

The Lambda Wizard MOO Programming Manual

© 2001 Nicholas Robinson and Eastern Goldfields Senior High School.

This document may be reproduced in its unaltered form for any educational purpose. Queries against the use of this document may be directed to egshs@egshs.wa.edu.au.

Welcome to MOO

Welcome to MOO Training. During this and the subsequent sessions we will be examining how to create a text based virtual environment based on the LambdaMOO MOO database.

MOO stands for MUD Object Orientated. MUD stands for Multi-User Dungeon (and herein lies the origins of the environment- a game). As the name suggests, a MOO uses a form of programming called Object Orientated programming.

An object is simply a collection of things that hold a numerical or textual value (called a *property*) and computer programs that operate on these properties and other objects. In a MOO these programs are called *verbs*.

When you first enter into the MOO programming environment there are a number of objects that have already been created so that you dont start from scratch and go insane. By using just these basic items, you can create a functional MOO very quickly. The time factor comes into play when you want to be creative and start having the MOO environment act on the players within it.

A player is simply a person who has connected into the MOO Server using a computer. This can be done using a network (like here) or by using a modem to connect to the internet and from there onto a network containing a MOO Server. (such as WinMOO).

Players can move about in the virtual world you create and interact with each other and things that you create for them to interact with.

Settingup WinMOO and Lambdacore

For those of us out there using NT Server (sorry to you Linux people, youll have to find your own software) the best MOO server engine is WinMOO which is available at <http://www.stanford.edu/~cunkel/WinMOO/winmoo-faq.html>. You will also need a copy of the latest Lambdacore database which is available from the same site. Follow the setup instructions.

Help

First off, if you ever need to know about a command or get stuck somewhere with something, LambdaMOO has a great help facility. The format of the help command is *help* followed by the command you want help on. Simply typing *help* also gives you a list of possible help topics.

For example type *help movement* and read about how to move about in the MOO.

MOO Basics

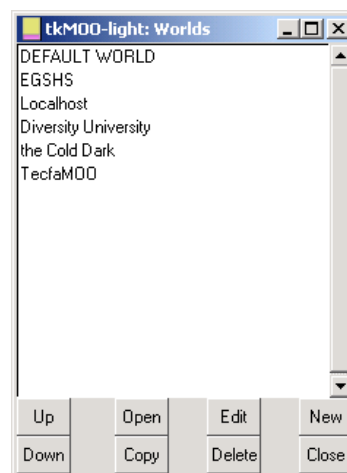
Before we get into building our MOO virtual reality, it is worth some time to experience using the MOO so that you can understand what your prospective MOO players are doing. In fact it is well worthwhile going and using some of the internet based MOOs out there to get a feel for textual virtual reality.

Connecting to a MOO

Connecting to any MOO is done using a MOO client. There are a number of these programs available free on the internet. Three of the most common are tkMOO, Vmoo and Pueblo. Of these Pueblo is my favourite, but since it requires registration we will be using the free program tkMoo which looks like this:

To connect to any MOO you need to know two things, its address (also called its host name) and its port number. If you do a search on the internet for public MOOs you can find this information normally on the MOOs introductory page.

In the *connect* pull-down-menu of tkMOO select the *Worlds* option. This dialogue allows you to connect to various MOOs that you like in an easy manner. Try connecting to TecfaMOO.



You will see a screen that informs you that you have connected to A Virtual Space for EDUCATIONAL TECHNOLOGY, EDUCATION, RESEARCH and LIFE at TECFA, School of Psychology and Education, University of Geneva, Switzerland and

that you can Connect as a registered person by typing connect (name) (password)
Guests: type 'connect guest' – Type 'who' to see who is connected.

Just to mess about, type *connect guest* and move around (type *help movement* for extra help).

Talking and Interacting

Straight away you are able to do a number of things with your player.

You can talk to other people that are in a room with you simply by typing a quote () symbol and typing what you want to say. For example:

Wow, isnt this MOO great!

Would result in everyone in the same virtual room as you seeing the text you type.

You are also able to look around and look at each other using the *look* command. This can be abbreviated to simply the first letter as well. For example try:

Look
L

You can also look at things in the room (this includes other players). As a player you are able to set what is displayed if someone looks at you by using the *@describe* command. This is a very useful command, and you will be using it many times. For example:

@describe me as A small, squelchy fungus.

Try setting your description and then looking at each other in the MOO. The LambdaMOO environment is quite clever is sorting out what you are wanting to look at or do. There is normally no need to type an entire name of a command or what it is you wish to operate on. For example , the following command would all work (the enter at the end of the command is assumed form now on):

look at Nick
lo nick
loo nic
l ni.
L n

The environment tends to work to the lowest point of differentiation. What I mean by

this is that you need only type as much of a command or name of an object to make it uniquely recognisable.

You can also set what gender you are using the *@gender* command. If you type the command by itself you will get a list of the available options. Try it.

Another very useful command is the *examine <object>* command. This command is used to provide a summary of the things you can do with the examined object. Try the command *examine me* to get a list of all the commands you can issue that do something to yourself. Yes, there are lots, but all things in their own good time.

For now, wander around in TecfaMOO and look at things. Talk to each other. Then when you are all done, type the command *@quit* which disconnects you from the MOO.

Making a New Connection

You will now need to connect to your MOO. This is done by making a new entry in our Worlds list. Do this by selecting the *Worlds* option from the *Connect* pull-down-menu. Click the *New* button.

In the World text box, type the name of your MOO (eg *EGSHS MOO*). In the Host text box type the IP address (such as *192.168.0.1*) or (if your MOO server is on a local network with WINS facilities) the NETBIOS name of the MOO server (such as *romeo*). In the port text box type the port number that the MOO server is operating on (such as *7777*).

Note that funny-looking rectangle next to where it says *General Settings*. If you click on it, you can set other things (like the font displayed and colours used).

Click *Save* when you are done, and then connect to your MOO. Since we have no players in your MOO yet, tkMOO will produce a dialogue box us for a player name and password, pick cancel for now.

Wizards and Programmers

By default your brand new LambdaMoo database will has a player called wizard. The wizard is all powerful and is able to construct and destruct the MOO environment and perform actions on other players. However, it is not advisable to use the wizard player to do your MOO programming. You should create a normal

player and then use the wizard player to give it programming rights using the *@programmer* command (see below).

Creating a Player

When you first connect to any MOO a welcome message is displayed. In the case of the EGSHS MOO we are instructed to type help for a list of available commands. The important one here is *create*. Type *create* and press enter.

Follow the format indicated to create a new player.

Changing your password

If you wish to change your password at any time typing the command as shown below.

```
@password <old> <new>
```

You will be using the @ symbol quite a bit. It tells the MOO Server that what you are about to type is a command that changes settings on the computer, rather than a command that interacts with the virtual world.

Making a Player a Programmer

If a player is to be able to make things in your MOO, you need to make that player a programmer. This is done by using the *@programmer* command in the format.

```
@programmer <player name or ID>
```

Home, Home on the MOO

When you enter the MOO it is also possible to set the location that you automatically arrive at. This is done using the *@sethome* command. Then, whenever you want to return to your home location you can simply type *home*.

Building the MOO

OK enough of that. Lets do something creative. Right now we are going to create a new virtual reality (make sure you are connected as a programming capable player, otherwise nothing we do from here on will work). This is done using the *@dig* command. The *@dig* command can take a number of arguments (the programming term for things you tell the command) or be quite simple. Lets start simply:

@dig <name of room>

This command creates a new room called *<name of room>*. You must include the quotation marks or the command will get upset. Here is the results of the command I typed.

@dig "The Hole"

The Hole (#110) created.

Now you try with your own room name.

Whatever you typed for the name will be displayed as well as an odd looking number in brackets that begins with a hash (#). This is the Object Number (remember what I was saying about a MOO being an object orientated environment). It is unique and will become important in about four seconds.

Right now although you have made a new room in the MOO, you have to get to it to do anything useful. This is where the *@move* command is used.

@move me to <object number>

Replace *<object number>* with the Object Number of your room as it was shown when you created your room. Here is the results of the command as I typed it.

@move me to #110

The Hole

You see nothing special.

You probably noticed that the description of where we moved to appeared automatically. In this case is was You see nothing special. This is the default description for a room (boring isnt it).

Lets make our new room more exciting. To do this we use the *@describe* command again, but this time with a slight difference.

@describe here as <description>

@describe here as "You are in a warm and cosy hole. A few roots dangle from the ceiling and the air smells pleasantly of damp soil."

Description set.

Now when you type the look command, you and everyone else that enters the room gets the description you just set. Easy huh.

look

The Hole

You are in a warm and cosy hole. A few roots dangle from the ceiling and the air smells pleasantly of damp soil.

Being a bit more technical about the *@describe* command the format of this command is actually:

@describe <object> as <description>

<Object> can refer to the name of an object in the current room, or an object number of any object in the MOO that you wish to set the description of. The special keyword *here* is used to refer to the room you are currently in. As a note for later, this keyword can not be used in a verb program it can only be used as a direct command.

Creating More Rooms

Right we now have one room. Not very interesting. Lets make some more. This time we will use the *@dig* command with a few more options that will create not only the new room, but also make it possible for us to move between them. The format of the extended dig command is:

@dig <exits> to <name of room>

The *<exits>* part is the tricky bit. You can name your exits whatever you like and a player will need to type what you have typed to go there. If you specify only one exit, then the passage is one way and the player will be unable to get back! Therefore, in most instances, you will specify the name of the exit to get to the new room and the name of the exit to get back to the one you are in now. This is accomplished easily using the format below.

<name of exit>, <another name for the exit>, <etc> | <name of exit back>, <another name>, <etc>

Note the | symbol between the exits to and from. This is located usually on the same key on your keyboard that has the backslash (\). The *<another name for the exit>* sections are what are known as aliases in MOO world. An alias is simply another name that an object can be called by and still be recognised. So, a full example could be:

@dig d,down|up,u to The Root Cellar

This is what happens when I type this command.

@dig down,d|up,up to "The Root Cellar"

The Root Cellar (#113) created.

Exit from The Hole (#110) to The Root Cellar (#113) via {"down", "d"}created with id #114.

Exit from The Root Cellar (#113) to The Hole (#110) via {"up", "u"}created with id #115.

Wow, loads of info. Basically the MOO is telling you the object numbers of the objects that have just been created, which you dont need to know except in special circumstances.

