LambdaMOO Programmer's Manual

For LambdaMOO Version 1.8.0p5
May 1996

by Pavel Curtis
aka Haakon
aka Lambda

1 Introduction

LambdaMOO is a network-accessible, multi-user, programmable, interactive system well-suited to the construction of text-based adventure games, conferencing systems, and other collaborative software. Its most common use, however, is as a multi-participant, low-bandwidth virtual reality, and it is with this focus in mind that I describe it here.

Participants (usually referred to as *players*) connect to LambdaMOO using Telnet or some other, more specialized, *client* program. Upon connection, they are usually presented with a *welcome message* explaining how to either create a new *character* or connect to an existing one. Characters are the embodiment of players in the virtual reality that is LambdaMOO.

Having connected to a character, players then give one-line commands that are parsed and interpreted by LambdaMOO as appropriate. Such commands may cause changes in the virtual reality, such as the location of a character, or may simply report on the current state of that reality, such as the appearance of some object.

The job of interpreting those commands is shared between the two major components in the LambdaMOO system: the *server* and the *database*. The server is a program, written in a standard programming language, that manages the network connections, maintains queues of commands and other tasks to be executed, controls all access to the database, and executes other programs written in the MOO programming language. The database contains representations of all the objects in the virtual reality, including the MOO programs that the server executes to give those objects their specific behaviors.

Almost every command is parsed by the server into a call on a MOO procedure, or *verb*, that actually does the work. Thus, programming in the MOO language is a central part of making non-trivial extensions to the database and thus, the virtual reality.

In the next chapter, I describe the structure and contents of a LambdaMOO database. The following chapter gives a complete description of how the server performs its primary duty: parsing the commands typed by players. Next, I describe the complete syntax and semantics of the MOO programming language. Finally, I describe all of the database conventions assumed by the server.

> **Note:** This manual describes only those aspects of LambdaMOO that are entirely independent of the contents of the database. It does not describe, for example, the commands or programming interfaces present in the LambdaCore database.

2 The LambdaMOO Database

In this chapter, I begin by describing in detail the various kinds of data that can appear in a LambdaMOO database and that, therefore, MOO programs can manipulate. In a few places, I refer to the *LambdaCore* database. This is one particular LambdaMOO database, created every so often by extracting the "core" of the current database for the original LambdaMOO.

> **Note**: The original LambdaMOO resides on the host `lambda.parc.xerox.com` (the numeric address for which is `192.216.54.2`), on port 8888. Feel free to drop by! A copy of the most recent release of the LambdaCore database can be obtained by anonymous FTP from host `ftp.parc.xerox.com` in the directory `pub/MOO`.

2.1 MOO Value Types

There are only a few kinds of values that MOO programs can manipulate:

- integers (in a specific, large range)
- real numbers (represented with floating-point numbers)
- strings (of characters)
- objects (in the virtual reality)
- errors (arising during program execution)
- lists (of all of the above, including lists)

MOO supports the integers from −2^31 (that is, negative two to the power of 31) up to 2^31 − 1 (one less than two to the power of 31); that's from −2147483648 to 2147483647, enough for most purposes. In MOO programs, integers are written just as you see them here, an optional minus sign followed by a non-empty sequence of decimal digits. In particular, you may not put commas, periods, or spaces in the middle of large integers, as we sometimes do in English and other natural languages (e.g., '2,147,483,647').

Real numbers in MOO are represented as they are in almost all other programming languages, using so-called *floating-point* numbers. These have certain (large) limits on size and precision that make them useful for a wide range of applications. Floating-point numbers are written with an optional minus sign followed by a non-empty sequence of digits punctuated at some point with a decimal point ('.') and/or followed by a scientific-notation marker (the letter 'E' or 'e' followed by an optional sign and one or more digits). Here are some examples of floating-point numbers:

```
325.0   325.   3.25e2   0.325E3   325.E1   .0325e+4   32500e-2
```

All of these examples mean the same number. The third of these, as an example of scientific notation, should be read "3.25 times 10 to the power of 2".

> *Fine points:* The MOO represents floating-point numbers using the local meaning of the C-language `double` type, which is almost always equivalent to IEEE 754 double precision floating point. If so, then the smallest positive floating-point number is no larger than `2.2250738585072014e-308` and the largest floating-point number is `1.7976931348623157e+308`.
>
> IEEE infinities and NaN values are not allowed in MOO. The error `E_FLOAT` is raised whenever an infinity would otherwise be computed; `E_INVARG` is raised whenever a NaN would otherwise arise. The value `0.0` is always returned on underflow.

Character *strings* are arbitrarily-long sequences of normal, ASCII printing characters. When written as values in a program, strings are enclosed in double-quotes, like this:

```
"This is a character string."
```

To include a double-quote in the string, precede it with a backslash ('\'), like this:

```
"His name was \"Leroy\", but nobody ever called him that."
```

Finally, to include a backslash in a string, double it:

```
"Some people use backslash ('\\') to mean set difference."
```

MOO strings may not include special ASCII characters like carriage-return, line-feed, bell, etc. The only non-printing characters allowed are spaces and tabs.

> *Fine point:* There is a special kind of string used for representing the arbitrary bytes used in general, binary input and output. In a *binary string*, any byte that isn't an ASCII printing character or the space character is represented as the three-character substring "~XX", where XX is the hexadecimal representation of the byte; the input character '~' is represented by the three-character substring "~7E". This special representation is used by the functions `encode_binary()` and `decode_binary()` and by the functions `notify()` and `read()` with network connections that are in binary mode. See the descriptions of the `set_connection_option()`, `encode_binary()`, and `decode_binary()` functions for more details.

Objects are the backbone of the MOO database and, as such, deserve a great deal of discussion; the entire next section is devoted to them. For now, let it suffice to say that every object has a number, unique to that object. In programs, we write a reference to a particular object by putting a hash mark ('#') followed by the number, like this:

```
#495
```

Object numbers are always integers.

There are three special object numbers used for a variety of purposes: `#-1`, `#-2`, and `#-3`, usually referred to in the LambdaCore database as `$nothing`, `$ambiguous_match`, and `$failed_match`, respectively.

Errors are, by far, the least frequently used values in MOO. In the normal case, when a program attempts an operation that is erroneous for some reason (for example, trying to add a number to a character string), the server stops running the program and prints out an error message. However, it is possible for a program to stipulate that such errors should not stop execution; instead, the server should just let the value of the operation be an error value. The program can then test for such a result and take some appropriate kind of recovery action. In programs, error values are written as words beginning with 'E_'. The complete list of error values, along with their associated messages, is as follows:

`E_NONE`	No error
`E_TYPE`	Type mismatch
`E_DIV`	Division by zero
`E_PERM`	Permission denied
`E_PROPNF`	Property not found
`E_VERBNF`	Verb not found
`E_VARNF`	Variable not found
`E_INVIND`	Invalid indirection
`E_RECMOVE`	Recursive move
`E_MAXREC`	Too many verb calls
`E_RANGE`	Range error
`E_ARGS`	Incorrect number of arguments

`E_NACC`	Move refused by destination
`E_INVARG`	Invalid argument
`E_QUOTA`	Resource limit exceeded
`E_FLOAT`	Floating-point arithmetic error

The final kind of value in MOO programs is *lists*. A list is a sequence of arbitrary MOO values, possibly including other lists. In programs, lists are written in mathematical set notation with each of the elements written out in order, separated by commas, the whole enclosed in curly braces ('`{`' and '`}`'). For example, a list of the names of the days of the week is written like this:

```
{"Sunday", "Monday", "Tuesday", "Wednesday",
 "Thursday", "Friday", "Saturday"}
```

Note that it doesn't matter that we put a line-break in the middle of the list. This is true in general in MOO: anywhere that a space can go, a line-break can go, with the same meaning. The only exception is inside character strings, where line-breaks are not allowed.

2.2 Objects in the MOO Database

Objects are, in a sense, the whole point of the MOO programming language. They are used to represent objects in the virtual reality, like people, rooms, exits, and other concrete things. Because of this, MOO makes a bigger deal out of creating objects than it does for other kinds of value, like integers.

Numbers always exist, in a sense; you have only to write them down in order to operate on them. With objects, it is different. The object with number '`#958`' does not exist just because you write down its number. An explicit operation, the '`create()`' function described later, is required to bring an object into existence. Symmetrically, once created, objects continue to exist until they are explicitly destroyed by the '`recycle()`' function (also described later).

The identifying number associated with an object is unique to that object. It was assigned when the object was created and will never be reused, even if the object is destroyed. Thus, if we create an object and it is assigned the number '`#1076`', the next object to be created will be assigned '`#1077`', even if '`#1076`' is destroyed in the meantime.

Every object is made up of three kinds of pieces that together define its behavior: *attributes*, *properties*, and *verbs*.

2.2.1 Fundamental Object Attributes

There are three fundamental *attributes* to every object:

1. A flag (either true or false) specifying whether or not the object represents a player,
2. The object that is its *parent*, and
3. A list of the objects that are its *children*; that is, those objects for which this object is their parent.

The act of creating a character sets the player attribute of an object and only a wizard (using the function `set_player_flag()`) can change that setting. Only characters have the player bit set to 1.

The parent/child hierarchy is used for classifying objects into general classes and then sharing behavior among all members of that class. For example, the LambdaCore database contains an object representing a sort of "generic" room. All other rooms are *descendants* (i.e., children or children's children, or ...) of that one. The generic room defines those pieces of behavior that are common to all rooms; other rooms specialize that behavior for their own purposes. The notion of classes and specialization is the very essence of what is meant by *object-oriented* programming. Only the functions `create()`, `recycle()`, `chparent()`, and `renumber()` can change the parent and children attributes.

2.2.2 Properties on Objects

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?

The '`name`' property is used to identify the object in various printed messages. It can only be set by a wizard or by the owner of the object. For player objects, the '`name`' property can only be set by a wizard; this allows the wizards, for example, to check that no two players have the same name.

The '`owner`' identifies the object that has owner rights to this object, allowing them, for example, to change the '`name`' property. Only a wizard can change the value of this property.

The '`location`' and '`contents`' properties describe a hierarchy of object containment in the virtual reality. Most objects are located "inside" some other object and that other object is the value of the '`location`' property. The '`contents`' property is a list of those objects for which this object is their location. In order to maintain the consistency of these properties, only the `move()` function is able to change them.

The '`wizard`' and '`programmer`' bits are only applicable to characters, objects representing players. They control permission to use certain facilities in the server. They may only be set by a wizard.

The '`r`' bit controls whether or not players other than the owner of this object can obtain a list of the properties or verbs in the object. Symmetrically, the '`w`' bit controls whether or not non-owners can add or delete properties and/or verbs on this object. The '`r`' and '`w`' bits can only be set by a wizard or by the owner of the object.

The '`f`' bit specifies whether or not this object is *fertile*, whether or not players other than the owner of this object can create new objects with this one as the parent. It also controls whether or not non-owners can use the `chparent()` built-in function to make this object the parent of an existing object. The '`f`' bit can only be set by a wizard or by the owner of the object.

All of the built-in properties on any object can, by default, be read by any player. It is possible, however, to override this behavior from within the database, making any of these properties readable only by wizards. See the chapter on server assumptions about the database for details.

As mentioned above, it is possible, and very useful, for objects to have other properties aside from the built-in ones. These can come from two sources.

First, an object has a property corresponding to every property in its parent object. To use the jargon of object-oriented programming, this is a kind of *inheritance*. If some object has a property named '`foo`', then so will all of its children and thus its children's children, and so on.

Second, an object may have a new property defined only on itself and its descendants. For example, an object representing a rock might have properties indicating its weight, chemical composition, and/or pointiness, depending upon the uses to which the rock was to be put in the virtual reality.

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 fertile, 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.

2.2.3 Verbs on Objects

The final kind of piece making up an object is *verbs*. A verb is a named MOO program that is associated with a particular object. Most verbs implement commands that a player might type; for example, in the LambdaCore database, there is a verb on all objects representing containers that implements commands of the form 'put *object* in *container*'. It is also possible for MOO programs to invoke the verbs defined on objects. Some verbs, in fact, are designed to be used only from inside MOO code; they do not correspond to any particular player command at all. Thus, verbs in MOO are like the 'procedures' or 'methods' found in some other programming languages.

As with properties, every verb has an owner and a set of permission bits. The owner of a verb can change its program, its permission bits, and its argument specifiers (discussed below). Only a wizard can change the owner of a verb. The owner of a verb also determines the permissions with which that verb runs; that is, the program in a verb can do whatever operations the owner of that verb is allowed to do and no others. Thus, for example, a verb owned by a wizard must be written very carefully, since wizards are allowed to do just about anything.

