboard

atm – at the moment
bbl – be back later
bcnu – Be seeing you.
bf – boyfriend
brb – be right back
btw – by the way
f2f – face to face
fdl – falls down laughing
filfre – feel free (to do something or other)
gf – girlfriend
ianal – I am not a lawyer
iirc – if I recall correctly
imho – in my humble opinion
imnsho – in my not so humble opinion
imo – in my opinion
istr– I seem to recall
j/k – just kidding
l8r – later
lol – laughs out loud
ltns – long time no see
oic – Oh, I see.
otoh – on the other hand
pov – point of view
ppl – people
qooc – quoted out of context
rotfl – rolling on the floor laughing
rtfm – read the manual
R U M or F? – Are you male or female? (This phrase is now eschewed by experienced players except to make fun of inexperienced players.)
stfu – shut the fuck up
tmi – too much information
ttfn – Ta ta for now
ttyl – Talk to you later
wrt – with regard to
wrte – we regret the error
ymmv – your mileage may vary

Addendum to the Glossary

The glossary entries for `call` and `subroutine` draw an analogy between a program calling a subroutine and a cookbook referring to one recipe from within another recipe.

The following two recipes are provided as an adjunct to those entries, *just in case* someone actually decided to check my cross-references.

Béchamel Sauce

2 Tablespoons butter
2 Tablespoons flour
1 Cup milk, heated
Salt
Freshly ground pepper

Melt the butter in a small shallow pan. Stir in the flour and cook, stirring constantly, until it bubbles a bit, but don't let it turn brown. 2-3 minutes.

Add the milk a little bit at a time, stirring to incorporate each addition completely before adding more. You should have a smooth paste. Bring just to a boil, add salt and pepper to taste, then lower the heat and simmer for 2-3 minutes more. Remove from heat.

This can be stored for later use. After it has cooled somewhat, place a piece of plastic wrap directly on the surface to prevent a skin from forming.

Chocolate Soufflé

Contrary to popular belief, a soufflé is not difficult to make, though it takes a while and must be served immediately. Most of the work can be done ahead of time.

2 1/2 ounces unsweetened chocolate
5 Tablespoons sugar
2 Tablespoons butter
2 Tablespoons flour
1/8 teaspoon salt
3/4 Cup whole milk
3 eggs, separated
1 teaspoon vanilla
1 pint vanilla ice cream (for the sauce)

Preheat the oven to 325° F. Butter a 1 1/2 quart soufflé dish and dust with granulated sugar. Set aside.

Put the chocolate, 2 Tablespoons of the sugar and 2 Tablespoons of hot water in a small pan and heat slowly, stirring occasionally, until the chocolate is melted and smooth. Remove from the heat and set aside.

Follow the procedure for béchamel sauce (see page 257) using 2 tablespoons butter, 2 tablespoons flour and 3/4 cup milk, but salting only lightly and leaving out the pepper. Blend in the chocolate mixture.

Beat the egg yolks well. Stir a little of the hot sauce into the yolks, then add the yolks to the remaining sauce. Stir well, then set aside to cool.

This much can be done in advance.

With clean beaters, beat the egg whites until foamy, then slowly add the remaining 3 tablespoons of sugar, and continue beating until stiff but not dry. Stir about 1/4 of the whites into the chocolate mixture, then fold in the remainder. Stir in the vanilla.

Pour all into the soufflé dish, sprinkle the top with sugar, and bake for 35 minutes. Meanwhile, set out the ice cream to melt at room temperature.

Serve immediately with a “cold vanilla sauce” made from the melted ice cream.

Appendix A – Summary of Commands

This appendix details various commands that are available on MOOs. It includes commands available on the player classes and feature objects provided with LambdaCore, plus others such as those that are available on $room.

A note about the syntax specifications: Text enclosed in angle brackets "<>" must be specified at the time it is typed in, e.g. instead of "<name>" you must supply an actual name, without the angle brackets. Text enclosed in square brackets "[]" is optional to a command. If you include it, don't type the square brackets. A vertical bar "|" separates either-or cases. For example, @notedit <object | object>.<property> means you must either supply an object, or an object and a property name separated by a period (notice that the period isn't in the angle brackets). Verb names with an asterisk in them (*) may be abbreviated to the part that comes before the asterisk.

The form of each entry is as follows

<command> <arguments>

<location/source>

<Usage notes>

Commands are listed alphabetically, ignoring punctuation. Object numbers for command locations will vary from MOO to MOO, so I only give them for commands that are LambdaMOO-specific. "LambdaMOO" is abbreviated as "LM".

@abort
Embedded with certain $command_utils verbs.

When prompted for text input or a yes-or-no answer, you can type @abort to exit a task entirely.

@abort [<object>[:<verb>]]
LM player class #7069

Causes queued tasks to be aborted. You may specify either all tasks associated with a particular object, or only tasks associated with a particular verb on an object.

@addalias <alias>[,...,<alias>] to <object>
@addalias <alias>[,...,<alias>] to <object>:<verb-name>
@addalias# <alias>[,...,<alias>] to <object>:<verb-number>
$player

The first form adds one or more aliases to an existing object. Note, player aliases may not have spaces in them.) The second form adds one or more aliases to an existing verb on an object. The third form unambiguously adds one or more aliases to a particular verb on an object when there are two or more verbs with the same name. (See also @verbs and @list#.)

@adddict <one or more words>
$frand_class
Adds a word or words to $spell, if present. (Many MOOs have recycled $spell because it is cumbersomely large.) Only wizards and players in $spell.trusted may use this verb.

@addfeature <feature object>
@add-feature <feature object>
$player
Adds a feature object to your list of features. See also @rmfeature. A feature is an object that provides additional commands that you can use.

@addlag
LM feature object #26787
Turns off the lag reduction FO and any features similar to it. (Re-)enables @gag, @check-full, and any other commands that utilize a player's :tell verb to filter and/or otherwise pre-process text before it is displayed to your screen.

@add-notify me to <player>
@add-notify <requestor> to me
$mail_recipient_class
This verb, used cooperatively between two players, allows one player to be notified when the other player receives MOOmail. The first form sends a MOOmail message to <player> indicating that one wishes to receive notification. The second form adds the requestor to a person's .mail_notify property. This might be useful if several players shared a group character (such as LambdaMOO's Grand_Master, for example) and wanted to be notified if the group character received mail, so as to be able to respond to it in a timely manner.

@add-owned <object>
$builder
Adds an object to your .owned_objects property in the highly unlikely event that it wasn't added upon creation of the object.

@addr*oom [<name>] [<place>]
@addr*oom [<place>] [<name>]
$frand_class
Adds a room to the list of rooms you know "by name". See also @rooms. If <name> is not specified, then the room is remembered by its actual name (as opposed to a nickname you provide). If <place> is not specified, then the current room is remembered.

@addword <word or words>
$frand_class
Add a word or words to your personal dictionary, if one is kept. (This may be disabled on some MOOs that do not use $spell.)

@age [<player>]
$player

Tells a person's MOO age, i.e. how long it has been since e first connected. On LambdaMOO, the difference between a person's actual MOO age and eir official MOO age is because system down time doesn't count for aging of people. This arose from a ballot to "stop the clock" on legislative issues (including the determination of voting age, etc.).

@answer [<message-number>] [on <mail-recipient>] [sender | all | followup] [include | noinclude]
$mail_recipient_class

See @reply.

@arb [all]
LM player class #322

Displays a list of the connected members of the LambdaMOO Architecture Review Board. If `all` is specified, then displays all members, whether connected or not.

@arb-ballots
LM player class #322

Lists open ballots for the LambdaMOO Architecture Review Board.

@arb-nominate <player>
LM player class #322

Obsolete. See `@nominate`.

@arb-petitions [all]
LM player class #322

Displays a list of all petitions nominating candidates to the office of LambdaMOO Architecture Review Board, except those you may have declined. Using the argument `all` shows all ARB nominating petitions.

@args <object>:<verb name> [<dobj> [<prep> [<iobj>]]]
@args# <object>:<verb number> [<dobj> [<prep> [<iobj>]]]
$prog

Changes the argument specifiers for an exisiting verb. Any omitted argument specifiers remain unchanged. If no arguments are given, then this prints out the current argument specifiers for the indicated verb. The second form (`@args#`) is used to specify a verb by number rather than by name, and is useful if an object has two verbs with the same name.

@at <object>
$frand_class

Prints a brief list of connected players either with or in `<object>` (depending on whether the specified object is a player, some other kind of object, or a room).

@audit [<player>] [for <string>] [from <object number>] [to <object number>]
$builder
Shows a list of objects that you own or that a specified player owns, with their object numbers. You may optionally specify a string, and see a list of objects with names or aliases that begin with that string. (Note, if the string has a space in it, then you must enclose it in double quotes.) You may optionally restrict the listing to a range of object numbers.

@ballots [all | open | closed | passed | failed | defeated]
LM player class #322
Prints a list of ballots. If no arguments are provided, prints a list of open ballots, if any.

@ban* <object>
@ban! <object>
LM player class #322
Prevents the specified object from entering any room that you own. If used with the exclamation point, prevents the specified object *and any descendents of it* from entering any room that you own. See also @unban.

@banned
LM player class #322
Prints a list of all objects you have banned from rooms you own using the **@ban** command.

boring [on | off]
LM player class #5803
Boring on makes you impervious to food fights. Boring off enables you to participate again. Boring with no argument toggles the setting and tells you what the new setting is. In addition, you will not lose things from your inventory into the couch cushions if you are boring.

@boot <guest>
LM player class #322
Disconnects a guest player's connection and disallows new connections from that guest's site for the following one hour. You must be at least four months old to use the command, and must give a reason, which is posted to *boot-log. The command must be seconded by another player.

@bug [<text>]
$mail_recipient_class
Sends MOOmail containing <text> to the owner of the room you're in as a bug report. If you do not specify text in the command line, then you are moved to the mail room to compose your message (presumably at greater length).

@build-o*ptions [<option> | <option setting>]
Also: **@buildo*ptions @builder-o*ptions @buildero*ptions**
$builder
Used without arguments, this command displays your current builder-options (settings that modify various builder commands). Used with a single option, displays the current setting of that option. Used with an option setting, modifies the specified option.

@check-chp*arent <object> to <new parent>
$builder
An object cannot be changed to a new parent if that object or any of its descendents defines a property that is also defined on the intended new parent. This command prints out all instances of conflicting properties that would interfere with @chparent in this manner.

@ch*eck <number of lines> [[!]<player>[,…,[!]<player>]]
$player
Prints a list of "best guesses" about where a line or lines of text originated, looking for "distrusted" players. By default, you and all the wizards are trusted, but you may specify additional players to be trusted (<player>) or not to be trusted (!<player>). You must have @paranoid on for this to work; LambdaMOOers will have to type @rmlag for @paranoid to work.

@ch*eck-full <number-of-lines> | <search string>
$player
Used to identify the source of text that is of dubious origin. @check-full prints out information about all the verbs responsible for a line of text displayed to your screen. You may specify either a number of lines or a string of text whose origin you wish to know more about. You must have @paranoid on for this to work; LambdaMOOers will have to type @rmlag for @paranoid to work.

@check-p*roperty <object>.<property name>
$prog
Prints a list of all descendents of <object> that define <property name>. See also @check-chparent.

@chmod <object> [+|-]<any substring of "rwf">
@chmod <object>.<property> [+|-]<any substring of "rwc">
@chmod <object>:<verb> [+|-]<any substring of "rwxd">
@chmod# <object>:<verb number> [+|-]<any substring of "rwxd">
$prog
Sets or changes permission flags. Objects – and also objects' individual properties and verbs – have *permission flags* that control whether non-owners can or cannot: read (objects, verbs, and properties), write (objects, verbs, and properties), execute (verbs), manually setting the value of properties, and make children (objects) (i.e. is the object fertile?). The "c" flag determines whether the owner of an object may change the value of a property that is defined on a parent or ancestor. (There is a longer explanation of the "c" flag and its use beginning on page 165.) If used with the "+" or "-" signs, it incrementally sets or clears the specified values. If used

without the “+” or “-” signs, it sets the permission flags to the specified values (clearing values as necessary). See also `help @chmod`.

@chparent `<object> to <new parent>`
`$builder`
Changes the parent of `<object>` to `<new parent>`. The object now has all the new parent’s properties and verbs, and all the new parent’s ancestors’ properties and verbs.

@cl*asses
`$builder`
Prints a list of object classes that the wizards have identified as “useful”. (This information is stored in `#0.class_registry` as a list of sublists. Each sublist consists of: `{<category>, <one-line description>, {<objects>}}`. The list is maintained manually.)

@clearp*roperty `<object>.<property>`
@clprop*erty `<object>.<property>`
`$prog`
Clears the value of the specified property on an object. This means the property will henceforth inherit its value from the object’s parent and will change as the value of the propety on the parent changes. The property will remain clear until it is set or changed on the child object itself.

@clear-tell-filter*-hook
LM player class `#33337`
Removes any tell-filter that is in use. (See `@set-tell-filter`.)

@comment `[<text>]`
`$mail_recipient_class`
Sends MOOmail containing `<text>` to the owner of the room you’re in as a comment. If you do not specify text in the command line, then you are moved to the mail room to compose your message (presumably at greater length).

@complete `<beginning of a word>`
`$frand_class`
Lists all the words in the dictionary (if present) that begin with the text you supply. E.g. `@complete silh` will give “silhouette”. (This may have been disabled in MOOs that don’t support `$spell`.)

connect `guest | <specific guest name>`
connect `<player name> [<password>]`
`$login`
Connects you to the MOO. Guest connections do not require a password. Omitting the password for a player connection will provide a separate prompt for your password, so that it will not be displayed on your screen.

@contents `[<object>]`
`$builder`
Gives a definitive list of `<object>`’s contents. If `<object>` is not specified, then lists the contents of the room you’re currently in.

@copy <object>:<verb> to <target object>[:<new verb>]
@copy-x <object>:<verb> to <target object>[:<new verb>]
@copy-move <object>:<verb> to <target object>[:<new verb>]
$prog

Copies a verb from one object to another, or to a new verb on the same object. It's better to have an object inherit a verb from a parent object than to copy verbs to objects directly, but occasions arise when copying a verb is the only way to get something done. @copy-x copies the verb without its "x" (executable) flag set, and would be used to archive a verb before making modifications to the working verb. @copy-move deletes the original verb after the copy is complete.

@count [<player>]
$builder

Tells you how many objects <player> owns and the total number of bytes used by those objects. <player> defaults to yourself.

@countDB [<player>]
$builder

This verb is related to @count, differing only in the counting method. @count inspects a player's .owned_objects property. A very few system characters (notably Hacker) do not participate in the object ownership system. To count these players' objects, it is necessary to consider every object in the database and see if <player> is its .owner. In large databases, this takes a long time and hogs system resources. Use @count instead, whenever possible.

@create <parent object> named <name>[,<alias>,…,<alias>]
$builder

Creates an object with the specified parent, name, and aliases. The object's parent can be changed at a later time with the @chparent command. The name and/or aliases can be changed with the @rename command. Aliases can be added or removed with the @addalias and @rmalias commands respectively. For rooms and exits, it's better to use @dig than @create.

@cspell <any number of words> | <object>.<property> |
<object>:<verb>
$frand_class

For those MOOs that utilize $spell, this command will check for misspelled words. It tends to run slowly.

@db*size
$prog

Reports the number of valid objects and allocated objects in the database.

decline <petition>
LM #55266 (Generic-Petition)

Removes <petition> from the list you see when you type @petitions.

@define <variable> as <value>
LM player class #8855
This command will probably be of interest only to programmers. It lets you predefine a value for use in a subsequent call to eval. See also @listdefs and **@undef**.

@denewt <player> [<comment>]
$wiz
Reverses the effect of @newt or @temp-newt.

@describe <object> as ["]<description>["]
$player
Sets the description of the specified object. If you omit the quotation marks, then sentences will be separated by a single space only, regardless of how many spaces you use to separate them when you type the description in. For multi-line descriptions, edit the .description property with the note editor.

@detail me with <detail name>[,<alias>,…<alias>] as <detail description>
@detail me with <detail name> is
LM player class #6669
Let's you add a detail to your description. Use the null string to remove a detail. Use the second form to display a detail. See help #6669:@detail for more detailed examples.

@details me | <nearby player> | <player object number>
LM player class #6669
Shows what details the specified player has defined. Note, this does not match on players' names the way many verbs do (e.g. page). If the player you want to know about isn't in the same room, you will have to use eir object number.

@dig <new room name>
@dig <exit>[,<aliases>][|<return-exit>[,<aliases>]] to <new room name> | <existing room object number>
$builder
Creates a new room, or exits to (and optionally back from) either a new room or a specified existing room. Note, the vertical bar "|" separating exits to and from a room is actually part of the command, rather than the meta syntax. Example:

```
@dig north,n | south,s to The Mosh Pit
```

@disown <object> [from <parent>]
Also: **@disinherit**
$prog
Contrary to what the name might suggest, this command does not alter an object's ownership. Rather, it alters an object's parentage, changing an object's parent to its grandparent. This command would be used if you did not own the object, but owned its parent, and no longer wanted the object to be a child of that parent. Changing the permission flag on the parent to -f (see @chmod) will prevent people from using that object as a parent in the future.

@d*isplay <object>[.][,][<property>]
@d*isplay <object>[:][;][<verb>]
@d*isplay <object>[.|,][:|;]
$prog
This command displays <object> and/or <object>.<property> and/or <object>:<verb> ownership, permissions, and values (for properties). The period and colon indicate information about properties or verbs defined on the object itself; the comma and semi-colon indicate information about inherited properties and verbs. See `help @display` for a more detailed explanation.

@display-o*ptions [<option> [<setting>]]
Also: **@displayo*ptions**
$player
Used without arguments, this command displays your current `@display` options (settings that modify various aspects of the output from `@display`). Used with a single option, displays the current setting of that option. Used with an option and setting, modifies the specified option.

@dump <object> [with [id=#<new object number>] [noprops] [noverbs] [create]]
$prog
Prints out all of an object's verbs and properties. If you specify `with create`, it will print it out in a form that can be used for porting an object to another MOO rather than merely investigating it on the source MOO. You can optionally omit properties and/or verbs, and can ask it to use a different object number in its output (this, too, can be useful for porting an object to another MOO).

@edit <object>[.<property>]
@edit <object>:<verb> [<dobj> <prep> <iobj>]
$player
Invokes the note editor or verb editor as appropriate. If no property is specified, then it defaults to `.text` if `<object>` is a descendent of `$note` or `.description` for any other kind of object.

@edit-o*ptions [<option> [<setting>]]
Also: **@edit-o*ptions**
$player
Used without arguments, this command displays your current `@edit` options (settings that control various aspects of the in-MOO editors). Used with a single option, displays the current setting of that option. Used with an option setting, modifies the specified option.

@egrep <regular expression> in <object> | <list of objects>
$prog
Searches the specified object or list of objects for verbs containing a substring matching `<regular expression>`. A regular expression is a template for expressing generalized strings. See `help regular-expressions`. See also `@grep`.

@eject <player or other object> [from <location>]
@eject! <player or other object> [from <location>]
@eject!! <player or other object> [from <location>]
$player
The usual way to move something is with the @move command, and it's considered polite to try to @move a thing before ejecting it. If the object won't move and you own the object's location (this includes yourself), then you should use @eject. With no exclamation points, this moves an object to its .home if indicated and possible. With one exclamation point, moves the offending object to the location $nothing, but notifies the object that it's being moved. With two exclamation points, moves the object to $nothing but doesn't notify the object.

eprint <expression>
eprint<n> <expression>
LM player class #5803
This command is useful for printing out complicated MOOcode expressions with indentation intended to make them easier to read and understand. For example if you had a complicated conditional clause and wanted to sort out what was actually being checked:

```
eprint (caller == this && args[2] ||
this.tally_board.registry:primary_char(dude) in
this.tally_board.public_access || (caller != this &&
$local.second_char_registry:trust(caller_perms())))
```

would yield:

```
 ((caller == this) && args[2])
     || ((   this.tally_board.registry:primary_char(dude)
          in this.tally_board.public_access)
     ||  (   (caller != this)
          &&
$local.second_char_registry:trust(caller_perms())))
```

The output can optionally be restricted to <n> columns. There is no space between eprint and <n>, e.g,

```
eprint10 ((ticks_left() < 3000) && suspend(0)).
```

eval <MOO-code>
eval-d <MOO-code>
;<MOO-code>
$prog
Evaluates a line of text as if it were MOO-code. Eval-d prints errors as values rather than generating a traceback. See help eval.

exam*ine <object>
$player
Provides more information about an object than you can get just by looking at it, including its full name, aliases, object number, owner, description, and any

obvious verbs that you can use on it. Unlike @examine, its output can be modified by the object's owner.

@exam*ine <object>
$player
Provides the same information that examine does, except that its output can't be controlled by the object's owner. This has advantages and disadvantages. The advantage is that you may see more information. The disadvantage is that the information might not be printed as nicely or might not be relevant to you (e.g. "obvious verbs" that are really only intended to be used by the object's owner).

@features [<name>] [for <player>]
$player
Lists all of a player's features matching <name>, or, if <name> is not supplied, all features for that player. Lists your own features if no player is specified.

@find <object number> | <player name or alias> | .<property name> | :<verb name> | ?<help topic>
$frand_class
Prints the location of the specified thing. This is especially useful for verbs and properties because it prints out *all* instances that it finds in your vicinity (including your known objects, see @remember).

@flush-cache
LM player class #322
When you use a feature object, the server must look through all your feature objects to find the appropriate verb. This takes time. In order to help reduce lag, the system records your most recently used feature object verbs and checks those first. The place where your most-recent-usage information is stored is called a *cache*. This command clears out the cache of your recently-used feature object verbs.

follow <player>
LM player class #8855 (This command is also provided on some other MOOs, but not is included with LambdaCore.)
Causes you to follow <player>. See also unfollow, stop-following, @list-followers and lose.

@forget <object>
LM player class #26026
Removes an object from the list of those you keep track of using @remember.

@forked [<player>]
$prog
Displays a list of all your suspended and forked tasks with their respective task_id numbers. Only a wizard may specify a player other than emself. If a wizard invokes this command without specifying a particular player, then this command will display all forked tasks in the system. This command is particularly useful for identifying tasks that you may want to @kill.

@forked-v*erbose [<player>]
$prog
This command displays the same information as @forked, except that for tasks that are suspended rather than forked off, shows the full callers() stack.

@forward <msg> [on *<recipient>] to <recipient>[,<other recipients>]
$mail_recipient_class
Forwards a MOOmail message to the designated recipient(s). See help @forward for a discussion of the nuances of this command.

@gag*! <player or object>
$player
This command prevents you from seeing any text emanating from the specified player or object. (Note, this does not include posts to mailing lists or MOOmail. See @refuse.) If an object has children, then you must use the exclamation point; this will have the effect of @gagging the object and all its descendents. It is not possible to @gag a parent object only.

@gaglist [all]
$player
The first form, with no arguments, displays a list of players and objects that you are gagging. The second form looks for and displays players who are gagging you. The second form is slow to run and adds to lag, so it should be used sparingly.

@gag-site <guest> for <duration>
LM player class #322
Prevents you from seeing any text from any guest connecting from the same site as the designated guest, for the specified duration of time.

@gag-sites
LM player class #322
Displays a list of all guests whose sites you have gagged, along with when you gagged the site and how much time is remaining before the site-gag expires.

@gender [m | f | n | <other>]
$player
Sets your gender (and pronouns) to male, female, neuter, etc.. Without an argument, displays your current gender setting and other available genders to

@gethelp [<topic> [from <db or dblist>]]
$prog
Locates and prints out the raw text of a help topic in a form that can be cut, modified, and pasted back in (like @dump). With no argument, gets the blank ("") help topic.

@gms [all]
LM player class #322
Prints an @who listing of connected (or all) LambdaMOO RPG game masters.

@go <location>
$frand_class
Teleports you to the specified location, which can either be the object number of a room or the name of a room in your `.rooms` database.

go <direction>
$room
Moves you in the specified direction (e.g. north). You may specify more than one direction, in which case you will go in those directions in sequence. `Go north east north` would move you first north, then east, then north again.

@grep <string> in <object> | {object list}
$prog
Searches the specified object or list of objects for verbs containing `<string>`. See also `@egrep`.

@gripe [<text>]
$mail_recipient_class
Moves you to the mail editor, ready to send a mail with subject heading `<text>` to the mail recipient(s) specified by the wizards in `$gripe_recipients`.

heartbeat
Syntax: ;me:heartbeat(<n>)
LM player class #5803
Starts up a task that prints a time stamp to your screen every `<n>` minutes. For those who idle for long periods of time, this can help identify when someone paged you. Note, this command can only be used by programmers (because it has to be started up using `eval`).

help [<topic>]
$player
Displays online help text for the specified topic. If no topic is specified, then it displays a list of some of the topics for which help text is available.

home
$player
Moves you to your home, or to the default player starting place if your home is invalid or won't accept you for some reason.

@idea [<text>]
$mail_recipient_class
Sends MOOmail containing `<text>` to the owner of the room you're in as an idea suggestion. If you do not specify `<text>` in the command line, then you are moved to the mail room to compose your message (presumably at greater length).

inv*entory
$player
Prints a list of things you are holding, with their object numbers.

@join <player>
$frand_class
Teleports you to the specified player's location. You can specify <player> by object number, name, or any alias.

@keep-m*ail [<message sequence>]
Also: **@keepm*ail**
$mail_recipient_class
Prevents the designated mail message(s) from expiring (i.e. being automatically deleted after a certain amount of time). See help @keepmail.

@kids <object>
$builder
Prints out a list (with object numbers) of all an object's children. (Note, not all descendents, just children.)

@kill <task id> | <object>:<verb> | soon <number of seconds> | all | %<trailing id>
Also: **@killq*uiet**
$prog
Kills one or more background tasks (see @forked). The second form, @killquiet, is better for killing large numbers of tasks, as it prints a summary of the number of tasks killed rather than a line for each one (especially useful if you've accidentally created a chain of forked tasks, each of which is sending text to your screen). Task id numbers tend to be large. You can use the %<trailing id> form to abbreviate the number to its last few digits: Instead of typing @kill 2053554299, you can type @kill %299, and the system will kill all tasks in your queue that end in the numbers 299.

@known*_objects
Also: **@known*-objects**
LM player class #26026
Prints a list of objects you've made note of with @remember.

@last-c*onnection [all]
$player
Reports your most recent connection information (when and from where) or, if all is specified, your last ten connection times and sites.

@lastlog [<player>[,...,<player>]]
$player
LM player class #5803
Shows the last disconnect time of the specified player or players. If called with no arguments, shows the last connection times of *all* players.

@linelen*gth [<number>]
$player
This command is used in conjunction with @wrap, to cause the MOO to perform word-wrapping for you. Without arguments, informs you of your current setting,

along with whether word-wrapping is currently turned on or off. With a number as an argument, sets your line length to that number of characters.

@list*# <object>:[<verb name> [<dobj> <prep> <iobj] | <verb number>] [with | without parentheses | numbers] [all] [<start>..<end>]
$prog
Lists the MOO code associated with the specified verb. Normally, this command lists only the code found either on the specified object itself or on its nearest ancestor. The optional argument all causes the corresponding code on the object and *all* ancestors to be displayed. By default, lines are numbered and show only those parentheses necessary to the meaning of the code. You can specify a range of line numbers to list if you know you only want to see part of the verb. These defaults can be changed with the @programmer-options command.

@listdefs
LM player class #8855
Lists variables you have defined using the @define command.

@list-followers
LM player class #8855
Prints a list of people who are programmatically following you. (See also follow.)

@listgag [all]
$player
See @gaglist.

@location*s <object>
$builder
Prints out the names and numbers of all objects that contain the specified object.

@lock <object> with <key expression>
$builder
This command is used to specify (via <key expression>) locations to which an object may be moved (and by extension, locations to which an object may *not* be moved). See help keys.

lose <player> | all
LM player class #8855
This command causes <player> (or everyone) to stop following you programmatically. See follow.

@mail <message-sequence> [on <recipient>]
$mail_recipient_class
Displays headers of the specified mail messages. (See the section on reading mail that begins on page 51 for a detailed explanation.)

@mail-all-new*-mail
$mail_recipient_class
Displays the headers of all unread mail messages on yourself and lists to which you are subscribed.

