

```

-----
      T H E      U N O F F I C I A L      M O O      P R O G R A M M I N G
      T      U      T      O      R      I      A      L
      (an experimental beginner's guide to making cool stuff in MOOcode)
-----
(c) 1994 by canton becker. This is freetext... do whatever you like with it
but please make sure to include these few lines in any copies you
distribute. If you make changes, please note what you changed, and put
down your name or something.

```

DISCLAIMER:

I'm really not all that good of a programmer. In fact, I'm much more what people call a "hacker", someone who borrows code, modifies it, patches broken stuff with hideously kludgy fixes, values "elegance" somewhere in between Alpo and Velveeta in the culinary correlate to programming elements. In other words, this isn't really an all-inclusive, politically correct, terrifically complete volume on MOOcoding... It's more of a "how to make cool stuff in Canton's kludgy way" instruction booklet. And what cool stuff you will make. Whatever you can imagine.

BEFORE YOU DO THIS TUTORIAL:

This tutorial assumes that you've already fooled around in CTDMOO for quite a while now. This means that you know how to move around, talk with people, pick up things, use the on-line help system, etcetera. If you haven't yet, you might want to explore a bit so as to get a feel for what's being talked about here. Also, this manual is prejudiced towards MOO coding as it can be implemented on MOOs whose foundations are formed by the unabridged LambdaCore database. (This applies to most MOOs.)

HERE WE GO:

TUTORIAL 1: MOO technology as implemented in snot absorbtion

MOO is just a programming language in which you design objects. Everything is an object. Rooms are objects, exits are objects, possessions are objects, even your MOO alter-ego/avatar is an object. We'll be looking at how you (1) make objects, and (2) write verbs that allow you to do Interesting Things with those objects. I'm not going to talk too much about the philosophy of object-oriented programming (that's what the OO in MOO stands for) so let's go ahead and make your first object:

(NOTE: Anything that I want you to type I will enclose in single quotes. In other words, if I write down: 'smell flower' then you will type smell flower and hit return, without the single quotes.)

Go somewhere quiet where you can work, and type the following commands: (Note that when you type a command, the computer may spit back lots of stuff at you. Be warned.)

```
'@create $thing called kleenex'
```

The computer should return something like:

You now have kleenex with object number #797 and parent Techno's detailed thing (#86).

The only thing that will be different is the object number it assigns your kleenex. This is a cool thing to know about: EVERY OBJECT HAS A UNIQUE OBJECT NUMBER. And, often times, it's useful to know the object number for a particular object you're working with. This is because if you're not in the

same room as an object, CTDMOO will have no idea what you're talking about if you refer to the object by its name.

So, let's see if you now have an object called kleenex:

```
'@audit me'
```

This will show you all of the objects you own. Note that you should own yourself, in addition to your kleenex. @audit is a really useful command when you (1) have forgotten the number of one of your objects, or (2) want to spy on someone and see what sort of stuff they've made. You should be able to type something like '@audit canton' to see everything I've made, for example. And then, if you want to look at something I've made, even if you're not in the same room as the object, you can type 'look #___'. For example, 'look #230' would show you what object #230 looks like.

Anyhow, make a note of what object number your kleenex got. Every time I put #__ in the commands that follow, replace the __ with the number of your kleenex. For example, if I write '@describe #__ as "blah"' then you would type @describe #233 as "blah" if #233 was the object number your kleenex was assigned.

Do this, now:

```
'@move #__ to here'
```

This moves the kleenex into the same room as you, so that we can stop using the #__ and call it by its name.

Now describe the kleenex however you like:

```
'@describe kleenex as "Put whatever description you like inside of these double quotes."'
```

Now look at it:

```
'look kleenex'
```

Now give it a smell:

```
'@detail kleenex smell is "It smells nummy."'
```

Try it out:

```
'smell kleenex'
```

Now give it a taste:

```
'@detail kleenex taste is "Yuch!."'
```

Try it out:

```
'taste kleenex'
```

Now give it any other quality you like. For example:

```
'@detail kleenex feel is "It feels soft."'
```

All these things are called 'details', which are part of the heart of CTDMOO. Everything should be as detailed as you can make it, without going overboard. When you do '@detail <object> <verb> is "<description>" you set a message that players will see when they type '<verb> <object>'.
'@detail kleenex feel is "It feels soft."'