Now you create yourself one or two new rooms and get ready to set their descriptions the easy way (that is, make the rooms but dont bother describing them yet). Move around your rooms.

Did you notice something annoying? The rooms do not tell you where you can move to. You need to let players know where they can move by including the exits they can take in the rooms description. For example:

look

The Dungeon

The air is dank and stale. Water seeps from cracks in the wall. There is a cell to the south.

You can see here that I have included in the description that you can move south from here. My naming my exit south with an alias of s (to make things simple), players can guess that they can type south or a variation of it to go into the cell.

If you have forgotten what you have called your exits and where they go, you can use the *@exits* command to get a list of all the exits currently in the room you are in. For example:

@exits

up (#101) leads to The Entrance Hall (#62) via {up, u}.

s (#104) leads to Cell (#103) via {s, south}.

n (#107) leads to Torture Chamber (#106) via {n, north}.

Did you pick up the error in my room? The description of the room The Dungeon

(see above) did not mention that you could go north to the Torture Chamber. I need to alter our description of the room The Dungeon to include this. This is where the Note Editor is really great, but before we look at using that a small diversion.

Describing Exits

If you go back to when you created your rooms using the *@dig* command you will see that the exits that were created also have object numbers. All objects have descriptions, and exits are no different. Try looking at one of your exits.

@exits

up (#101) leads to The Entrance Hall (#62) via {up, u}.

s (#104) leads to Cell (#103) via {s, south}.

n (#107) leads to Torture Chamber (#106) via {n, north}.

look n

You see nothing special.

Any objects description can be set using the *@describe* command or using the *@edit* command, and exits are no exception. Heres an example of what I did.

@describe north as "It looks ominous."

Description set.

look n

It looks ominous.

As you can see, building your virtual universe takes a fair bit of thought and work. Descriptions are the key for setting the mood of the environment, so spend time on them.

No Homies

Remember from earlier that the *@sethome* command allows a player to set their home location to whatever room they like. Then by simply typing *home* they can go there. Sometimes you may not wish players to be able to do this. To make a room not *@sethome* settable, type the following.

```
@set here.free_entry to 0
```

To make it home-settable again type:

```
@set here.free_entry to 1
```

The Note Editor

The MOO environment includes a special command called *@edit* which provides a simple text editor used to program verbs (more on that one later) and set long descriptions. It is not very user friendly, but it does the job well and you soon get the hang of it.

For a start, lets look at how to use the note editor to set our room descriptions. Dont type this for the moment.

@edit here.description

Looking tricky? Not really. What we are doing now is starting to work with properties. If you can remember back to the introduction I talked about objects having verbs (programs that do things) and properties (which contained number values or text strings).

When we want to do something with an objects property, we use the following format to refer to it:

<Object name>.<property name>

The *@describe* command that we have been using until now simply sets the *<object name>.description* property in an easy manner. However, if we need to add to our description or make a really long one, using *@describe* is really annoying. In this case using the note editor is a much better way to go.

When we type *@edit here.description* the MOO automatically replaces *here* with the object number of the room that you are currently in. Ill edit my room now and show you what it looks like.

@edit here.description

Note Editor

Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing "The Dungeon"(#99).description. [string mode]

Notice the bottom line. It tells us the name and Object ID of the object that we are editing. It also tells us that the editor is in string mode which means that we are able to write text.

The note editor has its very own little subset of commands that you use to edit

things. The most basic is the *list* command. This shows you what has already been written. This is what it looks like:

list

__1_ The air is dank and stale. Water seeps from cracks in the wall. There is a cell to the south.

^^^

Now use the note editor to edit your own room description. Use the *list* command to display the current room description. It should look something like this:

@edit here.description

Note Editor

Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing "The Root Cellar"(#113).description. [string mode]

lis

Note is devoid of text.

Oooh. Whats this bit at the end? Well, if you are in a room with no description (as you should be) this is telling you that the description of this room has not been set yet. But (do I here you ask), where did the default message You see nothing special that you get when you look at a new come from? Well in this case the MOO does this when a room has no description set.

OK, lets add a description to your room. This is done using the *say* command. Whatever follows the *say* command is added as a string of text to the *here.description* property. Type your own description in and then use the *list* command to see that it worked. Heres what I typed:

say An ominous doorway beckons to the north.

Line 2 added.

lis

1: The air is dank and stale. Water seeps from cracks in the wall. There is a cell to the south.

__2_ An ominous doorway beckons to the north.

^^^

I've moved ahead a bit here to show you something. I already had a room description set for my The Dungeon room, so when I used the `say` command it added the extra text as another paragraph. Notice that each paragraph is numbered. This becomes important soon. Add a second line of text to your description using the `say` command. Use the `list` command to see the finished result.

Now use the `save` command to save what you have done (just type `save`). Exit the note editor using the `quit` command. *Look* at your location.

Notice that we have more than one paragraph of text. Now let's say that you did not want this, that you wanted your description as one blob of text. Going back and using the note editor (do this now) we can use the `join` command to join two (or more) lines / paragraphs of text together. Here it is in action on my The Dungeon room description.

join 1 2

```
__1_ The air is dank and stale. Water seeps from cracks in the wall. There is a cell to the south. An ominous doorway beckons to the north.
```

save

Text written to "The Dungeon"(#99).description as a single string.

The `join` command requires that you specify the range of text to join together. My command `join 1 2` instructed the note editor to join lines / paragraphs 1 and 2 together. If we had more paragraphs, we could specify a greater range.

Use the `join` command to join your text together. Use `save` and `quit`. *Look* to see the finished result. Edit some room descriptions elsewhere for practice.

Show and Tell

Now we are going to look at another very useful command called `@show`. This command gives a list of the current properties and verbs that any object has defined. The command looks like:

```
@show <object name or number>
```

For example my north exit looks like this:

@show north

```
Object ID: #107
```

```
Name: n
```

```
Parent: generic exit (#7)
```

```
Location: *** NONE ***
```

Owner: The Crimson Wizard (#2)

Flags:

Properties:

key: 0

aliases: {"n", "north"}

description: "It looks ominous."

object_size: {0, 0}

obvious: 1

source: #99

dest: #106

nogo_msg: 0

onogo_msg: 0

arrive_msg: 0

oarrive_msg: 0

oleave_msg: 0

leave_msg: 0

Look closely and you will see that the *.description* property has been set to our north description. Nothing complex going on here. Whenever a player looks at an object the MOO simply grabs whatever the *<object>.description* property is set to and displays it. You might even be able to guess what the other properties are (their names are fairly self-explanatory). Lets give this whole thing a good going over.

The *Object ID* and *Name* properties you know about. But what is this *Parent* thing? This is another very important concept with object orientated programming. When an object is created it can either be based on an existing object (its parent) or be a completely new object. With a MOO we already have many objects made for us to make life simple so that we can get on with doing things and not fiddling around programming.

When you create an object based on another, it inherits all the properties and verbs of that object. You can then add additional properties and verbs to the newly created object and if you made another object based on that one, it would get all the work you have done on the earlier model automatically. Neat huh?

When you create a new room your room is based on the object called *\$exit* which is the Generic Exit object (the \$ symbol here is a special symbol in MOO world that refers to objects that exist as part of the MOO initial environment). Try typing *@show \$exit* and see what happens.

Location simply refers to the location of the object (ie what room it is in or whom is carrying it). Exits have no locations, so this is set to none. For comparison try *@show*

me and have a look at the location.

Owner refers to what player created this object. Only the owner of an object can alter its properties and verbs. Only special types of players called Wizards or Programmers can create objects. Normal players can do any of the stuff we are messing about with.

Flags are special. We will be getting to them later. One that is of interest is the f flag (do a *@show \$room* to see it). This means the object is fertile which means that new objects can be created based on this object.

The *properties* section lists all the objects properties. We are able to set an objects properties through a verb program, or sometimes by using the *@set* command (a properties flags set if this can be done or not). The *@set* command format looks like this:

@set <object name or id>.<property name> to <value>

So we could also set the description of an object by setting the value of the description property by going *@set <object>.description to Another description*. As they say, there are many ways to skin a cat (poor cat).

Joining Rooms

Its all very nice for everyone to have their own virtual universe to themselves, but after a while it gets all very boring if no one comes to visit. You need to link your room to other peoples rooms.

This is actually quite complicated as the owner of the room that you wish to connect two has to invoke a special command called *@add-room* once you have created your exits.

Let us say that we have a player called MooTest that creates a room called Moo World. MooTest wants to connect their world to a room that was created by another player called The Crimson Wizard. First MooTest needs to know the object ID number of the room that they wish to connect to. Using the *@show here* command in the room that MooTest wants to connect to gets this info. Let us say that the room has an object ID of #62.

MooTest would issue a command such as this to make the exits from its room to the Crimson Wizards room (object ID #62).

@dig east,e|w,west to #62

Exit from Moo World (#117) to The Entrance Hall (#62) via {"east", "e"} created with id #118.

However, I couldn't add #118 as a legal entrance to The Entrance Hall. You may have to get its owner, The Crimson Wizard to add it for you.

Exit to Moo World (#117) via {"w", "west"} created with id #119. However, I couldn't add #119 as a legal exit from The Entrance Hall. Get its owner, The Crimson Wizard to add it for you.

Notice that MooTest has not been given permission to create the exit as The Crimson Wizard is the owner of the room (although the exit object itself has been created). You can ignore the first error message about the entrance to the room, the bit you are interested is this part:

Exit to Moo World (#117) via {"w", "west"} created with id #119. However, I couldn't add #119 as a legal exit from The Entrance Hall. Get its owner, The Crimson Wizard to add it for you.

It is the exit from The Entrance Hall to Moo World that is the problem. The exit can not be created as The Entrance Hall is owned by The Crimson Wizard. Thus you need to ask The Crimson Wizard to add the exit with object ID #119 to The Entrance Hall.

If you know The Crimson Wizard personally, you could ask them. Otherwise there are other methods of communication available.

Paging

The easiest way to talk to another player who is not in the same room as yourself is by using the *page* command. Use the *@who* command first to get a list of players and their IDs.

@who

Player name	Connected	Idle time	Location
mootest (#102)	20 minutes	0 seconds	The Entrance Hall
The Crimson Wizard (#2)	28 minutes	a minute	The Entrance Hall

Total: 2 players, both of whom have been active recently.

Then use the *page* command. The format is:

Page <player ID> with <message>

For example if wanted The Crimson Wizard to create our exit for us we would type:

Page #2 with "Could you please create an exit from The Entrance Hall via exit #119"

Your message has been sent.