@mail-o*ptions [<option> | <option setting>]
Also: **@mail*ptions**
$player
Used without arguments, this command displays your current @mail options (settings to customize various aspects of the mail system). Used with a single option, displays the current setting of that option. Used with an option setting, modifies the specified option.

@make-petition <name>[,<alias>,...,<alias>]
LM player class #322
Creates a LambdaMOO petition with the specified name and aliases.

@measure <object>
@measure summary [<player>]
@measure new [<player>]
@measure breakdown <object>
@measure recent [<number of days>] [<player>]
$builder
For MOOs that use byte-based quota, objects are measured approximately once a week by a background measurement task. The various forms of the @measure command provide a way to update the measurement records on an incremental basis, when needed. @measure <object> measures the size of an object on demand. This is appropriate if an object's size is known to have undergone a recent significant change. @measure summary updates the summary information displayed by the @quota command. @measure new measures all of a player's objects that have never been measured. (This might be needed if one were creating a large number of small objects in a short time span, as one is only permitted to have a fixed number of unmeasured objects at a time.) Use @measure breakdown if you need to find out what part(s) of an object are taking up large amounts of space. @measure recent measures those things which have not been measured automatically with the specified number of days.

@mess*ages <object>
$player
Lists all the messages on the specified object, and their values.

@mode [brief | verbose]
$player
Sets your viewing mode. If brief, then only the name of a room will display on your screen when you enter that room. If verbose, then a room's name and description will display when you enter it. The default is verbose.

@more [rest | flush]
$player
If you have @pagelength set and the system has produced more lines of output than will fit on your screen, you will see a message of the form

```
*** More ***  <n> lines left.  Do @more [rest | flush] for
more.
```

@more without arguments prints sufficiently many lines to fill your screen, or all that remain, if they will fit. @more rest will print all of the remaining lines, regardless of whether they will fit or not. @more flush discards all remaining lines instead of displaying them on your screen.

@move <object> to <location>
$frand_class
Teleports the specified object to the specified location.

mu*rmur <person> <text>
LM player class #33337
This command does the same thing as whisper, except that the syntax is such that you don't need to enclose the whispered text in quotation marks.

news [all | new | contents | archive]
$player
Read the contents of the newspaper, which is a subset of messages on the *news mailing list that the wizards have designated as being of current interest or relevance to the entire MOO. news new will display news items that you have not yet read. news all will display all news items. news contents will display headers of all news items. news archive will display all messages on the *news mailing list.

@netforw*ard [<message-sequence> [on <mail-recipient>]]
$mail_recipient_class
Forwards the designated message(s) to your registration email address. Defaults to the current message on your current folder.

@newmess*age <message-name> [<message-text>] [on <object>]
$builder
In general, only programmers can add new properties to objects. This command lets non-programmer builders add message properties to objects they own.

@newt <player> [<reason>]
$wiz
A wizard-only command. Inhibits the specified player's ability to connect to the system. A MOOmail message is automatically sent to *site-locks. See also @denewt, @temp-newt.

@next [<how-many>] [on <mail_recipient>]
$mail_recipient_class
Prints out the next <how-many> mail messages on your current folder or the designated mail recipient (yourself or a mailing list). Defaults to one message.

@nominate <player> for <office>
LM player class #322
Nominates a person for public office on LambdaMOO. This can only be done during a two-week nominating period before any election. The offices are: ARB, Reaper, and Registrar.

@notedit <note-object> | <object>.<property>
$player
Moves you to the note editor, working either on the text of the specified note or on the text in the designated property on the designated object.

@nprop*erty <object>.<prop-name> [<initial-value> [<perms> [<owner>]]]
LM player class #5803
This is just like @prop*erty, except that <initial-value> is evaluated, first.

@owner <object>
$player
This command shows you who the owner of an object is. (It was added to LambdaMOO in July, 2000, and may or may not be available on other MOOs.)

page <player> <text>
$player, LM player class #5803
Sends <text> to <player> as if you were paging em from a distance. The fancier version on LM player class #5803 lets you page more than one player simultaneously; you must enclose the players' names in quotation marks: page "<player1> <player2> <player3>" <text>.

@pagelen*gth [<number>]
$player
This command is used in conjunction with @more to control the display of lines on your screen. When a number is specified, it sets your page length to a number of rows (of text). The system will prompt you with a message to type "@more" if there is more text about to display than will fit at one time. Without an argument, it will show you your current setting. To turn off page buffering and see all the lines of text at once, set your page length to zero. This would be an appropriate choice if you were switching from telnet to a client that lets you scroll back, for example.

@paranoid [off | immediate | <number>]
$player
This command is used to record and investigate lines of text that print to your screen. If invoked with immediate as the argument, it will prepend each line you see with the name of the player it thinks is responsible for generating that line. If invoked with <number>, then that number of lines is stored for later inspection with @check or @check-full. On LambdaMOO, players must first type @rmlag to disable the lag-reduction FO for @paranoid to work.

@parent <object>
LM player class #8855
Tells you the name and object number of an object's immediate parent.

@parents <object>
$builder
Displays the names and object numbers of an object's parent and ancestors.

party
LM player class #5803
This command prints a list of rooms and occupants in order of decreasing crowd size and increasing idle time (i.e. the liveliest parties first). For each, it comments on the security arrangements and asks if you want to go there. You may discontinue the listing at any time by typing @abort.

@password <old-password> <new-password>
$player
Changes your password.

@paste
Pasting Feature Object, LM player class #8855
Prompts for lines of text (terminated by a period on a line by itself) then displays the text to the entire room.

@pasteto <player>
Pasting Feature Object
Prompts for lines of text, then displays them (privately) to the specified player.

@pc-news
LM player class #33337
The author of the Politically Correct Featureful Player Class Created Because Nobody Would @copy Verbs to 8855 provided himself with a way to broadcast news items to users of his player class in a manner similar to the MOO-wide news command. This command is used to read those news items.

@pc-options
LM player class #33337
This player class provides an options package that works the same way that @mail-options does. Type @pc-options to list these options, and help @pc-options for additional information on setting them.

@pedit <object>.<property>
LM player class #5803
This command moves you to the property editor, described as "highly experimental" by its author. If you are editing a property whose value is a string or list of strings, you are probably better off using the note editor, instead, but this facility might be useful for editing properties with a more complex structure. See help @pedit for more detailed information.

@petition-options
LM player class #322
Lists options that pertain to various aspects of the LambdaMOO petition and ballot system. Use @petition-option +noannounce to suppress announcements of open ballots every time you log in.

@petitions [all | public | signed | vetted]
LM player class #322
Lists all or some petitions, as specified. The default is to list signed petitions. You can use @petition-options to customize the order in which petitions are presented.

@prettylist <object>:<verb>
LM player class #5803
Prints a verb with line breaks and indentations intended to make it easier to read.

@prev*ious [<how-many>] [on <mail_recipient>]
$mail_recipient_class
Prints out the previous <how-many> mail messages on your current folder or the designated mail recipient (yourself or a mailing list). Defaults to one message.

@prog*ram <object>:<verb> [<dobj> <preposition> <iobj>]
@program# <object>:<verb-number>
$prog
These commands put you into a line-reading mode. The lines you type in are saved as the content of the designated verb on the designated object, if that verb exists (otherwise the lines of text are still read, but are ignored).

@progo*ptions
Also: **@prog-o*ptions @programmero*ptions @programmer-o*ptions**
$prog
Lists a set of options available to programmers to customize certain system behaviors related to programming, e.g. whether line numbers print out when you list a verb. The @programmer-options package works like the @mail-options package.

@prop*erty <object>.<property-name> [<initial value> [<permission-flags> [<owner>]]]
$prog
Adds a property to an object. The initial value defaults to 0. The permission flags default to "rc", but this can be changed as one of the @programmer-options. A wizard may specify an owner other than emself.

@pros*pectus <player>
$prog
This command is like @audit, but provides additional information about each object, such as whether it has kids, how many verbs are defined on it, etc. See help @prospectus for more detailed information.

@quickr*eply <msg> [on <recipient>] [sender | all | followup]
Also: **@qreply**
$mail_recipient_class
This command lets you reply to a mail message without actually going to the mail editor. It prompts you for lines of input and then sends them directly.

@quick*send <mail-recipient> [subj = "text"] [<one-line-message>]
Also: **@qsend**
$mail_recipient_class
Sends a message to a player or a list without moving you to the mail editor. If you do not specify a one-line message, it prompts you for lines of input, then sends them directly.

@quit
$player
Disconnects you from the MOO. Your player object is automatically moved to its home a short time later.

@quota [<player>]
$builder
Prints out your current quota and measured usage, or the quota of a specified player.

@ranm
$mail_recipient_class
See @read-all-new*-mail.

@read [<message-sequence> [on <mail-recipient>]]
$mail_recipient_class
Reads the specified messages on the specified mail recipient (yourself or a MOO mailing list). If no message sequence or recipient is specified, reads your current message on your current folder. Updates your current message pointer.

@read-all-new*-mail
$mail_recipient_class
Reads all new messages on all mailing lists to which you are subscribed. Prompts you at the end to verify that you got all the information, and if you answer yes, updates your current message pointer. If the system crashes or you somehow disconnect before being able to answer the prompt, then your current message pointer is not updated, and these messages will still appear as new messages next time you log on. This command can be abbreviated as `@ranm`.

@reaper-ballots
LM player class #322
Lists ballots for the office of LambdaMOO Reaper.

@reaper-petitions
LM player class #322
Prints a list of all petitions nominating candidates to the office of LambdaMOO Reaper, except those you may have declined. Using the argument `all` shows all reaper nominating petitions.

@reapers [all]
LM player class #322
Prints a list of connected (or all) LambdaMOO Reapers. Reapers recycle players who have not logged on for a certain amount of time, and oversee the distribution of their owned objects if deemed appropriate. See `help reaping`.

@recreate <object> as <parent> named <new-name>
$builder
Takes an existing object and totally recreates it as a new kid of <parent> as if with **@create**. Verbs and properties on the object are stripped off, and inherited properties are reset to be clear.

@recycle <object>
$builder
Destroy an object irretrievably. If you have your @builder-options set to -bi_create (the preferred setting), the object will be turned into a kid of $garbage for re-use the next time someone invokes @create. Players may not be recycled unless they have first been made non-players with the @toad command or an equivalent.

@refusal-r*eporting [on | off]
$frand_class
If set to on, notifies you when someone whom you are refusing attempts a refused action while you are connected ("so that you can thumb your nose," says the documentation). If invoked without an argument, displays whether refusal reporting is currently on or off. Refusal reporting works for page, whisper, and mail, but doesn't work for move, join, or accept.

@refusals [for <player>]
$frand_class
Lists players and actions that you are refusing or that the specified player is refusing.

@ref*use <action(s)> [from <player>] [for <duration>]
$frand_class
The MOO provides a way to refuse certain actions, either universally or from a specified player. The actions that can be refused are page, whisper, mail, move, join (only works in certain rooms that support it), accept, flames, politics (LambdaMOO only), and all of the above. See also help @refuse.

@registerme [as <email-address>]
$player
Displays your current MOO registration email address, or changes it to a new one. If you are changing it, a new password is generated and mailed to the new address. You can change the new password back again with the @password command.

@registrar-ballots
LM player class #322
Lists ballots for the office of LambdaMOO Registrar.

@registrar-petitions [all]
LM player class #322
Prints a list of all petitions nominating candidates to the office of LambdaMOO Registrar, except those you may have declined. Using the argument all shows all registrar nominating petitions.

@registrars [all]
LM player class #322
Prints a list of connected (or all) LambdaMOO Registrars. Registrars assist the wizards in creating new players. They have access to players' email addresses.

@remember <object>
LM player class #26026
Remember an object's number. You can see a list of these objects by typing @known. See also @forget.

@remove-feature
See **@rmfeature**.

@rename <object> to "<new-name>"[,"<alias>",...,"<alias>"]
$player
Rename an object, with or without additional aliases. See help @rename for some detailed examples.

@renumber [me | <mail-recipient>]
$mail_recipient_class
Renumbers, from 1 to the total number of messages, all MOOmail messages on yourself or on a mailing list you own. Renumbering a public mailing list is inadvisable because it disrupts other players' current message pointers to that list. No messages are actually lost, but @rn will show that there are new messages on the list while @nn will say that there are no new messages. See also help zombie-messages.

@repl*y [<message-number>] [on <mail-recipient] [sender | all | followup] [include | noinclude]
$mail_recipient_class
Takes you to the mail editor and sets up a reply to the specified message. Specify sender to reply to the sender only, all to send your reply to all recipients who received the original post, or followup to send your reply to the first non-player recipient (i.e. a list). Specify include or noinclude to include or omit (respectively) the text of the original message. If these options are omitted, the defaults are sender and noinclude, but these can be changed with @mail-options.

@request <character-name> for <email-address>
$guest
This is the command used to request a player-character on a MOO.

@resend <message-sequence> [on <mail-recipient>] [to <recipient(s)>]
$mail_recipient_class
This is like @forward, except that it keeps the original body of the forwarded message intact and modifies the header to indicate that you resent it.

@resident <player-or-object>
@resident !<player-or-object>
@residents
$room
The first form adds a player or object to a room's list of allowable residents. If a player, that player may then set eir home to that room. The second form removes a player or object from a room's list of residents. The third form displays a room's current list of residents.

@rmalias <alias> from <object>
@rmalias <alias> from <object>:<verb-name>
@rmalias# <alias> from <object>:<verb-number>
$player
Removes an alias from the specified object or verb. `@rmalias#` is for unambiguously identifying a verb when an object may have more than one verb with the same name.

@rmdict <word>
$frand_class
Remove a word from `$spell`, if present. (Many MOOs have recycled `$spell` because it is cumbersomely large). Only wizards and players in `$spell.trusted` may use this verb.

@rmfeature <feature-object>
$player
Remove a feature from your `.features` list. Feature objects are used to extend the set of commands available to a player. See also `@add-feature`.

@rmlag
LM feature object #26787
Turns on the lag reduction FO and any features similar to it. Disables `@gag`, `@check-full`, and any other commands that utilize a player's `:tell` verb to filter and/or otherwise pre-process text before it is displayed to your screen.

@rmm*ail [<message-sequence>] [from <recipient>]
$mail_recipient_class
Removes one or more MOOmail messages from yourself or a specified mail recipient (mailing list). See also `@unrmmail`.

@rmprop*erty <object>.<property-name>
$prog
Removes the named property from the specified object.

@rmr*oom <name>
$frand_class
Remove the named room from the list of rooms you remember by name. See also `@addroom` and `@rooms`.

@rmverb <object>:<verb-name> [<dobj> <prep> <iobj>]
@rmverb# <object>:<verb-number>
$prog
Remove the specified verb from the specified object. If there are two or more verbs with the same name, removes the most recently defined one. If the argument specifiers are provided, then it removes the most recently defined one matching both verb name and argument specifiers. The second form, @rmverb#, is used to unambiguously remove a verb as specified in the (1-based) list given by the built-in function verbs(<object>).

@rmword <word>
$frand_class
Remove a word from your personal dictionary (stored in a player's .dict property).

@rn
$mail_recipient_class
Lists a summary of new messages on mailing lists to which you are subscribed, similar to that displayed when you log in.

@rooms
$frand_class
Displays a list of rooms you have remembered using the @addroom command.

seek <player>
LM player class #7069
Tries to move you to the designated player's location using an exit, thus simulating walking (as opposed to teleporting). (See also help #27325:@seek).

@send [<recipient> [<recipient(s)>] [subj[ect]="<subject>"]
$mail_recipient_class
Moves you to the mail editor and prepares you to compose a MOOmail message to the designated recipient(s). If no recipient is specified, resumes an earlier mail editor session, if there was one.

@setenv <environment string>
$prog
Sets a string that is evaluated before the eval command evaluates anything else. Example: @setenv me=player;here=player.location

@sethome
$player
Tries to set your home to your current location. Some rooms permit players to set their homes there, while others do not. This is at the discretion of the room's owner. If you are a room owner and want to make it so that anyone may set eir home there, @set <room>.free_home to 1. To permit an individual player to set eir home to a room you own, use the @resident command.

@set*prop <object>.<property name> to <value>
$builder
Changes the value of an existing property on an object to the specified new value.

@set-tell-filter*-hook <tell-filter-object>
LM player class #33337
A tell-filter is an object that intercepts text which is about to be displayed to your screen and may (or may not) modify that text in some way before it is actually displayed to you. One possible example would be to put a special symbol before text that was generated with the emote verb. A tell filter is usually custom programmed by the player intending to use it – if you use a tell-filter owned by someone else, its owner would theoretically be able to see most of the text that you see if e chose to. (See also @clear-tell-filter and help tell-filter.)

@s*how <object> | <object>.<property> | <object>:<verb>
$prog
This command is very similar to @display, but the information it displays about objects, properties and verbs is in a different format and more detailed. See also @ss and @display.

@skip <mail recipient>
$mail_recipient_class
Skip to the end of a mailing list, as if you had read all the messages. (This resets your current message pointer for that list.)

@sort-owned*-objects object | size
$builder
The @audit command displays a list of objects you own in the order that you created them. This command lets you change that so that your objects are sorted by object number or by size. Once this is done, however, there is no way to go back to having them sorted by when they were created.

@spell <any number of words> | <object>.<property> |
<object>:<verb>
$frand_class
Checks the spelling of a sequence of words, the words in an object's property (the property must be a string or list of strings), or the quoted parts of a verb. (This command may be disabled on some MOOs.)

@spellm*essages <object>
$frand_class
Checks the messages (all properties whose name ends in _msg) on the specified object for correct spelling. See also help spelling.

@spellp*roperties <object>
$frand_class
Check all properties on the specified object for correct spelling. Properties that are not a string or a list of strings will be ignored. See also help spelling.

@spurn [!]<object>
$frand_class
Prevent an object or any of its descendents from entering your inventory. Used with the exclamation point, this command removes an object from your list of spurned objects.

@spurned
$frand_class
Displays a list of spurned objects.

@ss*how <object> | <object>.<property> | <object>:<verb>
LM player class #5803
A short version of @show.

stop-following <player>
LM player class #8855
Cease programmatically following <player> wherever e goes. See also **follow**.

@subscribe [<mailing list>]
@subscribe*-quick [<mailing list>]
$mail_recipient_class
Subscribes you to a mailing list. If you type @subscribe without specifying a mailing list, then the system will print out all the lists to which you are not subscribed, along with their descriptions. @subscribe-quick, without specifying a mailing list, will print out only the names of the mailing lists to which you are not subscribed, i.e. the lists' descriptions are omitted.

@subscribed
$mail_recipient_class
Displays a list of all mailing lists to which you are subscribed, whether or not they have new messages on them. See also @rn.

@suggest*ion [<text>]
$mail_recipient_class
Sends MOOmail containing <text> to the owner of the room you're in as a suggestion. If you do not specify text in the command line, then you are moved to the mail room to compose your message (presumably at greater length).

@sweep
$player
This verb searches your local environment for objects that might be relaying information. It omits objects and verbs owned by yourself or by a wizard. Programmers wishing to customize what is displayed by their objects when someone uses the @sweep command should add a :sweep_msg verb.

@teleport
See @move.

@tell-filter
LM player class #33337

Displays information about the tell-filter object in use, if any. See @set-tell-filter.

@toad <player> [graylist | blacklist | redlist] [<comment>]
@toad! <player> [graylist | blacklist | redlist] [<comment>]
@toad!! <player> [graylist | blacklist | redlist] [<comment>]
$wiz

A wizard-only command. Deactivates a player object's status *as* a player, but does not recycle the object. The player's owned objects are left in the database as orphans, so it's a good idea to @audit em first and @recycle the objects listed. If used with one exclamation mark, the victim is also @blacklisted. If used with two exclamation marks, the victim is @redlisted. The optional comment is included in a post to *site-locks. See also help @toad and help @blacklist.

@tutorial
LM player class #322

Starts a tutorial of basic MOO commands. Type quit at any time to discontinue it.

@typo [<text>]
$mail_recipient_class

Sends MOOmail containing <text> to the owner of the room you're in, to report a typographical error. If you do not specify text in the command line, then you are moved to the mail room to compose your message (presumably at greater length).

@unban <player or object> | everyone
LM player class #322

Cease banning someone or something from all rooms you own. See also @ban, @banned.

@undef*ine <label>
LM player class #8855

Remove a definition for eval. See @define.

unfollow <player>
LM player class #8855

Stop following <player> wherever e goes. See also follow.

@ungag <player or object>
$player

Cease @gagging a player or object. You will once again see text originating from em or it. See @gag.

@ungag-site all | last | <guest> [<date>]
LM player class #322

Cease @gagging a site associated with a guest. See @gag-site, @gag-sites. Specify the date if you used @gag-site for guests with the same name on different occasions.

@unlock <object>
$builder

Clear any lock you may have placed on the object. See @lock, and help locking.

@unmess*age <message-name> [from <object>]
$builder

Remove a message property from an object you own (defaults to yourself). (Normally only programmers can add and remove properties. But anyone can add or remove a message.) See @newmessage.

@unread <msg> [on <recipient>]
$mail_recipient_class

Reset your message pointer, as if you haven't yet read the specified message on the specified mailing list.

@unrefuse <actions> from <player> | <actions> | everything
$frand_class

Cease @refusing specified actions. See @refuse.

@unrmm*ail [list | expunge] [on <recipient>]
$mail_recipient_class

When you remove a MOOmail message from yourself or a mailing list using @rmm, it isn't really deleted from the database, but rather is saved in a sort of limbo as a zombie message. The main purpose of the @unrmm command is to undo the removal of a message, restoring it to yourself or a mailing list. You can also use this command to view a mailing list's associated zombie messages, or to expunge any zombie messages so that they are well and truly gone forever.

There are a few idiosyncrasies of this verb that the formal syntax, while correct, doesn't make very clear. First, notice that while we remove messages from a list, we unremove them on a list. Second, unremove is an all-or-nothing proposition – you can't specify a message sequence. For a specified recipient, you list *all* zombie messages, restore *all* zombie messages, or expunge *all* zombie messages. See also help @unrmmail and help zombie-messages.

@unsend [<message-sequence>] from <player>
$mail_recipient_class

This command enables one, in some circumstances, to retract a post that one has sent to a player. There are several exceptions. See help @unsend. It was added to LambdaMOO in 1999, and therefore is not present in older versions of the core database.

@unset-tell-filter*-hook
LM player class #33337
Removes any tell-filter that is in use. (See @set-tell-filter.)

@unsubscribe [<list or lists>]
$mail_recipient_class
Unsubscribes you from the specified mailing lists, or your current mailing list if no mailing list is specified.

@unsubscribed
@unsubscribed-quick
$mail_recipient_class
Prints out the names and descriptions of all mailing lists to which you are not currently subscribed. The quick version prints out names only.

@uptime
$player
This command displays the amount of time since the last system restart.

@users
$player
Lists names of all connected players, in alphabetical order. See also @who.

@verb <object>:<verb name(s)> [<dobj> <preposition> <iobj>
[<permission flags> [<verb owner>]]]
$prog
Adds a new verb to an object. If more than one alias is given to the verb, then the names should be separated by spaces and all enclosed in double quotes, e.g., @verb me:"fee fie fo fu*m". Default argument specifiers can be specified with @prog-option. The verb owner defaults to yourself; only wizards can specify a different verb owner. You must own an object or be a wizard in order to add a verb to it.

@verbs <object>
$prog
Prints a concise list of the verbs defined on an object. See also @display.

@verify-owned
$builder
Verifies that your owned objects are all, in fact, owned by you. (A situation where this might not be the case could occur when an inexperienced wizard tried to change an object's ownership by manually setting its .owner property rather than using the wizard-only @grant command.)

@version
$player
Prints out the version number of the currently-running MOO server, and the date that the database was extracted from the core.

@watch [<player> | none | off]
LM player class #33337

This verb notifies you when the watched player ceases to be idle, i.e., when e types something. With no arguments, tells you whom you are watching. With none or off, turns watching off.

@ways [<room>]
$frand_class

Lists a room's obvious exits and their aliases.

ways
LM player class #5803

Lists all of the obvious exits from your current location.

@web me is [<web information>]
$frand_class

This verb lets you specify web information about yourself (e.g. your home page) for others to view. If <web information> is omitted from the command, it shows you your current web information. Programmers may access others' web information either via a player's .web_info property or :web_info verb. This verb and its associated properties are not included with the LambdaCore (even though $frand_class is included).

where*is [<player> [<player(s)>]]
@where*is [<player> [<player(s)>]]
$player

Lists the name(s), object number(s), location(s) and location object number(s) of the specified player or players. If no argument is given, then lists all players.

wh*isper "<text>" to player
$player, $frand_class, LM player class #7069

Communicates <text> to the specified player. Other players in the room do not see the exchange. The version on $frand_class permits @refusal of whispers. The version on LM player class #7069 stores the identity of the person who most recently whispered something to you for use with its respond verb, **=**. See also murmur, @refuse, **=**.

@who [<player> [<player(s)>]]
$player

Shows the names, numbers, idle durations and locations of the specified players, or all players if none are specified. Many different versions of this verb have since been written; some player classes provide ways to select which version of @who you prefer to use. See also @users.

@will recycle <item designator>
@will bequeath <item designator> to <player>
@will refuse <item designator> to <player>
@will keep <item designator>
@will display
@will forget <item designator>
@will clear
LM player class #322

This command provides a variety of ways for one to specify how one wishes one's objects to be disposed of in the event that one is reaped. See help @will on LambdaMOO for more detailed information.

@witness [on]
@witness off
@witness show [<number>]
@witness display [<number>]
@witness delete <number>
@witness email <number>
@witness publish <number>
LM player class #322

This command provides a way to log conversations. Witness logs cannot be modified, even by the person who is doing the logging. See help @witness.

@wizards [all]
$player

Lists connected (or all) wizards.

@wrap [on | off]
$player

If the words you see get cut off at the right edge of the screen, this means that you are either using telnet, or, for some other reason, you don't have word-wrap. @wrap on causes the MOO to perform word-wrapping for you. This command is used in conjunction with the @linelen command, which tells the MOO how long a line may be before it is wrapped to the next line. @wrap off discontinues this behavior; you may need to do this if you switch from using telnet to using a client program. Typing @wrap with no arguments will tell you whether word-wrapping is currently on or off.

'<player> <text>
LM player class #8855, LM player class #33337

This command is a short cut for the page command. Though not part of the core, it has been ported to various other MOOs. (The version on #33337 permits you to omit <player> to respond to the person who paged you most recently.)

? [<topic>]
$player

This is a short cut for the help command.

!<text>
LM player class #5803

This command lets you vary the forms of your statements if you get tired of beginning everything with your name. If your name is included anywhere in <text>, then <text> is displayed as you typed it in. Otherwise, your name is appended to the end as an attribution. See help !. Here are a couple of examples (with Yib doing the typing):

```
!Two thumbs up!
```

displays:

```
Two thumbs up!      --Yib
```

```
!A coconut cream pie sails into the room and smashes into
Yib's face!
```

displays:

```
A coconut cream pie sails into the room and smashes into
Yib's face!
```

#<string>[.<property>|.parent] [e*xit | p*layer | i*nventory] [for <code>]
$prog

Prints information about the object named by <string>. This is a very powerful shortcut for some of the things that the eval command does. In particular, it enables you to look at properties of an object without having to know its object number in advance. Properties can be chained in sequence. Here are a few examples:

```
#rock.color_list
```
displays the .color_list property of rock.

```
#yib p
```
displays Yib's name and object number.

```
#yib.description p
```
displays Yib's description – foils look-detection.

```
#yib.location p
```
displays Yib's location.

```
#yib.location.description
```
displays a description of Yib's location.

```
#yib.location.owner p
```
displays who is the owner of Yib's location.

See help #.

+`<player> <text>`
LM player class `#5803`

This is the remote-emote command. It lets you display `<text>` to `<player>` as if it were an emote, but you do not have to be in the same room with em.

=`[<text>]`
LM player class `#7069`

Responds with `<text>` to the player who most recently paged you. Used with no argument, displays the player to whom you are ready to respond.