The permission bits on verbs are drawn from this set: '`r`' (read), '`w`' (write), '`x`' (execute), and '`d`' (debug). Read permission lets non-owners see the program for a verb and, symmetrically, write permission lets them change that program. The other two bits are not, properly speaking, permission bits at all; they have a universal effect, covering both the owner and non-owners.

The execute bit determines whether or not the verb can be invoked from within a MOO program (as opposed to from the command line, like the '`put`' verb on containers). If the '`x`' bit is not set, the verb cannot be called from inside a program. The '`x`' bit is usually set.

The setting of the debug bit determines what happens when the verb's program does something erroneous, like subtracting a number from a character string. If the '`d`' bit is set, then the server *raises* an error value; such raised errors can be *caught* by certain other pieces of MOO code. If the error is not caught, however, the server aborts execution of the command and, by default, prints an error message on the terminal of the player whose command is being executed. (See the chapter on server assumptions about the database for details on how uncaught errors are handled.) If the '`d`' bit is not set, then no error is raised, no message is printed, and the command is not aborted; instead the error value is returned as the result of the erroneous operation.

> **Note:** the '`d`' bit exists only for historical reasons; it used to be the only way for MOO code to catch and handle errors. With the introduction of the `try-except` statement and the error-catching expression, the '`d`' bit is no longer useful. All new verbs should have the '`d`' bit set, using the newer facilities for error handling if desired. Over time, old verbs written assuming the '`d`' bit would not be set should be changed to use the new facilities instead.

In addition to an owner and some permission bits, every verb has three 'argument specifiers', one each for the direct object, the preposition, and the indirect object. The direct and indirect specifiers are each drawn from this set: `this`, `any`, or `none`. The preposition specifier is `none`, `any`, or one of the items in this list:

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

The argument specifiers are used in the process of parsing commands, described in the next chapter.

3 The Built-in Command Parser

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 six "built-in" commands: '`.program`', '`PREFIX`', '`OUTPUTPREFIX`', '`SUFFIX`', '`OUTPUTSUFFIX`', or the connection's defined *flush* command, if any ('`.flush`' by default). The first one of these is only available to programmers, the next four are intended for use by client programs, and the last can vary from database to database or even connection to connection; all six 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 next gives code in the database a chance to handle the command. If the verb `$do_command()` exists, it is called with the words of the command passed as its arguments and `argstr` set to the raw command typed by the user. If `$do_command()` does not exist, or if that verb-call completes normally (i.e., without suspending or aborting) and returns a false value, then the built-in command parser is invoked to handle the command as described below. Otherwise, it is assumed that the database code handled the command completely and no further action is taken by the server for that command.

If the built-in command parser is invoked, the server 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, or the object does not have an '`aliases`' property, then the empty list is used. In any case, the value of the '`name`' property is added to the list for the purposes of matching.

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

The value returned by the program, if any, is ignored by the server.

4 The MOO Programming Language

MOO stands for "MUD, Object Oriented." MUD, in turn, has been said to stand for many different things, but I tend to think of it as "Multi-User Dungeon" in the spirit of those ancient precursors to MUDs, Adventure and Zork.

MOO, the programming language, is a relatively small and simple object-oriented language designed to be easy to learn for most non-programmers; most complex systems still require some significant programming ability to accomplish, however.

Having given you enough context to allow you to understand exactly what MOO code is doing, I now explain what MOO code looks like and what it means. I begin with the syntax and semantics of expressions, those pieces of code that have values. After that, I cover statements, the next level of structure up from expressions. Next, I discuss the concept of a task, the kind of running process initiated by players entering commands, among other causes. Finally, I list all of the built-in functions available to MOO code and describe what they do.

First, though, let me mention comments. You can include bits of text in your MOO program that are ignored by the server. The idea is to allow you to put in notes to yourself and others about what the code is doing. To do this, begin the text of the comment with the two characters '`/*`' and end it with the two characters '`*/`'; this is just like comments in the C programming language. Note that the server will completely ignore that text; it will *not* be saved in the database. Thus, such comments are only useful in files of code that you maintain outside the database.

To include a more persistent comment in your code, try using a character string literal as a statement. For example, the sentence about peanut butter in the following code is essentially ignored during execution but will be maintained in the database:

```
for x in (players())
  "Grendel eats peanut butter!";
  player:tell(x.name, " (", x, ")");
endfor
```

4.1 MOO Language Expressions

Expressions are those pieces of MOO code that generate values; for example, the MOO code

```
3 + 4
```

is an expression that generates (or "has" or "returns") the value 7. There are many kinds of expressions in MOO, all of them discussed below.

4.1.1 Errors While Evaluating Expressions

Most kinds of expressions can, under some circumstances, cause an error to be generated. For example, the expression `x / y` will generate the error `E_DIV` if `y` is equal to zero. When an expression generates an error, the behavior of the server is controlled by setting of the '`d`' (debug) bit on the verb containing that expression. If the '`d`' bit is not set, then the error is effectively squelched immediately upon generation; the error value is simply returned as the value of the expression that generated it.

> **Note:** this error-squelching behavior is very error prone, since it affects *all* errors, including ones the programmer may not have anticipated. The '`d`' bit exists only for

historical reasons; it was once the only way for MOO programmers to catch and handle errors. The error-catching expression and the `try-except` statement, both described below, are far better ways of accomplishing the same thing.

If the 'd' bit is set, as it usually is, then the error is *raised* and can be caught and handled either by code surrounding the expression in question or by verbs higher up on the chain of calls leading to the current verb. If the error is not caught, then the server aborts the entire task and, by default, prints a message to the current player. See the descriptions of the error-catching expression and the `try-except` statement for the details of how errors can be caught, and the chapter on server assumptions about the database for details on the handling of uncaught errors.

4.1.2 Writing Values Directly in Verbs

The simplest kind of expression is a literal MOO value, just as described in the section on values at the beginning of this document. For example, the following are all expressions:

```
17
#893
"This is a character string."
E_TYPE
{"This", "is", "a", "list", "of", "words"}
```

In the case of lists, like the last example above, note that the list expression contains other expressions, several character strings in this case. In general, those expressions can be of any kind at all, not necessarily literal values. For example,

```
{3 + 4, 3 - 4, 3 * 4}
```

is an expression whose value is the list `{7, -1, 12}`.

4.1.3 Naming Values Within a Verb

As discussed earlier, it is possible to store values in properties on objects; the properties will keep those values forever, or until another value is explicitly put there. Quite often, though, it is useful to have a place to put a value for just a little while. MOO provides local variables for this purpose.

Variables are named places to hold values; you can get and set the value in a given variable as many times as you like. Variables are temporary, though; they only last while a particular verb is running; after it finishes, all of the variables given values there cease to exist and the values are forgotten.

Variables are also "local" to a particular verb; every verb has its own set of them. Thus, the variables set in one verb are not visible to the code of other verbs.

The name for a variable is made up entirely of letters, digits, and the underscore character ('_') and does not begin with a digit. The following are all valid variable names:

```
foo
_foo
this2that
M68000
two_words
This_is_a_very_long_multiword_variable_name
```

Note that, along with almost everything else in MOO, the case of the letters in variable names is insignificant. For example, these are all names for the same variable:

```
fubar
Fubar
FUBAR
fUbAr
```

A variable name is itself an expression; its value is the value of the named variable. When a verb begins, almost no variables have values yet; if you try to use the value of a variable that doesn't have one, the error value `E_VARNF` is raised. (MOO is unlike many other programming languages in which one must 'declare' each variable before using it; MOO has no such declarations.) The following variables always have values:

```
INT         FLOAT       OBJ
STR         LIST        ERR
player      this        caller
verb        args        argstr
dobj        dobjstr     prepstr
iobj        iobjstr     NUM
```

The values of some of these variables always start out the same:

`INT` an integer, the type code for integers (see the description of the function `typeof()`, below)

`NUM` the same as `INT` (for historical reasons)

`FLOAT` an integer, the type code for floating-point numbers

`LIST` an integer, the type code for lists

`STR` an integer, the type code for strings

`OBJ` an integer, the type code for objects

`ERR` an integer, the type code for error values

For others, the general meaning of the value is consistent, though the value itself is different for different situations:

`player` an object, the player who typed the command that started the task that involved running this piece of code.

`this` an object, the object on which the currently-running verb was found.

`caller` an object, the object on which the verb that called the currently-running verb was found. For the first verb called for a given command, '`caller`' has the same value as '`player`'.

`verb` a string, the name by which the currently-running verb was identified.

`args` a list, the arguments given to this verb. For the first verb called for a given command, this is a list of strings, the words on the command line.

The rest of the so-called "built-in" variables are only really meaningful for the first verb called for a given command. Their semantics is given in the discussion of command parsing, above.

To change what value is stored in a variable, use an *assignment* expression:

variable = *expression*

For example, to change the variable named 'x' to have the value 17, you would write 'x = 17' as an expression. An assignment expression does two things:

- it changes the value of of the named variable, and
- it returns the new value of that variable.

Thus, the expression

```
13 + (x = 17)
```

changes the value of 'x' to be 17 and returns 30.

4.1.4 Arithmetic Operators

All of the usual simple operations on numbers are available to MOO programs:

```
+    -    *    /    %
```

These are, in order, addition, subtraction, multiplication, division, and remainder. In the following table, the expressions on the left have the corresponding values on the right:

```
5 + 2        ⇒   7
5 - 2        ⇒   3
5 * 2        ⇒   10
5 / 2        ⇒   2
5.0 / 2.0    ⇒   2.5
5 % 2        ⇒   1
5.0 % 2.0    ⇒   1.0
5 % -2       ⇒   1
-5 % 2       ⇒   -1
-5 % -2      ⇒   -1
-(5 + 2)     ⇒   -7
```

Note that integer division in MOO throws away the remainder and that the result of the remainder operator ('%') has the same sign as the left-hand operand. Also, note that '-' can be used without a left-hand operand to negate a numeric expression.

> *Fine point:* Integers and floating-point numbers cannot be mixed in any particular use of these arithmetic operators; unlike some other programming languages, MOO does not automatically coerce integers into floating-point numbers. You can use the `tofloat()` function to perform an explicit conversion.

The '+' operator can also be used to append two strings. The expression

```
"foo" + "bar"
```

has the value

```
"foobar"
```

Unless both operands to an arithmetic operator are numbers of the same kind (or, for '+', both strings), the error value `E_TYPE` is raised. If the right-hand operand for the division or remainder operators ('/' or '%') is zero, the error value `E_DIV` is raised.

MOO also supports the exponentiation operation, also known as "raising to a power," using the '^' operator:

```
3 ^ 4         ⇒    81
3 ^ 4.5       error    E_TYPE
3.5 ^ 4       ⇒    150.0625
3.5 ^ 4.5     ⇒    280.741230801382
```

Note that if the first operand is an integer, then the second operand must also be an integer. If the first operand is a floating-point number, then the second operand can be either kind of number. Although it is legal to raise an integer to a negative power, it is unlikely to be terribly useful.

4.1.5 Comparing Values

Any two values can be compared for equality using '`==`' and '`!=`'. The first of these returns 1 if the two values are equal and 0 otherwise; the second does the reverse:

```
3 == 4                              ⇒  0
3 != 4                              ⇒  1
3 == 3.0                            ⇒  0
"foo" == "Foo"                      ⇒  1
#34 != #34                          ⇒  0
{1, #34, "foo"} == {1, #34, "FoO"}  ⇒  1
E_DIV == E_TYPE                     ⇒  0
3 != "foo"                          ⇒  1
```

Note that integers and floating-point numbers are never equal to one another, even in the 'obvious' cases. Also note that comparison of strings (and list values containing strings) is case-insensitive; that is, it does not distinguish between the upper- and lower-case version of letters. To test two values for case-sensitive equality, use the '`equals`' function described later.

> **Warning**: It is easy (and very annoying) to confuse the equality-testing operator ('`==`') with the assignment operator ('`=`'), leading to nasty, hard-to-find bugs. Don't do this.

Numbers, object numbers, strings, and error values can also be compared for ordering purposes using the following operators:

```
<        <=        >=        >
```

meaning "less than," "less than or equal," "greater than or equal," and "greater than," respectively. As with the equality operators, these return 1 when their operands are in the appropriate relation and 0 otherwise:

```
3 < 4             ⇒  1
3 < 4.0           error   E_TYPE
#34 >= #32        ⇒  1
"foo" <= "Boo"    ⇒  0
E_DIV > E_TYPE    ⇒  1
```

Note that, as with the equality operators, strings are compared case-insensitively. To perform a case-sensitive string comparison, use the '`strcmp`' function described later. Also note that the error values are ordered as given in the table in the section on values. If the operands to these four comparison operators are of different types (even integers and floating-point numbers are considered different types), or if they are lists, then `E_TYPE` is raised.