However, these aren't *real* verbs. Let's make a REAL verb:

```
'@verb kleenex:wipe any with this'
```

This says to MOO, "Make a verb on the kleenex object called 'wipe'. When someone in the same room as the kleenex types 'wipe <anything> with kleenex' then do something."

And now we have to tell it what to do:

```
'@edit kleenex:wipe'
```

Now we appear inside of the verb editor.

```
'list'
```

This is where the REAL programming happens in MOOcode. To insert a line, we precede a line of code with a double quote. (") Here's some other handy editing commands:

```
'" <code>'      Adds a line of code at your current insertion point.
'list'          Shows you the lines in your code.
'list x-y'      Lists lines x to y in your code.
'del x'         Deletes line x from your code, and puts your insertion point.
                where the deleted line used to exist.
'del x-y'       Deletes lines x through y.
'ins x'         Sets your insertion point to the line *before* line x.
                If you add a line now, it will appear as line (x-1).
': <code>'      Appends <code> to the last line before the insertion point.
                Really useful for typing extrordinarily long lines of code
                that exceed the three or four line length of some clients.
'compile'       Compiles your code... Tells you if there's a flagrant error.
                MUST be done before you:
'quit'         Exits the editor after you've compiled your code.
```

SO, let's put in the code for your wipe verb: (Don't forget all the double quotes and stuff:)

```
' " "Wipe verb... For using the kleenex";
' "if (dobjstr == "nose")
' " player:tell ("Okay. You wipe your nose with the ",this.name,".");
' " player.location:announce (player.name, " has the sniffles and uses
  the ",this.name,".");
' "else
' " player:tell ("Okay. You wipe ",dobjstr," with the ",this.name,".");
' " player.location:announce (player.name, " wipes ",dobjstr," with
  the ",this.name,".");
' "endif
'compile
'list
```

After you type compile, the computer should tell you that your verb was successfully compiled. If it doesn't, then you made a mistake, and you'll have to fix it using the editor techniques described above.

The output from 'list' should look like this:

```
1: if (dobjstr == "nose")
2:  player:tell ("Okay. You wipe your nose with the ",this.name,".");
3:  player.location:announce (player.name, " has the sniffles and uses the
",this.name,".");
4: else
5:  player:tell ("Okay. You wipe ",dobjstr," with the ",this.name,".");
6:  player.location:announce (player.name, " wipes ",dobjstr," with the
",this.name,".");
__7_ endif
^^^
```

The little carrots under the 7 tell you that the insertion point is right beneath line seven.

Let's try out the verb:

```
'quit'  
'wipe nose with kleenex'  
'get kleenex'  
'@move me to #231'  
'wipe gerlinda with kleenex'
```

Note that what other people see is quite different from what you see. They all see things like:

Canton has the sniffles and uses the kleenex.

or,

Canton wipes gerlinda with the kleenex.

TUTORIAL #2: Another silly verb

Mephisto wanted to know how to "randomize" the output of a verb. This is what I suggested:

In his room, he has a \$thing called a box in which the player suspects an either alive or dead cat is hiding. (He made this, one presumes, by typing @create \$thing called box,"cat's box".)
I suggested he type something like this in: (try it out for yourself, if you like.)

```
'@move <the box's number> to here'  
'@verb box:open this none none'  
'@edit box:open'  
  
"player:tell("You creep over to the box to open it. The Severe Looking Woman  
looks at you warily...");'  
"player.location:announce("A look of horror comes over your friend's face,  
as ",player.name," opens the cat's box and looks inside.");'  
"suspend (5);'  
"first={"You discover that","To your surprise, you find that","Unbelievably,"  
, "Completely believably,"}[random(4)];'  
"last={" the cat is quite dead.," the cat is quite alive.," the cat is  
neither dead, nor alive, and really resents having been questioned since  
before you opened the box, it was quite happy appreciating the state of  
simultaneous deadness and aliveness."}[random(3)];'  
"player:tell(first,last);'  
  
'compile'  
'quit'  
  
'open box'
```

Now try typing 'open box' a few more times, and see if you get different results...

DEMYSTIFICATION SESSION #1: How the box worked.