Appendix B – Verbs in $Utils Packages

$building_utils

```
:make_exit
:set_names
:recreate
:parse_names
:audit_object_category
:object_audit_string
:"do_audit do_prospectus"
:do_audit_item
:size_string
```

$byte_quota_utils

```
:initialize_quota
:init_for_core
:adjust_quota_for_programmer
:bi_create
:enable_create
:disable_create
:parse_create_args
:"creation_permitted
  verb_addition_permitted
  property_addition_permitted"
:all_characters
:display_quota
:get_quota
:charge_quota
:reimburse_quota
:set_quota
:get_size_quota
:display_quota_summary
:quota_remaining
:preliminary_reimburse_quota
:value_bytes
:"object_bytes object_size"
:do_summary
:summarize_one_user
:recent_object_bytes
:measurement_task
:can_peek
:can_touch
:do_breakdown
:object_overhead_bytes
:property_overhead_bytes
:verb_overhead_bytes
:add_owned_object
:measurement_task_nofork
:measurement_task_body
:schedule_measurement_task
:task_perms
:property_exists
```

$code_utils

```
:eval_d
:"toint tonum"
:toobj
:toerr
:error_name
:show_object
:show_property
:show_verbdef
:explain_verb_syntax
:"verb_p*erms
  verb_permi*ssions"
:verb_loc*ation
:verb_documentation
:set_verb_documentation
```

:parse_propref
:parse_verbref
:parse_argspec
:prepositions
:short_prep
:full_prep
:get_prep
:_fix_preps
:find_verb_named
:find_last_verb_named
:find_callable_verb_named
:find_verbs_containing
:_find_verbs_containing
:find_verbs_matching
:_find_verbs_matching
:_grep_verb_code
:_egrep_verb_code
:_parse_audit_args
:help_db_list
:help_db_search
:corify_object
:inside_quotes
:verb_or_property
:task_valid
:task_owner
:argstr
:verbname_match
:substitute
:show_who_listing
:_egrep_verb_code_all
:_grep_verb_code_all
:verb_usage
:verb_frame
:verb_all_frames
:move_verb
:move_prop*erty
:eval_d_util
:display_callers
:callers_text
:"set_property_value
 set_verb_or_property"
:owns_task
:dflag_on
:type_str
:dump_properties
:dump_preamble
:dump_verbs

$command_utils

:object_match_failed
:"player_match_result
 player_match_failed"
:read
:read_lines
:yes_or_no
:read_lines_escape
:suspend
:running_out_of_time
:suspend_if_needed
:dump_lines
:explain_syntax
:do_huh
:task_info
:init_for_core
:kill_if_laggy
:validate_feature

$convert_utils

:"dd_to_dms dh_to_hms"
:"dms_to_dd hms_to_dh"
:rect_to_polar
:polar_to_rect
:"F_to_C degF_to_degC"
:"C_to_F degC_to_degF"
:convert
:_do_convert
:_try_metric_prefix
:_format_units

```
:"K_to_C degK_to_degC"
:"C_to_K degC_to_degK"
:"F_to_R degF_to_degR"
:"R_to_F degR_to_degF"
:_do_value
```

$gender_utils

```
:set
:add
:get_pronoun
:get_conj*ugation
:_verb_plural
:_verb_singular
:_do
:pronoun_sub
```

$list_utils

```
:make
:range
:map_prop*erty
:map_verb
:map_arg*s
:map_builtin
:find_insert
:remove_duplicates
:arrayset
:setremove_all
:append
:reverse
:_reverse
:compress
:sort
:sort_suspended
:slice
:assoc
:iassoc
:iassoc_suspended
:assoc_prefix
:iassoc_prefix
:iassoc_sorted
:sort_alist
:sort_alist_suspended
:randomly_permute
:count
:flatten
:"longest shortest"
:check_nonstring_tell_lines
:reverse_suspended
:_reverse_suspended
:randomly_permute_suspended
  suspended
:swap_elements
:"random_item random_element"
:assoc_suspended
:amerge
:passoc
:setmove
:iassoc_new
:build_alist
```

$lock_utils

```
:init_scanner
:scan_token
:canonicalize_spaces
:parse_keyexp
:parse_E
:parse_A
:eval_key
:match_object
:unparse_key
:eval_key_new
:parse_A_new
```

$match_utils

```
:match
:match_nth
:match_verb
:match_list
:"parse_ordinal_reference
  parse_ordref"
:parse_possessive_reference
:object_match_failed
:init_for_core
```

$math_utils, $trig_utils

```
:xsin
:xcos
:factorial
:pow
:fibonacci
:geometric
:divmod
:combinations
:permutations
:simpson
:parts
:sqrt
:div
:mod
:exp
:aexp
:random
:random_range
:is_prime
:AND
:XOR
:OR
:NOT
:BLFromInt
:IntFromBL
:"gcd greatest_common_divisor"
:"lcm least_common_multiple"
:"are_rel_prime
  are_relatively_prime"
:base_conversion
:norm
:sin
:cos
:tan
:"arcsin asin"
:"arccos acos"
:"arctan atan"
:"deg2rads deg2rad"
:"rads2deg rad2deg"
:precision
:round
:"mean average"
:sum_float
:"sum_int sum"
:rint
```

$matrix_utils

```
:"vector_add vector_sub
  vector_mul vector_div"
:"matrix_add matrix_sub"
:transpose
:determinant
:inverse
:identity
:null
:is_square
:is_null
:is_identity
```

```
:"cross_prod outer_prod
  vector_prod"
:"norm length"
:submatrix
:"dot_prod inner_prod
  scalar_prod"
:dimension*s
:order
:"scalar_vector_add
  scalar_vector_sub
  scalar_vector_mul
  scalar_vector_div"
:subtended_angle
:column
:matrix_mul
:"scalar_matrix_mul
  scalar_matrix_div"
:is_matrix
:is_vector
:"is_reflexive is_areflexive"
:"is_symmetric is_asymmetric"
:"is_transitive
  is_atransitive"
:_relation_result
:is_partial_ordering
```

$object_utils

```
:has_property
:"all_properties all_verbs"
:has_verb
:has_callable_verb
:match_verb
:isa
:ancestors
:ordered_descendants
:contains
:all_contents
:findable_properties
:owned_properties
:property_conflicts
:descendants_with_property_sus
  pended
:locations
:"all_properties_suspended
  all_verbs_suspended"
:connected
:isoneof
:defines_verb
:defines_property
:"has_any_verb
  has_any_property"
:"has_readable_prop*erty hrp"
:"descendants descendents"
:leaves
:branches
:"descendants_suspended
  descendents_suspended"
:leaves_suspended
:branches_suspended
:"disown disinherit"
:accessible_verbs
:accessible_properties
```

$object_quota_utils

```
:initialize_quota
:init_for_core
:adjust_quota_for_programmer
:bi_create
:creation_permitted
:"verb_addition_permitted
  property_addition_permitted"
:display_quota
:"get_quota quota_remaining"
:charge_quota
:reimburse_quota
:set_quota
:preliminary_reimburse_quota
:can_peek
```

```
:can_touch
```

$perm_utils

```
:controls
:apply
:caller
:"controls_prop*erty
  controls_verb"
:_chparent
```

$quota_utils

```
:initialize_quota
:init_for_core
:adjust_quota_for_programmer
:bi_create
:enable_create
:disable_create
:parse_create_args
:"creation_permitted
  verb_addition_permitted
  property_addition_permitted"
:all_characters
:display_quota
:get_quota
:charge_quota
:reimburse_quota
:set_quota
:get_size_quota
:display_quota_summary
:quota_remaining
:preliminary_reimburse_quota
:value_bytes
:"object_bytes object_size"
:do_summary
:summarize_one_user
:recent_object_bytes
:measurement_task
:can_peek
:can_touch
:do_breakdown
:object_overhead_bytes
:property_overhead_bytes
:verb_overhead_bytes
:add_owned_object
:measurement_task_nofork
:measurement_task_body
:schedule_measurement_task
:task_perms
:property_exists
```

$seq_utils

```
:"add remove"
:contains
:complement
:union
:tostr
:for
:extract
:tolist
:from_list
:from_sorted_list
:first
:last
:size
:from_string
:firstn
:lastn
:range
:expand
:contract
:_union
:intersection
```

$set_utils

```
:union
:intersection
:diff*erence
:contains
:"exclusive_or xor"
:"difference_suspended
 diff_suspended"
:equal
:intersection_preserve_case
```

$string_utils

```
:space
:left
:right
:"centre center"
:"columnize columnise"
:from_list
:english_list
:names_of
:from_seconds
:trim
:triml
:trimr
:strip_chars
:strip_all_but
:"uppercase lowercase"
:"capitalize capitalise"
:literal_object
:match
:match_str*ing
:match_object
:match_player
:match_player_or_object
:find_prefix
:index_d*elimited
:"is_integer is_numeric"
:ordinal
:group_number
:english_number
:english_ordinal
:english_ones
:english_tens
:subst*itute
:substitute_d*elimited
:_cap_property
:pronoun_sub
:pronoun_sub_secure
:pronoun_quote
:alt_pronoun_sub
:explode
:words
:word_start
:to_value
:prefix_to_value
:_tolist
:_unquote
:_toscalar
:parse_command
:from_value
:"print print_suspended"
:reverse
:char_list
:regexp_quote
:connection_hostname_bsd
:connection_hostname
:from_value_suspended
:end_expression
:first_word
:common
:"title_list*c list_title*c"
:"name_and_number nn
 name_and_number_list
 nn_list"
 :"columnize_suspended
 columnise_suspended"
 :a_or_an
:index_all
```

```
:"match_stringlist
  match_string_list"
:from_ASCII
:to_ASCII
:abbreviated_value
:_abbreviated_value
:match_suspended
:incr_alpha
:is_float
:inside_quotes
:strip_all_but_seq
```

$time_utils

```
:day
:month
:ampm
:to_seconds
:sun
:from_ctime
:"dhms
  dayshoursminutesseconds"
:english_time
:from_day
:from_month
:dst_midnight
:time_sub
:"mmddyy ddmmyy"
:parse_english_time_interval
:seconds_until_date
:seconds_until_time
:rfc822_ctime
:"mmddyyyy ddmmyyyy"
```

$wiz_utils

```
:set_programmer
:set_player
:set_owner
:set_property_owner
:unset_player
:set_property_flags
:_set_property_flags
:random_password
:queued_tasks
:isnewt
:initialize_owned
:verify_owned_objects
:"connected_wizards
  connected_wizards_unadvertis
  ed"
:"all_wizards_advertised
  all_wizards
  all_wizards_unadvertised"
:rename_all_instances
:missed_help
:show_missing_help
:init_for_core
:show_netwho_listing
:show_netwho_from_listing
:"check_player_request
  check_reregistration"
:make_player
:send_new_player_mail
:do_make_player
:do_register
:do_new_password
:set_owner_new
:boot_idlers
:grant_object
:connection_hash
:newt_player
:unset_programmer
:is_wizard
:expire_mail
:expire_mail_weekly
:check_prog_restricted
:expire_mail_players
:expire_mail_lists
:flush_editors
```

```
:random_wizard                          :get_email_address
:set_email_address
```

Appendix C – Text of "LambdaMOO Takes a New Direction" (LTAND)

LambdaMOO Takes a New Direction

<Wednesday, December 9, 1992>

I'm sorry that what follows is so long, but I want to share the historical context that I perceive for current events and for an announcement, which appears at the end of this message.

I should note at the very beginning that I had planned to put this message together several weeks ago, very soon after I posted my last *social-issues note. I talked about the general idea with the other wizards at that time and I was supposed to draft a message and send it around for approval. It seemed to me tonight, though, that I was procrastinating an awful lot and that I'd better just write it and send it now, while I'm (temporarily) up-to-date on *social.

As a result, the other wizards are seeing all this at the same time as you are; I don't think they're going to be surprised (except to the extent that Haakon finally doing something is always surprising), but y'all should know that they haven't approved this note in advance.

++

Just over two years ago, I sent email to four people that I had met in the very first MOO, written by ghond and run on belch.berkeley.edu. I knew those four people by the same names they use(d) here: Gemba, Gary_Severn, Frand, and ghond. The email explained that I had opened the first "LambdaMOO", running a server derived from ghond's but with enough changes of my own that I gave it a new name.

There weren't very many of us in the early months, of course. Each of us pretty much knew everyone else and the only bureaucracy concerned taking care that the wizards didn't step on each other's toes in making changes to (and, really, creating from scratch) the first core of the LambdaMOO database. We were a very small cadre of friends (I remember the jubilation we all felt the first time there were more than 10 people connected at once) working together to build something that maybe, just maybe, somebody else would find interesting enough to visit more than once.

By the end of the third month or so, we had the core, the server, and the documentation in sufficiently good shape that we felt OK announcing the existence of LambdaMOO to rec.games.mud and thus inviting the world into our creation. By this time, however, we already had (as I recall) hundreds of players created by people who had heard of us simply by word of mouth. We were beginning to have a community, though it was so small ("How small was it?") that nearly every player who had ever connected had been personally greeted by me. (I know, it must be hard to believe that I used to venture regularly from my den, but it's so.)

We had, I think, already had some discipline problems, even then. I remember a couple of assholes from PSU who came in, changed their names to things I wouldn't want to say in front of my mother, and started cursing at everyone in sight. I remember going to try to talk to them about it, meeting stiff resistance, and finally recycling them in frustration.

After the public announcement, of course, the place took a little leap in popularity. We started seeing a wider variety of people coming through, I stopped being able to greet each new player personally, and we started having disagreements about what was and was not proper conduct here. Eventually, I was approached by a number of players and asked to draft a set of rules for proper MOO behavior. It was felt, by both myself and a number of the other players, that this was a new kind of place, that we had gained some useful experience with how well or badly certain kinds of behavior worked, and that at least some of the lessons we had learned would not be obvious to new users. With a written set of rules, we felt that new players could perhaps learn from our experience and that maybe the amount of friction would be reduced.

Accordingly (and after one of my usual periods of procrastination), I wrote a draft set of rules based entirely (as I recall) on the suggestions made by the players who had made the request. I showed the draft to a bunch of people and asked for their comments on its style, completeness, and correspondence with their impressions of the "right" way of things. After incorporating suggested changes, the first version of "help manners" was publicized in the newspaper; I had, I think, done as good a job as I could of trying to capture the public consensus of that (admittedly early) time.

Perhaps surprisingly, "help manners" worked quite well in reducing the number of incidents of people annoying each other. That society had a charter that reflected the general opinion and social pressure worked to keep the MOO society growing fairly smoothly.

We pretty much stopped growing over the summer of 1991, with a maximum of about 25-30 people commonly connected at once. At the end of the summer, though, as school restarted, we began growing almost alarmingly, with as many as 40 or 45 people often connected at the high points. I recall counting on the order of 350-400 people who had connected in the past week at that time.

As the society grew, so did the work load on the wizards. We were all spending a lot of time looking carefully at what players had built and deciding about requested quota increases, as well as other things, including arbitrating various inter-player disputes. The load of new players (with their understandable but frustrating disinclination to read documentation) and the ever-increasing number of quota requests were leading some of the wizards, including me, to feel stressed out and overworked. It became clear to me that something had to be done to reduce the wizardly workload, so at the very beginning of this year I created the Architecture Review Board, to try to shift some of the burden off of the wizards and onto a larger group of experienced players.

It took some working out, and I'm not saying that it didn't disturb a number of players, but the ARB did eventually relieve the wizards of what had become an intolerable burden. From our standpoint, anyway, it worked very well.

A couple of months later, at the plaintive and repeated requests of the other wizards, I agreed to move the MOO to a "registration" basis, where new players were only created by people sending RL email to one of the wizards. This has also worked to reduce some of the burden on wizards, since it introduced a degree of accountability and a concomitant reduction in certain kinds of disputes and discipline problems. I had resisted registration for months, worried that, among other things, it might stifle the continued growth and evolution of the MOO society.

I needn't have been concerned. The growth has continued and continued, forcing us to come up with new mechanisms and experimental solutions to the inevitable growing pains. We created red-listing, black-listing, and grey-listing. We created the @newt and @toad commands. We tried to block out a lot of people who we thought were causing problems and then stopped trying because it's too hard to be effective at that game. We were even forced to pop the top off of the limit on the number of connections in order to meet the demands of LambdaMOO's growth.

Of course, during this whole time, we were fighting an increasingly losing battle, to control and accommodate and soothe a larger and larger, more and more complex community. We were trying to take responsibility for, now, the behavior and mores of over 800 people a week, connecting from almost 30 countries of the world. We were frustrated, many of the players were frustrated; the center could not hold.

You can probably see where this is leading.

I realize now that the LambdaMOO community has attained a level of complexity and diversity that I've actually been waiting and hoping for since four hackers and I first set out to build this place: this society has left the nest.

I believe that there is no longer a place here for wizard-mothers, guarding the nest and trying to discipline the chicks for their own good. It is time for the wizards to give up on the 'mother' role and to begin relating to this society as a group of adults with independent motivations and goals.

So, as the last social decision we make for you, and whether or not you independent adults wish it, the wizards are pulling out of the discipline/manners/arbitration business; we're handing the burden and freedom of that role to the society at large. We will no longer be the right people to run to with complaints about one another's behavior, etc. The wings of this community are still wet (as anyone can tell from reading *social-issues), but I think they're strong enough to fly with.

There are a number of very important unresolved questions concerning the transition to an out-of-the-nest society:

- What should happen to the ARB and the quota-granting process?
- Who should be making decisions about granting or refusing programmer bits?
- What do we do with the current "help manners"?

and almost certainly a bunch of other things I'm not thinking of right now.

My personal model is that the wizards should move into the role of systems programmers: our job is to keep the MOO running well and getting better in a purely technical sense. That implies, though, that we're responsible for keeping people from getting "unauthorized" access; in particular, we still have to try to keep others from getting wizard bits since the functional integrity of the entire MOO is clearly at risk otherwise.

There are lots of details to be worked out, and I couldn't possibly try to lay them all out here even if I were capable of thinking of all of them in advance, but I am committed to removing the wizards from the social sphere of the MOO *entirely* and *soon*. Haakon, Nosredna, Geust, Slartibartfast, etc. will become technicians who work for the society. Lambda, yduJ, JoeFeedback, Ford, etc. will much more clearly become just another set of players in this community with no more power or moral authority than anyone else.

It's a brave new world outside the nest, and I am very much looking forward to exploring it with the rest of you. To those who have noted that I have the ability to shut down the MOO at any moment, that my finger is, after all, the one on the boot button: you have nothing to fear on that score for the foreseeable future; only an utter fool would put an end to such an exciting social experiment at so crucial a time in its evolution.

I think we're going to have a lot of fun, here... :-)

Haakon the technician and Lambda the lazy proletarian slob

Appendix D – Text of “LambdaMOO Takes Another Direction” (LTAD)

LambdaMOO Takes Another Direction

<Thursday, May 16, 1996>

On December 9, 1992, Haakon posted “LambdaMOO Takes A New Direction” (LTAND). Its intent was to relieve the wizards of the responsibility for making social decisions, and to shift that burden onto the players themselves. It indicated that the wizards would thenceforth refrain from making social decisions, and serve the MOO only as technicians. Over the course of the past three and a half years, it has become obvious that this was an impossible ideal: The line between “technical” and “social” is not a clear one, and never can be. The harassment that ensues each time we fail to achieve the impossible is more than we are now willing to bear.

So, we now acknowledge and accept that we have unavoidably made some social decisions over the past three years, and inform you that we hold ourselves free to do so henceforth.

1. We Are Reintroducing Wizardly Fiat

====================================

In particular, we henceforth explicitly reserve the right to make decisions that will unquestionably have social impact. We also now acknowledge that any technical decision may have social implications; we will no longer attempt to justify every action we take.

Players will still have a voice, however. Your input is essential. We will keep our existing institutions for now, with the modifications described below, but we encourage you to develop ideas for replacing these institutions (as will be described in section 2).

a. Petitions

The petition system will remain in its current form, with the following change:

In cases where difficulties arise that were unanticipated by the vetting process, we reserve the right to re-interpret and/or explicitly veto any clause of any passed ballot.

We will continue to vet petitions, in order to minimize the use of ballot veto, and we will continue to do so in terms of the existing vetting criteria in most cases. However, we will not rule out the possibilities of vetting being denied for other reasons, or of the vetting criteria being revised by fiat.

b. Arbitration

We explicitly reserve

- the right to veto any Arbitrator decision, particularly one that significantly impairs the ability of the wizards to do their jobs.
- the right to veto any Arbitration Change Proposal that is clearly not a “minor change” in the spirit of *Ballot:Arbitration (#50392) or that significantly impairs the ability of the wizards to do their jobs.

These may be temporary measures, as we hope to facilitate revision or replacement of Arbitration so that it may more adequately meet the needs of the community.

c. Wizardly Actions with Social Implications

The wizards will no longer refrain from taking actions that may have social implications. In three and a half years, no adequate mechanism has been found that prevents disruptive players from creating an intolerably hostile working environment for the wizards. The LTAND ideal that we might somehow limit ourselves solely to technical decisions has proven to be untenable.

2. Alternatives to Wizards Making Social Decisions

==

We encourage you, the players, to devise new mechanisms that will help minimize the need for the wizards to make unilateral social decisions. Several mechanisms, most notably the Arbitration system, seem less than ideal for the purpose, yet are too deeply entrenched to be changed with the petition system. We would like to try new mechanisms and to enable more radical changes than the current petition system will allow. We would like the players to propose ideas for major new institutions, and ways to select among the proposals. We hope this will introduce a new dynamism to LambdaMOO that will allow us to find better solutions to some of our more fundamental problems.

Similarly, we hope to facilitate an overhaul of the current petition and ballot system if the players want it.

Do keep in mind, though, that we cannot keep LambdaMOO running without the wizards Haakon has selected. “Cyberspace” and “new social reality” rhetoric aside, so long as the MOO is located on a single RL machine at a single RL site subject to RL laws and liabilities, there will be those deemed responsible for the use of that hardware. Part of the need for administrators is also inherent in the LambdaMOO security model and the organization of LambdaCore, while some of this need is a consequence of various quirks of LambdaMOO society (e.g., the correspondence between RL identities and MOO identities needing to remain secret and yet the need for someone to maintain it). While we might consider ways to decentralize some of these tasks, the fact remains that we simply can’t decentralize everything. We are still open to your suggestions for ways to decentralize what we can.

Suggestions such as:

- persons not well trusted by Haakon might be granted wizard bits as a result of popular election, or
- we might set up a “wizard machine” to run arbitrary wizardly code with NO human intervention at all

are not acceptable, however. There may be site administrators somewhere who will accept the risks involved in implementing these ideas, but we will not.

3. Rejection of the New Direction?

=================================

We realize that not everyone will agree that this is the best new direction LambdaMOO might take. We don’t doubt that some of the polemics among you will be able to come up with a different slant, e.g. (just to save you some trouble),

wizardly blackmail

military coup

martial law

nuclear terrorism

Some of you may find the new direction so disagreeable that you will consider ways to force an end to the new direction or ways to make the wizards’ lives miserable because of it. Instead of making the use of civil disobedience or wizard harassment be the necessary means for shutting down LambdaMOO, we will accept a *simple majority* decision of the following form:

> Any eligible voter may author a “shutdown” petition. This will be a pre-vetted petition with a specific, fixed wording. Should the petition reach ballot stage (by acquiring the usual signature threshold), a vote will be held to decide whether LambdaMOO should be shut down. If the number of YES (we should shut down) votes equals or exceeds the number of NO (we should not shut down) votes received, LambdaMOO will be shut down after an 8-week grace period. (Note, only one “shutdown” petition may be active at a time.)

Shutdown petitions will be implemented at the earliest opportunity.

4. The New Direction

===================

We hope that LambdaMOO will become a more dynamic and enjoyable place for the wizards and the players. We do not want to discourage lively debate or to deprive players of a voice, and we encourage all of you to develop new ideas, mechanisms, and social policies, so as to minimize the need for direct wizardly social intervention as much as possible.

The Wizards of LambdaMOO

Appendix E – A Compendium of LambdaMOO Ballots

This appendix provides a synopsis of (closed) LambdaMOO ballots, both passed and failed. Ballots require a 2/3 majority to pass, with the exception of *B:Shutdown (#100000), a special petition/ballot which requires only a simple majority to pass. A summary such as this one is naturally a moving target; it is current as of May 10, 2003.

For more information about any ballot, you can log onto LambdaMOO and type `read <ballot>`. Each ballot is also a mailing list, containing discussion from the petition stage through ballot closing.

#50392
Aliases: Arbitration
Title: Arbitration
Author: Grump (#122)
Closed: July 2, 1993
Votes: 270 in favor, 87 opposed, and 115 abstaining.
Status: Implemented

This ballot created the original LambdaMOO Arbitration system. It required the establishment of a registry of volunteer arbitrators, rules for calling a dispute, provision for people to join disputes as "interested parties", a provision for overturning an arbitrator's decision, and a mechanism for making "minor changes" to the dispute system. It also set some time limits for various phases of a dispute.

Much later, the wizards agreed that the legislation-as-written was insufficiently precise, and it probably would not have been vetted later in the life of the petition/ballot system. But it was the first piece of legislation written, and it was impossible to forecast the difficulties that lay ahead, including arguments about what did and did not constitute a "minor change" and abuse of loopholes in the system for sport.

(This ballot was repealed in February, 1999.)

#54055
Aliases: New-Arb, 5
Title: A New Structure For The ARB
Author: Xiombarg (#37636)
Closed: July 11, 1993
Votes: 188 in favor, 71 opposed, and 176 abstaining.
Status: Implemented

This ballot fixed the size of the Architectural Review Board (ARB) at 15 members, and provided for the election of new ARB members. (Previously ARB members had been appointed by the wizards.)

#50098
Aliases: Guest-Mail
Title: A Modification of Guest Mail Access
Author: Hagbard (#36271)
Closed: July 19, 1993
Votes: 219 in favor, 49 opposed, and 127 abstaining.
Status: Implemented

This ballot called for the provision of a mechanism whereby mailing list owners could disallow posts from guests.

#51664
Aliases: public-commentary, amendments
Title: Public Commentary and Amendments
Author: Mickey (#52413)
Closed: October 16, 1993
Votes: 182 in favor, 97 opposed, and 271 abstaining.
Status: Defeated

This ballot called for a mandatory period of public commentary on a petition, during which the petition could not accrue additional signatures. It furthermore stipulated that after vetting, public commentary and a second, affirming signature from the author, a petition would be removed from the author's control (i.e. the author would no longer be able to edit the petition text, and the petition would become "community property". The ballot also specified a formal amendment process.

#34167
Aliases: Time, Time_limitation_for_petitions
Title: Time Limitations for the Petitions Process
Author: Moriah (#50459)
Closed: November 13, 1993
Votes: 415 in favor, 97 opposed, and 183 abstaining.
Status: Implemented

This ballot established expiration periods of 14 days for unvetted petitions, and 90 days for vetted petitions.

#24179
Aliases: 5-percent, 5%
Title: Reduce the Amount of Sigs Required for Ballot to %5
Author: Quinn (#19845)
Closed: December 24, 1993
Votes: 362 in favor, 167 opposed, and 219 abstaining.
Status: Implemented

This ballot changed the number of signatures required to promote a petition to ballot status from 10% of the number of eligible voting populace logging in within the past 30 days to 5% of that number.

#40768
Aliases: guest-booting, gb, boot
Title: @booting of guests
Author: edd (#54917)
Closed: December 30, 1993
Votes: 470 in favor, 136 opposed, and 136 abstaining.
Status: Implemented

This ballot mandated the creation of the `@boot` command, whereby players older than four months could boot bothersome guests off the system. The boot command requires a reason for the booting, and its use is logged. Guests so booted have their site blocked for one hour.

#49201
Aliases: newhelpmanners
Title: New Help Manners
Author: PatGently (#37637)
Closed: January 7, 1994
Votes: 337 in favor, 75 opposed, and 181 abstaining.
Status: Implemented

This ballot instituted `help manners` text as voted-on-by-the-people, and required that the following be displayed to every character logging on for the first time: "If you have not already done so, please type 'help manners' and read the text carefully. It outlines the community standards of conduct, which each player is expected to follow while in LambdaMOO."

#55917
Aliases: @ban, @witness
Title: @ban and @witness
Author: Puff (#1449)
Closed: January 7, 1994
Votes: 295 in favor, 128 opposed, and 157 abstaining.
Status: Implemented

This ballot created two new commands. The `@ban` command provides an easy way for a player to ban another player (or any object) from all of the `@banning` player's rooms by typing in a single command. The `@witness` command provides a mechanism for logging conversations or transactions in a way that can't be faked. This is a credible way to prove that something actually happened, recognizing that logs can easily be counterfeited.