The Crimson Wizard would see:

You sense that mootest is looking for you in The Entrance Hall.

It pages, "Could you please create an exit from The Entrance Hall via exit #119"

MOO Mail

If The Crimson Wizard was not available we could use the MOO Mail service. Here is how it works. First we get the Object ID of the player we want to send mail to using the *@who* command. Then we use the mail *@send* command to send a mail message.

@send #2

Subject:

[Type a line of input or `@abort' to abort the command.]

Create an exit please

Mail Room

Do a 'look' to get the list of commands, or 'help' for assistance.

Composing a letter to The Crimson Wizard (#2) entitled "Create an exit please"
say Could you create an exit to object ID #119 in "The Entrance Hall".

Line 1 added.

send

Sending...

Mail actually sent to The Crimson Wizard (#2)

The Crimson Wizard is notified of the message and can read the mail by issuing the *@mail* command followed by the *@read <message number>* command.

@mail

1:+ Sep 1 16:17 mootest (#102) Create an exit please

----+

@read 1

Message 1:

Date: Fri Sep 1 16:17:57 2000 Taipei Standard Time

From: mootest (#102)

To: The Crimson Wizard (#2)

Subject: Create an exit please

Could you create an exit to object ID #119 in "The Entrance Hall".

By the way, to delete a mail message once you have read it use the *@rmmail* *<message number>* command.

Making the Room Link

Once contact has been established with the owner of the room we wish to link to, the owner of the room uses the *@add-exit* command.

@add-exit #119

You have added #119 as an exit that goes to Moo World (#117) via w and west.

Ta da! Now we can travel from one players virtual world to another players virtual world.

If you can, try organising to join two worlds together with a friend. This also makes for easier testing to make sure everything works as you expected it.

Creating Objects

Once you have made your rooms, you may wish to then populate them with objects for the players to interact with. Objects can be a simple or as complex as you desire. Anything from a key to an interactive character that responds to the players. All it takes is time and programming.

The command to create a new object is:

@create \$thing called <object name>, <aliases>, <more aliases>...

All new objects are created from the basic object *\$thing* which provides a few simple verbs that permit a player to look at the object, pick it up, give it to other people, etc. Try *@show \$thing* for more info (especially look at the verbs section which lists the verb programs that apply to this object). Your new object inherits all of this stuff automatically.

As an example we are going to <insert dramatic music here>:

Build a Rodent

The first step in designing any object is to decide what characteristics that object is going to have. In the case of our rodent I've decided that:

- Σ The rodent will grow in size when fed.
- Σ The rodent likes cheese only.
- Σ The rodent will respond in a cute way if dropped and left alone.
- Σ The rodent does not like to be thrown.

The first step then is to make a basic Rodent object. To do this we type:

@create \$thing called "rodent"

You now have rodent with object number #116 and parent generic thing (#5).

Your object number for the rodent will very likely be different. Remember all object numbers are unique.

When you create an object, you are automatically carrying it. You can see what you are carrying using the *inventory* command (just typing *i* will also work).

If you make a mistake, you can kill off your newly created object by using the *@recycle* command that looks like this:

@recycle <object name or ID>

This command can be used to kill any object that you no longer want (including rooms and exits that you have created).

Now we need to describe our rodent. I typed the following (feel free to alter this if you like):

@describe rodent as "A cute animal with a long twisty tail. It has small beady eyes and whiskers."

You can now play with your rodent already. You can pick it up (using the *get* command), give it to other players, drop it and look at it. It's not terribly interesting yet though and needs more work.

Making Properties

Remember I wanted our rodent to grow, so we need some way of storing our rodent's current size. This is done using a property that we make ourselves using the *@property* command. The *@property* command takes the format:

@property <object name or ID>.<property name> <initial value>

Here are a few examples of creating new object properties on our rodent object (dont actually do these).

```
@property rodent.teeth Sharp
@property rodent.age 3
@property rodent.decimal 123.76
@property rodent.text A long bit of text
@property rodent.empty_text
```

Not too hard is it? Now create your own property called size on your rodent object with an initial value of 1. This is what I typed:

```
@property rodent.size 1
```

If you make a mistake you can use the *@rmproperty* command to delete a property you have created, and the *@property* command to make it again.

Using Lists

Saying that a rodent had a size of 1 is not very descriptive is it. So lets also make use of another MOO feature called a list. A list (as the name suggests) is a set of numerical and / or textual values. The format of a list is simple, here are some examples:

```
{1,10,20,15}
{1,Hello There,12,Wow}
{Some Text,More Text,A extra bit}
```

Can you see the pattern? The list begins with a curly bracket, has values separated by commas (with text enclosed in quotation marks) and is closed by another curly bracket. We are able to get values from our list really simply, but we will look more at that later.

We are going to create another property called size_list that contains some descriptive text of our rodent size. Type the following as I did:

```
@property rodent.size_list {}
```

What we just did was create an empty list (the {}means empty). Now we need to fill it.

@edit rodent.size_list

Note Editor

Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing "rodent"(#116).size_list.

enter

[Type lines of input; use `.` to end or `@abort' to abort the command.]

small

large

huge

disgusting

.

Lines 1-4 added.

save

Text written to "rodent"(#116).size_list.

quit

More new stuff. This time we used the note editor to put entries in our list. Notice the *enter* command. If we are typing a whole bunch of stuff, rather than having to use *say* all the time we can issue this command so that we can just keep on typing. A period at the end on an empty line indicates that we are done.

Do a *@show rodent* command as see how we have changed our rodent object.

Note our two new properties at the bottom of the output. All very interesting isnt it? We will be getting to those other properties that you can see very shortly. In fact, right now.

Messages, Messages Everywhere

When players mess about with objects they are given a message indicating what happens when they do so. Not only that, and other players in the room with them also get a message telling them what that player is doing. The text displayed as a result of some action is normally (but not always) stored in a property with that object so that it can be easily changed if needed.

Look again at all those properties the rodent automatically inherited. Their names are fairly easy to understand. Messages that begin with o are messages that other players see (o meaning other), messages that do not are messages that the player doing the action sees.

Messages are used so much that there is actually a special command to see them.

@messages <object name or ID>

Try it on your rodent.

There is nothing special about a message property, it is a property just like any other but the naming of the property follows a special format all messages end with an `_msg` at their end. This lets the `@messages` command figure out what properties are messages and what are not.

Now you may ask why we would want to do this. Well, it makes setting our player interaction messages easy. We can use the shortcut format:

@<message name> <object> is <message>

Lets try it on our rodent drop and take messages (which is displayed when you drop or take a rodent object).

@drop_succeeded rodent is "It squeaks and looks forlorn."

@take_succeeded rodent is "It snuggles warmly against you."

Drop and *get* the rodent a few times. All of a sudden the thing has cuteness value and you start not wanting to drop it. Strange for something that doesnt even exist. Hmm...

Anyway, you may have noticed something else interesting about the default messages that the `$thing` parent object gave our rodent. If you look youll see a `%t` in the message text. This is a special command character sequence that is replaced by the name of the object that the message belongs to (i.e. in our case Rodent). There are a whole bunch of these special codes, but well look more at that later.

We want to add a message to our rodent so that when a player tries to pat it, we get a message reflecting this. This is what I typed:

@property rodent.pat_msg "You pat the %t. It looks happy."

You may remember that when we used the `@detail` command to make a room detail, we could change the detail message using the `@<detail name> here is <message>` command. Well now you know how it works. The `@detail` command creates a property called `<detail name>_msg` on the room object that is displayed when the player looks at a detail.

Creating Verbs

Remember that a verb is a custom command that you create for players to interact with your object. To create a new verb we use the `@verb` command. The `@verb` command has a format that looks like this:

```
@verb <object name>:<verb name> <dobj> <prep> <iobj>
```

Ok dont get too freaked out. Let me explain. Basic command syntax in a MOO follows the format `<verb> <direct object> <preposition> <indirect object>`. For example these are all valid MOO commands:

```
Put rodent in jar
Give rodent to james
Pat rodent
```

If we break these commands up (this is done by a thing in the MOO called the Parser) the structure looks like this:

Verb	Direct Object	Preposition	Indirect Object
Put	Rodent	In	Jar
Give	Rodent	To	James
Pat	Rodent	None	None

When we create our own verb, we need to tell that verb what to expect in the way of command format. We are starting with a simple verb that only has the direct object considered. Note the colon between the object name and verb name. Type the command below like I did:

```
@verb rodent:pat this none none
```

Here we are using a special MOO keyword called *this* in the `<direct object>` argument slot of the verb command. The keyword *this* means this object, i.e. the verb applies to the object that the verb is defined on. The possible values for each of the object arguments in the verb command are:

Object Argument Effect

This The verb applies to the actual object itself, not another object.
Any The verb applies to any object specified in the argument slot.
None The argument slot is not used.

The possible values for the prepositions argument slot are: with/using, at/to, in front of, in/inside/into, on top of/on/onto/upon, out of/from inside/from, over,

through, under/underneath/beneath, behind, beside, for/about, is, as and finally off/off of.

I know it all seems very complex now, but give it time and things will come together as you learn to program verbs and create your own objects.

Anyway, back to our rodent. Use the *@show* command to look at our rodent and you will notice that we now have a verb definitions section which contains the verb name *pat*. Now try the command *pat rodent*.

Nothing happens. That is because our verb is empty of programming code. Lets get it to actually do something.

Programming a Verb

To program our verb we use the note editor. Issue the command as I have below:

```
@edit rodent:pat
```

Now we are going to add our very first line of programming code using the *say* command. Do as I have typed below:

```
say player:tell(this.pat_msg);
```

All objects in a MOO automatically have a number of special variables set by the MOO. A variable is a term used in programming to refer to a name that when specified contains a numerical value or text. The *player* variable automatically contains the object number of the player that calls the verb.

The *tell* verb is part of every player object, and simply prints a text message to the player... so by issuing the programming code *player:tell(<message>)* we are telling the MOO to display the text *<message>* to that player by calling the verb *tell* which is part of the generic player object.

We also told the *player:tell* verb to get our message text from the property called *pat_msg* that we defined earlier as part of our rodent object. The special variable *this* is an easy way to refer to the object on which the verb is defined.

The semicolon at the end of it all is VERY important. If you do not include the semicolon you will get errors! The semicolon tells the MOO that that is the end of the line of code.