4.1.6 Values as True and False

There is a notion in MOO of *true* and *false* values; every value is one or the other. The true values are as follows:

- all integers other than zero,
- all floating-point numbers not equal to `0.0`,
- all non-empty strings (i.e., other than '`""`'), and
- all non-empty lists (i.e., other than '`{}`').

All other values are false:

- the integer zero,
- the floating-point numbers `0.0` and `-0.0`,
- the empty string ('`""`'),
- the empty list ('`{}`'),
- all object numbers, and
- all error values.

There are four kinds of expressions and two kinds of statements that depend upon this classification of MOO values. In describing them, I sometimes refer to the *truth value* of a MOO value; this is just *true* or *false*, the category into which that MOO value is classified.

The conditional expression in MOO has the following form:

expression-1 `?` *expression-2* `|` *expression-3*

First, *expression-1* is evaluated. If it returns a true value, then *expression-2* is evaluated and whatever it returns is returned as the value of the conditional expression as a whole. If *expression-1* returns a false value, then *expression-3* is evaluated instead and its value is used as that of the conditional expression.

```
1 ? 2 | 3             ⇒   2
0 ? 2 | 3             ⇒   3
"foo" ? 17 | {#34}    ⇒   17
```

Note that only one of *expression-2* and *expression-3* is evaluated, never both.

To negate the truth value of a MOO value, use the '`!`' operator:

`!` *expression*

If the value of *expression* is true, '`!`' returns 0; otherwise, it returns 1:

```
! "foo"      ⇒   0
! (3 >= 4)   ⇒   1
```

The negation operator is usually read as "not."

It is frequently useful to test more than one condition to see if some or all of them are true. MOO provides two operators for this:

expression-1 `&&` *expression-2*
expression-1 `||` *expression-2*

These operators are usually read as "and" and "or," respectively.

The '`&&`' operator first evaluates *expression-1*. If it returns a true value, then *expression-2* is evaluated and its value becomes the value of the '`&&`' expression as a whole; otherwise, the value of *expression-1* is used as the value of the '`&&`' expression. Note that *expression-2* is only evaluated if *expression-1* returns a true value. The '`&&`' expression is equivalent to the conditional expression

expression-1 `?` *expression-2* `|` *expression-1*

except that *expression-1* is only evaluated once.

The '`||`' operator works similarly, except that *expression-2* is evaluated only if *expression-1* returns a false value. It is equivalent to the conditional expression

expression-1 `?` *expression-1* `|` *expression-2*

except that, as with '`&&`', *expression-1* is only evaluated once.

These two operators behave very much like "and" and "or" in English:

```
1 && 1                    ⇒   1
0 && 1                    ⇒   0
0 && 0                    ⇒   0
1 || 1                    ⇒   1
0 || 1                    ⇒   1
0 || 0                    ⇒   0
17 <= 23  &&  23 <= 27    ⇒   1
```

4.1.7 Indexing into Lists and Strings

Both strings and lists can be seen as ordered sequences of MOO values. In the case of strings, each is a sequence of single-character strings; that is, one can view the string `"bar"` as a sequence of the strings `"b"`, `"a"`, and `"r"`. MOO allows you to refer to the elements of lists and strings by number, by the *index* of that element in the list or string. The first element in a list or string has index 1, the second has index 2, and so on.

4.1.7.1 Extracting an Element from a List or String

The indexing expression in MOO extracts a specified element from a list or string:

expression-1`[`*expression-2*`]`

First, *expression-1* is evaluated; it must return a list or a string (the *sequence*). Then, *expression-2* is evaluated and must return an integer (the *index*). If either of the expressions returns some other type of value, `E_TYPE` is returned. The index must be between 1 and the length of the sequence, inclusive; if it is not, then `E_RANGE` is raised. The value of the indexing expression is the index'th element in the sequence. Anywhere within *expression-2*, you can use the symbol `$` as an expression returning the length of the value of *expression-1*.

```
"fob"[2]                  ⇒   "o"
"fob"[1]                  ⇒   "f"
{#12, #23, #34}[$ - 1]    ⇒   #23
```

Note that there are no legal indices for the empty string or list, since there are no integers between 1 and 0 (the length of the empty string or list).

Fine point: The `$` expression actually returns the length of the value of the expression just before the nearest enclosing `[...]` indexing or subranging brackets. For example:

```
"frob"[{3, 2, 4}[$]]      ⇒   "b"
```

4.1.7.2 Replacing an Element of a List or String

It often happens that one wants to change just one particular slot of a list or string, which is stored in a variable or a property. This can be done conveniently using an *indexed assignment* having one of the following forms:

```
variable[index-expr] = result-expr
object-expr.name[index-expr] = result-expr
object-expr.(name-expr)[index-expr] = result-expr
$name[index-expr] = result-expr
```

The first form writes into a variable, and the last three forms write into a property. The usual errors (`E_TYPE`, `E_INVIND`, `E_PROPNF` and `E_PERM` for lack of read/write permission on the property) may be raised, just as in reading and writing any object property; see the discussion of object property expressions below for details. Correspondingly, if *variable* does not yet have a value (i.e., it has never been assigned to), `E_VARNF` will be raised.

If *index-expr* is not an integer, or if the value of *variable* or the property is not a list or string, `E_TYPE` is raised. If *result-expr* is a string, but not of length 1, `E_INVARG` is raised. Now suppose *index-expr* evaluates to an integer *k*. If *k* is outside the range of the list or string (i.e. smaller than 1 or greater than the length of the list or string), `E_RANGE` is raised. Otherwise, the actual assignment takes place. For lists, the variable or the property is assigned a new list that is identical to the original one except at the *k*-th position, where the new list contains the result of *result-expr* instead. For strings, the variable or the property is assigned a new string that is identical to the original one, except the *k*-th character is changed to be *result-expr*.

The assignment expression itself returns the value of *result-expr*. For the following examples, assume that `l` initially contains the list `{1, 2, 3}` and that `s` initially contains the string "foobar":

```
l[5] = 3             error   E_RANGE
l["first"] = 4       error   E_TYPE
s[3] = "baz"         error   E_INVARG
l[2] = l[2] + 3      ⇒   5
l                    ⇒   {1, 5, 3}
l[2] = "foo"         ⇒   "foo"
l                    ⇒   {1, "foo", 3}
s[2] = "u"           ⇒   "u"
s                    ⇒   "fuobar"
s[$] = "z"           ⇒   "z"
s                    ⇒   "fuobaz"
```

Note that the `$` expression may also be used in indexed assignments with the same meaning as before.

> *Fine point:* After an indexed assignment, the variable or property contains a *new* list or string, a copy of the original list in all but the *k*-th place, where it contains a new value. In programming-language jargon, the original list is not mutated, and there is no aliasing. (Indeed, no MOO value is mutable and no aliasing ever occurs.)

In the list case, indexed assignment can be nested to many levels, to work on nested lists. Assume that `l` initially contains the list

```
{{1, 2, 3}, {4, 5, 6}, "foo"}
```

in the following examples:

```
l[7] = 4                 error   E_RANGE
l[1][8] = 35             error   E_RANGE
l[3][2] = 7              error   E_TYPE
l[1][1][1] = 3           error   E_TYPE
l[2][2] = -l[2][2]    ⇒   -5
l                     ⇒   {{1, 2, 3}, {4, -5, 6}, "foo"}
l[2] = "bar"          ⇒   "bar"
l                     ⇒   {{1, 2, 3}, "bar", "foo"}
l[2][$] = "z"         ⇒   "z"
l                     ⇒   {{1, 2, 3}, "baz", "foo"}
```

The first two examples raise `E_RANGE` because 7 is out of the range of `l` and 8 is out of the range of `l[1]`. The next two examples raise `E_TYPE` because `l[3]` and `l[1][1]` are not lists.

4.1.7.3 Extracting a Subsequence of a List or String

The range expression extracts a specified subsequence from a list or string:

expression-1`[`*expression-2*`..`*expression-3*`]`

The three expressions are evaluated in order. *Expression-1* must return a list or string (the *sequence*) and the other two expressions must return integers (the *low* and *high* indices, respectively); otherwise, `E_TYPE` is raised. The `$` expression can be used in either or both of *expression-2* and *expression-3* just as before, meaning the length of the value of *expression-1*.

If the low index is greater than the high index, then the empty string or list is returned, depending on whether the sequence is a string or a list. Otherwise, both indices must be between 1 and the length of the sequence; `E_RANGE` is raised if they are not. A new list or string is returned that contains just the elements of the sequence with indices between the low and high bounds.

```
"foobar"[2..$]                       ⇒   "oobar"
"foobar"[3..3]                       ⇒   "o"
"foobar"[17..12]                     ⇒   ""
{"one", "two", "three"}[$ - 1..$]    ⇒   {"two", "three"}
{"one", "two", "three"}[3..3]        ⇒   {"three"}
{"one", "two", "three"}[17..12]      ⇒   {}
```

4.1.7.4 Replacing a Subsequence of a List or String

The subrange assigment replaces a specified subsequence of a list or string with a supplied subsequence. The allowed forms are:

variable`[`*start-index-expr*`..`*end-index-expr*`] =` *result-expr*
object-expr`.`*name*`[`*start-index-expr*`..`*end-index-expr*`] =` *result-expr*
object-expr`.(`*name-expr*`)[`*start-index-expr*`..`*end-index-expr*`] =` *result-expr*
`$`*name*`[`*start-index-expr*`..`*end-index-expr*`] =` *result-expr*

As with indexed assigments, the first form writes into a variable, and the last three forms write into a property. The same errors (`E_TYPE`, `E_INVIND`, `E_PROPNF` and `E_PERM` for lack of read/write permission on the property) may be raised. If *variable* does not yet have a value (i.e., it has never been assigned to), `E_VARNF` will be raised. As before, the `$` expression can be used in either *start-index-expr* or *end-index-expr*, meaning the length of the original value of the expression just before the `[...]` part.

If *start-index-expr* or *end-index-expr* is not an integer, if the value of *variable* or the property is not a list or string, or *result-expr* is not the same type as *variable* or the property, `E_TYPE` is raised. `E_RANGE` is raised if *end-index-expr* is less than zero or if *start-index-expr* is greater than the length of the list or string plus one. Note: the length of *result-expr* does not need to be the same as the length of the specified range.

In precise terms, the subrange assigment

```
v[start..end] = value
```

is equivalent to

```
v = {@v[1..start - 1], @value, @v[end + 1..$]}
```

if *v* is a list and to

```
v = v[1..start - 1] + value + v[end + 1..$]
```

if *v* is a string.

The assigment expression itself returns the value of *result-expr*. For the following examples, assume that `l` initially contains the list `{1, 2, 3}` and that `s` initially contains the string "foobar":

```
l[5..6] = {7, 8}        error   E_RANGE
l[2..3] = 4             error   E_TYPE
l[#2..3] = {7}          error   E_TYPE
s[2..3] = {6}           error   E_TYPE
l[2..3] = {6, 7, 8, 9}  ⇒  {6, 7, 8, 9}
l                       ⇒  {1, 6, 7, 8, 9}
l[2..1] = {10, "foo"}   ⇒  {10, "foo"}
l                       ⇒  {1, 10, "foo", 6, 7, 8, 9}
l[3][2..$] = "u"        ⇒  "u"
l                       ⇒  {1, 10, "fu", 6, 7, 8, 9}
s[7..12] = "baz"        ⇒  "baz"
s                       ⇒  "foobarbaz"
s[1..3] = "fu"          ⇒  "fu"
s                       ⇒  "fubarbaz"
s[1..0] = "test"        ⇒  "test"
s                       ⇒  "testfubarbaz"
```

4.1.8 Other Operations on Lists

As was mentioned earlier, lists can be constructed by writing a comma-separated sequence of expressions inside curly braces:

```
{expression-1, expression-2, ..., expression-N}
```

The resulting list has the value of *expression-1* as its first element, that of *expression-2* as the second, etc.

```
{3 < 4, 3 <= 4, 3 >= 4, 3 > 4}  ⇒  {1, 1, 0, 0}
```

Additionally, one may precede any of these expressions by the splicing operator, ‘`@`’. Such an expression must return a list; rather than the old list itself becoming an element of the new list, all of the elements of the old list are included in the new list. This concept is easy to understand, but hard to explain in words, so here are some examples. For these examples, assume that the variable `a` has the value `{2, 3, 4}` and that `b` has the value `{"Foo", "Bar"}`:

```
{1, a, 5}   ⇒  {1, {2, 3, 4}, 5}
{1, @a, 5}  ⇒  {1, 2, 3, 4, 5}
{a, @a}     ⇒  {{2, 3, 4}, 2, 3, 4}
{@a, @b}    ⇒  {2, 3, 4, "Foo", "Bar"}
```

If the splicing operator (‘@’) precedes an expression whose value is not a list, then `E_TYPE` is raised as the value of the list construction as a whole.