#37175

Aliases: Moderation, moderator, duty
Title: Moderation: Immediate action against harrassers [sic]
Author: Quantum-Vacuum (#53118)
Closed: January 7, 1994
Votes: 217 in favor, 160 opposed, and 182 abstaining.
Status: Defeated

This ballot proposed that players be able to issue a "mayday" signal and receive immediate assistance from a mediator. It proposed that there be a vote every 3 months to rank volunteer mediators for the degree to which they were trusted, and stipulated that this would be the order in which the system would attempt to assign a mediator to a particular "mayday" call.

#25812

Aliases: quota-restructuring, q-r, q-i
Title: A Rational Building Quota System
Author: yduJ (#68)
Closed: January 7, 1994
Votes: 271 in favor, 122 opposed, and 169 abstaining.
Status: Implemented

This ballot instituted byte-based quota in lieu of object-based quota.

#42212

Aliases: fix-ARB-elect, fix-arb, arb-elect
Title: Fixes to the ARB Election Process
Author: Dred (#49925)
Closed: January 30, 1994
Votes: 279 in favor, 36 opposed, and 200 abstaining.
Status: Implemented

This ballot instituted some changes to the ARB election process:

- Candidates must now accept nomination in order for their ARB petitions to be promoted to ballots.
- Candidates may withdraw at any time, including during the elections.
- The nomination period for ARB candidates is shortened to one week.
- Number of votes and abstentions will no longer be displayed until after voting closes.
- Allowable votes on ARB ballots become `yes`, `no`, or `abstain`.
- Calculation of who wins was modified.
- Number of available ARB seats determines number of candidates selected.

#47986
Aliases: ARB-Rules
Title: ARB Voting Rule Restructuring
Author: Xythian (#24436)
Closed: March 3, 1994
Votes: 207 in favor, 90 opposed, and 311 abstaining.
Status: Implemented

This ballot instituted some specific rules for how the ARB would consider quota requests:

- A waiting period before a quota request may be acted upon (i.e. approved or denied).
- A request must be sponsored by an ARB member to be considered.
- Votes will be one of: `yes`, `no`, `abstain`, or `delay`.
- Limits on the amount of quota an ARB member may vote to grant, if `yes`: Not more than the request is for, not less than half the amount the request is for.
- Requests may close either by exceeding a specified time limit, or by accruing enough votes to pass or fail. To pass, a request must get three more `yes` votes than `no` votes. If the request is for more than 100K, then it must get four more `yes` votes than `no` votes to pass.
- After closure of a request, a voting summary will be sent to the requester and to `*Public-ARB`.
- A mechanism was created to automatically grant the quota, so that the ARB actually grants the quota, rather than serving in an advisory capacity to the wizards.
- A mechanism was created by which the ARB may change its own rules.
- The ballot specifies that the rules and any subsequent changes to the rules shall be available to the public at all times.

#57380
Aliases: 57380
Title: Removal of `@addfeatures` on Guests
Author: Dodger (#59267)
Closed: March 5, 1994
Votes: 432 in favor, 162 opposed, and 130 abstaining.
Status: Implemented

This ballot removed guests' ability to add and use feature objects (FOs). It also permanently added the stage talk feature and the social verb core feature to the generic guest so that guests would have the use of these.

#54631
Aliases: @boot2, Add-On-@boot
Title: Adding a second @boot
Author: Ox (#54875)
Closed: March 6, 1994
Votes: 372 in favor, 134 opposed, and 192 abstaining.
Status: Implemented

This ballot created the requirement that a second player must ratify a first player's request to @boot a guest.

#63052
Aliases: qt, tq, transferrable_quota, quota_transfers
Title: Quota Transfers
Author: dr (#7003)
Closed: April 11, 1994
Votes: 264 in favor, 122 opposed, and 137 abstaining.
Status: Implemented

This ballot enabled players to transfer unused quota among themselves. Quota transfers are public, posted to the mailing list, `*Quota-Transfer-Log`.

#75104
Aliases: mpg, minimal_population_growth
Title: Minimal Population Growth
Author: legba (#26603)
Closed: May 4, 1994
Votes: 719 in favor, 191 opposed, and 181 abstaining.
Status: Implemented

This ballot established a waiting list for new-player requests, and limited the number of new players created to 5 per day. Additional players may be created on a par with number of players reaped.

New players are assigned numbers rather than names, so that they won't tie up aliases while awaiting their character assignments. They may `@rename` themselves when they first connect.

Players who request characters but never log on are reaped after 30 days.

The creation-limit of 5 per day, the wait list size limit of 500 requests, and the time periods of expiration may be changed at the wizards' discretion, though the wizards are requested to notify the public if they do, in fact, make such changes.

#60535
Aliases: AntiRape, Anti-Rape, StopRape, Stop-Rape, Virtual_Rape_Consequences, p:a, a
Title: Virtual Rape Consequences
Author: Nancy (#57980)
Closed: July 12, 1994
Votes: 541 in favor, 379 opposed, and 167 abstaining.
Status: Defeated

This ballot proposed a definition of "rape" within the context of MOO, and also definitions for "acts" and "speech", and furthermore stipulated that the act of rape be punishable by `@toading` (permanent expulsion from the community).

#22581
Aliases: Loopholes
Title: Closing Loopholes in Petition Timeouts
Author: Lambda (#50)
Closed: July 14, 1994
Votes: 462 in favor, 78 opposed, and 335 abstaining.
Status: Implemented

This ballot revised the time limits for petitions and ballots to fix the fact that under the previous limits, there were some loopholes such that some petitions would never expire. The new limits are as follows:

- If a petition has no signatures, then it also has no expiration date.
- If a petition has signatures and vetting has not been requested, then it expires 14 days after the last moment when there were no signatures.
- If vetting has been requested and neither granted nor denied, then no new signatures may be gathered on the petition.
- If a petition has been vetted and has not become a ballot, then it expires 90 days after vetting was granted.
- If a petition has been denied vetting, then its signatures are removed.

The ballot included language to set limits for petitions that currently existed at the time.

#68149
Aliases: Abuse, NoRape, Ab, NR
Title: Abuse by Any Other Name...
Author: Linnea (#58017)
Closed: July 28, 1994
Votes: 522 in favor, 211 opposed, and 232 abstaining.
Status: Implemented

This ballot called for the following text to be added to `help manners` under the section called "Don't Abuse Other Players":

```
* Sexual harassment (particularly involving unsolicited acts which
       simulate rape against unwilling participants).  Such
       behavior is not tolerated by the LambdaMOO community.  A
       single incidence of such an act may, as a consequence of
       due process, result in permanent expulsion from LambdaMOO.
```

The ballot furthermore confirmed (through passage) that the community thinks that expulsion is within the scope of reasonable penalties for an act of this kind.

#55789

Aliases: 1-Month-Booting, 1mb, 1, 3, pt, 3mb, Sign_Me!!!
Title: Reduce The @Booting Age To 1 Month
Author: Stetson (#65101)
Closed: July 31, 1994
Votes: 315 in favor, 367 opposed, and 228 abstaining.
Status: Defeated

This ballot proposed the following:

- Extend `@booting` privileges to all players eligible to vote (on LambdaMOO). In the event the voting age changed, the `@booting` age would have changed alongside it.
- Increase the time limit for `@booting` from 2 minutes to 5 minutes.
- Require that a `@witness` log be taken prior to a @booting.
- Players involved in `@booting` a guest should be prepared to defend their action in mediation.
- The guidelines for whether a `@boot` was an acceptable action shall be simple, relying upon the common sense of all parties involved. These guidelines shall be decided upon by mediation precedents.

#37152

Aliases: IRN
Title: Information, not Restricting Newbies (IRN)
Author: Individual (#63209)
Closed: July 31, 1994
Votes: 487 in favor, 160 opposed, and 222 abstaining.
Status: Implemented

This ballot stipulated that at times when the number of people connected was so high that the system was inhibiting additional connection, information would be added to the login screen, identifying other MOOs that people might visit, instead.

It also called for a message to be added to the registration text giving suggestions of other MUDs that might be as enjoyable, and less crowded.

#68461
Aliases: DisbandMediation
Title: We Just Aren't Ready
Author: Sunny (#58292)
Closed: August 17, 1994
Votes: 230 in favor, 433 opposed, and 335 abstaining.
Status: Defeated

This ballot proposed that Mediation/Arbitration as passed by LambdaMOOers (Petition ARB, #50392, that went on to become a ballot, was passed and is now readable in #9221) be rescinded, and that the petitions process be the only legally-recognized way to effect changes to LambdaMOO society.

It also proposed that petitions concerning themselves with "behavioral or mannerly conduct" only be considered "valid" if they could be implemented programmatically.

It also proposed an elected board of 15 players (all at least three months of MOO age) and two wizards, serving in an advisory capacity. The board's charter would be to discuss and then write up a petition listing behaviors that would automatically no longer be an option for players. Special protocols were specified for this petition-by-committee.

It furthermore required that this board of 15 players and 2 wizards collaboratively craft a document that would explain manners, tools (commands) available to the players of LambdaMOO, e.g., `@gag`, `@refuse`, `@refuse move`, `@refuse spam`, etc., in addition to a general policy statement about promoting peace, goodwill, and neighborliness towards others that one meets on LambdaMOO. All players, new and old, would be required to read and sign this document within 24 hours of receiving it in order to continue connecting to LambdaMOO.

#46185
Aliases: guest-registration, g-registration, g-r, gr
Title: Guest Registration
Author: Mickey (#52413)
Closed: August 31, 1994
Votes: 289 in favor, 272 opposed, and 155 abstaining.
Status: Defeated

The stated purpose of this petition was "to reliably associate guests with RL identities." Therefore:

- The ability to do `connect guest` would be removed.
- Secondary player creation would be streamlined, no longer requiring the intervention of a human typist.
- A command that could be typed at the login screen would be created which would enable people to request a guest for a specified email address. A "Validated Guest Access" (VGA) password would be sent to the email address for later use.

- Guest e-mail address information would be accessible by the wizards and could legally be turned over to a mediator in the event that the guest is disputed. Mediators' powers would extend not only to that person in their guest identity but also to any primary or secondary characters held by the person at that e-mail address.
- `@booting` of guests would be modified only to affect connections from a particular email address and not an entire site.

#59983
Aliases: mooerofthemonth, motm
Title: MOOer of the Month Proposal
Author: Foobies (#23371)
Closed: October 4, 1994
Votes: 540 in favor, 313 opposed, and 287 abstaining.
Status: Defeated

This ballot proposed the establishment of the award, MOOer of the Month, the criteria for eligibility to receive the award, and a process for determining the winner. (Author's note: This ballot is particularly delightful to read.)

#7887
Aliases: Terms
Title: Term Durations for ARB Members
Author: Quinn (#19845)
Closed: October 20, 1994
Votes: 488 in favor, 95 opposed, and 450 abstaining.
Status: Implemented

This ballot set the term limit for ARB members to one year, placing no limits on serving consecutive terms. There are provisions to hold elections twice a year, and for what to do if a member resigns or is otherwise removed from office in the middle of eir term.

#11351
Aliases: Speed
Title: Speed Up the Petitions Process
Author: Quinn (#19845)
Closed: October 27, 1994
Votes: 449 in favor, 172 opposed, and 407 abstaining.
Status: Implemented

This ballot changed the number of signatures required to bring a petition to ballot to 10% of the average of all votes for and against all previous closed ballots, not to be less than 50.

#4430
Aliases: GuestFO
Title: FO Owner may OK Guest use
Author: cARRoT (#47498)
Closed: November 11, 1994
Votes: 517 in favor, 229 opposed, and 226 abstaining.
Status: Implemented

The ballot empowered Feature Object owners to make feature objects okay for guest use. It contains a provision to prevent certain features from being marked as okay, and for preventing specified players from marking their FOs as okay.

#54635
Aliases: NoNewt
Title: NoNewt
Author: Kilik (#2819)
Closed: November 11, 1994
Votes: 272 in favor, 140 opposed, and 451 abstaining.
Status: Defeated

This ballot attempted to remove a particular mediator's verb which enabled that mediator to newt two specified players for up to one year (as a result of a dispute). The measure was crafted as a ballot because of jurisdictional limitations within the arbitration system.

#6887
Aliases: Banish_Sunny_from_our_MOO, toadsunny
Title: @toad Sunny
Author: Euphistopheles (#50222)
Closed: November 11, 1994
Votes: 252 in favor, 519 opposed, and 315 abstaining.
Status: Defeated

This ballot called for the player then known as Sunny (#58292), along with all known secondary characters, to be `@toaded` and the typist prohibited from having a character on LambdaMOO.

#80344
Aliases: namechange, nc
Title: The Dawning of A New Era
Author: Foobies (#23371)
Closed: November 12, 1994
Votes: 168 in favor, 586 opposed, and 194 abstaining.
Status: Defeated

This ballot proposed that the welcome screen be changed to read, "* Welcome to FoobiesMOO *". It furthermore suggested that the MOO incorporate sound so we at least hear Muzak during the periods of lag.

#44060
Aliases: P-R, Petition-Restrictions, PR
Title: Petition Restrictions
Author: Mickey (#52413)
Closed: November 15, 1994
Votes: 259 in favor, 231 opposed, and 345 abstaining.
Status: Defeated

This ballot proposed to prohibit future petitions that would target individual players.

#78600
Aliases: No-singling-out-individuals, n
Title: No-individuals
Author: X'iina (#58335)
Closed: November 16, 1994
Votes: 322 in favor, 255 opposed, and 290 abstaining.
Status: Defeated

This ballot stipulated, "No petition which designates an individual player by name or by object number may be vetted."

#9895
Aliases: slow
Title: Slow Down the Petitions Process!
Author: jaime (#35330)
Closed: November 21, 1994
Votes: 293 in favor, 254 opposed, and 328 abstaining.
Status: Defeated

This ballot proposed that the number of signatures required to bring a petition to ballot be 25% of the average of all votes for and against all previous closed ballots, not to be less than 100.

#37489
Aliases: Guts
Title: I Wouldn't Vote Yes on This if I Were You
Author: Foobies (#23371)
Closed: November 27, 1994
Votes: 182 in favor, 403 opposed, and 362 abstaining.
Status: Defeated

The stated purpose of this ballot was to achieve the lowest Yes/No ratio in ballot history.

#81387
Aliases: unique, nomonames, Use-Imagination
Title: Disallow New Char Names as Extension of Existing Chars
Author: Abraxas (#42395)
Closed: November 30, 1994
Votes: 480 in favor, 256 opposed, and 213 abstaining.
Status: Defeated

This ballot proposed disallowing character names with the form of an existing character name with the suffix "-2", "-3", and so forth.

#67528
Aliases: ARBage
Title: Change ARB age to Four months
Author: Gryndel (#85344)
Closed: December 2, 1994
Votes: 217 in favor, 355 opposed, and 278 abstaining.
Status: Defeated

This ballot proposed making the minimum MOO age to run for the ARB four months instead of one year.

#71774
Aliases: RegisterGuests, RG, Register, Reg, R
Title: Registration for Guests
Author: Stetson (#65101)
Closed: December 10, 1994
Votes: 434 in favor, 306 opposed, and 192 abstaining.
Status: Defeated

This ballot proposed a password/code plan by which guests would register to an email address and have a code displayed in their descriptions. In the event of a dispute, a guest's email might be accessed by a wizard and potentially matched with that of an established player. Disputes and `@boot` actions against guests would also be narrowed to the particular associated email address rather than a guest's entire site.

#13428
Aliases: 100-Sigs, 100-Signatures, 100
Title: 100 Signatures
Author: Mickey (#52413)
Closed: December 18, 1994
Votes: 391 in favor, 267 opposed, and 374 abstaining.
Status: Defeated

This measure proposed changing the signature threshold for promoting a petition to a ballot to a flat 100 signatures.

#56935
Aliases: RA
Title: RescindArbitration-ThrowOutTheBums
Author: Sunny (#58292)
Closed: January 16, 1995
Votes: 296 in favor, 374 opposed, and 342 abstaining.
Status: Defeated

This ballot called for the repeal of *B:Arbitration (#50392) and would furthermore render all `@newtings` and `@toadings` as a result of arbitration null and void.

#4930
Aliases: Options
Title: Signature Options
Author: Miles (#50636)
Closed: January 20, 1995
Votes: 252 in favor, 278 opposed, and 333 abstaining.
Status: Defeated

This ballot proposed permitting petition signatories to designate either 'yes' or 'no' as part of the act of signing. It proposed to eliminate time limits on petitions, and substituted three necessary criteria for a petition to become a ballot, instead:

- The signature threshold is reached.
- The petition has been vetted.
- The number of "yes" signatures is greater than the number of "no" signatures.

Petitions would be able to gather signatures during the vetting process.

#84988
Aliases: 84988
Title: Beware of Foobies!
Author: Trees_Beatnik (#64541)
Closed: January 22, 1995
Votes: 227 in favor, 430 opposed, and 233 abstaining.
Status: Defeated

This ballot proposed appending the line, "Beware of Foobies...he's very silly." to the welcome screen.

#86488
Aliases: Guest_Description, GD
Title: Guest Description
Author: Gary (#56296)
Closed: January 24, 1995
Votes: 378 in favor, 295 opposed, and 196 abstaining.
Status: Defeated

This ballot proposed a method by which guests could optionally be associated with an existing player. Guest descriptions would either identify the player with which the guest was linked or display the site from which the (unlinked) guest was connecting.

#41859
Aliases: FixBoot
Title: Fix the @boot command, (FixBoot)
Author: Dave (#77480)
Closed: February 8, 1995
Votes: 634 in favor, 50 opposed, and 205 abstaining.
Status: Implemented

This ballot changed the `@boot` command so that the message notifying the room that a player had initiated the `@boot` sequence and wanted a second would only be displayed to other players eligible to use the `@boot` command, and specifically would not be displayed to guests.

#79422
Aliases: Restrain
Title: Restrain Mail Abusers
Author: Quinn (#19845)
Closed: February 15, 1995
Votes: 316 in favor, 231 opposed, and 332 abstaining.
Status: Defeated

This ballot proposed that a new `@restrain` command be created (similar to the `@refuse` command, with a refusee and a duration argument) having the following effects for the given duration upon the person on which it is used:

E cannot post to any list except those e owns, or the lists of disputes in which e is a disputant.

- E may not send MOOmail to any player.
- E may not call or arbitrate disputes.
- E may not make petitions.
- E may not be nominated for the ARB, or any other publicly elected office.
- Guests from all sites from which e has connected are forbidden from directly using the mail system for purposes other than mailing *Wiz or *Reg. (They may still `@request` characters.)

The command could be issued by wizards only, and only as a result of due process. “Due process” was defined as the Arbitration and Ballot systems.

#88119
Aliases: 99-Sigs, 99, Signatures, Sigs
Title: 99 Signatures
Author: jaime (#35330)
Closed: February 15, 1995
Votes: 372 in favor, 192 opposed, and 293 abstaining.
Status: Defeated

This measure proposed setting to 99 the number of signatures required to promote a petition to a ballot.

#62953
Aliases: NoNewt2, NN2
Title: NoNewt II
Author: Kilik (#2819)
Closed: February 19, 1995
Votes: 300 in favor, 139 opposed, and 456 abstaining.
Status: Implemented

This ballot removed a particular mediator's verb which enabled that mediator to newt two specified players for up to one year (as a result of a dispute). The measure was crafted as a ballot because of jurisdictional limitations within the arbitration system.

#76165
Aliases: ballot-restrictions
Title: No @toading by ballot
Author: HumbertHumbert (#64152)
Closed: February 24, 1995
Votes: 271 in favor, 261 opposed, and 367 abstaining.
Status: Defeated

This ballot proposed to prohibit the `@toading` of an individual by ballot, retroactively.

#54019
Aliases: Registration, GlobalRegistration
Title: Global Registration
Author: PatGently (#37637)
Closed: March 1, 1995
Votes: 499 in favor, 251 opposed, and 265 abstaining.
Status: Defeated

This ballot called for existing characters without registered email addresses to be brought into compliance with the then-current registration requirements for new players, by requiring those players to register and have validated an identified email account.

#85122
Aliases: Extend, xt
Title: Extended Time Limits
Author: Rat (#50816)
Closed: March 24, 1995
Votes: 537 in favor, 145 opposed, and 327 abstaining.
Status: Implemented

This ballot requires that time lost due to system down time be "reimbursed" to 'timed issues' such as ballots, petitions, disputes, ARB-ballots, ARB-petitions, ARB-issues, quota requests, newtings, toadings, and site locks.

#71687
Aliases: SpeedUpFOs&ReduceLag!
Title: Speed Up FOs and Reduce Lag
Author: Kilik (#2819)
Closed: March 25, 1995
Votes: 886 in favor, 31 opposed, and 150 abstaining.
Status: Implemented

This ballot called for feature objects to be cached, thus reducing the ticks required to look them up.

#16109
Aliases: BootFixII
Title: Don't Reassign Booted Guest Names for Five Minutes
Author: Topher (#55250)
Closed: March 27, 1995
Votes: 762 in favor, 70 opposed, and 193 abstaining.
Status: Implemented

This ballot called for `@booted` guests' names not to be reassigned for five minutes after the @booting, to reduce harassment of new guests who happened to draw the same name as a just-booted guest.

#72846
Aliases: reg2, registration2
Title: Registration 2
Author: PatGently (#37637)
Closed: April 5, 1995
Votes: 483 in favor, 235 opposed, and 265 abstaining.
Status: Implemented

This ballot sought to alleviate inequities between new characters, required to register to get their character, and older players who had been grandfathered out of that requirement. It required that unregistered characters become ineligible to do any of the following:

- sign petitions and vote on ballots
- volunteer as arbitrators
- propose or vote on arbitration changes
- act as peer review for arbitration through the use of overturn, bar, and any similar verbs that may be created
- serve on the ARB
- request quota from the ARB
- transfer quota to others

After 30 days, unregistered characters would have their quota set to 0. They could still register after that point, but the quota would not be restored.

Registration, which could be done at any time, restores voting rights and the other above-listed eligibilities to characters otherwise eligible.

Registration information shall be available only to wizards, who may not release it even at arbitrator request. (There had previously been some lay registrars, appointed by the wizards.)

#13218
Aliases: Progbit-regulation, prog-r
Title: Progbit-regulation
Author: Cable (#50066)
Closed: April 7, 1995
Votes: 373 in favor, 302 opposed, and 284 abstaining.
Status: Defeated

This ballot proposed that in order to receive a programmer bit, one would have to be one month old and have read `help manners`.

#50190
Aliases: TTONTT
Title: ToToadOrNotToToad
Author: Sunny (#58292)
Closed: April 14, 1995
Votes: 241 in favor, 466 opposed, and 282 abstaining.
Status: Defeated

This ballot proposed that anyone who authored a ballot to `@toad` someone would be `@toaded` emself if that ballot failed.

#87664
Aliases: NotifyVotersOfArbProposals
Title: *P:NotifyVotersOfArbiProposals
Author: JohnBoy (#85460)
Closed: April 15, 1995
Votes: 459 in favor, 125 opposed, and 316 abstaining.
Status: Implemented

The implementation of the arbitration system included a mechanism for making “minor changes” to the arbitration system. This ballot called for all eligible voters to be notified of new arbitration proposals that were up for consideration.

#17532
Aliases: Raise_Quota, Quota
Title: Raise Quota to 75K each
Author: Bats! (#91438)
Closed: April 17, 1995
Votes: 625 in favor, 320 opposed, and 169 abstaining.
Status: Defeated

This ballot proposed granting an additional 25K of quota to all current non-secondary characters, and increasing the initial amount of quota given to new characters from 50K to 75K.

#82103
Aliases: ChangeArbitration
Title: Changing Arbitration
Author: QUARTlow (#87310)
Closed: April 22, 1995
Votes: 218 in favor, 207 opposed, and 434 abstaining.
Status: Defeated

This ballot proposed dismantling the arbitration change system and requiring that all further changes to the arbitration system be accomplished using the petition/ballot system, instead.

#91099
Aliases: Hate-Crime, hate
Title: Hate-Crime
Author: Tapu (#90997)
Closed: April 25, 1995
Votes: 523 in favor, 271 opposed, and 207 abstaining.
Status: Defeated

This ballot proposed adding the following text to `help manners`:

```
Race Hate in the public areas is not tolerated by the
LambdaMOO community, and may be grounds for @toading, as a
consequence of due process.
```

#4223

Aliases: Patch-Arbitration-Loopholes, Patch_Arbitration_Loopholes, P-A-L, P_A_L, PAL, TT
Title: Patch Arbitration Loopholes
Author: Mickey (#52413)
Closed: April 26, 1995
Votes: 339 in favor, 166 opposed, and 354 abstaining.
Status: Implemented

This ballot sought to patch various loopholes in the arbitration system which some people had taken advantage of, and to establish that taking advantage of any loophole was unacceptable behavior and itself subject to the arbitration process.

It established the following general principles:

- All characters are subject to the arbitration system, even if they claim not to be.
- Disputes are between typists, not characters. Arbitrators may call for punitive action to be taken against all of someone's characters, not just the one formally involved in a particular dispute.
- No one may exploit multiple characters to beat the system.
- Events that occur off-MOO, or accounts thereof, shall not be considered relevant to any dispute.
- Disputes that have never been formally started shall not count for the purpose of determining conflict of interest status.
- Taking personal advantage of loopholes instead of reporting them is officially an antisocial act.

#13093

Aliases: p-c
Title: Pest Control:@blacklisting frequently-@booted Guests
Author: Naked_Guest (#50453)
Closed: April 30, 1995
Votes: 493 in favor, 263 opposed, and 203 abstaining.
Status: Defeated

This ballot proposed to track `@boot` requests against guest sites, `@blacklisting` sites (i.e. prohibiting guest login from them) if there were 10 or more `@boot` requests against guests from that site within 30 days.

It proposed that a mechanism be provided by which people could request characters without logging in as guests.

People requesting characters from sites @blacklisted through this mechanism would not be added to the waitlist until 14 days after making their initial request.

#82852
Aliases: Beware!
Title: Beware of Moo politics.
Author: Chris-22(politician-beware!) (#86562)
Closed: May 6, 1995
Votes: 304 in favor, 403 opposed, and 253 abstaining.
Status: Defeated

This ballot called for a 3-week modification to a player's description if e wrote a petition pertaining to MOO politics and that petition was vetted. Gilmore (#34435) was designated as arbiter in the event of question as to whether a petition pertained to MOO politics for purposes of this measure.

A petition would be considered to have MOO Politics as its subject if its primary purpose was to enact changes in the rules governing one of the following: petitions, ballots, arbitration, Architecture Review Board, `@toading`, `@newting`.

#76501
Aliases: Quiet-in-the-Coat-Closet, QCC
Title: Quiet in the Coat Closet
Author: Yib (#58337)
Closed: May 8, 1995
Votes: 384 in favor, 381 opposed, and 224 abstaining.
Status: Defeated

This ballot proposed disabling certain verbs in the LambdaMOO coat closet, which would have the effect of simulating each player being the only one present.

#72623
Aliases: QCC2
Title: Quiet-in-#100000!
Author: abstract (#94368)
Closed: May 8, 1995
Votes: 384 in favor, 319 opposed, and 218 abstaining.
Status: Defeated

This ballot proposed creating (or taking by eminent domain) object number #100000, changing players' first connect point to that location, and making that location quiet, to allow guests and new players to read help texts, etc. without noisy interference from others.

#10216
Aliases: Voter-Registration, vr
Title: Register Characters who Intend to Vote
Author: Stetson (#65101)
Closed: May 16, 1995
Votes: 317 in favor, 302 opposed, and 251 abstaining.
Status: Defeated

This ballot proposed requiring players to register in order to be able to vote on ballots or otherwise participate in the LambdaMOO political and arbitration systems (other than as a disputant). The measure directed that records of who voted on which measures would be kept, and specified penalties for voters who were found to have used multiple characters to bypass the one-vote-per-typist principle.

#90968
Aliases: Personal-Mail-Quota, Personal_Mail_Quota, P-M-Q, P_M_Q, PMQ
Title: Personal Mail Quota
Author: Mickey (#52413)
Closed: May 21, 1995
Votes: 467 in favor, 162 opposed, and 259 abstaining.
Status: Implemented

This ballot calls for the mandatory netforwarding and subsequent deletion of MOOmail messages marked as "kept" if the owner of those messages is over quota by more than 5K.