Now we need to tell the note editor that we have finished programming. To so this

we use the *compile* command. And then we want to quit the note editor using the *quit* command.

compile

#116:pat successfully compiled.

quit

If you got errors (called a traceback) dont worry. Just issue the *@edit* command again and retype the programming code.

Now try the *pat rodent* command.

pat rodent

You pat the %t. It looks happy.

Hmmm, that %t wasnt supposed to do that was it? That was supposed to be replaced by the name of the object. So what went wrong? Well, we actually need to use another verb called *\$string_utils:pronoun_sub* (if you are clever can you see how this is broken up? *\$string_utils* is the object and *pronoun_sub* is a verb on that object) to get the thing to work. Do as I do below.

@edit rodent:pat

Verb Editor

Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing #116:throw (this none none).

list

```
__1_ player:tell(this.throw_msg);
```

```
^^^
```

```
say player:tell($string_utils:pronoun_sub(this.throw_msg));
```

Line 2 added.

list

```
1: player:tell(this.throw_msg);
```

```
__2_ player:tell($string_utils:pronoun_sub(this.throw_msg));
```

```
^^^
```

delete 1

```
player:tell(this.throw_msg);
```

---Line deleted. Insertion point is before line 1.

compile

#116:throw successfully compiled.

Quit

Now try your *pat rodent* command.

pat rodent

You pat the rodent. It looks happy.

Telling Other Players what is Happening

We also need to let other players that are in the room with the player performing the verb action what they are doing. Remember by convention we prefix messages that are designated for other players with the letter o. Lets make a property called *opat_msg* that will contain our message for other players.

```
@property rodent.opat_msg "You see %n pat a %t. Isn't %s odd!"
```

Property added with value "You see %n pat a %t. Isn't %s odd!".

Oooh, now we have some more codes to play with. The *\$string_utils:pronoun_sub* verb has lots of codes that it will automatically replace with the appropriate pronoun for you. Heres a list.

Name Codes Effect

%%	Displays a % sign
%n	Players name
%t	Objects name
%l	The indirect objects name from the command line
%d	The direct objects name from the command line
%l	The location of the player

Pronoun Codes Effect

%s	Subject pronoun (i.e. he, she or it)
%o	Object pronoun (i.e. him, her or it)
%p	Possessive pronoun (i.e. his, her or its)
%q	Possessive pronoun (i.e. his, hers or its)
%r	Reflexive pronoun (i.e. himself, herself or itself)

Note: If the code letter is capitalised, then the fist letter of the appropriate returned text is also capitalised.

Now if you look at our *opat_msg* again, it will make more sense.

Now we need to edit our *rodent:pat* verb to get it to tell other players in the room what is going on.

```
@edit rodent:pat
```

Verb Editor

Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing #116:throw (this none none).

lis

```
__1_ player:tell($string_utils:pronoun_sub(this.pat_msg));
```

```
^^^^
```

```
say player.location:announce($string_utils:pronoun_sub(this.opat_msg));
```

Line 2 added.

com

#116:throw successfully compiled.

qui

Did you notice something about the note editor commands I was typing... they were shorthand. That's right, you need only type the first three letters of any note editor command to get it to work. Easier huh...

Now back to that line we added. Every player has a property called *location* that is set to the room location they are currently in. The *\$room* object that all rooms are based on has a verb called *announce* that sends a message to all players in a room. Thus by issuing the command *player.location:announce* we are telling the MOO to tell all players in the same room as the player performing the action whatever text string follows.

As before, we use the *\$string_utils:pronoun_sub* verb to do the fiddly pronoun substitution for us, and wizzo we have a message that works for all players.

Now pat your rodent. Anyone else in the room with you now gets the message (given that the player doing the throwing is called The Crimson Wizard):

You see The Crimson Wizard pat a rodent. Isn't he odd!

Adding Size to the Rodent

Remember we created a property called *rodent.size* that contained a numerical value specifying the size of our rodent and a list called *rodent.size_list* containing descriptive text. Well, now we are going to add to our description of the rodent its size.

Whenever a player issues the look command at an object, the MOO checks if it can call a verb called *description* on that object. If that verb does not exist it calls the parent's description verb. If we write our own *description* verb, we override the

parents *description* verb and our own works instead. Do as I have done below.

@verb rodent:description tnt

In the above example tnt is a shortcut way of typing this none this that the MOO accepts. The construct *this none this* never occurs in the English language, so this is a special verb. A tnt verb can be called from other objects as well as from the objects verbs. You can not issue a command such as *description rodent* using this construct, it just does not mean anything to the MOO parser!

Back to the editor. Do as I have done below.

@edit rodent:description

enter

```
base_description = pass(@args);
```

```
base_description = base_description + " " + "This %t is " +  
this.size_list[this.size] + ":";
```

```
return $string_utils:pronoun_sub(base_description);
```

.

com

qui

As always, Ive added something extra. This time we are using a local variable called *base_description* to store our message plus a new function called *pass()* and a programming statement called *return*.

A local variable is one that is used temporarily to store the result of a calculation or hold some other value. Unlike properties, they are not permanent, their values are held only for the current execution of the verb. The next time the verb is called they are reset and used again.

Just to get you into using the on-line help more. Use command *help pass()* to see what that *pass()* function is all about.

The *return* statement instructs the verb to return the result indicated to the calling verb. In this case the calling verb is the *look* verb which calls the *description* verb that you just wrote.

We also used an index to access the text values of our *size_list*. When we have a list we can access each element in that list by using the syntax *listname[index]*, where index is a value between 1 (for the first item) and the total number of elements in the list. The index value can either be a specific number or, as in our case, a variable that can be set to access different elements in the list.

Now try *look rodent*.

look rodent

A cute animal with a long twisty tail. It has small beady eyes and whiskers. This rodent is small.

Feeding the Rodent

For fun, when we feed our rodent cheese we want to get bigger. So first lets create another object called cheese to feed our rodent with.

@create \$thing called "cheese"

@describe cheese as "A small section of tasty cheese."

Now we need some way of checking of what a player is attempting to feed our rodent to make sure that it is cheese, and not something else (like a player). We will create a new verb to do this for us.

@verb rodent:willeat tnt

We are going to program this verb so that it returns a value of one (1) if it is something the rodent will eat and a value of zero (0) if it is something the rodent dislikes. Im going shorthand with the programming now as you should be familiar enough with the editor to edit the verb (Ive added the indentations below to make it easier to read).

```
{item}= args;  
First check to make sure that the thing the player tries to feed;  
to the rodent is an object and that it exists;  
if (valid(item))  
players may not be fed to the rodent;  
    if (is_player(item))  
        return 0;  
    endif  
however cheese can be;  
    if (item.name == "cheese")  
        return 1;  
    else  
        return 0;  
    endif  
else  
return -1;
```

endif

Now lets look at what we have done.

Comments

When we are programming verbs, it helps if we can include comments in it so that when we look back at what we have done, we know what the program is doing at any one point.

We do this by enclosing comments in quotation marks. Anything that immediately begins with a quotation mark and ends with a quotation mark is ignored by the MOO.

In the above program the following line is a comment:

```
    Players may not be fed to the rodent;
```

Scattering Assignments

The first line of the program is a thing called a scattering assignment. More on that it a second. The variable *args* is a special MOO list variable that automatically contains a list of any arguments passed to a verb (eg if we typed feed cheese to rodent, then the *args* list would be {feed,to,rodent}).

Now as *args* is a list we could access each of these elements by stating:

```
    item1 = args[1];  
    item2 = args[2];  
    item3 = args[3];
```

But by using a scattering assignment we can shorthand this to:

```
    {item1,?item2,?item3}= args;
```

Using the previous example of the *args* list, these variables would then contain.

```
    item1 => feed  
    item 2 => to  
    item3 => rodent
```

So then the statement in our program *{item}= args* simply means that variable *item* contains the first argument passed to the verb. As an exercise type *help scattering*

and see what comes up (and you can learn more about what that question mark means).

In our case when we call the *rodent:willeat* from our *rodent:feed* verb we are passing the argument *dobj* (more on this later) which is the object ID of an object that the player is trying to feed our rodent.

More MOO Functions

We also have a new MOO function in this code. The *valid(<object id>)* function looks for an object of number *<object id>* and returns a true / false (i.e. a 1 or a 0 value) if that object number corresponds to a valid object.

In our program this function makes sure that what the player tries to feed the rodent is an object that actually exists otherwise the player could try and feed anything at all to the rodent, and our program would crash.

If... Else... Endif Statements

This is the most basic decision making statement in MOO syntax. It is a conditional statement in that it evaluates some sort of expression to a true or false condition, and then makes a decision based on that outcome. The *if* statement has the following syntax (note that there is no semicolon after the *if*, *else* and *endif* statements but there are after any programming code you write):

```
IF (<condition>
    <>true code>;
    ...
    <>true code>;
ELSE
    <>false code>;
    ...
    <>false code>;
ENDIF
```

In our program the first if statement checks to see if the object that has been passed as an argument to our verb is a player. This is done using a function called *is_player()*, which simply returns a value of one (1) if the object is a player, or zero (0) if the object is not. Try typing *help is_player()* for more information.

With an *if* statement if the condition equates to a value of zero, then the false condition code is activated. Any other value results in the true condition code being

activated.

You would have noticed from the help on the *is_player()* function that it returns a true / false condition. Thus by using this function as the condition for our *if* statement, the appropriate code is activated.

We also use a *if* statement to check if the item name is cheese. Notice the double equals signs in the condition! You must use double equals signs to check for equality, as a single equals sign is used to assign a value to a variable. This is a common programming error, so be aware.

Lastly we have an *else* condition with our original *if* statement that causes the program to return a 1 value if the thing the player tries to feed our rodent is not an object at all (i.e. it is an object name that does not exist in the same room as the player or does not exist at all and the player is just making things up).

More information on the *if* statement (and all the other conditional statements) can be found by typing *help statements*. Do this and read up on the *return* statement to see what it does. You should now be able to understand why we are using the *return* statement to return values to the calling verb of one (1) and zero (0), i.e. true and false.

Feed Me Cheese!

When we feed the rodent we want to send a message to the player describing the result and a message to everyone else in the room telling them what the player is doing. So we will create four messages: *feed_msg*, *feed_failed_msg*, *ofeed_msg* and *ofeed_failed_msg*.

```
@property rodent.feed_msg "The rodent sniffs at the %d, then eats its fill.  
Then before your eyes, the rodent swells in size!"
```

```
@property rodent.feed_failed_msg "The rodent daintily nibbles at %d, gets  
a funny look on its face, and spits it back out. It looks at you reproachfully."
```

```
@property rodent.ofeed_msg "You see %n feed %d to the rodent. It gets  
bigger."
```

```
@property rodent.ofeed_failed_msg "You see a rodent spit a bit of %d back  
at %n. It obviously didnt like it"
```

Now we need to add another verb called *feed* to the rodent.