The list membership expression tests whether or not a given MOO value is an element of a given list and, if so, with what index:

expression-1 `in` *expression-2*

Expression-2 must return a list; otherwise, `E_TYPE` is raised. If the value of *expression-1* is in that list, then the index of its first occurrence in the list is returned; otherwise, the ‘`in`’ expression returns 0.

```
2 in {5, 8, 2, 3}               ⇒  3
7 in {5, 8, 2, 3}               ⇒  0
"bar" in {"Foo", "Bar", "Baz"}  ⇒  2
```

Note that the list membership operator is case-insensitive in comparing strings, just like the comparison operators. To perform a case-sensitive list membership test, use the ‘`is_member`’ function described later. Note also that since it returns zero only if the given value is not in the given list, the ‘`in`’ expression can be used either as a membership test or as an element locator.

4.1.9 Spreading List Elements Among Variables

It is often the case in MOO programming that you will want to access the elements of a list individually, with each element stored in a separate variables. This desire arises, for example, at the beginning of almost every MOO verb, since the arguments to all verbs are delivered all bunched together in a single list. In such circumstances, you *could* write statements like these:

```
first = args[1];
second = args[2];
if (length(args) > 2)
  third = args[3];
else
  third = 0;
endif
```

This approach gets pretty tedious, both to read and to write, and it's prone to errors if you mistype one of the indices. Also, you often want to check whether or not any *extra* list elements were present, adding to the tedium.

MOO provides a special kind of assignment expression, called *scattering assignment* made just for cases such as these. A scattering assignment expression looks like this:

`{`*target*`, `...`} = `*expr*

where each *target* describes a place to store elements of the list that results from evaluating *expr*. A *target* has one of the following forms:

variable This is the simplest target, just a simple variable; the list element in the corresponding position is assigned to the variable. This is called a *required* target, since the assignment is required to put one of the list elements into the variable.

?*variable* This is called an *optional* target, since it doesn't always get assigned an element. If there are any list elements left over after all of the required targets have been accounted for (along with all of the other optionals to the left of this one), then this variable is treated like a required one and the list element in the corresponding position is assigned to the variable. If there aren't enough elements to assign one to this target, then no assignment is made to this variable, leaving it with whatever its previous value was.

?*variable* = *default-expr*
This is also an optional target, but if there aren't enough list elements available to assign one to this target, the result of evaluating *default-expr* is assigned to it instead. Thus, *default-expr* provides a *default value* for the variable. The default value expressions are evaluated and assigned working from left to right *after* all of the other assignments have been performed.

@*variable* By analogy with the @ syntax in list construction, this variable is assigned a list of all of the 'leftover' list elements in this part of the list after all of the other targets have been filled in. It is assigned the empty list if there aren't any elements left over. This is called a *rest* target, since it gets the rest of the elements. There may be at most one rest target in each scattering assignment expression.

If there aren't enough list elements to fill all of the required targets, or if there are more than enough to fill all of the required and optional targets but there isn't a rest target to take the leftover ones, then E_ARGS is raised.

Here are some examples of how this works. Assume first that the verb me:foo() contains the following code:

```
b = c = e = 17;
{a, ?b, ?c = 8, @d, ?e = 9, f} = args;
return {a, b, c, d, e, f};
```

Then the following calls return the given values:

```
me:foo(1)                          error    E_ARGS
me:foo(1, 2)                       ⇒   {1, 17, 8, {}, 9, 2}
me:foo(1, 2, 3)                    ⇒   {1, 2, 8, {}, 9, 3}
me:foo(1, 2, 3, 4)                 ⇒   {1, 2, 3, {}, 9, 4}
me:foo(1, 2, 3, 4, 5)              ⇒   {1, 2, 3, {}, 4, 5}
me:foo(1, 2, 3, 4, 5, 6)           ⇒   {1, 2, 3, {4}, 5, 6}
me:foo(1, 2, 3, 4, 5, 6, 7)        ⇒   {1, 2, 3, {4, 5}, 6, 7}
me:foo(1, 2, 3, 4, 5, 6, 7, 8)     ⇒   {1, 2, 3, {4, 5, 6}, 7, 8}
```

Using scattering assignment, the example at the begining of this section could be rewritten more simply, reliably, and readably:

```
{first, second, ?third = 0} = args;
```

It is good MOO programming style to use a scattering assignment at the top of nearly every verb, since it shows so clearly just what kinds of arguments the verb expects.

4.1.10 Getting and Setting the Values of Properties

Usually, one can read the value of a property on an object with a simple expression:

expression.*name*

Expression must return an object number; if not, `E_TYPE` is raised. If the object with that number does not exist, `E_INVIND` is raised. Otherwise, if the object does not have a property with that name, then `E_PROPNF` is raised. Otherwise, if the named property is not readable by the owner of the current verb, then `E_PERM` is raised. Finally, assuming that none of these terrible things happens, the value of the named property on the given object is returned.

I said "usually" in the paragraph above because that simple expression only works if the name of the property obeys the same rules as for the names of variables (i.e., consists entirely of letters, digits, and underscores, and doesn't begin with a digit). Property names are not restricted to this set, though. Also, it is sometimes useful to be able to figure out what property to read by some computation. For these more general uses, the following syntax is also allowed:

expression-1.(*expression-2*)

As before, *expression-1* must return an object number. *Expression-2* must return a string, the name of the property to be read; `E_TYPE` is raised otherwise. Using this syntax, any property can be read, regardless of its name.

Note that, as with almost everything in MOO, case is not significant in the names of properties. Thus, the following expressions are all equivalent:

```
foo.bar
foo.Bar
foo.("bAr")
```

The LambdaCore database uses several properties on `#0`, the *system object*, for various special purposes. For example, the value of `#0.room` is the "generic room" object, `#0.exit` is the "generic exit" object, etc. This allows MOO programs to refer to these useful objects more easily (and more readably) than using their object numbers directly. To make this usage even easier and more readable, the expression

`$`*name*

(where *name* obeys the rules for variable names) is an abbreviation for

`#0.`*name*

Thus, for example, the value `$nothing` mentioned earlier is really `#-1`, the value of `#0.nothing`.

As with variables, one uses the assignment operator ('`=`') to change the value of a property. For example, the expression

```
14 + (#27.foo = 17)
```

changes the value of the '`foo`' property of the object numbered 27 to be 17 and then returns 31. Assignments to properties check that the owner of the current verb has write permission on the given property, raising `E_PERM` otherwise. Read permission is not required.

4.1.11 Calling Built-in Functions and Other Verbs

MOO provides a large number of useful functions for performing a wide variety of operations; a complete list, giving their names, arguments, and semantics, appears in a separate section later. As an example to give you the idea, there is a function named '`length`' that returns the length of a given string or list.

The syntax of a call to a function is as follows:

`name(expr-1, expr-2, ..., expr-N)`

where *name* is the name of one of the built-in functions. The expressions between the parentheses, called *arguments*, are each evaluated in turn and then given to the named function to use in its appropriate way. Most functions require that a specific number of arguments be given; otherwise, `E_ARGS` is raised. Most also require that certain of the arguments have certain specified types (e.g., the `length()` function requires a list or a string as its argument); `E_TYPE` is raised if any argument has the wrong type.

As with list construction, the splicing operator '`@`' can precede any argument expression. The value of such an expression must be a list; `E_TYPE` is raised otherwise. The elements of this list are passed as individual arguments, in place of the list as a whole.

Verbs can also call other verbs, usually using this syntax:

`expr-0:name(expr-1, expr-2, ..., expr-N)`

Expr-0 must return an object number; `E_TYPE` is raised otherwise. If the object with that number does not exist, `E_INVIND` is raised. If this task is too deeply nested in verbs calling verbs calling verbs, then `E_MAXREC` is raised; the default limit is 50 levels, but this can be changed from within the database; see the chapter on server assumptions about the database for details. If neither the object nor any of its ancestors defines a verb matching the given name, `E_VERBNF` is raised. Otherwise, if none of these nasty things happens, the named verb on the given object is called; the various built-in variables have the following initial values in the called verb:

`this` an object, the value of *expr-0*

`verb` a string, the *name* used in calling this verb

`args` a list, the values of *expr-1*, *expr-2*, etc.

`caller` an object, the value of `this` in the calling verb

`player` an object, the same value as it had initially in the calling verb or, if the calling verb is running with wizard permissions, the same as the current value in the calling verb.

All other built-in variables (`argstr`, `dobj`, etc.) are initialized with the same values they have in the calling verb.

As with the discussion of property references above, I said "usually" at the beginning of the previous paragraph because that syntax is only allowed when the *name* follows the rules for allowed variable names. Also as with property reference, there is a syntax allowing you to compute the name of the verb:

`expr-0:(expr-00)(expr-1, expr-2, ..., expr-N)`

The expression *expr-00* must return a string; `E_TYPE` is raised otherwise.

The splicing operator ('`@`') can be used with verb-call arguments, too, just as with the arguments to built-in functions.

In many databases, a number of important verbs are defined on `#0`, the *system object*. As with the '`$foo`' notation for properties on `#0`, the server defines a special syntax for calling verbs on `#0`:

`$name(expr-1, expr-2, ..., expr-N)`

(where *name* obeys the rules for variable names) is an abbreviation for

`#0:name(expr-1, expr-2, ..., expr-N)`

4.1.12 Catching Errors in Expressions

It is often useful to be able to *catch* an error that an expression raises, to keep the error from aborting the whole task, and to keep on running as if the expression had returned some other value normally. The following expression accomplishes this:

` *expr-1* `!` *codes* `=>` *expr-2* '

Note: the open- and close-quotation marks in the previous line are really part of the syntax; you must actually type them as part of your MOO program for this kind of expression.

The *codes* part is either the keyword `ANY` or else a comma-separated list of expressions, just like an argument list. As in an argument list, the splicing operator (‘`@`’) can be used here. The `=>` *expr-2* part of the error-catching expression is optional.

First, the *codes* part is evaluated, yielding a list of error codes that should be caught if they're raised; if *codes* is `ANY`, then it is equivalent to the list of all possible MOO values.

Next, *expr-1* is evaluated. If it evaluates normally, without raising an error, then its value becomes the value of the entire error-catching expression. If evaluating *expr-1* results in an error being raised, then call that error *E*. If *E* is in the list resulting from evaluating *codes*, then *E* is considered *caught* by this error-catching expression. In such a case, if *expr-2* was given, it is evaluated to get the outcome of the entire error-catching expression; if *expr-2* was omitted, then *E* becomes the value of the entire expression. If *E* is *not* in the list resulting from *codes*, then this expression does not catch the error at all and it continues to be raised, possibly to be caught by some piece of code either surrounding this expression or higher up on the verb-call stack.

Here are some examples of the use of this kind of expression:

```
‘x + 1 ! E_TYPE => 0’
```

Returns `x + 1` if `x` is an integer, returns `0` if `x` is not an integer, and raises `E_VARNF` if `x` doesn't have a value.

```
‘x.y ! E_PROPNF, E_PERM => 17’
```

Returns `x.y` if that doesn't cause an error, `17` if `x` doesn't have a `y` property or that property isn't readable, and raises some other kind of error (like `E_INVIND`) if `x.y` does.

```
‘1 / 0 ! ANY’
```

Returns `E_DIV`.

4.1.13 Parentheses and Operator Precedence

As shown in a few examples above, MOO allows you to use parentheses to make it clear how you intend for complex expressions to be grouped. For example, the expression

```
3 * (4 + 5)
```

performs the addition of 4 and 5 before multiplying the result by 3.

If you leave out the parentheses, MOO will figure out how to group the expression according to certain rules. The first of these is that some operators have higher *precedence* than others; operators with higher precedence will more tightly bind to their operands than those with lower precedence. For example, multiplication has higher precedence than addition; thus, if the parentheses had been left out of the expression in the previous paragraph, MOO would have grouped it as follows:

```
(3 * 4) + 5
```

The table below gives the relative precedence of all of the MOO operators; operators on higher lines in the table have higher precedence and those on the same line have identical precedence:

```
!       -  (without a left operand)
^
*       /       %
+       -
==      !=      <       <=      >       >=      in
&&      ||
... ? ... | ... (the conditional expression)
=
```

Thus, the horrendous expression

```
x = a < b && c > d + e * f ? w in y | - q - r
```

would be grouped as follows:

```
x = (((a < b) && (c > (d + (e * f)))) ? (w in y) | ((- q) - r))
```

It is best to keep expressions simpler than this and to use parentheses liberally to make your meaning clear to other humans.