#12309
Aliases: wiffle, bat, melee
Title: Choosing Justice
Author: darkrider (#7003)
Closed: May 31, 1995
Votes: 345 in favor, 376 opposed, and 268 abstaining.
Status: Defeated

This ballot proposed that an alternative to the Arbitration system be added, in which players would be issued plastic wiffle bats with which to whap each other in lieu of filing disputes. It specified a system of points, damage, and healing; the penalty for accumulating more than a certain number of points would be a 24-hour newting. The ballot also specified procedures for the cases where a person registered under the arbitration system and a wiffler came into conflict, and vice versa, and for switching the system under which one was registered.

#72724
Aliases: The-Linen-Closet, TLC, LC, The_Linen_Closet
Title: A Quiet Starting Place for Guests and New Players
Author: Yib (#58337)
Closed: June 2, 1995
Votes: 614 in favor, 153 opposed, and 178 abstaining.
Status: Implemented

This ballot established The Linen Closet (#47726) as the guest connection point and default home for new players. The Linen Closet differs from The Coat Closet (#11) in that it simulates a player being alone, thus providing a quiet place in which to read help text, news, etc. before joining a noisy environment.

#1319
Aliases: Login-Choice, lc-q
Title: ToBeOrNotToBe-Quiet
Author: active (#91798)
Closed: June 6, 1995
Votes: 481 in favor, 209 opposed, and 230 abstaining.
Status: Implemented

This ballot called for guests and new players to be offered a choice between a noisy or a quiet starting point as part of the connection process.

#54577
Aliases: Court, JRB, JuRB, Judiciary, Supreme-Court
Title: Judicial Review Board, a.k.a. The LambdaMOO Supreme Court
Author: Rog (#4292)
Closed: June 15, 1995
Votes: 321 in favor, 226 opposed, and 371 abstaining.
Status: Defeated

This ballot sought to establish an elected Judicial Review Board whose charter would be "to supply the final word on the interpretation of any given law/petition. They [would] also serve as an appeals court to hear various kinds of challenges concerning actions taken by the wizards or various organizations (i.e., executive bodies) like the ARB."

#88677
Aliases: Cool!
Title: I'm OK. You're OK. They're not OK.
Author: crayon (#39390)
Closed: June 17, 1995
Votes: 287 in favor, 578 opposed, and 187 abstaining.
Status: Defeated

This ballot called for an 8-month moratorium on the creation of new players.

#81878
Aliases: lower, lag
Title: Time to lower lag
Author: Avenger (#50204)
Closed: June 18, 1995
Votes: 568 in favor, 266 opposed, and 171 abstaining.
Status: Implemented

This ballot instituted a rule whereby only 2 newbies could be created for every 3 players reaped until the LambdaMOO population decreased to 5000, at which point 1 newbie could be created for every 1 player reaped.

#62341
Aliases: crime_and_manners, c_and_m, cm
Title: Crime and Manners
Author: HumbertHumbert (#64152)
Closed: June 21, 1995
Votes: 457 in favor, 175 opposed, and 258 abstaining.
Status: Implemented

This ballot created a change in the text of `help manners`. New/changed paragraphs were added to the section titled “Don’t Abuse Other Players”. Those paragraphs were, “Hate speech in public areas”, and “General”. Two intervening paragraphs were changed in format but not content.

#91235
Aliases: Nothing
Title: DoNothing
Author: Brack (#90845)
Closed: June 25, 1995
Votes: 251 in favor, 256 opposed, and 319 abstaining.
Status: Defeated

This ballot stipulated that upon passage, nothing would happen, and “Life on LambdaMOO will continue as normal. At least, normal for this MOO.”

#88952
Aliases: ChangeProposals
Title: Voting & validation on simple changes
Author: gru (#122)
Closed: June 25, 1995
Votes: 189 in favor, 144 opposed, and 372 abstaining.
Status: Defeated

This ballot proposed eliminating the then-current Arbitration Change Proposal system in favor of a more generalized change proposal system that had its limits specified (by the ballot) in greater detail.

#28677
Aliases: Social-Ballots, SB
Title: Social Ballot System
Author: GrendelFish (#88093)
Closed: June 26, 1995
Votes: 241 in favor, 177 opposed, and 297 abstaining.
Status: Defeated

This ballot proposed giving petitions/ballots with strictly social content (i.e. not requiring technical action on the part of the wizards) valid legal standing.

#78996
Aliases: no-arbitration, Arbitration-Schmarbitration, SchmArbitration
Title: Repeal Arbitration.
Author: Tchinek (#54886)
Closed: June 30, 1995
Votes: 207 in favor, 310 opposed, and 265 abstaining.
Status: Defeated

The petition called for the repeal of *B:Arbitration and the abolishment of all artifacts generated because of it.

#95947
Aliases: Adjustment, adj
Title: *P:Adjustment
Author: Angharad (#79047)
Closed: July 3, 1995
Votes: 330 in favor, 120 opposed, and 291 abstaining.
Status: Implemented

This ballot restored voting rights (and other rights) to previously-unregistered players who later registered.

#95983
Aliases: Lower_The_Lag, LTL
Title: Lower The Lag Even More
Author: Chris-22 (#86562)
Closed: July 10, 1995
Votes: 172 in favor, 248 opposed, and 443 abstaining.
Status: Defeated

This ballot ostensibly called for the repeal of #81878 (Time to lower lag), whose title touted lowering lag but whose content addressed lowering population. *B:Lower_The_Lag was fielded as an experiment, to try to see how many people voted for petitions based on title alone rather than the content of the ballot's text. "Educated voters" were encouraged to abstain.

#82944
Aliases: 51%, 51, SilentMajority, SM
Title: 51% OrItDoesn't Count, 51%, 51, 51% Must Vote, SilentMajority, SM
Author: Sunny (#58292)
Closed: July 20, 1995
Votes: 192 in favor, 419 opposed, and 191 abstaining.
Status: Defeated

This ballot proposed to nullify all (future) ballots if at least 51% of all eligible voters did not vote either "yes" or "no".

#68925
Aliases: Ch-Mail, ChMail, Change-Mail, C-M
Title: Change Mail - Trim DB
Author: Bats! (#91438)
Closed: July 20, 1995
Votes: 367 in favor, 155 opposed, and 207 abstaining.
Status: Implemented

This ballot changed the default mail option to forward MOOmail to a player's registration email address instead of keeping it on the MOO (one could change this back, manually, at any time). It also created the option of having MOOmail automatically netforwarded before being deleted by the expiration task.

#55541
Aliases: BurnBanHomo, BBH
Title: BurnBanHomo
Author: CaRrOT (#47498)
Closed: August 21, 1995
Votes: 576 in favor, 226 opposed, and 295 abstaining.
Status: Implemented

This ballot called for *Petition:Ban.Homo.Trash (#57440) to be burned. If #57440 had become a ballot, it was to be burned. If #57440 had passed, it was to be rescinded. If #57440 was no longer a petition or a ballot, a new petition of the same name was to be created and burned in effigy.

#11847
Aliases: Quota-Cap, Cap-Quota, qc
Title: Quota-Cap
Author: Profane (#30788)
Closed: September 5, 1995
Votes: 310 in favor, 186 opposed, and 382 abstaining.
Status: Defeated

This ballot sought to identify a maximum db size (that at which the MOO could safely checkpoint), and place a variety of restrictions on new player creation and quota allocation based on the difference between the established maximum size and the actual size of the LambdaMOO database.

#74657
Aliases: MoreAliasesDammit
Title: More Aliases. Dammit.
Author: Quinn (#19845)
Closed: September 10, 1995
Votes: 240 in favor, 397 opposed, and 214 abstaining.
Status: Defeated

This ballot sought to raise the allowed number of player aliases to 50.

#9015
Aliases: Fix-QT, FixQT, Fix-Quota-Transferral
Title: Fix the Quota Transferral Feature
Author: Brack (#90845)
Closed: September 14, 1995
Votes: 361 in favor, 103 opposed, and 301 abstaining.
Status: Implemented

This ballot established a minimum quota transfer amount of 100 bytes, specified that transfers smaller than 10K would be saved up and published collectively in a single MOOmail post (to the public quota transfer log) rather than one post per transfer, and limited to five the number of permitted consecutive transfers from a single player.

#83438
Aliases: email
Title: Email From LambdaMOO
Author: Indite (#93055)
Closed: September 19, 1995
Votes: 338 in favor, 268 opposed, and 243 abstaining.
Status: Defeated

This ballot stipulated that no email would be sent to a player's off-MOO email address without that player's express consent. This would set the default of the netforward mail option back to `no`. Players would also be prompted as part of the process associated with `@registerme` and `@request`.

#2927
Aliases: RR, Russian, RussianRoulette
Title: Russian Roulette
Author: Artbag (#91408)
Closed: November 10, 1995
Votes: 289 in favor, 788 opposed, and 202 abstaining.
Status: Defeated

This ballot specified that each day at approximately noon, a player would be selected at random and newted for 24 hours, with an accompanying public spectacle. System characters, players ineligible to vote, and players who signed the petition would be exempt.

#95555
Aliases: LambdaMOO-Bill-of-Rights, LBoR, LBR, L-B-o-R, L_B_o_R
Title: LambdaMOO Bill of Rights
Author: Mickey (#52413)
Closed: November 15, 1995
Votes: 379 in favor, 269 opposed, and 311 abstaining.
Status: Defeated

This ballot sought to define and establish a set of rights, “so fundamental to the basis of our community as to supersede the effect of simple legislation.” The ballot had five sections: “Rights of Citizens”, “Rights of Wizards”, “Rights of the ArchWizard”, “Definitions of Terms” and “Extra”.

#96171
Aliases: Quorum, Q
Title: Quorum
Author: Sunny (#58292)
Closed: November 21, 1995
Votes: 245 in favor, 327 opposed, and 320 abstaining.
Status: Defeated

This ballot sought to identify and establish a quorum for passage of a ballot. It also specified procedures for notifying players reaching voting age of the existence of the political system, specified the addition of certain help texts, a way to designate oneself as a non-participant in the political system, and called for scheduled election periods for petitions that had accrued enough signatures to become ballots.

#55018
Aliases: No-bribery
Title: No bribing of signatories
Author: Hookleg (#78127)
Closed: November 25, 1995
Votes: 508 in favor, 140 opposed, and 260 abstaining.
Status: Implemented

This ballot prohibits any petition from calling for any change that results in differential treatment between those who sign it and those who do not.

#12797
Aliases: MooRights
Title: MooRights, MR
Author: Boonton (#76209)
Closed: December 6, 1995
Votes: 269 in favor, 337 opposed, and 336 abstaining.
Status: Defeated

This ballot sought to establish three basic rights:

- The Right not to be `@toaded` or `@newted` against one's will.
- The right to free speech.
- The right to `@gag` or ignore people.

The ballot stipulated that in order to have a petition vetted which would violate any of these rights, one would first have to pass a meta-petition obtaining permission to author a petition which would violate these rights.

#33189
Aliases: ffqt, Fix-Fix-QT
Title: Fix Fix-QT
Author: darkrider (#7003)
Closed: December 21, 1995
Votes: 243 in favor, 194 opposed, and 450 abstaining.
Status: Defeated

This ballot sought to set the minimum quota transfer amount at 1 byte, and to abolish in-MOO logging of quota transfers.

#74167
Aliases: A-->J, Arbitration-->Judgement, A->J, A>J, AJ
Title: Change Arbitration to Judgement
Author: Uther_Locksley (#93141)
Closed: December 28, 1995
Votes: 181 in favor, 275 opposed, and 403 abstaining.
Status: Defeated

This ballot sought to rename Arbitration to "Judgement", rename arbitrators to judges, and rename peer reviewers to jurors.

#82371
Aliases: Remarklines.for.Petitions, RFP
Title: A Petition to Add Comment Lines to Petitions
Author: anj (#59447)
Closed: January 16, 1996
Votes: 282 in favor, 150 opposed, and 315 abstaining.
Status: Defeated

This ballot sought to add an option for one-per-player comments to petitions, in addition to the regular posts to the petition mailing list.

#1036
Aliases: Read_it_before_you_sign_it, read
Title: Read
Author: Avenger (#50204)
Closed: January 18, 1996
Votes: 448 in favor, 139 opposed, and 210 abstaining.
Status: Implemented

This ballot created the requirement (implemented as a technical change) that a player read a petition or ballot before signing or voting on it. Exceptions were specified for abstaining on a ballot and for ARB ballots and petitions.

#65449
Aliases: Elected-Judges, Elected_Judges, EJ, E-J, E_J
Title: Elected Judges
Author: Mickey (#52413)
Closed: January 24, 1996
Votes: 185 in favor, 221 opposed, and 264 abstaining.
Status: Defeated

This ballot proposed replacing the Arbitration system with a court system involving nine elected judges. The text of the ballot includes provision for a mailing list, defines the judges' charter, addresses process, voting, and judges' powers, discusses judges' going on vacation, and has sections about appeals, parties to a case, privacy, elections, re-election, impeachment, and the transition from Arbitration to the proposed new system.

#4579
Aliases: @snuff
Title: Allowing for Player to Player Booting
Author: Mack-the-Knife (#47551)
Closed: January 26, 1996
Votes: 261 in favor, 316 opposed, and 167 abstaining.
Status: Defeated

This ballot described and proposed a voluntary and participatory system by which players who opted in would be able to boot one another for a period of three hours.

#4715
Aliases: validity
Title: Petition System Referendum
Author: HumbertHumbert (#64152)
Closed: January 28, 1996
Votes: 280 in favor, 106 opposed, and 229 abstaining.
Status: Implemented

This ballot was a referendum on whether or not the citizens of LambdaMOO accepted the validity of the petition system (which was imposed by Haakon after `LTAND`).

#103918
Aliases: SOS
Title: SaveOurSunny
Author: WriTinG (#73920)
Closed: January 29, 1996
Votes: 160 in favor, 323 opposed, and 224 abstaining.
Status: Defeated

This ballot proposed granting reaper protection to the LambdaMOO character known as Sunny (#58292).

#58647
Aliases: AOL, AmericaOnLudes
Title: America On Ludes
Author: Gilmore (#34435)
Closed: January 30, 1996
Votes: 222 in favor, 350 opposed, and 177 abstaining.
Status: Defeated

This ballot proposed denying future character requests to persons with an email address ending in “aol.com” and denying access to characters and guests from the “aol.com” domain.

#5360
Aliases: Readable
Title: Create Objects with +r (Readable) Flag
Author: Griffen (#93228)
Closed: February 1, 1996
Votes: 305 in favor, 144 opposed, and 213 abstaining.
Status: Implemented

This ballot caused all newly-created objects to be readable (`+r`) by default.

#63854
Aliases: 7A77
Title: The 7’s have it!
Author: P7A77 (#53430)
Closed: February 5, 1996
Votes: 107 in favor, 492 opposed, and 209 abstaining.
Status: Defeated

This ballot proposed that all objects that are not primary characters and whose object number is of the form ‘#x7y77’, where x is a digit between 1-9 and y is a digit between 0-9, would have their ownership transferred, without regard to quota limits, to the ballot’s author.

#80591
Aliases: Proxies, Proxy
Title: Voting Proxies
Author: jaime (#35330)
Closed: February 17, 1996
Votes: 272 in favor, 334 opposed, and 293 abstaining.
Status: Defeated

This ballot proposed a system by which an eligible voter would be able to authorize another eligible voter to vote on eir behalf by proxy.

#57800
Aliases: BirthControl, Eugenics, PopulationCap, Cap, qc2, CondomsForYouAndYours
Title: BirthControl
Author: Profane (#30788)
Closed: February 23, 1996
Votes: 475 in favor, 229 opposed, and 241 abstaining.
Status: Implemented

This ballot created a maximum database size limit of 200,000,000 bytes, and stipulates that no new characters may be created when the database is larger than that size.

#14923
Aliases: Junkmail, JM
Title: Expiring useless messages, which waste space.
Author: abstract (#94368)
Closed: February 28, 1996
Votes: 671 in favor, 32 opposed, and 186 abstaining.
Status: Implemented

This ballot called for the MOOmail lists `*Boot-log` and `*Witness-published-logs` to have their expire times set to six months. Messages older than that are to be deleted.

#63062
Aliases: Comments, DisputeComments, DC
Title: A Change to the Method of Commenting on Disputes
Author: QUARTlow (#87310)
Closed: March 10, 1996
Votes: 237 in favor, 125 opposed, and 400 abstaining.
Status: Defeated

This ballot sought to limit who could post to dispute mailings lists, when they could do so, and the size of the posts that would be permitted.

#15592
Aliases: AddNotice, ANFR
Title: Add a Notice for Researchers to the Welcome Screen
Author: Peri (#86631)
Closed: March 25, 1996
Votes: 573 in favor, 93 opposed, and 215 abstaining.
Status: Implemented

This ballot called for the following text to be added to the LambdaMOO welcome screen:

```
NOTICE FOR JOURNALISTS AND RESEARCHERS:
The citizens of LambdaMOO request that you ask for permission from
        all direct participants before quoting any material
        collected here.
```

#3568
Aliases: InheritFastFO, iffo, inherit
Title: New Players Inherit Fast Lag Reduction FO
Author: Kilik (#2819)
Closed: April 5, 1996
Votes: 568 in favor, 77 opposed, and 197 abstaining.
Status: Implemented

This ballot stipulates that new players shall be born with the lag reduction feature object #26787 (Lag Reduction FO of Godlike Powers) as a feature, and it shall be active.

#64597
Aliases: AprilFools!, AF, Fools
Title: April Fools!
Author: jaime (#35330)
Closed: April 9, 1996
Votes: 575 in favor, 123 opposed, and 176 abstaining.
Status: Implemented

This ballot authorizes the wizards to do just about anything (some restrictions apply) whenever the date is April 1 anywhere in the world, and that they may not be disputed for doing so.

#99623
Aliases: Guest-accountability, GA
Title: Eliminating Anonymous Hate Mail
Author: Hibernian (#63402)
Closed: April 15, 1996
Votes: 405 in favor, 207 opposed, and 155 abstaining.
Status: Defeated

This ballot sought to authorize the release of guest site info to arbitrators in cases where guests were disputed.

#88247
Aliases: 51%_Majority, 51%M
Title: Ballots Need 51% To Pass
Author: GothGrrl (#96823)
Closed: April 15, 1996
Votes: 247 in favor, 332 opposed, and 165 abstaining.
Status: Defeated

This ballot proposed that future ballots require 51% or more of the total Yes + No votes cast in order to pass, instead of the then-current threshold, which required at least twice as many Yes votes as No votes.

#70377
Aliases: FBC, FixBirthControl
Title: Fix *B:BirthControl
Author: Sleeper (#98232)
Closed: April 15, 1996
Votes: 411 in favor, 102 opposed, and 207 abstaining.
Status: Implemented

This ballot mandated that on days when the database size is above the established maximum size limit, slots created by reaped players would be discarded, not saved up for the next time the database fell below the maximum established size limit.

#54235
Aliases: New_Reaping, New-Reaping, N-R, N_R
Title: A New Reaping System
Author: Uther_Locksley (#93141)
Closed: April 15, 1996
Votes: 265 in favor, 237 opposed, and 216 abstaining.
Status: Defeated

This ballot proposed some changes to the reaping process: Players would be able to make wills, indicating how they would like their objects distributed or disposed of. A pool of volunteer reapers would be created. These reapers would be elected. The ballot also stipulated that minor changes could be made to the reaping system via a mechanism similar to the arbitration change system.

#7690
Aliases: MoreJunkmail, MJM
Title: More Junkmail
Author: abstract (#94368)
Closed: May 5, 1996
Votes: 557 in favor, 32 opposed, and 207 abstaining.
Status: Implemented

This ballot specified time limits after which unpublished witness logs shall be deleted, and that unpublished guest logs shall be deleted when the guest logs off.

#97004
Aliases: Court2, JRB2, JuRB2
Title: Judicial Review Board, Revisited
Author: Rat (#50816)
Closed: May 14, 1996
Votes: 311 in favor, 170 opposed, and 316 abstaining.
Status: Defeated

This ballot called for the proposal specified in *B:Court (see above, #54577) to be implemented.

#63904
Aliases: Guest_Accountability-2
Title: Eliminating Anonymous Hate Mail
Author: Hibernian (#63402)
Closed: June 12, 1996
Votes: 452 in favor, 177 opposed, and 210 abstaining.
Status: Implemented

This ballot added a second login screen that guests see when logging in, informing them that their site information may be used in the event of a dispute and that continuing the login process implied consent to its use in this manner. Guests would then be presented with a yes/no prompt asking whether they accepted these terms. The connection would be severed at that point if the guest answered ‘no’. The ballot also called for various help texts to be modified to reflect these changes.

#105876
Aliases: NoGeeks
Title: NoGeeks
Author: Vida_Blue (#84906)
Closed: June 23, 1996
Votes: 192 in favor, 676 opposed, and 147 abstaining.
Status: Defeated

This ballot sought to have added to the text of `help manners` an additional paragraph detailing specific topics of conversation that were to be designated as unmannerly and offensive to players’ sensibilities.

#104668
Aliases: Lay_Registrar, LR
Title: Lay Registrars to take some pressure away from the LambdaMOO Wizards.
Author: Darkson (#100806)
Closed: Thursday, June 27, 1996
Votes: 415 in favor, 193 opposed, and 194 abstaining.
Status: Implemented

This ballot created the elected position of lay registrar. These non-wizard registrars would have access to players’ email addresses and to certain commands that used wizard powers. They would assist the wizards with the task of creating new players.

#98598
Aliases: Adult_Swim, AdltSwm, AdultSwim, Adult-Swim
Title: Wednesday is for oldbies
Author: carrOt (#47498)
Closed: July 2, 1996
Votes: 491 in favor, 302 opposed, and 134 abstaining.
Status: Defeated

This ballot proposed that each Wednesday between 00:00:00 LST and 23:59:59 LST, only non-guest players who were either older than 90-days or older than this ballot would be allowed to connect.

#82382
Aliases: Repeal-Arbitration, Rep, RARB
Title: Repeal Arbitration
Author: notabird (#105807)
Closed: July 5, 1996
Votes: 212 in favor, 244 opposed, and 232 abstaining.
Status: Defeated

This ballot called for the repeal of *B:Arbitration, including the recycling of all dispute objects and the disabling of arbitration-related verbs.

#98087
Aliases: MailingListReform, Mailing_List_Reform, mlr
Title: Mailing List Reform
Author: Profane (#30788)
Closed: July 10, 1996
Votes: 385 in favor, 66 opposed, and 172 abstaining.
Status: Implemented

This ballot established 30 days as the default expire time on all mailing lists, established a maximum expire time of 180 days for non-wizard-owned lists, and made provision for mailing lists to be registered to an email address -- expired messages are sent to that email address (if present) before being deleted from the system.

#62048
Aliases: No_Programmatic_Conflicts, NPC, No-Programmatic-Conflicts, NoProgrammaticConflicts, N_P_C, N-P-C
Title: No Programmatic Conflicts of Interest
Author: Edweirdo (#58468)
Closed: July 17, 1996
Votes: 239 in favor, 72 opposed, and 251 abstaining.
Status: Implemented

This ballot eliminated any programmatic determination of a conflict of interest between various parties to a dispute.

#94350
Aliases: Random-Fixes, rf
Title: Mostly Harmless
Author: Stetson (#65101)
Closed: July 22, 1996
Votes: 388 in favor, 45 opposed, and 182 abstaining.
Status: Implementation begun

This ballot called for a set of minor changes.

- Secondary characters' reading a petition would 'count' for purposes of *B:Read. If a secondary character signs the petition, it is the primary's character whose name will actually appear.
- Arbitration change proposals have to be read before signing, as for petitions.
- The ballot modifies the implementation of *B:Read so that other ways of reading besides typing the actual 'read' command are accepted as substitutes.
- This list of those who have read a petition will no longer be public.
- Players will be provided with the option of typing a one-line command to connect either in the linen closet or the coat closet, rather than having to answer 'noisy' or 'quiet'.
- Unpublished guest witness logs will be saved for 24 hours before being deleted.
- A list would be created for information about arbitration proposals newly up for voting, and about arbitration change proposals which have passed or failed.

#77883
Aliases: VOAB, Vote_on_ALL_barring
Title: Vote on ALL barring
Author: New-Player-11164 (#105699)
Closed: August 1, 1996
Votes: 215 in favor, 90 opposed, and 282 abstaining.
Status: Implemented

This ballot addressed inequities in the way arbitrators had been barred between an old system and a newer one instituted by arbitration change proposals.

#80483
Aliases: Undertakers_and_Executors_-_Elected, U&EE, UEE, New_Reaping_2, n_r2, n-r2
Title: Elected Undertakers and Executors
Author: Peri (#86631)
Closed: August 5, 1996
Votes: 287 in favor, 105 opposed, and 186 abstaining.
Status: Implemented

This ballot created the elected office of reapers. Reapers would be enabled, through a set of special commands, to disburse a soon-to-be-reaped player's objects, and to reap players. All players would be given a command `@will` with which to designate ways they would like their objects to be disposed of. Minor changes were specified, and the ballot included a provision for making minor changes without the requirement of a petition.

#13125
Aliases: ArbiChange, Arbitration_Change_Proposals, AC, ACP
Title: Update and validate ArbiChange
Author: Uther_Locksley (#93141)
Closed: August 13, 1996
Votes: 139 in favor, 91 opposed, and 319 abstaining.
Status: Defeated

This ballot proposed to change some of the procedures for arbitration change proposals, and gave a specific list of those things which might be changed via that mechanism.

#82908
Aliases: NDD
Title: No Disjoint Disputes
Author: Cable (#50066)
Closed: August 21, 1996
Votes: 228 in favor, 74 opposed, and 273 abstaining.
Status: Implemented

This ballot provided a way for multiple disputes between the same pair of players to be dismissed, if the disputes were deemed by the selected arbitrator to be "closely related".

#8926
Aliases: Kill_Lag, KL
Title: Kill The Lag
Author: TunaJesus (#86562)
Closed: August 23, 1996
Votes: 405 in favor, 161 opposed, and 140 abstaining.
Status: Implemented

This ballot called for the mailing list `*social-issues` to be ceremoniously recycled.

#3665
Aliases: FreeLove, RepealBirthControl, RBC, !BirthControl
Title: Repeal *B:BirthControl
Author: Profane (#30788)
Closed: August 24, 1996
Votes: 273 in favor, 217 opposed, and 155 abstaining.
Status: Defeated

This ballot called for the repeal and reversal of *B:BirthControl (#57800) and *B:FBC (#70377).

#78880
Aliases: ctr, Change_The_Rules, rules
Title: Change the @boot age.
Author: psign_*p:ctr (#96823)
Closed: September 17, 1996
Votes: 400 in favor, 317 opposed, and 141 abstaining.
Status: Defeated

This ballot proposed raising to 16 months the age at which one would be eligible to `@boot` guests.

#90702
Aliases: LTAD, LTAD_Consent, LTAD-Consent, Consent, Confidence_Motion
Title: Consent by the population to message 300 on *News
Author: Li2CO3 (#79261)
Closed: September 29, 1996
Votes: 321 in favor, 111 opposed, and 272 abstaining.
Status: Implemented

This ballot called for the text of `LTAD` (LambdaMOO Takes Another Direction) to be incorporated into help text and the history section of the museum. It furthermore established:

- The LTAD declaration is legal.
- The population has shown its confidence in the Wizards.
- The population has shown its consent to LTAD.

#60159
Aliases: apolitical, ap, apol
Title: A Standard Way to Declare Yourself Apolitical
Author: Brack (#90845)
Closed: September 30, 1996
Votes: 415 in favor, 71 opposed, and 183 abstaining.
Status: Implemented

This ballot mandated a standard way to declare oneself to be apolitical. Apolitical players still have the right to author and sign petitions and vote on ballots. Declaring oneself to be apolitical is an indication that one does not wish to be *lobbied* about any pending petitions or ballots.

This ballot was implemented to utilize the syntax of the `@refuse` command. Further information is available in `help apolitical`.

#15184
Aliases: RLR
Title: Repeal the Lay Registrars Ballot
Author: QUARTlow (#87310)
Closed: October 18, 1996
Votes: 245 in favor, 193 opposed, and 260 abstaining.
Status: Defeated

This ballot sought to repeal the ballot which created lay registrars (#104668), and guarantee that no site information would be available to any non-wizard character.