Well, it's probably time you learned about some fundamentals of the MOO world's internal architecture:

THE BIG IDEA: Everything in MOO (EVERYTHING, including rooms, weird quacking things, even you) is an OBJECT. (That's what the first "O" in "MOO" stands

for: Multi-user Dimension/Dungeon, Object-Oriented.) Object oriented programming is a terrific philosophy that lends itself well to MOO, since "parenting" (having objects be functional copies of other objects) has lots of applications. Every object has a unique object number. Also, every object has at least one parent object, from which it gets many of its qualities. (More on that later.)

THE BIG FIB: Actually, not **EVERYTHING** is an object, in MOO. There also happen to be two other sorts of things: Properties and verbs. However, since properties and verbs have to have a host object (they're sort of like barnacles that have to have rocks to attach to) they can sort of be considered facets of objects, which still compose **EVERYTHING**.

VERBS: We've already been dealing with verbs, with the kleenex and box example. Verbs are where all the programming happens in MOO. Verbs are executed by other verbs. An object can have as many verbs as you like. What verbs do is

- (1) execute other verbs
- (2) diddle with properties

PROPERTIES: To understand how the box example worked, you'll have to understand what properties are all about. In programming language, properties are what we call "variables"... Sorts of "slots" or "cubby holes" that hold different kinds of information. These are the different kinds of properties:

- (1) numerical : These hold numbers. MOO only groks integers (eg: 0,5, 4,-33,538323). They can get pretty big, but if you make them too big, MOO wraps 'em around and makes them negative, which can be irritating.
- (2) string : These hold ONE STRING of text that can pretty much be as long as you like. (Within reason.) Strings are notated as bracketed with double quotes. For example:
"foo"
"foo is a cool thing to start a string with"
"42"

(Wait a second... "42"? You told me that there were numerical properties for that sort of thing...)

YES, a trick property, that was! "42" is a VALID string property. However, 42 is NOT VALID. The quotes make "42" mean "the string '42'" instead of the number 42. If you told the computer to add a string property to a numerical property, it would happily barf in your face. HOWEVER, as you'll learn later, the commands `tonum()` and `tostr()` will help you do such things.

- (3) list : Lists are composed of several strings and/or numbers. Any long description you see on the MOO is stored in some list. For example, try typing (from the command line)

```
        ;#{your object #}.description
```

and you'll see the property that contains your description. That property is of the type LIST, since it contains a bunch of strings.

SO, WHAT DOES THIS HAVE TO DO WITH HOW THE BOX WORKED?

Do you remember this line of code from the cat box example?

```
'"first={"You discover that","To your surprise, you find that","Unbelievably,"  
,"Completely believably,"}[random(4)];'
```

What it says is this:

"make up a new variable called first, which should be made to randomly contain

one of the four strings contained in the list {"You discover..."}

Here's how: To make a variable that will remain in existence solely for the duration of a verb's execution, just make up a variable name and assign it to some sort of an element, be it a number, string, or list.

Here's how you pull elements out of a list:

to get "foo" out of the list {"dog","is","foo","bar"} you do (from within a verb):

```
{"dog","is","foo","bar"}[3]
```

Here's how you make a random number:

to generate a random number from 1 to 5, do:

```
random(5)
```

SO, to get a random element out of a list, you could do:

```
{"this","is","a","curious","list"}[random(5)]
```

You can use a similar technique to pull out characters from a string:

```
"This is a string"[7..12]
```

will return => "s a st"

COOL HACKER LIKE TIP #1: EVAL

Here's a good way to see how programming code will act before you stick it in a verb:

To see what the result of any line of code will be, just type the line of code in at the command line, but precede it with a semicolon. For example, from ANY room (not just within an editor) if you type:

```
;"I wonder if he's lying or not"[27..29]
```

and hit return, MOO will respond:

```
=> "not"
```

TUTORIAL #3: RECYCLING

Recycle often. If you're low on quota (type @quota) then you might want to @recycle (destroy) some of the objects you've made. (Such as the kleenex or whatever.) Just type @audit, pick an object you don't want anymore, and @recycle #____

What fun. Amaze friends and family by keeping MOO database under control.