4.2 MOO Language Statements

Statements are MOO constructs that, in contrast to expressions, perform some useful, non-value-producing operation. For example, there are several kinds of statements, called 'looping constructs', that repeatedly perform some set of operations. Fortunately, there are many fewer kinds of statements in MOO than there are kinds of expressions.

4.2.1 Errors While Executing Statements

Statements do not return values, but some kinds of statements can, under certain circumstances described below, generate errors. If such an error is generated in a verb whose '`d`' (debug) bit is not set, then the error is ignored and the statement that generated it is simply skipped; execution proceeds with the next statement.

> **Note:** this error-ignoring behavior is very error prone, since it affects *all* errors, including ones the programmer may not have anticipated. The '`d`' bit exists only for historical reasons; it was once the only way for MOO programmers to catch and handle errors. The error-catching expression and the `try-except` statement are far better ways of accomplishing the same thing.

If the '`d`' bit is set, as it usually is, then the error is *raised* and can be caught and handled either by code surrounding the expression in question or by verbs higher up on the chain of calls leading to the current verb. If the error is not caught, then the server aborts the entire task and, by default, prints a message to the current player. See the descriptions of the error-catching expression and the `try-except` statement for the details of how errors can be caught, and the chapter on server assumptions about the database for details on the handling of uncaught errors.

4.2.2 Simple Statements

The simplest kind of statement is the *null* statement, consisting of just a semicolon:

```
;
```

It doesn't do anything at all, but it does it very quickly.

The next simplest statement is also one of the most common, the expression statement, consisting of any expression followed by a semicolon:

expression;

The given expression is evaluated and the resulting value is ignored. Commonly-used kinds of expressions for such statements include assignments and verb calls. Of course, there's no use for such a statement unless the evaluation of *expression* has some side-effect, such as changing the value of some variable or property, printing some text on someone's screen, etc.

4.2.3 Statements for Testing Conditions

The '`if`' statement allows you to decide whether or not to perform some statements based on the value of an arbitrary expression:

```
if (expression)
  statements
endif
```

Expression is evaluated and, if it returns a true value, the statements are executed in order; otherwise, nothing more is done.

One frequently wants to perform one set of statements if some condition is true and some other set of statements otherwise. The optional '`else`' phrase in an '`if`' statement allows you to do this:

```
if (expression)
  statements-1
else
  statements-2
endif
```

This statement is executed just like the previous one, except that *statements-1* are executed if *expression* returns a true value and *statements-2* are executed otherwise.

Sometimes, one needs to test several conditions in a kind of nested fashion:

```
if (expression-1)
  statements-1
else
  if (expression-2)
    statements-2
  else
    if (expression-3)
      statements-3
    else
      statements-4
    endif
  endif
endif
```

Such code can easily become tedious to write and difficult to read. MOO provides a somewhat simpler notation for such cases:

```
if (expression-1)
  statements-1
elseif (expression-2)
  statements-2
elseif (expression-3)
  statements-3
else
  statements-4
endif
```

Note that '`elseif`' is written as a single word, without any spaces. This simpler version has the very same meaning as the original: evaluate *expression-i* for *i* equal to 1, 2, and 3, in turn, until one of them returns a true value; then execute the *statements-i* associated with that expression. If none of the *expression-i* return a true value, then execute *statements-4*.

Any number of '`elseif`' phrases can appear, each having this form:

```
elseif (expression) statements
```

The complete syntax of the '`if`' statement, therefore, is as follows:

```
if (expression)
  statements
zero-or-more-elseif-phrases
an-optional-else-phrase
endif
```

4.2.4 Statements for Looping

MOO provides three different kinds of looping statements, allowing you to have a set of statements executed (1) once for each element of a given list, (2) once for each integer or object number in a given range, and (3) over and over until a given condition stops being true.

To perform some statements once for each element of a given list, use this syntax:

```
for variable in (expression)
  statements
endfor
```

The expression is evaluated and should return a list; if it does not, `E_TYPE` is raised. The *statements* are then executed once for each element of that list in turn; each time, the given *variable* is assigned the value of the element in question. For example, consider the following statements:

```
odds = {1, 3, 5, 7, 9};
evens = {};
for n in (odds)
  evens = {@evens, n + 1};
endfor
```

The value of the variable '`evens`' after executing these statements is the list

```
{2, 4, 6, 8, 10}
```

To perform a set of statements once for each integer or object number in a given range, use this syntax:

```
for variable in [expression-1..expression-2]
  statements
endfor
```

The two expressions are evaluated in turn and should either both return integers or both return object numbers; `E_TYPE` is raised otherwise. The *statements* are then executed once for each integer (or object number, as appropriate) greater than or equal to the value of *expression-1* and less than or equal to the result of *expression-2*, in increasing order. Each time, the given variable is assigned the integer or object number in question. For example, consider the following statements:

```
evens = {};
for n in [1..5]
  evens = {@evens, 2 * n};
endfor
```

The value of the variable '`evens`' after executing these statements is just as in the previous example: the list

```
{2, 4, 6, 8, 10}
```

The following loop over object numbers prints out the number and name of every valid object in the database:

```
for o in [#0..max_object()]
  if (valid(o))
    notify(player, tostr(o, ": ", o.name));
  endif
endfor
```

The final kind of loop in MOO executes a set of statements repeatedly as long as a given condition remains true:

```
while (expression)
  statements
endwhile
```

The expression is evaluated and, if it returns a true value, the *statements* are executed; then, execution of the '`while`' statement begins all over again with the evaluation of the expression. That is, execution alternates between evaluating the expression and executing the statements until the expression returns a false value. The following example code has precisely the same effect as the loop just shown above:

```
evens = {};
n = 1;
while (n <= 5)
  evens = {@evens, 2 * n};
  n = n + 1;
endwhile
```

Fine point: It is also possible to give a 'name' to a '`while`' loop, using this syntax:

```
while name (expression)
  statements
endwhile
```

which has precisely the same effect as

```
while (name = expression)
  statements
```

```
endwhile
```

This naming facility is only really useful in conjunction with the '`break`' and '`continue`' statements, described in the next section.

With each kind of loop, it is possible that the statements in the body of the loop will never be executed at all. For iteration over lists, this happens when the list returned by the expression is empty. For iteration on integers, it happens when *expression-1* returns a larger integer than *expression-2*. Finally, for the '`while`' loop, it happens if the expression returns a false value the very first time it is evaluated.

4.2.5 Terminating One or All Iterations of a Loop

Sometimes, it is useful to exit a loop before it finishes all of its iterations. For example, if the loop is used to search for a particular kind of element of a list, then it might make sense to stop looping as soon as the right kind of element is found, even if there are more elements yet to see. The '`break`' statement is used for this purpose; it has the form

```
break;
```

or

`break` *name*;

Each '`break`' statement indicates a specific surrounding loop; if *name* is not given, the statement refers to the innermost one. If it is given, *name* must be the name appearing right after the '`for`' or '`while`' keyword of the desired enclosing loop. When the '`break`' statement is executed, the indicated loop is immediately terminated and executing continues just as if the loop had completed its iterations normally.

MOO also allows you to terminate just the current iteration of a loop, making it immediately go on to the next one, if any. The '`continue`' statement does this; it has precisely the same forms as the '`break`' statement:

```
continue;
```

or

`continue` *name*;

4.2.6 Returning a Value from a Verb

The MOO program in a verb is just a sequence of statements. Normally, when the verb is called, those statements are simply executed in order and then the integer 0 is returned as the value of the verb-call expression. Using the '`return`' statement, one can change this behavior. The '`return`' statement has one of the following two forms:

```
return;
```

or

`return` *expression*;

When it is executed, execution of the current verb is terminated immediately after evaluating the given *expression*, if any. The verb-call expression that started the execution of this verb then returns either the value of *expression* or the integer 0, if no *expression* was provided.

4.2.7 Handling Errors in Statements

Normally, whenever a piece of MOO code raises an error, the entire task is aborted and a message printed to the user. Often, such errors can be anticipated in advance by the programmer and code written to deal with them in a more graceful manner. The `try-except` statement allows you to do this; the syntax is as follows:

```
try
  statements-0
except variable-1 (codes-1)
  statements-1
except variable-2 (codes-2)
  statements-2
...
endtry
```

where the *variable*s may be omitted and each *codes* part is either the keyword `ANY` or else a comma-separated list of expressions, just like an argument list. As in an argument list, the splicing operator ('`@`') can be used here. There can be anywhere from 1 to 255 `except` clauses.

First, each *codes* part is evaluated, yielding a list of error codes that should be caught if they're raised; if a *codes* is `ANY`, then it is equivalent to the list of all possible MOO values.

Next, *statements-0* is executed; if it doesn't raise an error, then that's all that happens for the entire `try-except` statement. Otherwise, let *E* be the error it raises. From top to bottom, *E* is searched for in the lists resulting from the various *codes* parts; if it isn't found in any of them, then it continues to be raised, possibly to be caught by some piece of code either surrounding this `try-except` statement or higher up on the verb-call stack.

If *E* is found first in *codes-i*, then *variable-i* (if provided) is assigned a value containing information about the error being raised and *statements-i* is executed. The value assigned to *variable-i* is a list of four elements:

```
{code, message, value, traceback}
```

where *code* is *E*, the error being raised, *message* and *value* are as provided by the code that raised the error, and *traceback* is a list like that returned by the '`callers()`' function, including line numbers. The *traceback* list contains entries for every verb from the one that raised the error through the one containing this `try-except` statement.

Unless otherwise mentioned, all of the built-in errors raised by expressions, statements, and functions provide `tostr(`*code*`)` as *message* and zero as *value*.

Here's an example of the use of this kind of statement:

```
try
  result = object:(command)(@arguments);
  player:tell("=> ", toliteral(result));
except v (ANY)
  tb = v[4];
  if (length(tb) == 1)
    player:tell("** Illegal command: ", v[2]);
  else
    top = tb[1];
    tb[1..1] = {};
    player:tell(top[1], ":", top[2], ", line ", top[6], ":",
```

```
                 v[2]);
      for fr in (tb)
        player:tell("... called from ", fr[1], ":", fr[2],
                    ", line ", fr[6]);
      endfor
      player:tell("(End of traceback)");
    endif
  endtry
```

4.2.8 Cleaning Up After Errors

Whenever an error is raised, it is usually the case that at least some MOO code gets skipped over and never executed. Sometimes, it's important that a piece of code *always* be executed, whether or not an error is raised. Use the `try-finally` statement for these cases; it has the following syntax:

```
try
  statements-1
finally
  statements-2
endtry
```

First, *statements-1* is executed; if it completes without raising an error, returning from this verb, or terminating the current iteration of a surrounding loop (we call these possibilities *transferring control*), then *statements-2* is executed and that's all that happens for the entire `try-finally` statement.

Otherwise, the process of transferring control is interrupted and *statments-2* is executed. If *statements-2* itself completes without transferring control, then the interrupted control transfer is resumed just where it left off. If *statements-2* does transfer control, then the interrupted transfer is simply forgotten in favor of the new one.

In short, this statement ensures that *statements-2* is executed after control leaves *statements-1* for whatever reason; it can thus be used to make sure that some piece of cleanup code is run even if *statements-1* doesn't simply run normally to completion.

Here's an example:

```
try
  start = time();
  object:(command)(@arguments);
finally
  end = time();
  this:charge_user_for_seconds(player, end - start);
endtry
```

4.2.9 Executing Statements at a Later Time

It is sometimes useful to have some sequence of statements execute at a later time, without human intervention. For example, one might implement an object that, when thrown into the air, eventually falls back to the ground; the '`throw`' verb on that object should arrange to print a message about the object landing on the ground, but the message shouldn't be printed until some number of seconds have passed.

The '`fork`' statement is intended for just such situations and has the following syntax:

```
fork (expression)
  statements
endfork
```

The '`fork`' statement first executes the expression, which must return a integer; call that integer *n*. It then creates a new MOO *task* that will, after at least *n* seconds, execute the statements. When the new task begins, all variables will have the values they had at the time the '`fork`' statement was executed. The task executing the '`fork`' statement immediately continues execution. The concept of tasks is discussed in detail in the next section.

By default, there is no limit to the number of tasks any player may fork, but such a limit can be imposed from within the database. See the chapter on server assumptions about the database for details.