@verb rodent:feed any to this

Verb added (3).

And program it (I've included the indentations for readability).

```
result = this:willeat(dobj);
If (result != -1)
  if (result == 0)
    player:tell($string_utils:pronoun_sub(this.feed_failed_msg));
    player.location:announce($string_utils:pronoun_sub(this.ofeed_failed_msg));
  else
    player:tell($string_utils:pronoun_sub(this.feed_msg));
    player.location:announce($string_utils:pronoun_sub(this.ofeed_msg));
    this.size = this.size + 1;
  endif
else
  player:tell("I see no " + dobjstr + " here");
endif
```

The only thing new here is the variable *dobj*. When any verb is called the MOO parser creates a standard set of variables containing very useful values. These are:

Variable Type Description

player	An object	the player who typed the command
this	An object	the object on which this verb was found
verb	A string	the first word of the command
argstr	A string	everything after the first word of the command
args	A list of strings	the words in `argstr`
dobjstr	A string	the direct object string found during parsing
dobj	An object	the direct object value found during matching
prepstr	A string	the prepositional phrase found during parsing
iobjstr	A string	the indirect object string
iobj	An object	the indirect object value

Strings are MOO language for text (from a string of letters, which is how they are stored in the computer).

Now try feeding different things to the rodent (including yourself). When you feed your cheese object to the rodent, make sure you look at it afterwards. If all went to plan, the rodent's description has changed to reflect its size.

A Bug!

Our program deliberately has a bug (that's the programming term for an error

caused through our own negligence) in it. If we keep feeding our rodent cheese, it gets bigger and bigger but eventually our *rodent.size* index property is incremented beyond the items in our list. This results in an error like this when we *look* at the rodent:

look rodent

```
#116:description, line 2: Range error
... called from #1:look_self (this == #116), line 1
... called from #3:l*ook (this == #62), line 13
(End of traceback)
```

A range error is the error caused when we reference an item in a list that does not exist. We need to edit our feed verb so that a rodent can get no bigger than disgusting, which is the 4th element in the list. This is done using an *if* statement, see below.

```
if (this.size > 4)
    this.size = 4;
endif
```

Add this code to your program immediately below the *this.size = this.size + 1* line of code.

Now set our rodent *size* property back to one:

```
@set rodent.size to 1
Property #116.size set to 1.
```

Feed and look at the rodent a few times and make sure that the code works. Make sure that it gets to disgusting size and then stops growing.

But, what if we added more descriptive text to our list of rodent sizes, or reduced it. We would have to keep rewriting the code too fiddly. Im going to make an improvement.

```
len = length(this.size_list);
if (this.size > len)
  this.size = len;
endif
```

Replace your code with the improved code. Note that the note editor delete command can be used with a range to delete a number of lines of code. This is

done by typing *delete* <starting line>-<ending line> (note the hyphen between the starting and ending lines).

The improved code we typed gets the number of items in the list using the *length()* function which returns the number of elements in our list. We store this into a temporary variable called *len* and then use this to check to see if our *rodent.size* value is too big.

To see that this worked. Lets add some more descriptions to our *rodent.size_list* list.

@edit rodent.size_list

Note Editor

Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing "rodent"(#116).size_list.

lis

1: small

2: large

3: huge

__4_ disgusting

^^^

ins 4

__3_ huge

^^4^ disgusting

say fat and bulbous

Line 4 added.

ins 6

__5_ disgusting

^^^

say totally foul

Line 6 added.

save

Text written to "rodent"(#116).size_list.

qui

Now reset the *rodent.size* property back to a value of one and feed your rodent.

Exercise 1

When players feed the rodent they should get a message when it has reached its maximum size that it is no longer hungry. Edit the *feed* verb so that this happens (remember to include messages for both the player and others in the room with the

player).

Forking

After our rodent hasn't been fed for a while we want it to shrink again in size so that players can feed it again. This is where the fork statement can be used. The fork statement allows a verb to execute code after a specified interval in seconds.

```
Fork (<num seconds>)  
  <statements>  
endfork
```

Add this code to the end of the existing *feed* verb.

```
if (this.size <= length(this.size_list) && this.size > 1)  
  shrink = 1;  
else  
  shrink = 0;  
endif  
if (shrink == 1)  
  fork (10)  
    player:tell("The rodent shrinks.");  
    this.size = this.size - 1;  
  endfork  
endif
```

This code includes some error checking to ensure that we can not shrink our rodent too many times. The *if...endif* code checks to see that our rodent is not at its full size AND that it is not at its smallest size, using a variable called *shrink* to store a true / false value depending on the outcome.

The special character combination **&&** tells the MOO that one condition AND another condition must both be true to return a true result to the *if* statement. There is also the character combination **||** (that's two of the same character you used in the *@dig* command when specifying exits) which means OR, i.e. one condition OR another condition can be true for the result to the *if* statement to be true..

If *shrink* is true (i.e. a value of 1 or more), then the *fork...endfork* code is executed after a ten second delay. Now I know that ten seconds is a bit short (even for a rodent) for it to get hungry again, this is just to illustrate the use of the fork statement. Really we should use the MOOs time functions to give a decent / realistic delay.

Starving the Rodent

We are now going to shrink our rodent based on a real-time lapse in the MOO world by using the time functions. First we need to create two properties, one to store the time the rodent was last fed and another to set how much time needs to pass before rodent shrinks through lack of food.

```
@prop rodent.reference_time 0
```

```
@prop rodent.time_interval 5
```

Notice the shorthand for the *@property* command.

Now add the following line of code to the *rodent:feed* verb just before or after the point we increase the rodents *.size* property.

```
this.reference_time = time();
```

Now delete the entire section in the *rodent:feed* verb that made the rodent shrink in size. We need to change where our rodent shrinks to occur when a player looks at it. Remember that when a player looks at an object the *description* verb is called if it exists on that object. We previously changed out *rodent:description* verb to show the rodents size, now we are going to expand that code. This is what the finished *rodent:description* verb looks like (I've added the indentations for readability) Change yours to be the same.

```
base_description = pass(@args);  
if (this.reference_time != 0)  
    time_passed = time() - this.reference_time;  
    time_intervals = time_passed / this.time_interval;  
    this.size = this.size - time_intervals;  
    if (this.size < 1)  
        this.size = 1;  
    endif  
endif  
base_description = base_description + " " + "This %t is " +  
this.size_list[this.size] + " .";  
return $string_utils:pronoun_sub(base_description);
```

The second line of code error checks to make sure that the rodent has been fed at least once (remember the reference time is set in the *rodent:feed* verb), otherwise we would get an error message and the verb would crash.

Then we do some basic arithmetic to find out how much time has passed since the rodent was last fed, and break this time down to a number of time intervals that have passed. We then set the size of our rodent based on this.

By changing the value of *rodent.time_interval* we can make our rodent leaner with time. Currently we have set it to a period of 5 seconds, but we could make it anything we liked- eg 60 would be one minute, 3600 would be one hour, etc.

Exercise 2

Change the *rodent.size_list* elements to reflect the state of hunger of the rodent instead of its size.

The Rodent Lives

Now we are going to make our rodent come to life by making it react to things that happen in the same room as it.

Remember how when a player issues the look command the MOO checks to see if there is a verb called description on the direct object specified. If so, that verb is run. In the same way whenever a verb calls a rooms *:announce_all* verb, the MOO runs any verbs called *:tell* on any objects located in that room.

Thus by simply adding a verb called tell to our rodent, we now have the capacity to make it react to players. Before you do this, drop the rodent or nothing will happen.

@verb rodent:tell tnt

Add the following code to our *rodent:tell* verb.

this.location:announce_all("The rodent startles and scuttles to a corner of the room.");

Now say something.

Making the Rodent less Frisky

OK, you can see there is a problem here. What we have happening is a loop where the rodent is responding to the rodents responses. Poor thing will get exhausted at that rate!

What we need to do is use a variation of the announce verb called *:announce_all_but*. Replace our code in the *rodent:tell* verb with:

```
this.location:announce_all_but({this},"The rodent startles and scuttles to a corner of the room.");
```

Now try saying something. Much better wasn't it. The `:announce_all_but` verb takes a list of objects to exclude as its first parameter, hence the inclusion of the `{this}` part of the code.

Lets make our rodent more interesting. We will create a list of actions that our rodent will perform randomly whenever there is noise in the room.

```
@prop rodent.action_list {}
```

Now add some actions to the list. Add whatever you like (eg A rodent makes a cute squeaking noise, A rodent fluffs up and looks fierce.) but make at least four different actions.

Replace the `rodent:tell` code with this:

```
len = length(this.action_list);  
this.location:announce_all_but({this}, this.action_list[random(len)]);
```

Say something a few times and see the effect.

We've added another function this time called `random(<range>)`. This function simply returns a random number in the range `<range>`. In our code we used the number of items in the `action_list` property to get a random result based on this number. We then used the random number as an index to get a message from our `action_list` list.

The fact that the rodent always does something every time there is a noise in the room can wear thin very quickly. Thus it is better if we adjust our code so that the rodent only responds once in a while. Try this:

```
if (random(3) == 1)  
  len = length(this.action_list);  
  this.location:announce_all_but({this}, this.action_list[random(len)]);  
endif
```

What we have done here is make our rodent respond randomly with a random message. If we wanted a quieter rodent, we could increase the random range in the first `if` statement (or even assign a property to hold a value that was randomised).

Cheese... You have Cheese?

Just for fun, lets get our rodent to pay particular attention to any players that have cheese in their pockets (i.e. in their *player.contents* list). Here is the complete code (indents added for clarity):

```
if (random(3) == 1)
  len = length(this.action_list);
  this.location:announce_all_but({this}, this.action_list[random(len)]);
  player_list = players();
  for n in (player_list)
    if (n.location == this.location)
      for item in (n.contents)
        if (item.name == "cheese")
          n:tell("A rodent sniffs the air and eyes your pocket hungrily.");
          message = You see a rodent eye + n.name + hungrily.;
          this.location:announce_all_but({this,n}, message);
        endif
      endfor
    endif
  endfor
endif
```

This code contains a couple of new things. The most important of which is the *for... endfor* looping statement.