#80213
Aliases: ARBstandards, AS, ARBs
Title: Provide minimum standards for ARB members
Author: Mooshie (#106469)
Closed: November 1, 1996
Votes: 294 in favor, 145 opposed, and 262 abstaining.
Status: Implementation begun

This ballot provides a minimum standard of time/effort expected from members of the Architecture Review Board. That standard is to cast a vote (yes/no/abstain) on at least 40% of the applications that come before the board during a member's term, and to cast an influencing vote (yes or no) on at least 20% of the applications that come before the board during a member's term, with a check half-way through a member's term (six months after being elected). A member who does not meet these standards may not run for re-election the following term.

#27007
Aliases: ResponsibleARB, GoodARB, GARB
Title: Standards of Behavior for Architecture Review Board Members
Author: Profane (#30788)
Closed: December 5, 1996
Votes: 319 in favor, 123 opposed, and 287 abstaining.
Status: Implementation begun

This ballot established that a person must be a programmer to serve on the Architecture Review Board (ARB), and will be removed from the ARB if eir programmer bit is removed either by fiat or arbitration, or if e is `@newted` either by fiat or by arbitration.

#91597
Aliases: TLD, Time_Limit_for_Disputes
Title: Time Limit for Disputes
Author: Shalmaneser (#105194)
Closed: December 13, 1996
Votes: 324 in favor, 71 opposed, and 263 abstaining.
Status: Implemented

The ballot set a general time limit for disputes at 90 days. Disputes older than that would be closed automatically, with a result of “No Action”.

#2866
Aliases: VotingOptions
Title: VotingOption
Author: Farcan (#108472)
Closed: December 21, 1996
Votes: 205 in favor, 287 opposed, and 194 abstaining.
Status: Defeated

This ballot sought to create a voter registration system. Voters would be able to register and unregister at will. Unregistered voters would not be able to author or sign petitions or vote on ballots. It also stipulated that *Ballot:Apolitical would be repealed.

#55186
Aliases: NLR, NoLayRegistrars, PORA
Title: Protect Our Registered Addresses, No Lay Registrars
Author: Sunny (#58292)
Closed: January 10, 1997
Votes: 311 in favor, 227 opposed, and 153 abstaining.
Status: Defeated

This ballot sought to rescind *B:LayRegistrars.

#92378
Aliases: Kill_More_Lag, KML
Title: Kill More Lag
Author: RedFeather (#91419)
Closed: January 16, 1997
Votes: 339 in favor, 141 opposed, and 130 abstaining.
Status: Implemented

This ballot stipulates that when the owner of a mail recpient (e.g. a *list) is more than 50K over quota, new mail may not be sent to that mail recipient. (The text of this ballot is arranged in a particularly humorous format.)

#60398
Aliases: RNCQ
Title: Reduce New Character Quota
Author: Pictwe (#70967)
Closed: January 16, 1997
Votes: 226 in favor, 256 opposed, and 114 abstaining.
Status: Defeated

This ballot proposed reducing the default quota given to new characters to 30,000 bytes (from 50,000).

#50993
Aliases: Fix-Boot-3.14, FixBoot3.14, FixBootIII, Fix-Boot-3.14159265358979323846264
Title: Fix @boot Yet Again
Author: Quiet (#90845)
Closed: January 29, 1997
Votes: 329 in favor, 129 opposed, and 199 abstaining.
Status: Implemented

This ballot provides for a graduated way to block access from an entire domain (rather than just a single site) if a recently-booted guest returns from a similar-but-slightly-different site (e.g. a different terminal in a school's computer lab).

#90908
Aliases: AmendShutdown
Title: Amendments to Shutdown
Author: Hookleg (#78127)
Closed: March 9, 1997
Votes: 335 in favor, 153 opposed, and 209 abstaining.
Status: Implemented

This ballot calls for email to be sent to each player's registration email address in the event that the shutdown ballot passes, informing players of the date of the shutdown and allowing them to retrieve any information they wish.

It furthermore stipulates that should the shutdown ballot ever pass, the first signatory of that ballot shall have eir home set to a set of stocks in the living room (#17) and that “facilities shall be made available to allow the public to express their displeasure in the customary fashion, i.e., throwing rotten fruit/eggs at the subject, drawing silly moustaches on his/her face etc.”

#99036
Aliases: Reduce_the_Database, rdb
Title: Deleting Old Disputes
Author: loree (#59292)
Closed: March 10, 1997
Votes: 453 in favor, 68 opposed, and 156 abstaining.
Status: Implemented

This ballot calls for the deletion of all disputes that closed more than 120 days prior to passage of this measure. (Old disputes are archived at ftp://ftp.lambda.moo.mud.org/pub/MOO/lambda/disputes/.)

#100000
Aliases: Shutdown
Title: Shutdown
Author: #4 (Petitioner)
Closed: March 24, 1997
Votes: 95 in favor, 1406 opposed, and 68 abstaining.
Status: Defeated

This ballot specifies that LambdaMOO will be permanently shut down 8 weeks after passage. (This is a special ballot created as part of the policy statement known as “LambdaMOO Takes Another Direction” (LTAD) and requires only a simple majority to pass. See also `help LTAD`.)

#7606
Aliases: Vacations, V
Title: Vacations and Reaping
Author: Uther_O’Locksley (#93141)
Closed: March 30, 1997
Votes: 381 in favor, 300 opposed, and 167 abstaining.
Status: Defeated

This ballot proposed reaping players after two months (61 days) of inactivity, but providing players with a mechanism to declare themselves to be “on vacation”, which, if set, would extend the reaping period to six months.

#36324
Aliases: define-minor-2
Title: define-minor-2
Author: Cable (#50066)
Closed: April 14, 1997
Votes: 249 in favor, 85 opposed, and 391 abstaining.
Status: Implemented

This ballot provided a formal definition of “minor change” for purposes of determining which things could be changed via the Arbitration Change mechanism. Other, “non-minor” changes could only be made through the petition/ballot process.

#73786
Aliases: ReapWarning
Title: Send message to registered email addresses of inactive players to inform them of their possible reaping.
Author: Darkson (#100806)
Closed: May 16, 1997
Votes: 638 in favor, 148 opposed, and 122 abstaining.
Status: Implemented

This ballot provides for email to be sent to inactive players more than one year old, warning them of their potential reaping if they remain inactive.

#8109
Aliases: Recourse
Title: Lay Registrars are Accountable
Author: JohnBoy (#85460)
Closed: June 3, 1997
Votes: 427 in favor, 103 opposed, and 198 abstaining.
Status: Implemented

This ballot stipulates that a log will be kept of lay registrars’ access to players’ email addresses, that lay registrars will be prompted to provide a reason for such access, and that a copy of the access record will be mailed to the player whose information is accessed.

#7865

Aliases: Amend-Petition-Process, Amend_Petition_Process, APP, A-P-P, A_P_P, A-PP
Title: Amend the Petition Process
Author: Mooshie (#106469)
Closed: June 14, 1997
Votes: 172 in favor, 126 opposed, and 285 abstaining.
Status: Defeated

This ballot proposed the following changes to the petition process:

- Petition authors would have more flexibility in when to submit a petition for vetting, rather than only when ten signatures have been acquired.
- Petitions may acquire signatures while awaiting vetting.
- Petitions have a 90-day expiration period after the most recent edit; petitions do not age while awaiting vetting.
- Petition authors would be provided with a way to disapprove/protest a petition's implementation notes, and to resubmit the petition for vetting and revision of the implementation notes. The author must approve the implementation notes before a petition can be promoted to a ballot.
- Resubmitting a petition for vetting would not erase signatures on the petition. A maximum of two resubmissions would be allowed.

#77822

Aliases: WholeGag, wg
Title: Gag The Whole Site
Author: MugWump (#89069)
Closed: June 24, 1997
Votes: 437 in favor, 220 opposed, and 135 abstaining.
Status: Defeated

This ballot called for a way to `@gag` a player and any guests connecting from that player's site.

#92524

Aliases: nowhere-cleanup
Title: Clean up nowhere
Author: Xia (#95203)
Closed: June 27, 1997
Votes: 493 in favor, 88 opposed, and 136 abstaining.
Status: Implemented

This ballot called for the recycling of old ARB and reaper election petitions and ballots.

#82107
Aliases: MP, Mediation_Period
Title: A Mediation Period for Disputes
Author: Shalmaneser (#105194)
Closed: June 29, 1997
Votes: 228 in favor, 123 opposed, and 306 abstaining.
Status: Defeated

This ballot proposed extending the default duration of a dispute from two weeks to three weeks, and declaring the first week as a "mediation period" during which only the mediator, parties to the dispute, and formally interested parties would be permitted to post to the dispute mailing list.

#9954
Aliases: Social_Security, Soc, Sec, SS
Title: Social Security
Author: Scarab (#33633)
Closed: July 2, 1997
Votes: 589 in favor, 94 opposed, and 107 abstaining.
Status: Implemented

This ballot increased the reaping period from four months to four months plus one month per year of MOO age up to a maximum of one year.

#88116
Aliases: Toad-007
Title: @Toad Bond-007
Author: ZenWombat (#90331)
Closed: July 27, 1997
Votes: 325 in favor, 241 opposed, and 357 abstaining.
Status: Defeated

This ballot proposed that the character then known as Bond-007 (#103634) and all registered secondary characters be `@toaded`, and that no new characters be created or registered to that player's current email address(es).

#11775
Aliases: FinishFinishedDisputes, FFD
Title: Close Mailing Lists of Finished Disputes
Author: Nuveena (#86941)
Closed: August 25, 1997
Votes: 469 in favor, 44 opposed, and 166 abstaining.
Status: Implemented

This ballot stipulates that non-wizard players may no longer send messages to a dispute mailing list once a dispute has been either closed or withdrawn.

#86750
Aliases: MOOicide_Reform, mr, tnataard
Title: MOOicide Reform, or There's Nothing as Tedious as a Reformed Drunk
Author: TMFKAN64 (#110825)
Closed: August 26, 1997
Votes: 344 in favor, 143 opposed, and 210 abstaining.
Status: Implemented

This ballot changed the behavior of the LambdaMOO location known as "The Edge of the World" so that instead of `@recycling` a character, it would `@newt` em for one month less than the current reap period. Jumping off the edge would no longer require recycling all one's possessions or renaming oneself to "`Toad<n>`". Players who are eventually reaped after having jumped off the edge of the world and who subsequently wish to return may do so without special registrar intervention, just as if they had been reaped due to inactivity.

#52691
Aliases: UnlimitAliases, infinalias
Title: Revise Limits on Number of Player Aliases
Author: Quinn (#19845)
Closed: August 31, 1997
Votes: 219 in favor, 351 opposed, and 160 abstaining.
Status: Defeated

This ballot proposed that players be permitted to have as many aliases as their quota allows.

#60137
Aliases: ReaperPolicy, ReapPolicy, RPol, RP
Title: Establish a Policy for Reapers
Author: Peri (#86631)
Closed: September 4, 1997
Votes: 293 in favor, 80 opposed, and 257 abstaining.
Status: Implemented

This ballot established a set of guidelines that Reapers are expected to follow, including:

- a set of ethical guidelines for Reapers
- rules and guidelines for Reaper duties
- requiring active Reapers to read and consent to it
- defining some common terms, to pave the way for better and more thorough help files about Reapers
- allowing changes to the above by Reaper change proposals.

#66322
Aliases: Disallow_@toading_Petitions, dtp
Title: Disallow @toading Petitions
Author: Ox (#54875)
Closed: September 15, 1997
Votes: 259 in favor, 231 opposed, and 229 abstaining.
Status: Defeated

This ballot sought to disallow petitions which called for the @toading of an individual player.

#14802
Aliases: Fewer_Reapers_by_Attrition, Fewer_Reapers, Attrit_Reapers, fr
Title: Reduce Reapers by Attrition
Author: Peri (#86631)
Closed: November 1, 1997
Votes: 365 in favor, 86 opposed, and 197 abstaining.
Status: Implemented

This ballot reduced the number of elected reapers by reducing from six to three the number of reaper positions available in each of the two reaper elections that followed, and stipulating that only three reapers would be elected in all subsequent elections.

#5810
Aliases: COI
Title: Conflict of Interest
Author: Johns (#110288)
Closed: November 4, 1997
Votes: 433 in favor, 94 opposed, and 167 abstaining.
Status: Implemented

This ballot prevents any person holding an elected or appointed office and involved in a dispute with someone from acting in eir official capacity with respect to the person with whom e is involved in a dispute. Official actions of a neutral nature (such as abstentions) are not affected. Examples of elected or appointed offices would be ARB Member, Lay-Registrar, or Reaper.

#84994
Aliases: OPC
Title: One_Per_Customer, OPC
Author: pens (#84567)
Closed: November 5, 1997
Votes: 365 in favor, 157 opposed, and 170 abstaining.
Status: Implemented

This ballot stipulates that no player may hold more than one elected position at a time.

#87693
Aliases: EvilEdge
Title: GilEvilEdge
Author: Evil (#64052)
Closed: November 6, 1997
Votes: 222 in favor, 265 opposed, and 217 abstaining.
Status: Defeated

This ballot proposed to permit the players Evil (#64052) and Gilmore (#34435) to set their homes to West Lambda Street (#40309). (West Lambda Street is also known as the Edge of the World.)

#79608
Aliases: Truckloads_O_Justice, toj, First_Refusal
Title: Give Gilmore First Refusal on Disputes
Author: TMFKAN64 (#110825)
Closed: November 13, 1997
Votes: 148 in favor, 367 opposed, and 167 abstaining.
Status: Defeated

This ballot proposed making Gilmore (#34435) the default arbitrator in any new disputes which did not involve him directly. If Gilmore were to choose not to arbitrate any particular dispute, then the regular mechanism for selecting an arbitrator would take effect.

#2107
Aliases: Repeal_Kill_More_Lag, rkml
Title: Repeal *B:Kill_More_Lag
Author: MutantNemesis (#87103)
Closed: November 13, 1997
Votes: 239 in favor, 194 opposed, and 205 abstaining.
Status: Defeated

This ballot sought to repeal *B:Kill_More_Lag, and remove the restriction on sending mail to a list when its owner is more than 50K over quota.

#35404
Aliases: New-Wizards, NW, new
Title: A New Wizard Each Year (Whether We Need One Or Not)
Author: Yib (#58337)
Closed: November 14, 1997
Votes: 243 in favor, 278 opposed, and 153 abstaining.
Status: Defeated

This ballot sought to inject new blood into the wizardry by formally requesting that Haakon add one new wizard per year whether or not more manpower was actually needed to reduce the wizards' work load.

#77751
Aliases: ropc, Repeal-OPC
Title: Re-allow players to hold more than one elected office at a time
Author: Darkson (#100806)
Closed: November 29, 1997
Votes: 196 in favor, 322 opposed, and 200 abstaining.
Status: Defeated

This ballot sought to re-allow players to hold more than one elected position at a time.

#102897
Aliases: toad-nonsenso, toad_nonsenso
Title: Toad nonsenso!
Author: Fuel (#59658)
Closed: December 4, 1997
Votes: 244 in favor, 214 opposed, and 301 abstaining.
Status: Defeated

This ballot sought to toad the player nonsenso (#112743) and any of his known secondary characters. It also called for the blacklisting of his site.

#3856
Aliases: Hush
Title: Hush
Author: Gary (#110811)
Closed: December 8, 1997
Votes: 352 in favor, 246 opposed, and 188 abstaining.
Status: Defeated

This ballot sought to restrict players to sending only one post per midnight-to-midnight day to petition and ballot mailing lists and limit the length of said posts to 2000 characters or fewer.

#58025
Aliases: BringBackGilmore, BBG
Title: UnNewt Gilmore
Author: Peri (#86631)
Closed: December 9, 1997
Votes: 285 in favor, 329 opposed, and 222 abstaining.
Status: Defeated

This ballot sought to denewt the player Gilmore (#34435). (He was a newt at the time by virtue of having walked off the Edge of the World. His request to be denewted had been declined by the wizards in accordance with *B:MOOicide_Reform.)

#22540
Aliases: Append_Site_Information_To_Guest_Descriptions, info
Title: Append Site Information to Guest Descriptions
Author: Jayturkey (#105276)
Closed: December 21, 1997
Votes: 341 in favor, 351 opposed, and 111 abstaining.
Status: Defeated

This ballot proposed to append guests' connection site information to their descriptions.

#81155
Aliases: Reconcile_MR_and_SS, reconcile, Down_with_Ambiguity!, dwa
Title: Reconcile *B:MOOicide_Reform and *B:Social_Security
Author: Drippy (#109564)
Closed: December 23, 1997
Votes: 332 in favor, 82 opposed, and 229 abstaining.
Status: Still to do

*B:Social_Security increased a player's reap time to the base reap time of four months plus one month per year of MOO age. *B:MOOicide_Reform mandated that jumping off the Edge of the World would newt a person for "one month less than the reap time". This ballot specifies that in the event of MOOicide, the player will be newted for three months and reaped after four months if e does not subsequently log in.

#14067
Aliases: til, marred
Title: Truth in Lending
Author: O.M.I.N. (#97582)
Closed: January 7, 1998
Votes: 342 in favor, 80 opposed, and 203 abstaining.
Status: Implemented

This ballot stipulates that if a player is `@toaded`, the person who did the deed shall be identified on the `*obits` mailing list, and that if a player MOOicides, that will be noted as such (rather than as "reaped", as per *B:MR) on the `*obits` mailing list.

#15810
Aliases: NewMOOicideReform, NMR
Title: NewMooicideReform
Author: Fred_astaire (#112705)
Closed: January 10, 1998
Votes: 205 in favor, 210 opposed, and 175 abstaining.
Status: Defeated

This ballot called for the abolishment of the MOOicide mechanism all together, and stipulated that a player wishing to leave LambdaMOO shall refrain from connecting until eir character is reaped.

#91413
Aliases: bbc, BringBackChoppie
Title: Bring Back Choppie!
Author: Quadric (#105557)
Closed: January 26, 1998
Votes: 235 in favor, 312 opposed, and 274 abstaining.
Status: Defeated

This ballot requested the denewting of the character Choppie! (#110208), who had MOOicided and subsequently had a change of heart.

#71956
Aliases: Rescind_Arbitration, RAR
Title: Rescind Arbitration
Author: Jaybird (#105276)
Closed: February 12, 1998
Votes: 293 in favor, 153 opposed, and 273 abstaining.
Status: Defeated

This ballot called for the repeal of *B:Arbitration, and the undoing of any modifications made to the Arbitration system after its passage as a ballot.

#22772
Aliases: Never_Say_Never, never
Title: Never_Say_Never!
Author: Tapu (#98332)
Closed: March 3, 1998
Votes: 210 in favor, 213 opposed, and 287 abstaining.
Status: Defeated

This ballot specified a process by which an arbitrator who had been barred from arbitrating could apply for reinstatement.

#103954
Aliases: ISV, InnoucousShutdownVotes
Title: Shutdown votes shouldn't affect petition process
Author: Sleeper (#98232)
Closed: March 10, 1998
Votes: 209 in favor, 86 opposed, and 308 abstaining.
Status: Implemented

This ballot stipulates that votes cast on past and future Shutdown ballots (those involving object #100000) shall not affect the number of signatures required to convert petitions into ballots.

#65664
Aliases: CCS
Title: Change Checkpoint Schedule
Author: Selma (#109388)
Closed: March 16, 1998
Votes: 295 in favor, 227 opposed, and 145 abstaining.
Status: Defeated

This ballot sought to change the time interval between automated checkpoints from 24 hours to 23 hours so that it would occur at a different time each day. (System lag increases dramatically during the checkpoint process.)

#95714
Aliases: More_Free_Mailing, MFM
Title: FreeMailing,MassMailing
Author: QUARTlow (#87310)
Closed: March 19, 1998
Votes: 216 in favor, 165 opposed, and 220 abstaining.
Status: Defeated

This ballot proposed that the owners of "public use" mailing lists be permitted to set a list's expire period to 72 hours or less, and that those so doing be exempted from the restrictions specified by *B:Kill_More_Lag (i.e. that if the owner is more than 50K over quota, new messages to that list will not be accepted).

#61372
Aliases: Bring_Back_the_Blender, BBB
Title: Bring Back the Blender: MOOicide with a sharper edge
Author: Holgate (#65396)
Closed: March 24, 1998
Votes: 335 in favor, 155 opposed, and 162 abstaining.
Status: Implementation begun

This ballot called for the restoration of the original, permanent method of MOOiciding by climbing into the Cuisinart in the LambdaMOO kitchen and turning it on. It furthermore called for the Edge of the World to be modified so as to provide a newting of a random interval between one and six days only.

#97656
Aliases: AKA+
Title: 50 Aliases
Author: QuinnGrrl (#19845)
Closed: April 21, 1998
Votes: 241 in favor, 446 opposed, and 125 abstaining.
Status: Defeated

This ballot sought to increase to 50 the maximum number of player aliases allowed.

#51855
Aliases: Fun
Title: Fun
Author: Gary (#110811)
Closed: May 6, 1998
Votes: 232 in favor, 484 opposed, and 132 abstaining.
Status: Defeated

This ballot sought to:

- Remove the text of `help manners`
- Remove Arbitration
- Modify the text of `help theme` to include a pointer to `help fun`
- Add the help topic `help fun` which would state that the purpose of LambdaMOO is to have fun and that anything goes (except expect to be dealt with if your idea of fun is to crack or damage the system).

#67194
Aliases: RAR2, Rescind_Arbitration_Please, RAP, RAP2
Title: Rescind Arbitration Please
Author: Sleeper (#98232)
Closed: June 29, 1998
Votes: 244 in favor, 147 opposed, and 156 abstaining.
Status: Defeated

This ballot sought to rescind *B:Arbitration.

#107750
Aliases: Jury, Something-Completely-Different, SCD
Title: An Alternative to the Arbitration System
Author: Yib (#58337)
Closed: August 25, 1998
Votes: 216 in favor, 123 opposed, and 189 abstaining.
Status: Defeated

This ballot sought to create an alternative to the Arbitration system, consisting of a panel of jurors who would hear, consider, and decide on cases. The ballot specified:

- A method of jury selection
- Term of office
- Tools and accouterments for the jury to use
- A mechanism for tracking cases
- Jurors' privileges and limitations
- An appeals process
- A mechanism to remove jurors

#77315
Aliases: Make_Boonton_Honest
Title: Make Boonton Honest
Author: Downtime (#108986)
Closed: October 16, 1998
Votes: 224 in favor, 281 opposed, and 174 abstaining.
Status: Defeated

This ballot sought to force the player known as Boonton to append the words, "marginally smarter than a monkey" to his signature on all his posts to public lists.

#68943
Aliases: How_Long?
Title: How_Long?
Author: Jaybird (#105276)
Closed: October 28, 1998
Votes: 438 in favor, 52 opposed, and 114 abstaining.
Status: Implemented

This ballot called for the wizards to recycle all petitions that had not been modified for more than nine months and had no signatures, and to set up a mechanism so that petitions that have been denied vetting and remain untouched/unsigned for nine months or more will be recycled. Petitions that belong to characters that have a registered second who is a wizard are exempt, and *B:Shutdown (#100000) is exempt as well. A copy of the to-be-deleted petition's text will be sent to the author of the to-be-deleted petition via MOOmail.

#108101
Aliases: Voting_information_and_Options, VIO
Title: Voting_information_and_Options
Author: Farcan (#108472)
Closed: November 4, 1998
Votes: 250 in favor, 134 opposed, and 113 abstaining.
Status: Defeated

This ballot proposed sending MOOmail to each player when e reached voting age, explaining how to use the petitions process. It furthermore requested that the wizards provide a command by which players might remove themselves from the roster of registered voters and another command for rejoining the roster of registered voters. This command would, among other things, set certain of a player's petition options in specified ways. There would also be a one-line command to see whether a player was a registered voter or not.

#12133
Aliases: repeal, repealing, repeal_petitions, r50
Title: Simple majority to repeal petitions
Author: Sleeper (#98232)
Closed: November 7, 1998
Votes: 179 in favor, 212 opposed, and 123 abstaining.
Status: Defeated

This ballot specified that a petition that solely sought to repeal any other single ballot would only require a simple majority to pass.

#77415
Aliases: Simple_Majority, majority, maj
Title: Simple Majority
Author: TMFKAN64 (#110825)
Closed: November 7, 1998
Votes: 119 in favor, 297 opposed, and 96 abstaining.
Status: Defeated

This ballot stipulated that henceforth any petition becoming a ballot would require a simple majority to pass, not a 2/3 supermajority.

#95136
Aliases: Vendettas
Title: Keep personal vendettas out of the petition system
Author: soup (#110586)
Closed: November 8, 1998
Votes: 276 in favor, 169 opposed, and 85 abstaining.
Status: Defeated

This ballot stipulated that the wizards would no longer vet petitions aimed at affecting one specific typist.

#79238
Aliases: GagGuest, GagGuestSites, gg, ggs
Title: Gagging and Refusing Guests
Author: Nuveena (#86941)
Closed: January 29, 1999
Votes: 404 in favor, 84 opposed, and 81 abstaining.
Status: Implemented

This ballot provides a way by which players can `@gag` all guests from a particular guest's site.

#99022
Aliases: OMT
Title: One More Time!
Author: Jaybird (#105276)
Closed: February 1, 1999
Votes: 276 in favor, 137 opposed, and 128 abstaining.
Status: Implemented

This ballot repealed *B:Arbitration.

#25438
Aliases: Intolerance
Title: Intolerance
Author: Sage (#74894)
Closed: March 11, 1999
Votes: 404 in favor, 110 opposed, and 104 abstaining.
Status: Implemented

This ballot expresses LambdaMOO citizens' support for the wizards' "newting or toading players who have used LambdaMOO to enable, encourage or cause actions which maliciously threaten or cause harm to the Real Life well-being of other typists," and adds a similarly-worded paragraph to the text of `help manners`.

#73986
Aliases: Anonymity
Title: Anonymity
Author: Klaatu (#114081)
Closed: March 13, 1999
Votes: 405 in favor, 94 opposed, and 92 abstaining.
Status: Implemented

This ballot added a paragraph to the text of `help manners` stating that disclosing a player's offline identity without that player's consent "may be considered the worst form of unmannerly behavior and may result in swift, permanent expulsion from LambdaMOO."

#69024
Aliases: UnnecessaryDelay
Title: Unnecessary Delay
Author: JohnBoy (#85460)
Closed: March 28, 1999
Votes: 244 in favor, 136 opposed, and 138 abstaining.
Status: Defeated

This ballot proposed allowing players to sign a petition while it was under review for vetting. Signatures would be erased if vetting was denied, but preserved if vetting was granted.

#29452
Aliases: 40
Title: 40 Signatures
Author: Jaybird (#105276)
Closed: May 9, 1999
Votes: 188 in favor, 169 opposed, and 189 abstaining.
Status: Defeated

This ballot proposed changing the number of signatures required to promote a petition to ballot status to 40, and changing the number of signatures required to promote an ARB nomination petition to ballot status to 40.

#51783
Aliases: Guest-Idling-Limits, Guest-Idle, gidle
Title: Guest Idling Limits
Author: Roebare (#109000)
Closed: May 14, 1999
Votes: 478 in favor, 88 opposed, and 57 abstaining.
Status: Implemented

This ballot instituted the automatic disconnection of guests who have been idle for more than one hour.

#46748
Aliases: ToadWrit
Title: ToadWriting
Author: Bear(tm) (#88110)
Closed: June 3, 1999
Votes: 120 in favor, 143 opposed, and 177 abstaining.
Status: Defeated

This ballot called for the `@toading` of the player WriTinG (#73920) and all known secondaries.

#114559
Aliases: Player-Do_Command, do_command, docmd
Title: Allowing players to hook into $do_command
Author: xmath (#115429)
Closed: July 4, 1999
Votes: 158 in favor, 106 opposed, and 143 abstaining.
Status: Defeated

This ballot proposed allowing players and/or player class owners to process typed commands before the system object processes them.

#61719
Aliases: AD, Arbitration_Day, Lambda's_Bastille_Day
Title: Arbitration Day
Author: Qui-Gon_Jinn (#110777)
Closed: July 7, 1999
Votes: 195 in favor, 126 opposed, and 99 abstaining.
Status: Defeated

This ballot called for an annual MOO holiday to celebrate the repeal of *B:Arbitration, and specified some of the merriment to occur.

#99296
Aliases: NewtBoard
Title: NewtBoard
Author: Legion (#69858)
Closed: July 16, 1999
Votes: 164 in favor, 188 opposed, and 94 abstaining.
Status: Defeated

This ballot proposed establishing an elected board of players who would be empowered to `@newt` players who had egregiously violated the text set forth in `help manners`, and a process by which the populace might communicate its desire to have a player `@newted`.