Occasionally, one would like to be able to kill a forked task before it even starts; for example, some player might have caught the object that was thrown into the air, so no message should be printed about it hitting the ground. If a variable name is given after the '`fork`' keyword, like this:

```
fork name (expression)
  statements
endfork
```

then that variable is assigned the *task ID* of the newly-created task. The value of this variable is visible both to the task executing the fork statement and to the statements in the newly-created task. This ID can be passed to the `kill_task()` function to keep the task from running and will be the value of `task_id()` once the task begins execution.

4.3 MOO Tasks

A *task* is an execution of a MOO program. There are five kinds of tasks in LambdaMOO:

- Every time a player types a command, a task is created to execute that command; we call these *command tasks*.
- Whenever a player connects or disconnects from the MOO, the server starts a task to do whatever processing is necessary, such as printing out '`Munchkin has connected`' to all of the players in the same room; these are called *server tasks*.
- The '`fork`' statement in the programming language creates a task whose execution is delayed for at least some given number of seconds; these are *forked tasks*.
- The `suspend()` function suspends the execution of the current task. A snapshot is taken of whole state of the execution, and the execution will be resumed later. These are called *suspended tasks*.
- The `read()` function also suspends the execution of the current task, in this case waiting for the player to type a line of input. When the line is received, the task resumes with the `read()` function returning the input line as result. These are called *reading tasks*.

The last three kinds of tasks above are collectively known as *queued tasks* or *background tasks*, since they may not run immediately.

To prevent a maliciously- or incorrectly-written MOO program from running forever and monopolizing the server, limits are placed on the running time of every task. One limit is that no task is allowed to run longer than a certain number of seconds; command and server tasks get five seconds each while other tasks get only three seconds. This limit is, in practice, rarely reached. The reason is that there is also a limit on the number of operations a task may execute.

The server counts down *ticks* as any task executes. Roughly speaking, it counts one tick for every expression evaluation (other than variables and literals), one for every '`if`', '`fork`' or '`return`' statement, and one for every iteration of a loop. If the count gets all the way down to zero, the task is immediately and unceremoniously aborted. By default, command and server tasks begin with an store of 30,000 ticks; this is enough for almost all normal uses. Forked, suspended, and reading tasks are allotted 15,000 ticks each.

These limits on seconds and ticks may be changed from within the database, as can the behavior of the server after it aborts a task for running out; see the chapter on server assumptions about the database for details.

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

4.4 Built-in Functions

There are a large number of built-in functions available for use by MOO programmers. Each one is discussed in detail in this section. The presentation is broken up into subsections by grouping together functions with similar or related uses.

For most functions, the expected types of the arguments are given; if the actual arguments are not of these types, `E_TYPE` is raised. Some arguments can be of any type at all; in such cases, no type specification is given for the argument. Also, for most functions, the type of the result of the function is given. Some functions do not return a useful result; in such cases, the specification '`none`' is used. A few functions can potentially return any type of value at all; in such cases, the specification '`value`' is used.

Most functions take a certain fixed number of required arguments and, in some cases, one or two optional arguments. If a function is called with too many or too few arguments, `E_ARGS` is raised.

Functions are always called by the program for some verb; that program is running with the permissions of some player, usually the owner of the verb in question (it is not always the owner, though; wizards can use `set_task_perms()` to change the permissions 'on the fly'). In the function descriptions below, we refer to the player whose permissions are being used as the *programmer*.

Many built-in functions are described below as raising `E_PERM` unless the programmer meets certain specified criteria. It is possible to restrict use of any function, however, so that only wizards can use it; see the chapter on server assumptions about the database for details.

4.4.1 Object-Oriented Programming

One of the most important facilities in an object-oriented programming language is ability for a child object to make use of a parent's implementation of some operation, even when the child provides its own definition for that operation. The `pass()` function provides this facility in MOO.

`value` **pass** (*arg*, ...) Function

Often, it is useful for a child object to define a verb that *augments* the behavior of a verb on its parent object. For example, in the LambdaCore database, the root object (which is an ancestor of every other object) defines a verb called '`description`' that simply returns the value of `this.description`; this verb is used by the implementation of the `look` command. In many cases, a programmer would like the description of some object to include some non-constant part; for example, a sentence about whether or not the object was 'awake' or 'sleeping'. This sentence should be added onto the end of the normal description. The programmer would like to have a means of calling the normal `description` verb and then appending the sentence onto the end of that description. The function '`pass()`' is for exactly such situations.

`pass` calls the verb with the same name as the current verb but as defined on the parent of the object that defines the current verb. The arguments given to `pass` are the ones given to the called verb and the returned value of the called verb is returned from the call to `pass`. The initial value of `this` in the called verb is the same as in the calling verb.

Thus, in the example above, the child-object's `description` verb might have the following implementation:

```
return pass() + "  It is " + (this.awake ? "awake." | "sleeping.");
```

That is, it calls its parent's `description` verb and then appends to the result a sentence whose content is computed based on the value of a property on the object.

In almost all cases, you will want to call '`pass()`' with the same arguments as were given to the current verb. This is easy to write in MOO; just call `pass(@args)`.

4.4.2 Manipulating MOO Values

There are several functions for performing primitive operations on MOO values, and they can be cleanly split into two kinds: those that do various very general operations that apply to all types of values, and those that are specific to one particular type. There are so many operations concerned with objects that we do not list them in this section but rather give them their own section following this one.

4.4.2.1 General Operations Applicable to all Values

`int` **typeof** (*value*) Function

Takes any MOO value and returns an integer representing the type of *value*. The result is the same as the initial value of one of these built-in variables: `INT`, `FLOAT`, `STR`, `LIST`, `OBJ`, or `ERR`. Thus, one usually writes code like this:

```
if (typeof(x) == LIST) ...
```

and not like this:

```
if (typeof(x) == 3) ...
```

because the former is much more readable than the latter.

`str` **tostr** (*value*, ...) Function

Converts all of the given MOO values into strings and returns the concatenation of the results.

```
tostr(17)                 ⇒   "17"
tostr(1.0/3.0)            ⇒   "0.333333333333333"
tostr(#17)                ⇒   "#17"
tostr("foo")              ⇒   "foo"
tostr({1, 2})             ⇒   "{list}"
tostr(E_PERM)             ⇒   "Permission denied"
tostr("3 + 4 = ", 3 + 4)  ⇒   "3 + 4 = 7"
```

Note that `tostr()` does not do a good job of converting lists into strings; all lists, including the empty list, are converted into the string `"{list}"`. The function `toliteral()`, below, is better for this purpose.

`str` **toliteral** (*value*) Function

Returns a string containing a MOO literal expression that, when evaluated, would be equal to *value*.

```
toliteral(17)          ⇒   "17"
toliteral(1.0/3.0)     ⇒   "0.333333333333333"
toliteral(#17)         ⇒   "#17"
toliteral("foo")       ⇒   "\"foo\""
toliteral({1, 2})      ⇒   "{1, 2}"
toliteral(E_PERM)      ⇒   "E_PERM"
```

`int` **toint** (*value*) Function

`int` **tonum** (*value*) Function

Converts the given MOO value into an integer and returns that integer. Floating-point numbers are rounded toward zero, truncating their fractional parts. Object numbers are converted into the equivalent integers. Strings are parsed as the decimal encoding of a real number which is then converted to an integer. Errors are converted into integers obeying the same ordering (with respect to `<=` as the errors themselves. `Toint()` raises `E_TYPE` if *value* is a list. If *value* is a string but the string does not contain a syntactically-correct number, then `toint()` returns 0.

```
toint(34.7)        ⇒   34
toint(-34.7)       ⇒   -34
toint(#34)         ⇒   34
toint("34")        ⇒   34
toint("34.7")      ⇒   34
toint(" - 34  ")   ⇒   -34
toint(E_TYPE)      ⇒   1
```

`obj` **toobj** (*value*) Function

Converts the given MOO value into an object number and returns that object number. The conversions are very similar to those for `toint()` except that for strings, the number *may* be preceded by '`#`'.

```
toobj("34")       ⇒   #34
toobj("#34")      ⇒   #34
toobj("foo")      ⇒   #0
toobj({1, 2})     error   E_TYPE
```

`float` **tofloat** (*value*) Function

Converts the given MOO value into a floating-point number and returns that number. Integers and object numbers are converted into the corresponding integral floating-point numbers. Strings are parsed as the decimal encoding of a real number which is then represented as closely as possible as a floating-point number. Errors are first converted to integers as in `toint()` and then converted as integers are. `Tofloat()` raises `E_TYPE` if *value* is a list. If *value* is a string but the string does not contain a syntactically-correct number, then `tofloat()` returns 0.

```
tofloat(34)          ⇒   34.0
tofloat(#34)         ⇒   34.0
tofloat("34")        ⇒   34.0
tofloat("34.7")      ⇒   34.7
tofloat(E_TYPE)      ⇒   1.0
```

`int` **equals** (*value1*, *value2*) Function

Returns true if *value1* is completely indistinguishable from *value2*. This is much the same operation as "*value1* `==` *value2*" except that, unlike `==`, the `equals()` function does not treat upper- and lower-case characters in strings as equal.

```
"Foo" == "foo"          ⇒   1
equals("Foo", "foo")    ⇒   0
equals("Foo", "Foo")    ⇒   1
```

`int` **value_bytes** (*value*) Function

Returns the number of bytes of the server's memory required to store the given *value*.

`str` **value_hash** (*value*) Function

Returns the same string as `string_hash(toliteral(`*value*`))`; see the description of `string_hash()` for details.

4.4.2.2 Operations on Numbers

`int` **random** ([`int` *mod*]) Function

mod must be a positive integer; otherwise, `E_INVARG` is raised. An integer is chosen randomly from the range `[1..`*mod*`]` and returned. If *mod* is not provided, it defaults to the largest MOO integer, 2147483647.

`num` **min** (`num` *x*, ...) Function

`num` **max** (`num` *x*, ...) Function

These two functions return the smallest or largest of their arguments, respectively. All of the arguments must be numbers of the same kind (i.e., either integer or floating-point); otherwise `E_TYPE` is raised.

`num` **abs** (`num` *x*) Function

Returns the absolute value of *x*. If *x* is negative, then the result is `-`*x*; otherwise, the result is *x*. The number *x* can be either integer or floating-point; the result is of the same kind.

str **floatstr(float** *x*, int *precision* [, *scientific*]) Function

Converts *x* into a string with more control than provided by either `tostr()` or `toliteral()`. *Precision* is the number of digits to appear to the right of the decimal point, capped at 4 more than the maximum available precision, a total of 19 on most machines; this makes it possible to avoid rounding errors if the resulting string is subsequently read back as a floating-point value. If *scientific* is false or not provided, the result is a string in the form `"MMMMMMM.DDDDDD"`, preceded by a minus sign if and only if *x* is negative. If *scientific* is provided and true, the result is a string in the form `"M.DDDDDDe+EEE"`, again preceded by a minus sign if and only if *x* is negative.

float **sqrt** (float *x*) Function

Returns the square root of *x*. Raises `E_INVARG` if *x* is negative.

float **sin** (float *x*) Function
float **cos** (float *x*) Function
float **tan** (float *x*) Function

Returns the sine, cosine, or tangent of *x*, respectively.

float **asin** (float *x*) Function
float **acos** (float *x*) Function

Returns the arc-sine or arc-cosine (inverse sine or cosine) of *x*, in the range `[-pi/2..pi/2]` or `[0..pi]`, respectively. Raises `E_INVARG` if *x* is outside the range `[-1.0..1.0]`.

float **atan** (float *y* [, float *x*]) Function

Returns the arc-tangent (inverse tangent) of *y* in the range `[-pi/2..pi/2]` if *x* is not provided, or of *y*/*x* in the range `[-pi..pi]` if *x* is provided.

float **sinh** (float *x*) Function
float **cosh** (float *x*) Function
float **tanh** (float *x*) Function

Returns the hyperbolic sine, cosine, or tangent of *x*, respectively.

float **exp** (float *x*) Function

Returns *e* raised to the power of *x*.

float **log** (float *x*) Function
float **log10** (float *x*) Function

Returns the natural or base 10 logarithm of *x*. Raises `E_INVARG` if *x* is not positive.

float **ceil** (float *x*) Function

Returns the smallest integer not less than *x*, as a floating-point number.

float **floor** (float *x*) Function

Returns the largest integer not greater than *x*, as a floating-point number.

float **trunc** (float *x*) Function

Returns the integer obtained by truncating *x* at the decimal point, as a floating-point number. For negative *x*, this is equivalent to `ceil()`; otherwise it is equivalent to `floor()`.