TUTORIAL #4: FOOLING AROUND WITH PROPERTIES

Try this one out: (For sake of convenience, I'm going to stop using all those irritating ' marks to indicate when you should type exactly what you read: I assume that by this point you know when to do what... :)

```
@create $thing called "Dumb Object",dumb
@move #<the object number you have assigned to you> to here
@property dumb.times 0
@property dumb.dumblist {"furry creatures.,"Michael Jackson.,"Elvis."}
@property dumb.subject_msg "You suddenly realize that you now "
```

```

@verb dumb:touch this none none
@edit dumb:touch

"numberargs=length(args);
"if (numberargs != 1)
" player:tell ("No, silly, just type 'touch ",this.name,".");
"else
" emotion={"love ","lust after ","despise ","admire ","worship "}[random(5)];
" victim=this.dumblist[random(length(this.dumblist))];
" wholething=this.subject_msg + emotion + victim;
" player:tell("A strange feeling overcomes you...");
" player:tell(wholething);
" player.location:announce(player.name, " seems changed after having touched
  the ",this.name,".");
" this.times=this.times + 1;
" if (this.times > 5)
" player.location:announce_all("The ",this.name," seems pleased as it
  announces to everyone that it has been touched no less than ",this.times,"
  times.");
" endif
"endif

compile
quit

touch dumb
touch dumb
touch dumb
touch dumb
touch dumb
touch dumb

```

WHAT'S NEW IN THIS LAST VERB:

- o Though not entirely new, `this.name` and `player.name` and that sort of thing have been used here... What happens is that whenever you run a verb, you can use `this.____` to specify a property existing on the object on which the verb exists. (In case you haven't figured it out, every object has a property called `name` that should contain its name.) ALSO, whenever a verb is executed, it assigns the value of the player *calling* the verb to a variable (unsurprisingly) named `"player"`. So, from within a verb, `player:tell("Blah");` means ``Send the string "Blah" to the `:tell` verb of the player that called this verb.'' On the other hand, if you wanted to tell something to everything in the room, you'd do: `player.location:announce_all("Blah")`, which would mean ``Send the string "Blah" to the `:announce_all` verb of the object which is named in the `.location` property of the player that called this verb.''
- o Yes, not only you, but VERBS can change properties resident on objects.
- o When we did the `@property dumb.subject_msg "You suddenly realize..."` we were using a cool easy-to-use feature of MOO: Messages. Right now, type:

```
@messages me
```

What you see is a list of all the properties defined on you that end with `_msg`. What's neat about these properties is that if you want to edit them, you don't necessarily have to `@edit` them, or `command-line-evaluate` them (use a semicolon to set them), you can just type stuff like:

```
@ohome_depart me is "%N wanders homeward"
```

Messages are really useful when you want to make a verb that you imagine lots of non-programmer types might want to use. Use them when you want owners to be able to easily customize the messages the verb uses to create its output.

- o We used the IF conditional, and wasn't that sort of neat? Here's some other uses of if:

```
if (player.gender == "male" || player.gender == "female")
  player:tell ("Sorry, this verb is only for people without gender.");
endif
```

The double-pipe (||) means "OR". If you want to do "AND", use && instead. Of course, if the only genders we expect are male, female, and neuter, we would probably write the code this way:

```
if (player.gender != "neuter")
  player:tell ("Sorry, this verb is only for people without gender.");
endif
```

which means "If player's gender property is NOT equal to "neuter"..."

TIP:

If you want to test equality, BE SURE TO USE A DOUBLE EQUAL SIGN, AND NOT A SINGLE EQUAL SIGN, BECAUSE IF YOU USE A SINGLE EQUAL SIGN YOUR VERB WON'T WORK, AND YOU'LL SPEND HOURS LOOKING FOR THE BUG, AND YOU MIGHT END UP DOING SOMETHING REALLY NASTY TO YOURSELF OUT OF PURE FRUSTRATION.

- o What else is new? We used string addition in this line of code:

```
wholething=this.subject_msg + emotion + victim;
```

This line just adds a few strings to each other. If you do foo+bar, and foo is "dog" and bar is "food", then the output will be "dogfood".

SOME OF PAVEL'S MANUAL, and A PLUG FOR YOU TO READ IT

(You see, Pavel Curtis' delicious LambdaMOO manual says:

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.)