#72096
Aliases: FQT
Title: *p:Flexible_Quota_Transfer
Author: deLaMer (#111890)
Closed: July 27, 1999
Votes: 170 in favor, 162 opposed, and 108 abstaining.
Status: Defeated

This ballot proposed allowing players to transfer quota on either an absolute or a probationary basis.

#32418
Aliases: NoQT
Title: No Transferring Quota to Players Too Young to Transfer Quota Themselves
Author: Etoile (#113614)
Closed: July 30, 1999
Votes: 290 in favor, 96 opposed, and 73 abstaining.
Status: Implemented

This ballot prohibits the transfer of quota to players who are not old enough to transfer it back, except that a registered secondary character may receive quota from eir primary character regardless of age.

#12460
Aliases: Take_Out_The_Trash, TotT, Trash, $Garbage, Take
Title: Deleting Extra $Garbage
Author: Harry_Potter (#110777)
Closed: August 8, 1999
Votes: 280 in favor, 95 opposed, and 95 abstaining.
Status: Implemented

This ballot called for a one-time deletion of recycled objects from the database so as to bring the total number of recycled objects down to no more than 20,000.

#96112
Aliases: toadshard, toad-shard, toad_shard, ts, lynchshard, lynch-shard, lynch_shard, ls
Title: Toad shard
Author: fifelfoo (#79261)
Closed: August 13, 1999
Votes: 146 in favor, 205 opposed, and 176 abstaining.
Status: Defeated

This ballot called for the `@toading` of the player shard (#117190) and any known secondaries.

#111951
Aliases: 8MailAlias, EightMailAliases
Title: Limit Mailing lists to 8 Aliases
Author: Krate (#47498)
Closed: August 21, 1999
Votes: 356 in favor, 64 opposed, and 79 abstaining.
Status: Implemented

This ballot stipulates that a MOO mailing list may have no more than eight aliases.

#54623
Aliases: Stop_Lady-Dawn's_Scamming, SLDS
Title: Stop Lady-Dawn's Quota Fraud
Author: POSV (#112523)
Closed: August 24, 1999
Votes: 334 in favor, 143 opposed, and 106 abstaining.
Status: Implemented

This ballot called for the character Lady-Dawn (#117023) and any known secondaries to be prohibited from receiving quota transfers from other players, and for all quota previously transferred to her via the `@qt` quota transfer mechanism to be returned to the original donors.

#101306
Aliases: Better_Transfers, FQT2, better, bt
Title: Flexible Quota Transfer v. 2.0
Author: deLaMer (#111890)
Closed: September 3, 1999
Votes: 229 in favor, 107 opposed, and 101 abstaining.
Status: Implementation begun

This ballot calls for the addition of an option for players to transfer quota on a probationary basis.

#107473
Aliases: Wiffle2, w2, wiff2
Title: Wiffling's Second At Bat
Author: lights (#113418)
Closed: September 8, 1999
Votes: 199 in favor, 169 opposed, and 102 abstaining.
Status: Defeated

This ballot proposed establishing a slightly modified version of the system specified in *B:Wiffle (#12309)

#26342
Aliases: Bravely_Gag_Jaybird, BGJINFAFS, BGJ
Title: Bravely Gag Jaybird Ignoring Nipped Fingers and Frantic Squawking
Author: Musketeer (#112067)
Closed: September 14, 1999
Votes: 105 in favor, 261 opposed, and 151 abstaining.
Status: Defeated

This ballot proposed prohibiting the player Jaybird (#105276) from posting to petition and ballot mailing lists.

#106923
Aliases: speedboot
Title: speedboot
Author: mingaloid (#101361)
Closed: September 24, 1999
Votes: 219 in favor, 133 opposed, and 115 abstaining.
Status: Defeated

This ballot proposed a streamlined version of the command used to boot guests from the system.

#37998

Aliases: ToadtheToaders, ttt
Title: Toad the Toaders
Author: Hibernian (#63402)
Closed: September 25, 1999
Votes: 135 in favor, 251 opposed, and 76 abstaining.
Status: Defeated

This ballot proposed that should someone author a petition to `@toad` one or more players, a companion petition to toad the original petition's author would be created at the same time the original `@toad` petition was vetted.

#86159

Aliases: Informed_Quota_Consumers, IQC
Title: Informed_Quota_Consumers
Author: Hobgoblin (#105941)
Closed: September 25, 1999
Votes: 240 in favor, 101 opposed, and 85 abstaining.
Status: Implemented

This ballot specifies that new players who are entitled to receive an initial quota allotment shall receive this quota in two stages. The initial allotment shall be 20,000 bytes, with the remaining 30,000 bytes to be granted after new players have read text that briefly describes quota's significance on LambdaMOO. That text is as follows:

```
LambdaMOO is limited in size; it must stay below 200 megabytes in
        order to function smoothly.  You are sharing this 200
        megabyte environment with over 5000 other players.  When
        each of us starts out here on LambdaMOO, we each begin with
        a quota allowance which is our share of this world.

Quota is required for all builders and programmers who wish to
        create objects on LambdaMOO.  It is a limited resource.  It
        is important for you to recognize this as you withdraw the
        remainder of your initial quota allotment.  Carefully weigh
        your needs when deciding where your quota will be used or
        donated; not everyone has your best interests at heart in
        this regard.  Anything that you wish to build or create in
        the future will require quota.  Spend your quota wisely.
```

#77081
Aliases: Unsend
Title: Unsend
Author: Gear (#104262)
Closed: September 25, 1999
Votes: 240 in favor, 116 opposed, and 76 abstaining.
Status: Implemented

This ballot calls for a mechanism by which players may try to retract MOOmail that has been sent to another player. (There are some circumstances where this is a technical impossibility (e.g. when a player has eir mail options set to netforward MOOmail instead of saving it within the MOO); this ballot of necessity exempts those.)

#102883
Aliases: 16mailaliases, 16
Title: 16 Mail Aliases for Mail Folders
Author: spivak (#105570)
Closed: September 26, 1999
Votes: 194 in favor, 141 opposed, and 88 abstaining.
Status: Defeated

This ballot proposed increasing the numbers of aliases a mailing list may have from eight to sixteen.

#100000
Aliases: Shutdown
Title: Shutdown
Author: #4 (Petitioner)
Closed: October 8, 1999
Votes: 112 in favor, 790 opposed, and 39 abstaining.
Status: Defeated

This ballot specifies that LambdaMOO will be permanently shut down 8 weeks after passage. (This is a special ballot created as part of the policy statement known as "LambdaMOO Takes Another Direction" (LTAD) and requires only a simple majority to pass. See also `help LTAD`.) This is the second time that *B:Shutdown was brought to ballot (see page 353).

#64539
Aliases: Co-Authorship, ca, coa
Title: Allowing Petitions/Ballots to Have Multiple Authors
Author: Shimizu (#116587)
Closed: October 21, 1999
Votes: 252 in favor, 101 opposed, and 117 abstaining.
Status: Implementation begun

This ballot calls for the creation of a mechanism by which more than one player may co-author a petition.

#56428
Aliases: penguins_ahoy!, pa!, pa
Title: penguins ahoy!
Author: snarl (#106787)
Closed: October 22, 1999
Votes: 319 in favor, 196 opposed, and 80 abstaining.
Status: Defeated

This ballot specifies that each year on heyoka's (#109226) official birthday, all guests shall take the form of penguins, e.g., Red_Penguin, Ultraviolet_Penguin, etc. It also requests a few other specific forms of merriment.

#91237
Aliases: Stand_Up_Against_Gilmore_The_Opresor, opresor, op
Title: Stand Up Against Gilmore The Opresor
Author: 'Warhol' (#102178)
Closed: October 25, 1999
Votes: 196 in favor, 209 opposed, and 115 abstaining.
Status: Defeated

This ballot proposed giving the player known as Gilmore (#34435) the opportunity to swear that he will for all future times abstain from discriminating against players based upon their object number. It specified that if Gilmore failed to do this, his player object would be converted to and his possessions transferred to a new player object with a six-digit number.

#64102
Aliases: VI, Voting_Information, Voting_Information
Title: Voting_Information
Author: Farcan (#108472)
Closed: November 2, 1999
Votes: 323 in favor, 78 opposed, and 79 abstaining.
Status: Implemented

This ballot calls for MOOmail to be sent to players upon reaching voting age. The MOOmail would include the following information: how to read, sign, and create petitions, how to submit petitions for vetting, and how to vote on open ballots. It shall have a reference to `help @petition-commands`, have a reference to `help petitions` and have a reference to `help @refuse`.

#1807
Aliases: No-Special-Treatment, No_Special_Treatment, NST, INST
Title: No special treatment for specific characters
Author: Melki (#116569)
Closed: November 4, 1999
Votes: 167 in favor, 207 opposed, and 113 abstaining.
Status: Defeated

This ballot stipulates that no petition may call for any change which singles out any player for ongoing differential treatment (e.g. granting or denying privileges) based upon eir identity alone, except that petitions may assign players to established categories with different privileges (e.g. removing a builder or wizard bit), or expel them from the MOO (`@newting`/`@toading`). (The ballot text gives some examples of acceptable and unacceptable legislation.)

#65388
Aliases: Alt-des, Altd
Title: Legal Self-destruction of Alts
Author: Vorial (#113841)
Closed: November 18, 1999
Votes: 293 in favor, 71 opposed, and 117 abstaining.
Status: Still to do

This ballot specifies that players may unilaterally elect to recycle registered secondary characters.

#49066
Aliases: Last, LastPetition, LastBallot, LastPetition/Ballot, LastPetition-Ballot, LPB, LP, LB, FixLoophole, FixLoopholes, TheUltimateLoopholeFix
Title: The Last Petition/Ballot
Author: Sunny (#58292)
Closed: December 21, 1999
Votes: 109 in favor, 382 opposed, and 65 abstaining.
Status: Defeated

This ballot specified that, if passed, the wizards would dismantle the petition/ballot process.

#85089
Aliases: nmss, No_More_Secret_Seconds, Public_Seconds, ps
Title: No More Secret Seconds
Author: Utterly_Lame (#93141)
Closed: January 20, 2000
Votes: 196 in favor, 248 opposed, and 118 abstaining.
Status: Defeated

This ballot stipulated that illegal secondary characters would be publicly identified if and when caught. It also specified that players would be provided with a way to make the identities of their own registered secondary characters available to the public if desired.

#34281
Aliases: QTR2
Title: Quota Transfers to Registered Seconds.
Author: deLaMer (#111890)
Closed: March 15, 2000
Votes: 201 in favor, 114 opposed, and 127 abstaining.
Status: Defeated

This ballot specifies that players may transfer quota between their own registered secondary characters regardless of MOO age.

#11644
Aliases: bounce_in_the_LR, bounce
Title: @bounce in the Living Room
Author: hhsb (#115393)
Closed: April 24, 2000
Votes: 172 in favor, 229 opposed, and 115 abstaining.
Status: Defeated

This ballot called for a mechanism by which players could “bounce” a targeted player from the LambdaMOO Living Room (#17) for a particular duration of time. Players who had been `@bounced` would not be permitted to return during the next one hour.

#6821
Aliases: SHIP
Title: Seniority Has Its Privileges
Author: Fionaa (#93049)
Closed: April 30, 2000
Votes: 406 in favor, 115 opposed, and 56 abstaining.
Status: Implemented

This ballot specifies that each player will be granted an additional 5000 bytes of quota for every full year of MOO age, up to a specified maximum.

#3076
Aliases: Position-Papers, PP
Title: Take a Stand, One-To-A-Person, Modify it if You Want
Author: Yib (#58337)
Closed: May 3, 2000
Votes: 243 in favor, 69 opposed, and 138 abstaining.
Status: Still to do

This ballot calls for a command or mechanism to be added to the generic petition which would enable a player to submit a single position statement. These position statements shall be kept separately from regular posts to the petition mailing lists. A player may only have one such position statement per petition/ballot, but may modify it or remove it at any time. An additional command shall be provided to read the position statements in chronological order of addition and modification.

#10324
Aliases: Handling_Commands_Early, early_commands, commands, cmds
Title: Handling commands before the parser
Author: xmath (#115429)
Closed: May 9, 2000
Votes: 194 in favor, 124 opposed, and 146 abstaining.
Status: Defeated

This ballot called for players and or player classes to be able to process a command before the parser does. It is similar to *B:Player-Do_Command (#114559), except that by default the behavior would be turned off.

#104009
Aliases: cock, cockatoo, put_the_cock_back_in_cockatoo
Title: That's an awful lot of gagging for such a wee little cock
Author: soup (#110586)
Closed: May 26, 2000
Votes: 288 in favor, 177 opposed, and 78 abstaining.
Status: Defeated

This ballot proposed that the cockatoo (#1479) be modified in such a way that gagging it would only work one out of every three attempts, and that if the attempt to gag it should fail, the player attempting to gag it would have to wait until after someone else's successful gag attempt before being able to try again.

#110694
Aliases: Wiffle3, wiff3, w3
Title: Give Us Wiffling, Dammit
Author: GreyDruid (#110777)
Closed: June 23, 2000
Votes: 90 in favor, 84 opposed, and 28 abstaining.
Status: Defeated

This ballot called for the system specified in *B:Wiffle2 to be implemented, except modified so that no wiffler could be hit more than once every thirty seconds.

#45794
Aliases: Allow_Transfers_to_Registered_Seconds, ATRS, Allow, Allow_Transfers, arts
Title: Allow Transfers to Registered Seconds
Author: deLaMer (#111890)
Closed: August 21, 2000
Votes: 102 in favor, 54 opposed, and 30 abstaining.
Status: Defeated

This ballot proposed to eliminate the 4-month age restriction for transferring quota between registered secondary characters.

#62990
Aliases: Efficient_Registrar_Selection, Reg_Eff, RegEff, Reg_Ef, RegEf
Title: Reform of Registrar Selection Process
Author: Mediocrity (#106189)
Closed: August 27, 2000
Votes: 58 in favor, 74 opposed, and 35 abstaining.
Status: Defeated

This ballot sought to discontinue the periodic election of lay registrars in favor of a system whereby lay registrars would serve for life. It provided for wizardly appointment of new lay registrars in the event an existing one left the MOO or was otherwise found by the wizards not to be performing eir duties to wizardly standards.

#51338
Aliases: drama_queens, dq
Title: No More Melodramatic Mooicides
Author: hhsb (#115393)
Closed: September 11, 2000
Votes: 241 in favor, 111 opposed, and 98 abstaining.
Status: Still to do

*Ballot:Bring_Back_the_Blender stipulates that the cuisinart in the LambdaMOO kitchen shall be modified so as to kill off a character and permanently remove em from the LambdaMOO database. It furthermore stipulates that those jumping off the edge of the world shall endure a newting comparable to that of the Russian Roulette pistol (a period of between two and six days). As a partial implementation of *B:BBB, the wizards began letting people come back early from self-newting off the Edge of the World if they had been gone for at least a week. This ballot calls for that practice to cease until real teeth are actually installed in the Cuisinart (so to speak).

#10326
Aliases: Election
Title: Mock Election on LambdaMOO!
Author: Boonton (#76209)
Closed: September 30, 2000
Votes: 174 in favor, 172 opposed, and 86 abstaining.
Status: Defeated

This ballot called for a mechanism with which to hold a mock U.S. Presidential election on the MOO.

#101738
Aliases: Mail, Mail-Name-DB, mndb, mail-name, name-db
Title: Tastes Great, Less Tickage
Author: GhostDruid (#110777)
Closed: October 26, 2000
Votes: 282 in favor, 50 opposed, and 88 abstaining.
Status: Still to do

This ballot provides for a different, more efficient way of storing MOO mailing lists. It furthermore stipulates that mailing lists will be permitted to have up to 20 aliases each, and that destroyed lists will have their names cached for 60 days.

#68457
Aliases: closet
Title: Alternative starting points
Author: pupa (#118222)
Closed: October 27, 2000
Votes: 92 in favor, 153 opposed, and 71 abstaining.
Status: Defeated

This ballot proposed making the push-button in the linen closet send players to one of four randomly-selected destinations.

#41579
Aliases: More-Choices
Title: More-Choices
Author: Hannibal (#104762)
Closed: November 9, 2000
Votes: 40 in favor, 144 opposed, and 85 abstaining
Status: Defeated

This ballot proposed lowering the number of signatures required for promoting an ARB nomination petition to ballot from 50 to 30.

#86634
Aliases: ATTRS2, Allow_Transfers_To_Registered_Seconds_2
Title: Allow Transfers To Registered Seconds 2
Author: AndroidLust (#109830)
Closed: November 9, 2000
Votes: 144 in favor, 129 opposed, and 78 abstaining
Status: Defeated

This ballot proposed eliminating the 4-month age restriction for transferring quota between primary and registered secondary characters.

#76830
Aliases: aliens
Title: @request to be Abducted by Aliens
Author: RinkRabbit (#103685)
Closed: November 13, 2000
Votes: 225 in favor, 119 opposed, and 68 abstaining
Status: Defeated

This ballot proposed changing the departure message for the `@request` command to depict the requestor as being abducted by aliens.

#86867
Aliases: Notify-Seconds, notify2, n2
Title: Notify People when they are registered as seconds
Author: RussianMob (#117379)
Closed: November 21, 2000
Votes: 227 in favor, 62 opposed, and 77 abstaining
Status: Still to do

This ballot requires the wizards to notify all involved parties whenever they make or alter a registered-second relationship between player characters.

#115295
Aliases: impl
Title: Implementation Notes
Author: Elayne (#118426)
Closed: November 28, 2000
Votes: 119 in favor, 142 opposed, and 97 abstaining
Status: Defeated

This ballot proposed requiring draft implementation notes as an added condition for a petition's vetting.

#116799
Aliases: TITT, TIT
Title: Throw in the Towel
Author: Yib (#58337)
Closed: Friday, December 1, 2000
Votes: 187 in favor, 165 opposed, and 70 abstaining
Status: Defeated

This ballot proposed making the vetting of petitions either automatic or not required, and the implementation of ballots (either passed *or* failed) entirely optional, at the wizards' discretion.

#108571
Aliases: MOOmailreform, mmr, personalmoomail, pmm, kill_even_more_lag, keml
Title: People are Mail Folders Too
Author: spivak (#105570)
Closed: December 23, 2000
Votes: 196 in favor, 112 opposed, and 57 abstaining
Status: Defeated

This ballot proposed prohibiting individual players from receiving MOOmail if they were more than 50,000 bytes over quota.

#14009
Aliases: oldcoc
Title: Restoration of the Cockatoo
Author: graeme (#119071)
Closed: February 17, 2001
Votes: 203 in favor, 179 opposed, and 122 abstaining
Status: Defeated

This ballot proposed that the cockatoo (#1479) be returned to its state before the player Stetson (#65101) was granted ownership of it. It furthermore proposed that if the cockatoo was to remain a resident of the Living Room (#17), then any further development be done on a child object rather than on the original.

#114042
Aliases: laugh
Title: Let's laugh a little less ludicrously
Author: Kobot (#113671)
Closed: March 2, 2001
Votes: 250 in favor, 213 opposed, and 54 abstaining
Status: Defeated

This ballot proposed replacing the "tiresome and unpopular" text of the social verb `laugh` on #40842 (Social Verb Core and Feature Object). The unnecessarily exuberant text, "Munchkin falls down laughing," would have become, simply, "Munchkin laughs."

#52006
Aliases: impl-choice, Lend-A-Hand, opt-in-impl
Title: Lend a Hand with Implementation Notes (If You Choose)
Author: active (#91798)
Closed: April 4, 2001
Votes: 129 in favor, 60 opposed, and 103 abstaining
Status: Still to do

This ballot calls for a mechanism by which players may attach draft implementation notes to a petition prior to vetting, and also a mechanism by which a petition's author may indicate eir first choice among the draft implementation notes submitted.

#7518
Aliases: toadcobot, tc
Title: Toad #117343
Author: fifel (#79261)
Closed: May 1, 2001
Votes: 112 in favor, 227 opposed, and 123 abstaining
Status: Defeated

This ballot called for the `@toading` of the player cobot (#117343).

#104646
Aliases: ToadBlissboi, ToadWriT2, ToadDegage, ToadWriTinG2, TW2, tb
Title: ToadWriTinG II
Author: Benny (#114539)
Closed: May 17, 2001
Votes: 213 in favor, 99 opposed, and 116 abstaining
Status: Implemented

This ballot called for the `@toading` of Blissboi (#117469), all registered seconds, and any identifiable illegal seconds.

#111961
Aliases: rosa
Title: Redistribution of Short Aliases
Author: ObsessionRose (#78546)
Closed: June 14, 2001
Votes: 193 in favor, 254 opposed, and 45 abstaining
Status: Defeated

This ballot proposed limiting players to only one alias shorter than three characters long. It furthermore called for players holding more than one short alias to remove those aliases within one month's time, and stipulated that those who failed to do so would have all short aliases removed from them by the wizards.

#34584
Aliases: Resolved
Title: September 11th Resolution
Author: Boonton (#76209)
Closed: January 3, 2002
Votes: 193 in favor, 216 opposed, and 56 abstaining
Status: Defeated

This ballot proposed designating September 11 as a holiday in remembrance of the terrorist attack on the World Trade Center towers in New York City. It stipulated that elections and ballot voting would be suspended, that wizards would not vet or implement petitions (except this one) on that day, and that a request would be broadcast to all connected players for a moment of silence at 8:45 am, 9:02 am, 9:38 am and 9:45 am, Eastern U.S. time.

#71120
Aliases: Noisy-Default, noisy, nd
Title: Partially Rescind *B:Login-Choice
Author: GrinchDruid (#110777)
Closed: January 12, 2002
Votes: 218 in favor, 61 opposed, and 60 abstaining
Status: Implemented

This ballot abolished the noisy/quiet prompt that had been set up by *B:Login-Choice, and made it so that the (noisy) Coat Closet (#11) was once again the connection location for guests. New players still see the noisy/quiet prompt; guests and new players will have their home set to the Linen Closet (#47726). A lever was installed in each closet, appropriately labeled, that would transport someone to the other closet.

#22653
Aliases: Toad_this_be-otch
Title: TTb
Author: Jerri_Blank (#117367)
Closed: February 28, 2002
Votes: 159 in favor, 130 opposed, and 143 abstaining
Status: Defeated

This ballot called for the @toading of the following characters: Sabastian (#112623), Aglaia (#112632), Phalin (#112640), hiccup (#115676), PKelly (#119005), Hepper (#118091), on the grounds that they were all illegal seconds of Nancy's.

#70573
Aliases: NoBoot
Title: Players will be able to stop guests being booted.
Author: pupuck (#118222)
Closed: July 15, 2002
Votes: 201 in favor, 122 opposed, and 41 abstaining
Status: Defeated

This ballot called for a mechanism that would enable players to stop a guest from being booted.

#68178
Aliases: MTM
Title: Make @Tutorial a More Visible Option for New Guests
Author: Gabaldon (#117345)
Closed: July 27, 2002
Votes: 219 in favor, 46 opposed, and 35 abstaining
Status: Still to do

This ballot called for adding a yes/no prompt to the guest login script that would ask if the guest is "a first-time or fairly new visitor to LambdaMOO who would like assistance in learning basic MOO commands" and, if so, would transport the guest to the tutorial.

#22447
Aliases: Congrats
Title: Two Reserved Characters (Congrats yduJ!)
Author: jaime (#35330)
Closed: September 26, 2002
Votes: 257 in favor, 86 opposed, and 46 abstaining
Status: Still to do

This ballot called for the creation of two new characters in honor and celebration of the RL birth of twins to the wizard yduJ.

#112667
Aliases: guest_timeout
Title: shorten_guest_idle_timeout
Author: habibi_with_wild_rice (#115177))
Closed: December 3, 2002
Votes: 221 in favor, 90 opposed, and 32 abstaining
Status: Implemented

This ballot shortened the idle timeout limit for guests from one hour to fifteen minutes.

Bibliography

Busy, Andrew

1995 *Secrets of the MUD Wizards*. Indianapolis, IN: Sams.net. This book is mostly about MUDs, rather than MOOs. The chapter on MOO programming (contributed by Chris Stacy) is well written and informative.

Curtis, Pavel

1997 *LambdaMOO Programmer's Manual.* Available via ftp from `ftp.lambda.moo.mud.org`, `pub/MOO/ProgrammersManual.txt` and `pub/MOO/ProgrammersManual.ps.Z`. This is the definitive specification of the MOO programming language.

Dibbell, Julian

1998 *My Tiny Life: Crime and Passion in a Virtual World*. New York, NY: Owl Books, Henry Holt & Company, Inc. A memoir (mostly about LambdaMOO).

Haynes, Cynthia (editor) and Holmevik, Jan (editor)

1998 *High Wired: On the Design, Use, and Theory of Educational MOOs*. Ann Arbor, MI: University of Michegan Press. A collection of essays about educational MOOs.

Holmevik, Jan and Haynes, Cynthia

2000 *MOOniversity*. Boston, MA: Allyn & Bacon. Introduces educational MOOs and describes the EnCore MOO platform. Emphasizes the use of MOOs as a tool for collaborative writing.