For... Endfor Loops

A *for... endfor* loop allows us to look at the contents of a list one element at a time, and perform operations on that list element. It has the format:

```
For <name> in (<list name>)
  ...
  <statements>
  ...
endfor
```

If you look at our code, you will see that immediately before the for statement, another new MOO function is called named *players()*, the result of which is stored in a local variable called *player_list*. As you may have guessed, the *players()* function simply returns a list of all the object numbers of all current players in the MOO.

Thus, by using the *for... endfor* loop we are able to check every player in the MOO

and see if their location is the same as the rodents. If it is, then we use another *for... endfor* loop to check what they are carrying (their *contents* list). If they are carrying cheese then that player is told that the rodent is interested in them for some reason. Notice that the call of the *announce_all_but* verb in this section also excludes the player with the cheese from receiving the message.

Throwing the Rodent

Remember in our original specifications for our rodent we decided that it did not like to be thrown. All MOO objects by default have a *throw* verb which has the same effect as the *drop* verb. We need to override the default *throw* verb.

If you execute the *@show \$thing* command (do this now) you will note that the default *throw* verb is named *th*row*, which is a shorthand way of naming a verb so that a player can use a shorthand command (in this case *th*) as well as typing the whole thing.

Pick up the rodent (if you do not have it already) and *throw* it. As you can see all is normal. Now lets override the default *throw* verb. In order to do this we have to call our new verb exactly the same. To examine how the default *throw* verb has been created, use the *@show* command like this:

@show \$thing:throw

#5:d*rop th*row

Owner: The Crimson Wizard (#2)

Permissions: rxd

Direct Object: this

Preposition: none

Indirect Object: none

For the moment do not worry about the permissions, we will be looking at this in detail soon. The stuff we are interested in is the *dobj*, *prep* and *iobj* specifications for the verb. We need to use the same. Hence:

@verb rodent:th*row this none none

Now try throwing the rodent. As you can see, nothing at all happens (our verb contains no code as yet).

Exercise 3

Add code to the *rodent:throw* verb so that the player receives a message telling them that the rodent does not like to be thrown and that it clings to them. Also add a message that tells other players how nasty the player is being trying to throw a

poor rodent.

Now for the rodents retaliatory strike. Since the player has been mean enough to try and throw the rodent away, instead of gently dropping it, the rodent will go berserk and randomly steal one item of the players and run off with it. The MOO function *move(<object>, <location>)* is used to do this. By now you should be able to figure out how this function works (if not try the *help move()* command).

Exercise 4

Add the code to the *rodent:throw* verb to do the above. Remember to include player messages. Be creative as you like. Hint: Use the *length()* function to get the number of items a player has (*player.contents* is a list of objects remember).

If you get really stuck, Appendix B contains the code I used.

Blessed are the Cheese Makers

One of the things that is still wrong with our rodent object is that it never actually eats the cheese the player feeds it. However, right at this point if it did there would be no more cheese in the MOO as the cheese object would be destroyed.

Thus I thought it would be a good idea to create a machine that creates cheese so that players can feed the rodent.

Making the Cheese Fertile

The first thing we need to do is to make the existing cheese object able to produce offspring cheeses (i.e. we are going to change our existing cheese object so that it is able to become a parent to other cheese objects).

To avoid confusion between the original cheese object and any subsequent cheeses, we are going to rename it. Thus:

@rename cheese to Generic Cheese

Now we are going to make it fertile by adding a permission bit. Permissions tell the MOO what an object (or a verb or property) is permitted to have done to it. There are six possible permissions.

Property Definition

- r The object / verb / property may be read using the *@list* commands.
- w The object / verb / property may be written to by anyone. It is not recommended to set anything with this permission.
- f Means that an object is fertile and can have child objects created from it.
- c A property can be set from the command line using the *@set* command.
- x A verb may be executed (i.e. used)
- d A verb will debug. This is a default.

In summary, the permissions that objects, properties and verbs can have are:

```
<object-permissions> r, w, f
<property-permissions>      r, w, c
<verb-permissions>   r, w, x, d
```

Now that you know what these permissions mean, lets add a permission bit to our Generic Cheese object using the *@chmod* command. Type help *@chmod* for more info on how to use this command. We will type:

```
@chmod Generic Cheese +f
```

This makes our Generic Cheese object fertile. As a test, lets make a new cheese object. Type:

```
@create Generic Cheese called cheese
```

Now we need to alter our *rodent:feed* verb so that when we feed cheese to our rodent, the cheese object is destroyed. This is done using the MOO function `recycle(<object>)`.

Thus all we have to do is added the following line of code to our *rodent:feed* verb, anywhere in the section of the code that handles a successful feeding.

```
recycle(dobj)
```

Once you have done this, feed your cheese to the rodent. Notice that afterwards it is gone! You would have to keep using the *@create* command to make more cheeses to feed the rodent if you wanted to.

The Cheese Machine

What we need is to build a cheese machine that manufactures cheese objects and

gives them to players. Like this:

```
@create $thing called Cheese Machine,cm
```

Notice that I included an alias `cm` to make it simpler for players to do things to the machine. Now set its description.

```
@describe cm as "A strange bulbous machine with a lever on one side."
```

And create a new verb called `pull` with the arguments *any on this*. This tells the MOO that we expect the command to be in the format *pull <something> on cheese machine* or *pull <something> on cm*. Thus:

```
@verb cm:pull any on this
```

In order to create cheese objects, we will need to know the object ID of our Generic Cheese object. Type:

```
#Generic Cheese
```

The hash (`#`) symbol preceding the name of an object as a command causes the MOO to tell use the object ID of that object. Handy. Remember your object ID.

And edit the verb to have this code (indents added for clarity). Replace the `#120` near the bottom with the object ID of your Generic Cheese object (eg `#160`, `#130` or whatever).

```
if (dobjstr == "lever")  
  has_cheese = 0;  
  for n in (player.contents)  
    if (n.name == "cheese")  
      has_cheese = 1;  
    endif  
  endfor  
  if (has_cheese)  
    player:tell("You allready have some cheese.");  
  else  
    object = player:_create(#120);  
    $building_utils:set_names(object, "cheese");  
    move(object, player);  
    player:tell("A chunk of cheese shoots out from a slot in the machines side, and  
lands in your pocket.");  
  endif  
else  
  player:tell("The machine has no " + dobjstr + " to pull.");
```

endif

This code first performs some error checking to ensure that the player is trying to pull the lever on the machine and not something else, displaying an error message to the player if they try something odd.

It then checks to ensure that the player does not already have some cheese. If not, we then use the `_create(<object id>)` verb which is part of every player to create a new Generic Cheese object (note the unusual use of the underscore character). We then name the new object using the `$building_utils:set_names` verb and finally move the finished object to the players contents list (i.e. their inventory).

Now you can have players pull the lever on our cheese machine and get cheese to feed to the rodents. Oooh, we have a little ecosystem going here.

Locking Up

Sometimes in your virtual world you may want to stop a player for accessing a room if they do not own a specific object (such as a key) or stop a player from taking an object from a room. The `@lock` command does this for us. Type `help locking` to get an idea of the command. Now type `help keys` and have a read.

To remove any locking that you have done to an object, use the `@unlock` command.

Stop Stealing the Machine

Our cheese machine has a problem in that any player can come along and pick the cheese machine up and take it somewhere else. Well use the `@lock` command to stop this:

@lock cm with here

Now try and take the machine. The message you are getting is a message property called `take_failed_msg`, so you can set that with `@take_failed cm is Blah`. You can also set the `otake_failed` message to tell others in the room of the failure.

Keys for Doors

As a more interesting example of using the `@lock` command, lets make a key that a player must have in order to enter a door. First we make the key:

@create \$thing called key

Now we can use the `@lock` command to stop players from exiting a specific direction unless they have the key object. As an example, if we had an exit called down we could lock it to the key object by issuing the command:

`@lock down with key`

Easy huh. With exit objects, if a player does not have the required object to let them pass then they receive the message set by the `.nogo_msg` property of the exit. For example we could set the `.nogo_msg` property of our down exit thus:

`@nogo down is The door to the dungeon is locked. You need a key.`

Other Stuff

This section deals with some of the miscellaneous programming and commands that are useful at various points in MOO programming.

Help Messages

We can add a help message to an object by defining a new list property called `.help_msg`. Once added, use the `@edit` command to edit the help message to whatever you like. Then when a player types `help <name of object>`, the MOO engine will display your help message.

The Examine Verb

When a player uses the examine command on an object, the default behaviour of the MOO is to return all the verbs that have been defined on that object. At times you may wish to alter this behaviour so that players are left in the dark as to how an object works.

When a player issues the examine command a the MOO looks for a verb called `examine_verbs` on the object first of all. If it does not exist, the default `examine` verb is called from the generic player object. So, if we want to change what players see with the examine command, then we simply add a `:examine_verbs` verb to our object thus:

`@verb <my object>:examine_verbs tnt`

We then simply edit the new verb, and do whatever we like. When a player comes along and types *examine <my object>*, your code is executed.

Asking for Player Input

Sometimes you may want to get information off a player that can then be processed. This is where the *read()* function is used. Here is an example of code that uses the *read()* function.

```
player:tell(What is your name?);  
buffer = read();  
player:tell(Really, your name is actually + buffer + that is bizarre!);
```

Looking at Code

Often you may wish to refer to code that you have written previously. Now you could go to the trouble of editing the old verb to see your code and then edit your new verb, but there is an easier way. The *@list <object>:<verb>* command.

The *@list* command simply displays the code of a verb that you own. So for example you could issue the command:

```
@list rodent:feed
```

Try it on some of your verbs.

Changing Verb Arguments

Every now and then you start writing a verb and then realise that you have stuffed up the verb arguments. The *@args* command allows you to change them without having to start all-over-again. It has the format:

```
@args <object>:<verb> <dob> <prep> <iob>
```

Lets say that we create a verb called feed thus:

```
@verb rodent:feed this to any
```

Now you can see that this verb would be wrong because the player would have to type *feed rodent to <something>* for it to work. Now rather than using the *@rmverb* command and recreating it, we can simply use the *@args* command thus:

@args rodent:feed any to this

Appendix A: Programming Cheat Sheet

The following section provides a quick reference sheet for the functions and special variables in the MOO programming environment. Enjoy.