I REALLY STRONGLY SUGGEST THAT YOU READ PAVEL'S MANUAL IN ITS ENTIRETY AFTER YOU'VE FINISHED READING THIS TUTORIAL. Not only is his manual more extensive and detailed: it's *correct*! A lot of what I'm writing here is a pack of lies that would probably insult Pavel. However, they're precisely the sorts of lies that help me program, and will get you programming as quickly as possible. (With all respectful apologies) I must admit that when I first started programming, Pavel's manual was about as helpful to me as a bicycle is to a fish. It's a beautiful statement of the philosophy and structure that is embodied in LambdaMOO. However, it doesn't say too much about the LambdaCore database, which is what you need to know about to program in (most) MOO's. (Notable exceptions exist, where the core DB has been significantly altered. However, most MOOs are just extensions of the LambdaCore DB.)

End of plug. To get his manual (in post script or text formats) just find it via anonymous FTP from ftp.parc.xerox.com, or, more easily, via the CTDNet gopher server. Currently, it's in this path: (ctdnet.acns.nwu.edu 70)

Path=0/CTDNET_Information/CTDMOO Reference/MOO manuals and such/

SOME OF PAVEL'S MANUAL: (for real)

Properties

A "property" is a named "slot" in an object that can hold an arbitrary MOO value. Every object has eight built-in properties whose values are constrained to be of particular types. In addition, an object can have any number of other properties, none of which have type constraints. The built-in properties are as follows:

name	a string, the usual name for this object
owner	an object, the player who controls access to it
location	an object, where the object is in virtual reality
contents	a list of objects, the inverse of `location'
programmer	a bit, does the object have programmer rights?
wizard	a bit, does the object have wizard rights?
r	a bit, is the object publicly readable?
w	a bit, is the object publicly writable?
f	a bit, is the object fertile?

..

```
{HERE'S SOME IMPORTANT STUFF ON PERMISSIONS}
```

Every defined property (as opposed to those that are built-in) has an owner and a set of permissions for non-owners. The owner of the property can get and set the property's value and can change the non-owner permissions. Only a wizard can change the owner of a property.

The initial owner of a property is the player who added it; this is usually, but not always, the player who owns the object to which the property was added. This is because properties can only be added by the object owner or a wizard, unless the object is publicly writable (i.e., its `w' property is 1), which is rare. Thus, the owner of an object may not necessarily be the owner of every (or even any) property on that object.

The permissions on properties are drawn from this set: `r' (read), `w' (write), and `c' (change ownership in descendants). Read permission lets non-owners get the value of the property and, of course, write permission lets them set that value. The `c' permission bit is a little more complicated.

Recall that every object has all of the properties that its parent does and perhaps some more. Ordinarily, when a child object inherits a property from its parent, the owner of the child becomes the owner of that property. This is because the `c' permission bit is "on" by default. If the `c' bit is not on, then the inherited property has the same owner in the child as it does in the parent.

As an example of where this can be useful, the LambdaCore database ensures that every player has a `password' property containing the encrypted version of the player's connection password. For security reasons, we don't want other players to be able to see even the encrypted version of the password, so we turn off the `r' permission bit. To ensure that the password is only set in a consistent way (i.e., to the encrypted version of a player's password), we don't want to let anyone but a wizard change the property. Thus, in the parent object for all players, we made a wizard the owner of the password property and set the permissions to the empty string, `""'. That is, non-owners cannot read or write the property and, because the `c' bit is not set, the wizard who owns the property on the parent class also owns it on all of the descendants of that class.

Another, perhaps more down-to-earth example arose when a character named Ford started building objects he called "radios" and another character, yduJ, wanted to own one. Ford kindly made the generic radio object publicly readable, allowing yduJ to create a child object of it, her own radio. Radios had a property called `channel' that identified something corresponding to the frequency to which the radio was tuned. Ford had written nice programs on radios (verbs, discussed below) for turning the channel selector on the front

of the radio, which would make a corresponding change in the value of the `channel' property. However, whenever anyone tried to turn the channel selector on yduJ's radio, they got a permissions error. The problem concerned the ownership of the `channel' property.