4.4.2.3 Operations on Strings

int **length** (str *string*) Function

Returns the number of characters in *string*. It is also permissible to pass a list to `length()`; see the description in the next section.

```
length("foo")   ⇒  3
length("")      ⇒  0
```

str **strsub** (str *subject*, str *what*, str *with* [, *case-matters*]) Function

Replaces all occurrences in *subject* of *what* with *with*, performing string substitution. The occurrences are found from left to right and all substitutions happen simultaneously. By default, occurrences of *what* are searched for while ignoring the upper/lower case distinction. If *case-matters* is provided and true, then case is treated as significant in all comparisons.

```
strsub("%n is a fink.", "%n", "Fred")   ⇒  "Fred is a fink."
strsub("foobar", "OB", "b")             ⇒  "fobar"
strsub("foobar", "OB", "b", 1)          ⇒  "foobar"
```

int **index** (str *str1*, str *str2* [, *case-matters*]) Function
int **rindex** (str *str1*, str *str2* [, *case-matters*]) Function

The function `index()` (`rindex()`) returns the index of the first character of the first (last) occurrence of *str2* in *str1*, or zero if *str2* does not occur in *str1* at all. By default the search for an occurrence of *str2* is done while ignoring the upper/lower case distinction. If *case-matters* is provided and true, then case is treated as significant in all comparisons.

```
index("foobar", "o")        ⇒  2
rindex("foobar", "o")       ⇒  3
index("foobar", "x")        ⇒  0
index("foobar", "oba")      ⇒  3
index("Foobar", "foo", 1)   ⇒  0
```

int **strcmp** (str *str1*, str *str2*) Function

Performs a case-sensitive comparison of the two argument strings. If *str1* is lexicographically less than *str2*, the `strcmp()` returns a negative integer. If the two strings are identical, `strcmp()` returns zero. Otherwise, `strcmp()` returns a positive integer. The ASCII character ordering is used for the comparison.

list **decode_binary** (str *bin-string* [, *fully*]) Function

Returns a list of strings and/or integers representing the bytes in the binary string *bin_string* in order. If *fully* is false or omitted, the list contains an integer only for each non-printing, non-space byte; all other characters are grouped into the longest possible contiguous substrings. If *fully* is provided and true, the list contains only integers, one for each byte represented in *bin_string*. Raises `E_INVARG` if *bin_string* is not a properly-formed binary string. (See the early section on MOO value types for a full description of binary strings.)

```
decode_binary("foo")          ⇒  {"foo"}
decode_binary("~~foo")        ⇒  {"~foo"}
decode_binary("foo~0D~0A")    ⇒  {"foo", 13, 10}
```

```
decode_binary("foo~0Abar~0Abaz")   ⇒   {"foo", 10, "bar", 10, "baz"}
decode_binary("foo~0D~0A", 1)      ⇒   {102, 111, 111, 13, 10}
```

str **encode_binary** (*arg*, ...) Function

Each argument must be an integer between 0 and 255, a string, or a list containing only legal arguments for this function. This function translates each integer and string in turn into its binary string equivalent, returning the concatenation of all these substrings into a single binary string. (See the early section on MOO value types for a full description of binary strings.)

```
encode_binary("~foo")                      ⇒   "~7Efoo"
encode_binary({"foo", 10}, {"bar", 13})    ⇒   "foo~0Abar~0D"
encode_binary("foo", 10, "bar", 13)        ⇒   "foo~0Abar~0D"
```

list **match** (str *subject*, str *pattern* [, *case-matters*]) Function
list **rmatch** (str *subject*, str *pattern* [, *case-matters*]) Function

The function `match()` (`rmatch()`) searches for the first (last) occurrence of the regular expression *pattern* in the string *subject*. If *pattern* is syntactically malformed, then `E_INVARG` is raised. The process of matching can in some cases consume a great deal of memory in the server; should this memory consumption become excessive, then the matching process is aborted and `E_QUOTA` is raised.

If no match is found, the empty list is returned; otherwise, these functions return a list containing information about the match (see below). By default, the search ignores upper-/lower-case distinctions. If *case-matters* is provided and true, then case is treated as significant in all comparisons.

The list that `match()` (`rmatch()`) returns contains the details about the match made. The list is in the form:

{*start*, *end*, *replacements*, *subject*}

where *start* is the index in *subject* of the beginning of the match, *end* is the index of the end of the match, *replacements* is a list described below, and *subject* is the same string that was given as the first argument to the `match()` or `rmatch()`.

The *replacements* list is always nine items long, each item itself being a list of two integers, the start and end indices in *string* matched by some parenthesized sub-pattern of *pattern*. The first item in *replacements* carries the indices for the first parenthesized sub-pattern, the second item carries those for the second sub-pattern, and so on. If there are fewer than nine parenthesized sub-patterns in *pattern*, or if some sub-pattern was not used in the match, then the corresponding item in *replacements* is the list {0, -1}. See the discussion of '`%)`', below, for more information on parenthesized sub-patterns.

```
match("foo", "^f*o$")        ⇒  {}
match("foo", "^fo*$")        ⇒  {1, 3, {{0, -1}, ...}, "foo"}
match("foobar", "o*b")       ⇒  {2, 4, {{0, -1}, ...}, "foobar"}
rmatch("foobar", "o*b")      ⇒  {4, 4, {{0, -1}, ...}, "foobar"}
match("foobar", "f%(o*%)b")
        ⇒  {1, 4, {{2, 3}, {0, -1}, ...}, "foobar"}
```

Regular expression matching allows you to test whether a string fits into a specific syntactic shape. You can also search a string for a substring that fits a pattern.

A regular expression describes a set of strings. The simplest case is one that describes a particular string; for example, the string 'foo' when regarded as a regular expression matches 'foo' and nothing else. Nontrivial regular expressions use certain special constructs so that they can match more than one string. For example, the regular expression 'foo%|bar' matches either the string 'foo' or the string 'bar'; the regular expression 'c[ad]*r' matches any of the strings 'cr', 'car', 'cdr', 'caar', 'cadddar' and all other such strings with any number of 'a''s and 'd''s.

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression that matches that character and nothing else. The special characters are '$', '^', '.', '*', '+', '?', '[', ']' and '%'. Any other character appearing in a regular expression is ordinary, unless a '%' precedes it.

For example, 'f' is not a special character, so it is ordinary, and therefore 'f' is a regular expression that matches the string 'f' and no other string. (It does *not*, for example, match the string 'ff'.) Likewise, 'o' is a regular expression that matches only 'o'.

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression which matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string.

As a simple example, we can concatenate the regular expressions 'f' and 'o' to get the regular expression 'fo', which matches only the string 'fo'. Still trivial.

The following are the characters and character sequences that have special meaning within regular expressions. Any character not mentioned here is not special; it stands for exactly itself for the purposes of searching and matching.

'.'
: is a special character that matches any single character. Using concatenation, we can make regular expressions like 'a.b', which matches any three-character string that begins with 'a' and ends with 'b'.

'*'
: is not a construct by itself; it is a suffix that means that the preceding regular expression is to be repeated as many times as possible. In 'fo*', the '*' applies to the 'o', so 'fo*' matches 'f' followed by any number of 'o''s.

  The case of zero 'o''s is allowed: 'fo*' does match 'f'.

  '*' always applies to the *smallest* possible preceding expression. Thus, 'fo*' has a repeating 'o', not a repeating 'fo'.

  The matcher processes a '*' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, it backtracks, discarding some of the matches of the '*''d construct in case that makes it possible to match the rest of the pattern. For example, matching 'c[ad]*ar' against the string 'caddaar', the '[ad]*' first matches 'addaa', but this does not allow the next 'a' in the pattern to match. So the last of the matches of '[ad]' is undone and the following 'a' is tried again. Now it succeeds.

'+'
: '+' is like '*' except that at least one match for the preceding pattern is required for '+'. Thus, 'c[ad]+r' does not match 'cr' but does match anything else that 'c[ad]*r' would match.

'?' '?' is like '*' except that it allows either zero or one match for the preceding pattern. Thus, 'c[ad]?r' matches 'cr' or 'car' or 'cdr', and nothing else.

'[...]' '[' begins a *character set*, which is terminated by a ']'. In the simplest case, the characters between the two brackets form the set. Thus, '[ad]' matches either 'a' or 'd', and '[ad]*' matches any string of 'a''s and 'd''s (including the empty string), from which it follows that 'c[ad]*r' matches 'car', etc.

Character ranges can also be included in a character set, by writing two characters with a '-' between them. Thus, '[a-z]' matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in '[a-z$%.]', which matches any lower case letter or '$', '%' or period.

Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: ']', '-' and '^'.

To include a ']' in a character set, you must make it the first character. For example, '[]a]' matches ']' or 'a'. To include a '-', you must use it in a context where it cannot possibly indicate a range: that is, as the first character, or immediately after a range.

'[^ ...]' '[^' begins a *complement character set*, which matches any character except the ones specified. Thus, '[^a-z0-9A-Z]' matches all characters *except* letters and digits.

'^' is not special in a character set unless it is the first character. The character following the '^' is treated as if it were first (it may be a '-' or a ']').

'^' is a special character that matches the empty string – but only if at the beginning of the string being matched. Otherwise it fails to match anything. Thus, '^foo' matches a 'foo' which occurs at the beginning of the string.

'$' is similar to '^' but matches only at the *end* of the string. Thus, 'xx*$' matches a string of one or more 'x''s at the end of the string.

'%' has two functions: it quotes the above special characters (including '%'), and it introduces additional special constructs.

Because '%' quotes special characters, '%$' is a regular expression that matches only '$', and '%[' is a regular expression that matches only '[', and so on.

For the most part, '%' followed by any character matches only that character. However, there are several exceptions: characters that, when preceded by '%', are special constructs. Such characters are always ordinary when encountered on their own.

No new special characters will ever be defined. All extensions to the regular expression syntax are made by defining new two-character constructs that begin with '%'.

'%|' specifies an alternative. Two regular expressions *a* and *b* with '%|' in between form an expression that matches anything that either *a* or *b* will match.

Thus, '`foo%|bar`' matches either '`foo`' or '`bar`' but no other string.

'`%|`' applies to the largest possible surrounding expressions. Only a surrounding '`%( ... %)`' grouping can limit the grouping power of '`%|`'.

Full backtracking capability exists for when multiple '`%|`''s are used.

'`%( ... %)`' is a grouping construct that serves three purposes:

1. To enclose a set of '`%|`' alternatives for other operations. Thus, '`%(foo%|bar%)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for a following '`*`', '`+`', or '`?`' to operate on. Thus, '`ba%(na%)*`' matches '`bananana`', etc., with any number of '`na`''s, including none.
3. To mark a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature that happens to be assigned as a second meaning to the same '`%( ... %)`' construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

'`%`*digit*' After the end of a '`%( ... %)`' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '`%`' followed by *digit* to mean "match the same text matched by the *digit*'th '`%( ... %)`' construct in the pattern." The '`%( ... %)`' constructs are numbered in the order that their '`%(`''s appear in the pattern.

The strings matching the first nine '`%( ... %)`' constructs appearing in a regular expression are assigned numbers 1 through 9 in order of their beginnings. '`%1`' through '`%9`' may be used to refer to the text matched by the corresponding '`%( ... %)`' construct.

For example, '`%(.*%)%1`' matches any string that is composed of two identical halves. The '`%(.*%)`' matches the first half, which may be anything, but the '`%1`' that follows must match the same exact text.

'`%b`' matches the empty string, but only if it is at the beginning or end of a word. Thus, '`%bfoo%b`' matches any occurrence of '`foo`' as a separate word. '`%bball%(s%|%)%b`' matches '`ball`' or '`balls`' as a separate word.

For the purposes of this construct and the five that follow, a word is defined to be a sequence of letters and/or digits.

'`%B`' matches the empty string, provided it is *not* at the beginning or end of a word.

'`%<`' matches the empty string, but only if it is at the beginning of a word.

'`%>`' matches the empty string, but only if it is at the end of a word.

'`%w`' matches any word-constituent character (i.e., any letter or digit).

'`%W`' matches any character that is not a word constituent.

`str` **substitute** (`str` *template*, `list` *subs*) Function

Performs a standard set of substitutions on the string *template*, using the information contained in *subs*, returning the resulting, transformed *template*. *Subs* should be a list like those returned by `match()` or `rmatch()` when the match succeeds; otherwise, `E_INVARG` is raised.