Special variables in functions

this the object
caller value of "this" in calling function, or player at top-level
player player that invoked the task
verb verb named used to invoke verb
args list of arguments

[For commands]

argstr everything after first word of command
dobjstr direct object string
dobj direct object value
prepstr preposition string
iobjstr indirect object string
iobj indirect object value

Exprs

cond ? expr1 | expr2 conditional expression shorthand
\$foo Easy way to refer to objects on object #0, ie #0.foo
"str" + "str" join two strings together
expr in list list membership, returns index or 0 (false)

Statements

if (expr); ... elseif (expr); ... else; ... endif
for variable in (expression); ... endfor
for variable in [expr1..expr2]; ... endfor
while (expr); ... endwhile
fork (expr); ... endfork
fork name (expr); ... endfork

Note that strings comparisons are case-insensitive.

Maths Functions

min(n1, n2, ...) -- minimum of n1,n2,...
max(n1, n2, ...) -- maximum of n1,n2,...
abs(n) -- absolute value of n
sqrt(n) -- square root of n, rounded down
random(n) -- random integer between 1 and n inclusive

\$math_utils:

Trigonometric/Exponential functions

:sin(a),cos(a),tan(a) -- returns 10000*(the value of the corresponding trigonometric function) angle a is in degrees.
:arctan([x],y) -- returns arctan(y/x) in degrees in the range -179..180. x defaults to 10000. Quadrant is that of (x,y).
:exp(x[,n]) -- calculates e^x with an nth order taylor polynomial

Statistical functions

:combinations(n,r) -- returns the number of combinations given n objects taken r at a time.
:permutations(n,r) -- returns the number of permutations possible given n objects taken r at a time.

Number decomposition

:div(n,d) -- correct version of / (handles negative numbers correctly)
:mod(n,d) -- correct version of % (handles negative numbers correctly)
:divmod(n,d) -- {div(n,d),mod(n,d)}
:parts(n,q[,i]) -- returns a list of two elements {integer, decimal fraction}

Other math functions

:sqrt(x) -- returns the largest integer $n \leq$ the square root of x
:pow(x,n) -- returns x^n
:factorial(x) -- returns $x!$

Series

:fibonacci(n) -- returns the 1st n fibonacci numbers in a list
:geometric(x,n) -- returns the value of the nth order geometric series at x

Integer Properties

:gcd(a,b) -- find the greatest common divisor of the two numbers
:lcm(a,b) -- find the least common multiple of the two numbers
:are_relatively_prime(a,b) -- return 1 if a and b are relatively prime
:is_prime(n) -- returns 1 if the number is a prime and 0 otherwise

Miscellaneous

:random(n) -- returns a random number from 0..n if n > 0 or n..0 if n < 0
:random_range(n[,mean]) -- returns a random number from mean - n..mean + n with mean defaulting to 0
:simpson({a,b},{f(a),f((a+b)/2),f(b)}) -- returns the numerical approximation of an integral using simpson's rule

Bitwise Arithmetic

:AND(x,y) -- returns x AND y
:OR(x,y) -- returns x OR y
:XOR(x,y) -- returns x XOR y (XOR is the exclusive-or function)
:NOT(x) -- returns the complement of x

Note that all bitwise manipulation is of 32-bit values.

Time

time() -- current time in seconds since midnight GMT, 1 Jan 70
ctime([time]) -- time (or current time) converted to a human-readable string

\$time_utils:

Note that time values are given in seconds-since-1970

:dhms (time) => string .DD:HH:MM:SS
:english_time (time[, reference time]) => string of y, m, d, m, s

Converting to seconds

:to_seconds ("hh:mm:ss") => seconds since 00:00:00
:from_ctime (ctime) => corresponding time-since-1970
:from_day (day_of_week, which) => time-since-1970 for the given day*
:from_month (month, which) => time-since-1970 for the given month*

(* the first midnight of that day/month)

:day ([c]time) => what day it is
:month ([c]time) => what month it is
:ampm ([c]time[, precision]) => what time it is, with am or pm
:time_sub (string, time) => substitute time information
:sun ([time]) => angle between sun and zenith

Strings

<code>index(str1, str2 [, case-matters])</code>	index of first str2 in str1
<code>rindex(str1, str2 [, case-matters])</code>	index of last str2 in str1
<code>strcmp(str1, str2)</code>	case-sensitive string comparison
<code>strsub(subject, what, with [, case-matters])</code>	substitution in a string
<code>crypt(string [, salt])</code>	one-way string encryption
<code>match(str1, str2 [, case-matters])</code>	match first pattern str2 in str1
<code>rmatch(str1, str2 [, case-matters])</code>	match last pattern str2 in str1
<code>substitute(template, subs)</code>	perform substitutions on template

\$string_utils

Conversion routines

`:from_list (list [,sep]) => "foo1foo2foo3"`
`:english_list (str-list[,none-str[,and-str[, sep]]) => "foo1, foo2, and foo3"`
`:title_list*c (obj-list[,none-str[,and-str[, sep]]) => "foo1, foo2, and foo3" or => "Foo1, foo2, and foo3"`
`:from_value (value [,quoteflag [,maxlistdepth]]) => "{foo1, foo2, foo3}"`
`:english_number(42) => "forty-two"`
`:english_ordinal(42) => "forty-second"`
`:ordinal(42) => "42nd"`
`:group_number(42135 [,sep]) => "42,135"`

Type checking

`:is_numeric(string) => return true if string is composed entirely of digits`

Parsing

`:explode (string,char) -- string => list of words delimited by char`
`:words (string) -- string => list of words (like command line parser)`
`:word_start (string) -- string => list of start-end pairs.`

Matching

`:match_string(string, pattern, options) => * wildcard matching`
`:find_prefix(prefix, string-list) => list index of elt starting with prefix`
`:index_delimited(string,target[,case]) => index of delimited string occur.`
`:match(string, [obj-list, prop-name]+) => matching object`
`:match_player(string-list[,me-object]) => list of matching players`
`:match_object(string, location) => default object match...`

Pretty printing

:space (n/string[,filler]) => n spaces
:left (string,width[,filler]) => left justified string in field
:right (string,width[,filler]) => right justified string in field
:center/re (string,width[,filler]) => centered string in field
:columnize/se (list,n[,width]) => list of strings in n columns

Substitutions

:substitute (string,subst_list [,case]) -- general substitutions.
:pronoun_sub (string/list[,who[,thing[,location]]) -- pronoun substitutions.
:pronoun_sub_secure (string[,who[,thing[,location]]],default) -- substitute and check for names.
:pronoun_quote (string/list/subst_list) -- quoting for pronoun substitutions.

Miscellaneous string munging:

:trim (string) => string with outside whitespace removed.
:triml (string) => string with leading whitespace removed.
:trimr (string) => string with trailing whitespace removed.
:strip_chars (string,chars) => string with all chars in `chars' removed.
:strip_all_but(string,chars) => string with all chars not in `chars' removed.
:capitalize/se(string) => string with first letter capitalized.
:uppercase/lowercase(string) => string with all letters upper or lowercase.
:names_of (list of OBJ) => string with names and object numbers of items.

:alphabet => "abcdefghijklmnopqrstuvwxyz"

Lists

listappend(list, value [, index])	adding an element at the end of a list
listinsert(list, value [, index])	adding an element at the head of a list
listset(list, value, index)	updating a list at some index
listdelete(list, index)	removing an element from a list
setadd(list, element)	adding an element to a set represented as a list
setremove(list, element)	removing an element from such a set

\$list_utils:

:append (list,list,..)	result of concatenating the given lists
:reverse (list)	reversed list
:remove_duplicates (list)	list with all duplicates removed
:compress (list)	list with consecutive duplicates removed
:setremove_all (list,elt)	list with all occurrences of elt removed

:find_insert (sortedlist,e) index of first element > e in sortedlist
 :sort (list[,keys]) sorted list

 :make (n[,e]) list of n copies of e
 :range (m,n) => {m,m+1,...,n}
 :arrayset (list,val,i[,j,k...]) => array modified so that list[i][j][k]==val

Mapping functions (take a list and do something to each element)

map_prop ({o...},prop) => list of o.(prop) for all o
 :map_verb ({o...},verb[,args]) => list of o:(verb)(@args) for all o
 :map_arg ([n],obj,verb,{a...},args) => list of obj:(verb)(a,@args) for all a

Association list functions

An association list (alist) is a list of pairs (2-element lists), though the following functions have been generalized for lists of n-tuples (n-element lists). In each case i defaults to 1.

:assoc (targ,alist[,i]) => 1st tuple in alist whose i-th element is targ
 :iassoc (targ,alist[,i]) => index of same.
 :assoc_prefix (targ,alist[,i]) => ... whose i-th element has targ as a prefix
 :iassoc_prefix(targ,alist[,i]) => index of same.
 :slice (alist[,i]) => list of i-th elements
 :sort_alist (alist[,i]) => alist sorted on i-th elements.

\$seq_utils

A sequence is a set of integers (*)

:add (seq,f,t) => seq with [f..t] interval added
 :remove (seq,f,t) => seq with [f..t] interval removed
 :range (f,t) => sequence corresponding to [f..t]
 {} => empty sequence
 :contains (seq,n) => n in seq
 :size (seq) => number of elements in seq
 :first (seq) => first integer in seq or E_NONE
 :firstn (seq,n) => first n integers in seq (as a sequence)
 :last (seq) => last integer in seq or E_NONE
 :lastn (seq,n) => last n integers in seq (as a sequence)

 :complement(seq) => [-2147483648..2147483647] - seq
 :union (seq,seq,...)
 :intersect(seq,seq,...)

:extract(seq,array) => array[@seq]
:for([n,]seq,obj,verb,@args) => for s in (seq) obj:verb(s,@args); endfor

:tolist(seq) => list corresponding to seq
:tostr(seq) => contents of seq as a string
:from_list(list) => sequence corresponding to list
:from_sorted_list(list) => sequence corresponding to list (assumed sorted)
:from_string(string) => sequence corresponding to string

For boolean expressions, note that the representation of the empty sequence is {} (boolean FALSE) and all non-empty sequences are represented as nonempty lists (boolean TRUE).

The representation used works better than the usual list implementation for sets consisting of long uninterrupted ranges of integers. For sparse sets of integers the representation is decidedly non-optimal (though it never takes more than double the space of the usual list representation).

(*) Actually what this package implements is sets of integers-mod- 2^{32} , but this assumes the underlying machine on which the server runs has 32-bit integers. If not, you need to change this.maxneg to be the largest negative ("smallest"?) integer available.