As I explain later, programs run with the permissions of their author. So, in this case, Ford's nice verb for setting the channel ran with his permissions. But, since the `channel' property in the generic radio had the `c' permission bit set, the `channel' property on yduJ's radio was owned by her. Ford didn't have permission to change it! The fix was simple. Ford changed the permissions on the `channel' property of the generic radio to be just `r', without the `c' bit, and yduJ made a new radio. This time, when yduJ's radio inherited the `channel' property, yduJ did not inherit ownership of it; Ford remained the owner. Now the radio worked properly, because Ford's verb had permission to change the channel.

..

Command Parsing *****

The MOO server is able to do a small amount of parsing on the commands that a player enters. In particular, it can break apart commands that follow one of the following forms:

```
VERB
VERB DIRECT-OBJECT
VERB DIRECT-OBJECT PREPOSITION INDIRECT-OBJECT
```

Real examples of these forms, meaningful in the LambdaCore database, are as follows:

```
look
take yellow bird
put yellow bird in cuckoo clock
```

Note that English articles (i.e., `the', `a', and `an') are not generally used in MOO commands; the parser does not know that they are not important parts of objects' names.

To have any of this make real sense, it is important to understand precisely how the server decides what to do when a player types a command.

First, the server checks whether or not the first non-blank character in the command is one of the following:

```
"      :      ;
```

If so, that character is replaced by the corresponding command below, followed by a space:

```
say      emote      eval
```

For example, the command

```
"Hi, there.
```

is treated exactly as if it were as follows:

```
say Hi, there.
```

The server next breaks up the command into words. In the simplest case, the command is broken into words at every run of space characters; for example, the command `foo bar baz' would be broken into the words `foo', `bar', and `baz'. To force the server to include spaces in a "word", all or part of a word can be enclosed in double-quotes. For example, the command

```
foo "bar mumble" baz "fr"otz" bl"o"rt
```

is broken into the words `foo', `bar mumble', `baz frotz', and `blort'. Finally, to include a double-quote or a backslash in a word, they can be preceded by a backslash, just like in MOO strings.

Having thus broken the string into words, the server next checks to see if the first word names any of the three "built-in" commands: `.program', `PREFIX', or `SUFFIX'. The first one of these is only available to programmers and the other two are intended for use by client programs; all three are described in the final chapter of this document, "Server Commands and Database Assumptions". If the first word isn't one of the above, then we get to the usual case: a normal MOO command.

The server now tries to parse the command into a verb, direct object, preposition and indirect object. The first word is taken to be the verb. The server then tries to find one of the prepositional phrases listed at the end of the previous section, using the match that occurs earliest in the command. For example, in the very odd command `foo as bar to baz', the server would take `as' as the preposition, not `to'.

If the server succeeds in finding a preposition, it considers the words between the verb and the preposition to be the direct object and those after the preposition to be the indirect object. In both cases, the sequence of words is turned into a string by putting one space between each pair of words. Thus, in the odd command from the previous paragraph, there are no words in the direct object (i.e., it is considered to be the empty string, `') and the indirect object is `"bar to baz"'.

If there was no preposition, then the direct object is taken to be all of the words after the verb and the indirect object is the empty string.

The next step is to try to find MOO objects that are named by the direct and indirect object strings.

First, if an object string is empty, then the corresponding object is the special object `#-1' (aka `\$nothing' in LambdaCore). If an object string has the form of an object number (i.e., a hash mark (`#') followed by digits), and the object with that number exists, then that is the named object. If the object string is either `"me"' or `"here"', then the player object itself or its location is used, respectively.

Otherwise, the server considers all of the objects whose location is either the player (i.e., the objects the player is "holding", so to speak) or the room the player is in (i.e., the objects in the same room as the player); it will try to match the object string against the various names for these objects.

The matching done by the server uses the `aliases' property of each of the objects it considers. The value of this property should be a list of strings, the various alternatives for naming the object. If it is not a list, then the empty list is used; if an object does not have an `aliases' property, then a list containing the value of the `name' property is used instead.

The server checks to see if the object string in the command is either exactly equal to or a prefix of any alias; if there are any exact matches, the prefix matches are ignored. If exactly one of the objects being considered has a matching alias, that object is used. If more than one has a match, then the special object `#-2' (aka `\$ambiguous_match' in LambdaCore) is used. If there are no matches, then the special object `#-3' (aka `\$failed_match' in LambdaCore) is used.