Index

', 290
!, 291
#, 243, 291
#-1, 153
$
  designator for special objects, 37
  referring to the last element of a list, 141
$container, 37
$limbo, 248
$note, 37
$object_utils, 297
$player_start, 250
$room, 37
$spell, 260
$thing, 37
%N, 116
%s, 116
:, 10
::, 11
;, 125
;;, 164
?, 290
?…| construct, 133
@, 243
  convention (in command names), 27
@abort, 259
@addalias, 259
@addalias#, 259
@adddict, 260
@addfeature, 42, 260
@addlag, 260
@add-notify, 260
@add-owned, 260
@addroom, 45, 260
@addword, 260
@age, 261
@answer, 261
@arb, 261
@arb-ballots, 261
@arb-petitions, 261
@args, 261
@args#, 261
@at, 261
@audit, 83, 262
@ballots, 262
@ban, 34, 262, 313
@banned, 262
@boot, 262, 316, 325
@bug, 262
@build-options, 263
@check, 263
@check-chparent, 263
@check-full, 30, 260, 263
@check-property, 263
@chmod, 263
@chparent, 43, 86, 264
  troubleshooting, 263
@classes, 264
@clearproperty, 264
@clear-tell-filter-hook, 264
@clproperty, 264
@comment, 264
@complete, 264
@contents, 45, 88, 99, 264
@copy, 265
@copy-move, 265
@copy-x, 265
@count, 265
@countDB, 265
@create, 38, 45, 79, 80, 82, 84, 265
  resource-limit-exceeded error, 92
@cspell, 265
@dbsize, 265
@define, 266
@denewt, 266
@describe, 45, 80, 266
@detail, 266
@details, 266
@dig, 45, 79, 80, 84, 266
  exits, 85
  rooms, 84
@disown, 266

@display, 45, 127, 267
@display-options, 267
@dump, 45, 267
@dview, 224
@edit, 267
@edit-options, 267
@egrep, 267
@eject, 32, 34, 40, 268
@eval-d, 268
@examine, 269
@features, 269
@find, 42, 269
@flush-cache, 269
@forget, 269
@forked, 132, 269
@forked-verbose, 270
@forward, 270
@gag, 29, 260, 270
@gaglist, 29, 270
@gag-site, 270
@gag-sites, 270
@gender, 45, 270
@gethelp, 270
@gms, 270
@go, 45, 271
@grep, 271
@gripe, 271
@idea, 271
@join, 45, 272
@keep-mail, 272
@kids, 83, 272
@kill, 151, 272
@killquiet, 272
@known-objects, 272
@last-connection, 272
@lastlog, 272
@linelen, 8
@linelength, 272
@list, 273
@list#, 273
@listdefs, 273
@list-followers, 273
@listgag. *See* @gaglist
@locations, 273
@lock, 31, 88, 273
@mail, 45, 273
@mail-all-new-mail, 274
@mail-options, 45, 274
@make-petition, 274
@measure, 91, 274
@messages, 46, 80, 274
@mode, 274
@more, 8, 275
@move, 40, 275
@netforward, 275
@newmessage, 275
@newt, 275
@next, 45, 275
@nominate, 276
@notedit, 276
@nproperty, 276
@owner, 276
@pagelen, 8
@pagelength, 276
@paranoid, 14, 30, 276
@parent, 276
@parents, 43, 82, 277
@password, 45, 277
@paste, 277
@pasteto, 277
@pc-news, 277
@pc-options, 277
@pedit, 277
@petition-options, 277
@petitions, 278
@prettylist, 278
@previous, 278
@prog-options, 278
@program, 45, 278
@program#, 278
@property, 45, 101, 103, 278
@prospectus, 278
@qreply, 278
@qsend, 279
@quickreply, 278
@quicksend, 279
@quit, 9, 18, 45, 279
@quota, 90, 279
@ranm. *See* @read-all-new-mail
@read, 45, 279
@read-all-new-mail, 279
@reaper-ballots, 279
@reapers, 279
@recreate, 86, 280
@recycle, 81, 280
@refusal-reporting, 280
@refusals, 31, 280
@refuse, 30, 45, 280

@registerme, 280
@registrar-ballots, 280
@registrar-petitions, 280
@registrars, 281
@remember, 281
@remove-feature. *See* @rmfeature
@rename, 281
@renumber, 281
@reply, 281
@request, 14, 15, 281
@resend, 281
@resident, 282
@residents, 282
@rmalias, 282
@rmalias#, 282
@rmdict, 282
@rmfeature, 43, 282
@rmlag, 282
@rmmail, 282
@rmproperty, 282
@rmroom, 282
@rmverb, 283
@rmverb#, 283
@rmword, 283
@rn, 283
@rooms, 45, 283
@send, 45, 283
@set, 86
  with messages, 88
@setenv, 164, 283
@sethome, 283
@setprop, 284
@set-tell-filter-hook, 284
@show, 284
@skip, 284
@sort-owned-objects, 284
@spell, 284
@spellmessages, 284
@spellproperties, 284
@spurn, 34, 285
@spurned, 285
@sshow, 285
@subscribe, 45, 285
@subscribed, 285
@subscribe-quick, 285
@suggestion, 285
@sweep, 14, 32, 285
@teleport. *See* @move
@tell-filter, 286
@tutorial, 9, 286
@typo, 286
@unban, 286
@undefine, 286
@ungag, 286
@ungag-site, 287
@unlock, 31, 287
@unmessage, 287
@unread, 287
@unrefuse, 31, 287
@unrmmail, 287
@unsend, 287
@unset-tell-filter-hook, 288
@unsubscribe, 288
@unsubscribed, 288
@unsubscribed-quick, 288
@uptime, 288
@users, 288
@verb, 45, 103, 104, 288
@verbs, 288
@verify-owned, 288
@version, 288
@watch, 289
@ways, 45, 289
@web, 289
@whereis, 289
@who, 289
@will, 290
@witness, 290, 313
@wizards, 290
@wrap, 8, 290
` ! => ' construct, 158
+, 292
+c property permission flag, 165
<>, 2
abbreviations, conversational typing, 255
Abraxas, 323
abstract, 331, 342, 344
accept vs. acceptable, 146
accountability, 305
active, 333, 381
Adventure, 25, 37
age
  determining a player's MOO-age, 261
  official (LambdaMOO), 261
Aglaia, 383
Aladdin, 179

alias, 243
  adding an alias to your MOO character, 16
ancestors (of an object), 38
ancestry, 82
AndroidLust, 379
Angharad, 335
angle brackets, usage, 2
anj, 339
annoyances, countering, 29
antisocial feature object, 218
ARB, 91, 190, 235, 243, 261, 304, 305, 311, 314, 315, 323, 328, 331, 350, 351
  members
    listing, 261
    term limits, 320
arbitration system, LambdaMOO, 233, 308, 311, 319, 321, 324, 325, 326, 328, 329, 330, 331, 332, 334, 335, 339, 340, 344, 346, 347, 348, 351, 354, 362, 364, 367, 369
Architecture Review Board. *See* ARB
args, built-in variable, 145, 159
argstr, built-in variable, 159
argument list, 106
argument specifiers, 102, 104, 112, 163, 164, 243, 244, 288
  changing the argument specifiers for an exisiting verb, 261
arguments, 102, 114, 243
Artbag, 337
articles, usually omitted, 23
assignment, scattering, 160
asterisk, 259
  in verb names, 109
  to designate a mailing list, 38
audit. *See* @audit
Avenger, 334, 340
awareness, general, 32
background task, 102, 132, 162, 163, 244, 253, 255, 272
backspace, with raw telnet, 7
balloons
  hot air, 200
ballots, 327. *See also* petition and ballot system
  LambdaMOO
    compendium of, 311
    listing, 262
banning someone from rooms you own, 34
Barth, 11
Bartlebooth, 24, 189, 194, 200, 201, 203, 212
bash, 244
basic communications, 9
Bats!, 329, 336
Bear(tm), 368
Béchamel Sauce, 257
behavioral norms, 36
Benny, 382
Big Brother (or someone else) may be watching you, 32
Bits, 115
Blissboi, 382
Blob, 178
Bond-007, 356
Boo, 12, 218
Boonton, 339, 365, 378, 382
boring, 262
bot, 244
Brack, 334, 337, 350
bug(s)
  reporting, 262
bugging, 32. *See also* @sweep
  of conversations, 14
building, 79, 90
  overview, 79
build-options, 89
built-in
  function (defined), 244
  functions, 168
  variables, 159
byte, 245
byte-based quota, 90, 314
Cable, 328, 348, 354
call, 245
caller
  built-in variable, 159
camouflaging a room or object's contents, 98
cARRoT, 195, 200, 211, 321, 336, 346
channels, 13, 245
  on LambdaMOO, 225
character, 245
  requesting, 14
child. *See* parent

chocolate soufflé, 257
Choppie!, 362
chparent. *See* @chparent
Chris-22, 331, 335
class, 245
client program
  if you don't have one, 8
  switching from raw telnet to, 8
client programs, 7
cobot, 382
command, 101, 245
command line, 245
command-line verbs, 105
commas, in names and aliases, 111
communication, recording, 13
communications, basic, 9
community, 303, 305
  expectations, 9
compass directions
  moving, 25
compound statements, 161
connect, 264
connected (room), 245
connecting to a MOO, basics, 7
container room, 26
containment hierarchy, 246
contents. *See also* @contents
  :contents verb vs .contents property, 99, 100
  of an object, 37, 39, 88, 98, 245
conventions of acceptable behavior, 28
conversational typing abbreviations, 255
conversations, coping in a crowded room, 11
copyright, of communication that occurs in real time, 14
core, 246
core database. *See also* LambdaCore
  defined, 101, 246
crayon, 333
create. *See* @create
creating rooms and other objects, 79
ctype
  property of $room
    examples, 86
cultural and behavioral norms, 36
Curtis, Pavel, 104
customizing objects, 86
customs, 28
darkrider, 332, 339
Darkson, 345, 354, 360
data types, 105, 156, 246
database, 2, 101, 246
  changing
    as opposed to generating text with the emote command, 11
  size, 265
Dave, 325
decline
  (a LambdaMOO petition), 265
defense mechanisms, 29
deLaMer, 369, 371, 376, 377
describe. *See* @describe
describing yourself, 16, 17
detailed rooms, 24
dig. *See* @dig
directed say, 11
discipline problems, 304
disconnecting from a MOO, 9
display. *See* @display
dispute system
  LambdaMOO. *See* arbitration system
dobj
  built-in variable, 148, 159
dobjstr, 120
  built-in variable, 159
Dodger, 315
doing vs. emoting, 10
Doug, 27, 196
Downtime, 365
dr, 316
Dred, 208, 314
Drippy, 57, 361
due diligence, 82
dynamic allocation (of variables), 105
e, 3
edd, 313
editors
  aborting an editing session, 68
  deleting text, 74
  ending an editing session, 68
  entering text, 69
  importing text from another source on the MOO (yank), 70
  insertion point, 66
    finding and moving, 68
  invoking, 67

listing text, 68
miscellaneous commands, 75
modifying existing text, 71
moving text, 73
options, 76
searching for text (find), 75
using, how to, 66
Edweirdo, 346
eep, 218
eir, 3
eirs, 3
eject. *See* @eject
Elayne, 380
em, 3
emacs with mud.el, client program, 7
email
changing your registration email address. *See* @registerme
emote, 10, 11
remote-emote, 13
emself, 3
EnCore, 246
eprint, 268
ERR, data type, 158
error, trapping, 158
etiquette
teleporting into a private room, 28
Etoile, 369
Euphistopheles, 321
eval, 45, 164
Evil, 359
examine, 268
command, introduced, 23
contrasted with @examine, 109
customizing the list of obvious verbs, 119
exit, 246
exit messages, 94, 96
exits
attaching an exit owned by someone else, 85
creating with @dig, 85
finding available exits from a room, 25
seeing what exits are available in/from a room. *See* @ways
expectations
cultural/behavioral, 9, 36
exploring, 22
expression
regular, 267
expressions, 106, 161, 164, 246, 251
Farcan, 351, 365, 374
feature objects, 41
adding a feature object, 42
LambdaMOO, 217
antisocial feature object, 218
APHiD's socializing feature object, 219
Carrot's social interaction feature, 223
Carrot's viewing feature, 224
Compass Rosette FO, 224
dancing feature object, 220
fast & dangerous Info FO, 221
Karma FO, 224
LambdaMOO museum search FO, 224
login watcher, 220
Mazer's object utilities FO, 224
Multi-communications feature, 225
Obvious feature object, 224
Pasting FO, 225
Quota-Transferral feature, 225
social verb core, 218
stage-talk feature, 217
programming, 167
removing a feature object, 43
sequence matters, 42
fertile, 83, 90, 246
fiat, wizard (LambdaMOO), 234, 307
fifel, 382
fifelfoo, 370
Fionaa, 376
flag, 103, 247
FLOAT, data type, 156
follow (another player), 269
Foobies, 320, 321, 322, 324
food fights
opting out, 262
Ford, 306
foreground task, 101, 102, 163, 244, 247, 253. *See also* background task
fork bomb, 247
forked task, defined, 247
forked tasks
listing, 269

Frand, 303
Frand's player class, commands, 45
Fred_astaire, 361
Fuel, 360
functions
  built-in, 168
  built-in (defined), 244
Gabaldon, 383
gag, 247. *See* @gag
game master, 247
Gary, 325, 360, 364
Gary_Severn, 214, 303
Gear, 373
Gemba, 303
gender, 2. *See* @gender
  agreement, in exit messages, 97
  presenting, 18
  pronouns, 17
  specifying your MOO character's gender, 17
gender-neutral pronouns, 3
general awareness, 32
generic, 247
generic builder commands, 45
generic mail receiving player commands, 45
generic objects, 79, 81
generic player commands, 45
generic programmer commands, 45
genna, 189
Geust, 306
ghond, 303
GhostDruid, 379
Gilmore, 331, 341, 359, 360, 374
global variables, 105
glossary of terms, 243
go
  command, 26, 271
GothGrrl, 344
graeme, 381
Grand_Master, 247
Greene, 201
greetings
  unexpected, 19
GrendelFish, 335
GreyDruid, 377
Griffen, 341
GrinchDruid, 383
gru, 188, 194, 197, 334
Grump, 311
Gryndel, 323
guest
  connecting as, 7
guests
  booting off the system, 316
gurst, 247
Haakon, 233, 303, 306, 307, 308, 309, 340, 359
habibi_with_wild_rice, 384
Hagbard, 312
Hannibal, 379
harassment, 35
Harry_Potter, 370
heartbeat, 271
help, 45, 271
  online, 8
help manners. *See* manners
help text
  adding help text to an object, 118
Hepper, 383
heyoka, 374
hhsb, 376, 378
Hibernian, 343, 345, 372
hiccup, 383
hidden treasures, 166
Hobgoblin, 372
Holgate, 363
home, 45, 271
  creating a home for yourself, 18
  designating a room as your home, 18
  going home, 19
  letting someone else designate a room you own as eir home, 18
Hookleg, 338, 352
hot air balloons, 200
HumbertHumbert, 326, 334, 340
idle, 247
Indite, 337
Individual, 318
inherited properties, 81
input, 247
insertion point
  in MOO editors, 66
INT, data type, 156
integrating room, 24
integration
  advantages and disadvantages, 99

room, 94
invalid, 247
invalid objects, 37
inventory, 80, 248, 271
invoke, 248
iobj
built-in variable, 148, 160
iobjstr, 131
built-in variable, 160
Jack, 253
Jacobson, David, 14
jaime, 322, 326, 342, 343, 384
Jaybird, 362, 365, 367, 368, 371
Jayturkey, 361
Jerri_Blank, 383
JHCore, 45, 246
Jill, 253
JoeFeedback, 306
JohnBoy, 328, 354, 367
Johns, 358
join. *See* @join
Jon_BonJarleycorn, 197
Jool, 197
journalists, at LambdaMOO, 343
judicial review board
LambdaMOO (failed ballot), 333
Karma, 206
kids. *See also* parent. *See* @kids
Kilik, 187, 321, 326, 327, 343
Kirlan, 218
Klaatu, 13, 30, 38, 115, 200, 218, 229, 230, 367
Kobot, 381
Krate, 370
Lady-Dawn, 370
lag, 248
$login.current_lag, 163
@addlag (command), 260
@rmlag (command), 282
effect on @paste, 225
Lag Reduction FO of Godlike Powers, 129
Lambda, 306, 317
LambdaCore, 2, 13, 27, 34, 45, 47, 66, 82, 162, 170, 173, 217, 225, 226, 237, 246, 248, 259, 269, 288, 289, 290, 303, 308
LambdaMOO, 2
arbitration. *See* arbitration system
elected offices, 235
feature objects, 217
Guide to Interesting Places, 175
petition system
2/3 majority, 235
creating a petition, 239
elected offices, 235
eligibility requirements, 235
historical overview, 233
how to participate, 237
nomination petitions, 241
opting out, 234
petition options, 238
signature thresholds, 235
time limits for petitions and ballots, 235
vetting criteria, 236
political system, 232
registration policy, 14
RPG (Role Playing Game). *See* RPG
Takes a New Direction, 234, 303, 307, 308, 340
Takes Another Direction, 234, 307, 349, 353, 373
VR atlas, 25
LambdaMOO Programmer's Manual, 104
Lamont_Cranston, 202
legba, 316
Legion, 369
Li2CO3, 349
lights, 371
limbo. *See* $limbo
line breaks
in verbs, 110
Linnea, 317
LIST, 248
data type, 157
listening devices. *See* @sweep
literals, 112
local variables, 105
lock. *See* @lock. *See* @lock
log files, 14
logging, 32
login watcher (feature object), 220
login watchers, 19
look detection, 17
look_msg, 98
look_self, 99
looking around, 22

looping, 162
loree, 353
lose, 273
lovecraft, 57
LTAD, 234, 307, 349, 353, 373
LTAND, 234, 303, 307, 308, 340
lurk, 248
Mack-the-Knife, 340
mail
  @read and @peek, 55
  aborting an editing session if you change your mind about sending mail, 59
  automatically forwarding mail to your email address, 63
  checking for new messages, 52
  current folder, 52
  current message, 52
  customizing aspects of the mail system, 59
  folder, 52
  list, topic description, 52
  message headers, 54
  message sequences, 54
  options, 59
  reading, 51
  reading a MOOmail list, 51
  reading all new mail at once (@ranm), 52
  recipient, 51
  removing messages (@rmm), 56
  renumbering messages, 56
  searching MOOmail lists, 54
  seeing what MOOmail lists exist, 52
  sending, 57
  skipping unread messages, 56
  subscribing to a MOOmail list, 52
  using the mail editor (basic), 57
mail options, 59
  listing, 60
  modifying, 61
mailing lists, 38, 51
  creating, 93
manners, 9, 22, 28, 48, 233, 304, 305, 313, 317, 319, 328, 329, 334, 345, 364, 367, 369. *See also* help manners
map, 25
matching, 102, 248
mav, 249
measuring an object. *See* @measure
Mediocrity, 378
Melki, 375
message
  property, defined, 46
message sequences, 54
messages. *See* @messages
  setting, 46
meta-VR, 1, 2, 25, 88, 94, 243, 249
Mickey, 312, 319, 322, 323, 330, 332, 338, 340
Miles, 324
mingaloid, 371
mockturtle, 42, 248
MOO
  definition, 1
MOObash, 244
MOOer of the Month, 320
MOOicide, 357, 363
MOOmail
  reading, 51
MOOs
  finding. *See* Rachel's Super MOO List
MOOsex, 35
Mooshie, 350, 355
Moriah, 312
morph, 249
Morpheus, 26
moving around, 25
  walking vs teleporting, 25
Mr_Bungle, 233
MUD, 37
  Object Oriented, 37
MUDDweller, 7
MugWump, 355
multiple characters, 249
multi-room, 186
multi-tasking, 249
multi-user environment
  draws and drawbacks, 28
murmur, 275
Musketeer, 371
MutantNemesis, 359
Naked_Guest, 330
name
  changing the name of your MOO character, 16
Nancy, 317

New-Player-11164, 347
news, 8, 275
newt, 249
Nim, 12, 219, 229
noise abatement, 29
nonsenso, 360
non-VR, 1, 25, 249. *See also* meta-VR
norms
  cultural and behavioral, 36
nosiness, 32
Nosredna, 254, 306
notabird, 346
note editor, 57. *See* @notedit
null object, 153
NUM
  data type, 156
Nuveena, 356, 366
O.M.I.N., 361
OBJ, data type, 156
object, 249
  ancestry, 82
  hierarchy, 37, 83
  measurement task, 91
  number, 37
objects, 1, 2, 37
  brief introduction, 21
  contents, 37, 39, 88, 98, 245. *See also* @contents
  creating a new object, 38. *See also* @dig. *See also* @create
  customizing, 86
  feature objects, 41
  fertile, 83, 90, 246
  generic, 81
  getting rid of unwanted objects (@eject), 40
  identifying interactive objects in your vicinity, 23
  invalid, 37
  location, 38
  measuring, 91
  moving, 38
  parent hierarchy, 43
  parent objects (ancestors), 38
  parenthood, 37
  referring to by number or by name, 38
  root object, 37
  seeing a list of objects you own, 38
  teleporting, 40
  unmeasured, 91
  valid, 37
ObsessionRose, 382
obvious verbs, 23
  customizing, for examine, 120
official age (LambdaMOO), 261
operators, 160
options package, 53, 249
  defined, 59
outing, 36
output, 101, 250
ownership
  and permissions, 165
Ox, 316, 358
P7A77, 341
page, 12, 276
parameters, 250
paranoid. *See* @paranoid
parent, 250
  changing the parent of an object, 86, 264
parent hierarchy, 43
parents. *See* @parents
parser, 41, 102, 250
parsing, 41
party, 277
password
  changing, 15. *See* @password
PatGently, 313, 326, 327
pens, 358
Peri, 343, 348, 357, 358, 360
permission flags
  changing, 263
  object, 90, 266
  property, 103, 278
  verb, 104, 168, 288
permissions
  error (E_PERM), 165
  ownership and, 165
  task, 165
petition and ballot system (LambdaMOO), 226, 232, 250, 252, 253, 261, 265, 274, 277, 278, 279, 280, 307, 308, 309, 311, 312, 314, 317, 319, 320, 322, 323, 324, 325, 326, 327, 328, 329, 331, 333, 335, 338, 339, 340, 347, 350, 351, 355,

358, 360, 362, 365, 366, 367, 368, 372, 373, 374, 375, 376, 379, 380, 381
Phalin, 383
Pictwe, 352
PKelly, 383
Plaid_Guest, 217
player, 250
  built-in variable, 112, 159
player class, 250
player class owners
  powers, 44
player classes, 43
  Frand's player class, 45
  generic builder, 45
  generic mail receiving player, 45
  generic player, 45
  generic programmer, 45
  LambdaMOO, 226
    Detailed Player Class, 232
    Experimental Guinea Pig Class with Even More Features of Dubious Utility, 228
    Generic LambdaMOO Citizen, 226
    Generic Player Class With Additional Features of Dubious Utility, 227
    Generic Super_Huh Player, 231
    Politically Correct Featureful Player Class Created Because Nobody Would @Copy Verbs To 8855, 230
    SSPC, 232
    Super-Schmoo, 232
player object, 250
player-character, 250
players, 1
port, 250
port number, 7
portable room, 26
POSV, 370
Power Elite (LambdaMOO), 196
prepstr
  built-in variable, 160
primary character, 235, 250, 252, 320, 347, 369, 379. *See also* secondary character
privacy, 31
  player class owners and, 44
  which MOOmail lists you read, 51
Profane, 336, 342, 346, 349, 351
program, 251
programmer, 251
programmer bit, 108, 305
  obtaining, 45
Programmer's Manual
  (by Pavel Curtis)
    available via ftp, 109
programming
  concise reference, 156
  tutorial, 108
pronoun substitutions, 115
pronouns
  gender, 17
  Spivak, 3
proper conduct
  disagreements about, 304
property, 2, 101. *See also* $object_utils. *See also* @show. *See also* @property. *See also* @dump. *See also* @display
  +c permission flag, 165
  adding a new property to an object, 103
  changing 'c' permissions on descendants of an object you own, 165
  changing with @set, 284
  defined, 251
  defined vs inherited, 93
  displaying a list of an object's properties, 127
  displaying an object's inherited properties, 142
  dynamic access, 159
  edit, 68
  editor (@pedit), 228, 277
  inherited, 43, 81, 86
  measuring a property's size, 92
  message, 46, 88
  not found (error) E_PROPNF, 158
  ownership, 165
  ownership of inherited properties, 165
  permission flags, 263
  quota required to add a property to an object, 90
  removing a property from an object. *See* @rmproperty

viewing with eval, 291
psign_*p:ctr, 349
Pueblo, 7
Puff, 313
pupa, 379
puppet, 251
pupuck, 383
Quadric, 362
Quantum-Vacuum, 314
QUARTlow, 329, 342, 350, 363
queue, 103, 251
Quiet, 352
Qui-Gon_Jinn, 369
Quinn, 312, 320, 325, 337, 357
QuinnGrrl, 363
quit
disconnecting from a MOO, 9. *See also* @quit
quota, 329. *See also* ARB
byte-based, 90, 314
defined, 90
displaying amounts. *See* @quota
getting more, 91
measuring, 274
policy (LambdaMOO), 372
things to do if you are over quota, 91
transfer (LambdaMOO), 225, 316, 337, 339, 369
Rachel's Super MOO List, 8
RAM
random access memory, 13
rape, 317, 318
Rat, 327, 345
raw telnet, 7
reading mail, 51
reapers, 348. *See also* @reapers
reaping
(LambdaMOO), 358
reaping (LambdaMOO), 222, 251, 254, 316, 357
defined, 251
eligibility requirements to be a reaper, 235
timing, 356
recreate. *See* @recreate
recycle. *See* @recycle
RedFeather, 352
refuse. *See* @refuse
registrars. *See* @registrars
regular expression, 267
remote-emote, 13
requesting a MOO character, 8, 14. *See also* @request
LambdaMOO
checking your status on the waitlist, 15
researchers
at LambdaMOO, 343
resource limit exceeded
when trying to create a new object, 92
response latency, 251
return value, 251
RinkRabbit, 380
risk
associated with adding a player class, 45
Road_Dog, 206
robot, 244
Roebare, 368
Rog, 333
room, 251
container, 26
detailed, 24
integrating, 24
portable, 26
remembering a room's object number, 27
room integration, 94
rooms
connected, 26
creating with @dig, 84
dark, 99
rooms database, 28, 251
root object, 37
RPG
LambdaMOO Role Playing Game, 184, 209, 212, 252
game masters, 226, 247, 270
Grand_Master, 247
rules, 9, 29, 304
Russian roulette pistol
LambdaMOO, 203
RussianMob, 380
Sabastian, 383
Sage, 367
say

double quote abbreviation, 10
Scarab, 356
scattering assignment, 160
Schmoo, 231
searching
mailing lists, 54
secondary character, 250, 252, 319, 320, 347, 369, 375, 376, 377, 379. *See also* primary character
illegal, 375
seek, 283
Selma, 363
sending mail, 57
server, 2, 41, 101, 252
set. *See* @set
setting messages, 46
sexual advances, 35
sexual harassment, 318
Shalmaneser, 351, 356
shard, 370
Shimizu, 373
Shmool, 39
shouting, 29, 252
shutdown
built-in command, 170
LambdaMOO ballot, 234, 309, 311, 352, 353, 362, 365, 373
side effect, 106, 252
silent teleporting, 46
silly moustaches, 353
skeptopotamus, 57
Slartibartfast, 306
Sleeper, 344, 362, 364, 366
snarl, 374
social pressure
on player class owners, not to spy, 44
social security
LambdaMOO, 356
social verb core (feature object), 218
soufflé, chocolate, 257
soup, 366, 377
spam, 252
spamming, 29
Spivak
(LambdaMOO player), 373, 381
Michael, 3
pronouns, 3
spoofing, 30, 228
defined, 252
spying, 14
stage-talk feature, 217
Stetson, 318, 323, 332, 347, 381
stop-following, 285
STR
data type, 156
string, 253
subroutine, 105, 115
subscribing to a MOOmail list, 52
subscripting, 158
Sunny, 319, 321, 324, 328, 336, 338, 341, 351, 375
sweep. *See* @sweep
syntax, 253
syntax specification for commands, 259
system, 253
system character, 253
Tapu, 329, 362
Tartan_Guest, 13, 220
task, 101, 102, 253
background, 102, 244
foreground, 102, 247
forked, 247
permissions, 165
task_id, 103
Tchinek, 335
teleport, 26, 253
silent teleporting, 46
tell_contents, 99
tell-filter, 253
telnet, 7. *See also* @wrap
terminate, 106
text
generating text as opposed to changing the database, 11
unattributed (tracing), 30
The_Walrus, 212
theft, 33
prevention, 33
theme, 1, 22
themely transition, 178, 189, 206, 207
this
built-in variable, 148, 159
this (built-in variable), 140
ticks, 162, 254
tiny scenery, 99, 196, 200, 254
tkMOO, 7

TMFKAN64, 357, 359, 366
toad, 254
toad scar, 254
Tomb of the Unknown MOOers, 204
Topher, 327
Tower, 55
treasures
  hidden, 166
Trees_Beatnik, 324
trespass, 33
tricks of the (programming) trade, 167
troll, 254
trust
  player class owners, 44
truth values, 161
try/except/endtry construct, 158
TunaJesus, 348
tutorial. *See also* @tutorial
  LambdaMOO, 383
  programming (Yib's Pet Rock), 108
  yduJ's Wind-up Duck, 109, 207
typing abbreviations
  conversational, 255
typist, 1, 254
unattributed text, 252
  tracing, 30
unexpected greetings, 19
unfollow, 286
unmeasured objects, 91
user, 254
user-extensible, 2, 41
Uther_Locksley, 339, 344, 348
Uther_O'Locksley, 353
Utterly_Lame, 375
valid, 254
variable, 159. *See also* flag. *See also* data types
  built-in variables, 105, 159
  defined, 105, 255
  global, 105
  local, 105. *See also* @setenv
verb, 23
  adding a verb to an object, 288
  built-in variable, 159
  copying a verb from one object to another, 265
  defined, 101, 255
  displaying a verb on an object, 127
  displaying inherited verbs on an object, 267
  dynamic access, 159
  editor, 57
  inherited, 43
  introduced, 2
  measuring a verb's size, 92
  not found (error) E_VERBNF, 158
  obvious verbs on an object, 111
  ownership, 165
  removing from an object. *See* @rmverb
Veren, 57
vetting petitions (LambdaMOO), 307
Vida_Blue, 345
virtual reality, 1, 25, 243
  breaking, 25
Vorial, 375
VR, 25, 255. *See also* virtual reality
  actions, 243
waffle, 176, 207
Warhol, 374
ways, 289
Werebull, 16, 27, 67, 220, 252
wheel, 255
whisper, 12, 289
wiffle bats, 332
Wintermute, 188, 199
wizard, 255
wizards, 303, 304, 305, 306, 307, 308, 309
  fiat (LambdaMOO), 234, 307
  listing, 290
  powers, 33, 44, 51, 57, 104, 163, 235, 247, 249, 250, 254, 255, 260, 269, 275, 278, 282, 286, 343
WriTinG, 341, 368
X'iina, 322
Xia, 355
Xiombarg, 311
xmath, 368, 377
Xorbon, 193
Xythian, 189, 315
yduJ, 109, 194, 207, 306, 314, 384
Yib
  arriving in a shower of sparks, 46
  ballots by, 331, 333, 359, 364, 376, 380
  begins her career as a bugbear, 189

dancing with Werebull, 220
lifting an elephant, 11
ZenWombat, 356
Zork, 25, 37, 208