\$set_utils

This object is useful for operations that treat lists as sets (i.e., without concern about order and assuming no duplication).

:union(set, set, ...) => union
:intersection(set, set, ...) => intersection

:diff*erence(set 1, set 2, ..., set n)
=> result of removing all elements of sets 2..n from set 1.
:exclusive_or(set, set, set, ...)
=> all elements that are contained in exactly one of the sets

:contains(set 1, set 2, ..., set n)
=> true if and only if all of sets 2..n are subsets of set 1

Objects

valid(object) -- testing whether an object exists
create(parent [, owner(*)]) -- creating a new MOO object

recycle(object) -- destroying a MOO object
 move(object, where) -- altering the object-containment hierarchy
 chparent(object, new-parent) -- altering the object-inheritance hierarchy
 parent(object) -- object's parent in the inheritance hierarchy
 children(object) -- object's children in the inheritance hierarchy
 max_object() -- the highest-numbered object in the MOO
 renumber(obj) -- changes an object's number to lowest available one*
 reset_max_object() -- resets max_object() to the largest valid object*

properties(object) -- a list of the properties defined on an object
 add_property(object, prop-name, value, info) -- add a new property
 delete_property(object, prop-name) -- remove a property
 property_info(object, prop-name) -- {owner, perms}info on a prop
 set_property_info(object, prop-name, info) -- setting same
 is_clear_property(object, prop-name) -- find out if a prop. is "clear"
 clear_property(object, prop-name) -- make a property "clear"

verbs(object) -- a list of the verbs defined on an object
 add_verb(object, info, args) -- add a verb to an object
 delete_verb(object, verb-name) -- remove a verb from an object
 verb_info(object, verb-name) -- {owner, perms, names}info for a verb defn.
 verb_args(object, verb-name) -- {dobj, prep, iobj}argument info for a verb
 verb_code(object, verb-name [, fully-paren [, indent]]) -- program listing
 set_verb_info(object, verb-name, {owner, perms, names})
 set_verb_args(object, verb-name, {dobj, prep, iobj})
 set_verb_code(object, verb-name, {line, line, ...})

\$object_utils

Examining everything an object has defined on it

:all_verbs (object) => like it says
 :all_properties (object) => likewise
 :findable_properties(object) => tests to see if caller can "find" them
 :owned_properties (object[, owner]) => tests for ownership

Investigating inheritance

:ancestors(object[,object...]) => all ancestors
 :descendants (object) => all descendants
 :ordered_descendants(object) => descendants, in a different order
 :leaves (object) => descendants with no children
 :branches (object) => descendants with children
 :isa (object,class) => true iff object is a descendant of class (or ==)

Considering containment

:contains (obj1, obj2) => Does obj1 contain obj2 (nested)?
:all_contents (object) => return all the (nested) contents of object

Verifying verbs and properties

:has_property(object,pname) => false/true according as object.(pname) exists
:has_verb (object,vname) => false/{#obj}according as object:(vname) exists
:has_callable_verb => same, but verb must be callable from a program
:match_verb (object,vname) => false/{location, newvname}
(identify location and usable name of verb)

Commands

\$command_utils

Detecting and Handling Failures in Matching

:object_match_failed(match_result, name)
Test whether or not a :match_object() call failed and print messages if so.
:player_match_failed(match_result, name)
Test whether or not a :match_player() call failed and print messages if so.
:player_match_result(match_results, names)
...similar to :player_match_failed, but does a whole list at once.

Reading Input from the Player

:read() -- Read one line of input from the player and return it.
:yes_or_no([prompt])
-- Prompt for and read a `yes' or `no' answer.
:read_lines() -- Read zero or more lines of input from the player.
:dump_lines(lines)
-- Return list of lines quoted so that feeding them to
:read_lines() will reproduce the original lines.

Utilities for Suspending

:running_out_of_time()
-- Return true if we're low on ticks or seconds.
:suspend_if_needed(time)
-- Suspend (and return true) if we're running out of time.

Client Support for Lengthy Commands

:suspend(args) -- Handle PREFIX and SUFFIX for clients in long commands.

Code

\$code_utils:

:parse_propref("foo.bar") => {"foo","bar"}(or 0 if arg. isn't a prop. ref.)
:parse_verbref("foo:bar") => {"foo","bar"}(or 0 if arg. isn't a verb ref.)
:parse_argspec("any","in","front","of","this","baz"...) => {"any", "in front of", "this"}, {"baz"...}
(or string if args don't parse)

:tonum(string) => number (or E_TYPE if string is not a number)
:toobj(string) => object (or E_TYPE if string is not an object)
:toerr(number or string) => error value (or 1 if out of range or unrecognized)
:error_name(error value) => name of error (error_name(E_PERM) => "E_PERM")

:verb_perms() => the current task_perms (as set by set_task_perms()).
:verb_location() => the object where the current verb is defined.
:verb_documentation([object,verbname]) => documentation at beginning of verb code, if any -- default is the calling verb

Preposition routines

:prepositions() => full list of prepositions
:full_prep ("in") => "in/inside/into"
:short_prep("into") => "in"
:short_prep("in/inside/into") => "in"
:get_prep ("off", "of", "the", "table") => {"off of", "the", "table"}

Verb routines

:verbname_match (fullname,name) => can `name' be used to call `fullname'
:find_verb_named (object,name[,n]) => verb number or -1 if not found
:find_callable_verb_named (object,name[,n]) => verb number or -1 if not found
:find_verbs_containing (pattern[,object|objlist])

Verbs that do the actual dirty work for @show:

:show_object (object)
:show_property(object,propname)
:show_verbdef (object,verbname)

Dirty work for explain_syntax

:explain_verb_syntax(thisname,verbname,@verbargs)

A random but useful verb

:verb_or_property(object,name[@args]) => result of verb or property call,
or E_PROPNF

Building

\$building_utils

:make_exit(spec,source,dest[,don't-really-create]) => a new exit
spec is an exit-spec as described in `help @dig'

:set_names(object, spec) - sets name and aliases for an object

:parse_names(spec) => list of {name, aliases}

in both of these, spec is of the form

<name>[[,:]<alias>,<alias>,...]

(as described in `help @rename')

:recreate(object, newparent) - effectively recycle and recreate object
as a child of newparent

:transfer_ownership(object, old-owner, new-owner) - just what it sounds like

Other

pass(arg, ...) -- calling a verb defined on this object's parent

eval(string) -- parsing and executing strings as MOO code

notify(player, string) -- sending text to a player's terminal

read() -- reading a line of input from the player (*)

output_delimiters(player) -- return {prefix,suffix}set by PREFIX/SUFFIX cmds

typeof(value) -- determining the data type of a value

tostr(value, ...) -- converting any set of values into a string

tonum(value) -- converting any non-list value into a number

toobj(value) -- converting any non-list value into an object

length(value) -- returns the length of a string or list

is_player(object) -- testing whether or not object is a player

players() -- a list of all players, active or not

connected_players() -- a list of all currently-connected players

idle_seconds(player) -- seconds since given player typed anything

connected_seconds(player) -- seconds given player has been logged in

boot_player(player) -- disconnect player from the MOO immediately*

set_player_flag(player, value) -- set/clear player bit; boot player if clear*

connection_name(player) -- server-assigned name for player's connection

open_network_connection(@args) -- open a connection to another network site

caller_perms() -- the player whose permissions your caller was using

set_task_perms(player) -- changing permissions of the running task *
callers() -- list of {obj, verb, owner, vloc, player}: this task's stack
suspend(secs) -- suspending the current task for a number of seconds
seconds_left() -- number of seconds left in the current task
ticks_left() -- number of ticks left in the current task
task_id() -- a random number representing the currently-running task
queued_tasks() -- list of {id,start,0,20000,owner,obj,verb,line,this}
kill_task(id) -- delete one of your tasks from the queue

server_log(string) -- add a comment to the server log file
server_version() -- a string of three numbers "major.minor.release"
memory_usage() -- {{blocksize, nused, nfree}, ...}, server's memory stats
shutdown(msg) -- print msg and kill the server *
dump_database() -- what it says *

\$wiz_utils

The following functions are substitutes for various server builtins.
Anytime one feel tempted to use one of the expressions on the right,
use the corresponding one on the left instead. This will take care
of various things that the server (for whatever reason) does not handle.

:set_programmer(object) object.programmer = 1;
 chparent object to \$prog
 send mail to \$prog_log

:set_player(object[,nochown]) set_player_flag(object,1);
 set player flag,
 add name/aliases to \$player_db,
 and maybe do a self chown.

:unset_player(object[,newowner]) set_player_flag(object,0);
 unset player flag,
 remove name/aliases from \$player_db
 chown to newowner if given

:set_owner(object, newowner) object.owner = newowner;
 change ownership on object
 change ownership on all +c properties
 juggle .ownership_quotas

:set_property_owner(object, property, newowner)
 change owner on a given property
 if this is a -c property, we change the owner on all descendants

for which this is also a -c property.

Polite protest if property is +c and newowner != object.owner.

`:set_property_flags(object, property, flags)`

change the permissions on a given property and propagate these to *all descendants*. property ownership is changed on descendants where necessary.

`$match_utils`

`:match`

`:match_nth`

`:match_verb`

`:match_list`

`$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`

`$perm_utils`

`:controls`

`:apply`

`:caller`

Gender

`$gender_utils:`

Defines the list of standard genders, the default pronouns for each, and routines for adding or setting pronoun properties on any gendered object.

-- Properties

`.genders` -- list of standard genders

`.pronouns` -- list of pronoun properties

`.ps .po .pp .pq .pr .psc .poc .ppc .pqc .prc`

-- lists of pronouns for each of the standard genders

If foo is of gender this.gender[n],
then the default pronoun foo.p is this.p[n]
(where p is one of ps/po/pp/pq...)

-- Verbs

:set(object,newgender) -- changes pronoun properties to match new gender.

:add(object[,perms[,owner]]) -- adds pronoun properties to object.

:get_pronoun (which,object) -- return pronoun for a given object

:get_conj*ugation(verbspec,object) -- return appropriately conjugated verb

Appendix B: Exercise Answers

Exercise 4:

```
player:tell("The rodent glares at you as it clings viscosly to your hand.");  
len = length(player.contents);  
item_no = random(len);  
item = player.contents[item_no];  
player:tell("In a blinding flash of speed it runs around your body and steals your " +  
item.name + ", dumping it in a corner of the room.");  
move(item, player.location);
```