So, now the server has identified a verb string, a preposition string, and direct- and indirect-object strings and objects. It then looks at each of the verbs defined on each of the following four objects, in order:

1. the player who typed the command,

2. the room the player is in,
3. the direct object, if any, and
4. the indirect object, if any.

For each of these verbs in turn, it tests if all of the the following are true:

- * the verb string in the command matches one of the names for the verb,
- * the direct- and indirect-object values found by matching are allowed by the corresponding argument specifiers for the verb, and
- * the preposition string in the command is matched by the preposition specifier for the verb.

I'll explain each of these criteria in turn.

Every verb has one or more names; all of the names are kept in a single string, separated by spaces. In the simplest case, a verb-name is just a word made up of any characters other than spaces and stars (i.e., ` ` and `*`). In this case, the verb-name matches only itself; that is, the name must be matched exactly.

If the name contains a single star, however, then the name matches any prefix of itself that is at least as long as the part before the star. For example, the verb-name `foo*bar` matches any of the strings `foo`, `foob`, `fooba`, or `foobar`; note that the star itself is not considered part of the name.

If the verb name *ends* in a star, then it matches any string that begins with the part before the star. For example, the verb-name `foo*` matches any of the strings `foo`, `foobar`, `food`, or `foogleman`, among many others. As a special case, if the verb-name is `*` (i.e., a single star all by itself), then it matches anything at all.

Recall that the argument specifiers for the direct and indirect objects are drawn from the set `none`, `any`, and `this`. If the specifier is `none`, then the corresponding object value must be `#-1` (aka `\$nothing` in LambdaCore); that is, it must not have been specified. If the specifier is `any`, then the corresponding object value may be anything at all. Finally, if the specifier is `this`, then the corresponding object value must be the same as the object on which we found this verb; for example, if we are considering verbs on the player, then the object value must be the player object.

Finally, recall that the argument specifier for the preposition is either `none`, `any`, or one of several sets of prepositional phrases, given above. A specifier of `none` matches only if there was no preposition found in the command. A specifier of `any` always matches, regardless of what preposition was found, if any. If the specifier is a set of prepositional phrases, then the one found must be in that set for the specifier to match.

So, the server considers several objects in turn, checking each of their verbs in turn, looking for the first one that meets all of the criteria just explained. If it finds one, then that is the verb whose program will be executed for this command. If not, then it looks for a verb named `huh` on the room that the player is in; if one is found, then that verb will be called. This feature is useful for implementing room-specific command parsing or error recovery. If the server can't even find a `huh` verb to run, it prints an error message like `I couldn't understand that.` and the command is considered complete.

At long last, we have a program to run in response to the command typed by the player. When the code for the program begins execution, the following built-in variables will have the indicated values:

```
player    an object, the player who typed the command
this      an object, the object on which this verb was found
caller    an object, the same as `player'
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
```

{END OF THIS SET OF BITS FROM PAVEL'S MANUAL.}

SOME TIPS AND TRICKS, AND THEN YOU'RE ON YOUR OWN:

If the last set of stuff out of Pavel's manual was confusing, don't worry- It will become more clear if you refer to it after you've been programming for a while. Also, if you read it in context (ie: if you read the WHOLE manual) that might help. Before I part and leave you to fend for yourself in the world of LambdaCore-based MOO programming, here's some assorted unrelated tips, commands, and notes that might help you out:

TELLING WHETHER OR NOT SOMETHING IS A PLAYER:

something like this might help:

```
pl =$string_utils:match_player(dobjstr)
  if ($command_utils:player_match_failed(pl, dobjstr))
    player:tell(dobjstr, " isn't a player, or refers to multiple players.");
  return 0;
endif
```

be sure to check out 'help \$string_utils'. The \$x_utils are REALLY powerful sets of verbs you can use in your programs. Become familiar with them.

ALSO CHECK OUT:

```
help $math_utils
help $time_utils
help $list_utils
help $set_utils
help $match_utils
help $object_utils
help $lock_utils
help $gender_utils
help $seq_utils
help $trig_utils
help $string_utils
help $perm_utils
```

All these utilities are sort of like FREE CODE for you to use. Also note that these are great deposits of code that you can analyze! All of it should be publicly-readable. Just do stuff like @list \$string_utils:trim