In *template*, the strings '`%1`' through '`%9`' will be replaced by the text matched by the first through ninth parenthesized sub-patterns when `match()` or `rmatch()` was called. The string '`%0`' in *template* will be replaced by the text matched by the pattern as a whole when `match()` or `rmatch()` was called. The string '`%%`' will be replaced by a single '`%`' sign. If '`%`' appears in *template* followed by any other character, `E_INVARG` will be raised.

```
subs = match("*** Welcome to LambdaMOO!!!", "%(%w*%) to %(%w*%)");
substitute("I thank you for your %1 here in %2.", subs)
        ⇒   "I thank you for your Welcome here in LambdaMOO."
```

`str` **crypt** (`str` *text* [, `str` *salt*]) Function

Encrypts the given *text* using the standard UNIX encryption method. If provided, *salt* should be a string at least two characters long, the first two characters of which will be used as the extra encryption "salt" in the algorithm. If *salt* is not provided, a random pair of characters is used. In any case, the salt used is also returned as the first two characters of the resulting encrypted string.

Aside from the possibly-random selection of the salt, the encryption algorithm is entirely deterministic. In particular, you can test whether or not a given string is the same as the one used to produce a given piece of encrypted text; simply extract the first two characters of the encrypted text and pass the candidate string and those two characters to `crypt()`. If the result is identical to the given encrypted text, then you've got a match.

```
crypt("foobar")          ⇒   "J3fSFQfgkp26w"
crypt("foobar", "J3")    ⇒   "J3fSFQfgkp26w"
crypt("mumble", "J3")    ⇒   "J3D0.dh.jjmWQ"
crypt("foobar", "J4")    ⇒   "J4AcPxOJ4ncq2"
```

`str` **string_hash** (`str` *text*) Function
`str` **binary_hash** (`str` *bin-string*) Function

Returns a 32-character hexadecimal string encoding the result of applying the MD5 cryptographically secure hash function to the contents of the string *text* or the binary string *bin-string*. MD5, like other such functions, has the property that, if

```
string_hash(x) == string_hash(y)
```

then, almost certainly,

```
equals(x, y)
```

This can be useful, for example, in certain networking applications: after sending a large piece of text across a connection, also send the result of applying `string_hash()` to the text; if the destination site also applies `string_hash()` to the text and gets the same result, you can be quite confident that the large text has arrived unchanged.

4.4.2.4 Operations on Lists

int **length** (list *list*) Function

Returns the number of elements in *list*. It is also permissible to pass a string to `length()`; see the description in the previous section.

```
length({1, 2, 3})   ⇒   3
length({})          ⇒   0
```

int **is_member** (*value*, list *list*) Function

Returns true if there is an element of *list* that is completely indistinguishable from *value*. This is much the same operation as "*value* `in` *list*" except that, unlike `in`, the `is_member()` function does not treat upper- and lower-case characters in strings as equal.

```
"Foo" in {1, "foo", #24}            ⇒   2
is_member("Foo", {1, "foo", #24})   ⇒   0
is_member("Foo", {1, "Foo", #24})   ⇒   2
```

list **listinsert** (list *list*, *value* [, int *index*]) Function
list **listappend** (list *list*, *value* [, int *index*]) Function

These functions return a copy of *list* with *value* added as a new element. `listinsert()` and `listappend()` add *value* before and after (respectively) the existing element with the given *index*, if provided.

The following three expressions always have the same value:

```
listinsert(list, element, index)
listappend(list, element, index - 1)
{@list[1..index - 1], element, @list[index..length(list)]}
```

If *index* is not provided, then `listappend()` adds the *value* at the end of the list and `listinsert()` adds it at the beginning; this usage is discouraged, however, since the same intent can be more clearly expressed using the list-construction expression, as shown in the examples below.

```
x = {1, 2, 3};
listappend(x, 4, 2)   ⇒   {1, 2, 4, 3}
listinsert(x, 4, 2)   ⇒   {1, 4, 2, 3}
listappend(x, 4)      ⇒   {1, 2, 3, 4}
listinsert(x, 4)      ⇒   {4, 1, 2, 3}
{@x, 4}               ⇒   {1, 2, 3, 4}
{4, @x}               ⇒   {4, 1, 2, 3}
```

list **listdelete** (list *list*, int *index*) Function

Returns a copy of *list* with the *index*th element removed. If *index* is not in the range `[1..length(`*list*`)]`, then `E_RANGE` is raised.

```
x = {"foo", "bar", "baz"};
listdelete(x, 2)   ⇒   {"foo", "baz"}
```

list **listset** (list *list*, *value*, int *index*) Function

Returns a copy of *list* with the *index*th element replaced by *value*. If *index* is not in the range `[1..length(`*list*`)]`, then `E_RANGE` is raised.

```
x = {"foo", "bar", "baz"};
listset(x, "mumble", 2)   ⇒   {"foo", "mumble", "baz"}
```

This function exists primarily for historical reasons; it was used heavily before the server supported indexed assignments like `x[i] = v`. New code should always use indexed assignment instead of '`listset()`' wherever possible.

`list` **setadd** (`list` *list*, *value*) Function
`list` **setremove** (`list` *list*, *value*) Function

Returns a copy of *list* with the given *value* added or removed, as appropriate. `setadd()` only adds *value* if it is not already an element of *list*; *list* is thus treated as a mathematical set. *value* is added at the end of the resulting list, if at all. Similarly, `setremove()` returns a list identical to *list* if *value* is not an element. If *value* appears more than once in *list*, only the first occurrence is removed in the returned copy.

```
setadd({1, 2, 3}, 3)         ⇒   {1, 2, 3}
setadd({1, 2, 3}, 4)         ⇒   {1, 2, 3, 4}
setremove({1, 2, 3}, 3)      ⇒   {1, 2}
setremove({1, 2, 3}, 4)      ⇒   {1, 2, 3}
setremove({1, 2, 3, 2}, 2)   ⇒   {1, 3, 2}
```

4.4.3 Manipulating Objects

Objects are, of course, the main focus of most MOO programming and, largely due to that, there are a lot of built-in functions for manipulating them.

4.4.3.1 Fundamental Operations on Objects

`obj` **create** (`obj` *parent* [, `obj` *owner*]) Function

Creates and returns a new object whose parent is *parent* and whose owner is as described below. Either the given *parent* object must be `#-1` or valid and fertile (i.e., its '`f`' bit must be set) or else the programmer must own *parent* or be a wizard; otherwise `E_PERM` is raised. `E_PERM` is also raised if *owner* is provided and not the same as the programmer, unless the programmer is a wizard. After the new object is created, its `initialize` verb, if any, is called with no arguments.

The new object is assigned the least non-negative object number that has not yet been used for a created object. Note that no object number is ever reused, even if the object with that number is recycled.

The owner of the new object is either the programmer (if *owner* is not provided), the new object itself (if *owner* was given as `#-1`), or *owner* (otherwise).

The other built-in properties of the new object are initialized as follows:

```
name         ""
location     #-1
contents     {}
programmer   0
wizard       0
r            0
```

```
w            0
f            0
```

The function '`is_player()`' returns false for newly created objects.

In addition, the new object inherits all of the other properties on *parent*. These properties have the same permission bits as on *parent*. If the '`c`' permissions bit is set, then the owner of the property on the new object is the same as the owner of the new object itself; otherwise, the owner of the property on the new object is the same as that on *parent*. The initial value of every inherited property is *clear*; see the description of the built-in function `clear_property()` for details.

If the intended owner of the new object has a property named '`ownership_quota`' and the value of that property is an integer, then `create()` treats that value as a *quota*. If the quota is less than or equal to zero, then the quota is considered to be exhausted and `create()` raises `E_QUOTA` instead of creating an object. Otherwise, the quota is decremented and stored back into the '`ownership_quota`' property as a part of the creation of the new object.

`none` **chparent** (`obj` *object*, `obj` *new-parent*) Function

Changes the parent of *object* to be *new-parent*. If *object* is not valid, or if *new-parent* is neither valid nor equal to `#-1`, then `E_INVARG` is raised. If the programmer is neither a wizard or the owner of *object*, or if *new-parent* is not fertile (i.e., its '`f`' bit is not set) and the programmer is neither the owner of *new-parent* nor a wizard, then `E_PERM` is raised. If *new-parent* is equal to `object` or one of its current ancestors, `E_RECMOVE` is raised. If *object* or one of its descendants defines a property with the same name as one defined either on *new-parent* or on one of its ancestors, then `E_INVARG` is raised.

Changing an object's parent can have the effect of removing some properties from and adding some other properties to that object and all of its descendants (i.e., its children and its children's children, etc.). Let *common* be the nearest ancestor that *object* and *new-parent* have in common before the parent of *object* is changed. Then all properties defined by ancestors of *object* under *common* (that is, those ancestors of *object* that are in turn descendants of *common*) are removed from *object* and all of its descendants. All properties defined by *new-parent* or its ancestors under *common* are added to *object* and all of its descendants. As with `create()`, the newly-added properties are given the same permission bits as they have on *new-parent*, the owner of each added property is either the owner of the object it's added to (if the '`c`' permissions bit is set) or the owner of that property on *new-parent*, and the value of each added property is *clear*; see the description of the built-in function `clear_property()` for details. All properties that are not removed or added in the reparenting process are completely unchanged.

If *new-parent* is equal to `#-1`, then *object* is given no parent at all; it becomes a new root of the parent/child hierarchy. In this case, all formerly inherited properties on *object* are simply removed.

`int` **valid** (`obj` *object*) Function

Returns a non-zero integer (i.e., a true value) if *object* is a valid object (one that has been created and not yet recycled) and zero (i.e., a false value) otherwise.

```
valid(#0)    ⇒   1
valid(#-1)   ⇒   0
```

`obj` **parent** (`obj` *object*) Function
`list` **children** (`obj` *object*) Function

These functions return the parent and a list of the children of *object*, respectively. If *object* is not valid, then `E_INVARG` is raised.

`none` **recycle** (`obj` *object*) Function

The given *object* is destroyed, irrevocably. The programmer must either own *object* or be a wizard; otherwise, `E_PERM` is raised. If *object* is not valid, then `E_INVARG` is raised. The children of *object* are reparented to the parent of *object*. Before *object* is recycled, each object in its contents is moved to `#-1` (implying a call to *object*'s `exitfunc` verb, if any) and then *object*'s '`recycle`' verb, if any, is called with no arguments.

After *object* is recycled, if the owner of the former object has a property named '`ownership_quota`' and the value of that property is a integer, then `recycle()` treats that value as a *quota* and increments it by one, storing the result back into the '`ownership_quota`' property.

`int` **object_bytes** (`obj` *object*) Function

Returns the number of bytes of the server's memory required to store the given *object*, including the space used by the values of all of its non-clear properties and by the verbs and properties defined directly on the object. Raised `E_INVARG` if *object* is not a valid object and `E_PERM` if the programmer is not a wizard.

`obj` **max_object** () Function

Returns the largest object number yet assigned to a created object. Note that the object with this number may no longer exist; it may have been recycled. The next object created will be assigned the object number one larger than the value of `max_object()`.

4.4.3.2 Object Movement

`none` **move** (`obj` *what*, `obj` *where*) Function

Changes *what*'s location to be *where*. This is a complex process because a number of permissions checks and notifications must be performed. The actual movement takes place as described in the following paragraphs.

what should be a valid object and *where* should be either a valid object or `#-1` (denoting a location of 'nowhere'); otherwise `E_INVARG` is raised. The programmer must be either the owner of *what* or a wizard; otherwise, `E_PERM` is raised.

If *where* is a valid object, then the verb-call

    where:accept(*what*)

is performed before any movement takes place. If the verb returns a false value and the programmer is not a wizard, then *where* is considered to have refused entrance to *what*; `move()` raises `E_NACC`. If *where* does not define an `accept` verb, then it is treated as if it defined one that always returned false.

If moving *what* into *where* would create a loop in the containment hierarchy (i.e., *what* would contain itself, even indirectly), then `E_RECMOVE` is raised instead.

The '`location`' property of *what* is changed to be *where*, and the '`contents`' properties of the old and new locations are modified appropriately. Let *old-where* be the location of *what* before it was moved. If *old-where* is a valid object, then the verb-call

old-where`:exitfunc(`*what*`)`

is performed and its result is ignored; it is not an error if *old-where* does not define a verb named '`exitfunc`'. Finally, if *where* and *what* are still valid objects, and *where* is still the location of *what*, then the verb-call

where`:enterfunc(`*what*`)`

is performed and its result is ignored; again, it is not an error if *where* does not define a verb named '`enterfunc`'.

4.4.3.3 Operations on Properties

`list` **properties** (`obj` *object*) Function

Returns a list of the names of the properties defined directly on the given *object*, not inherited from its parent. If *object* is not valid, then `E_INVARG` is raised. If the programmer does not have read permission on *object*, then `E_PERM` is raised.

`list` **property_info** (`obj` *object*, `str` *prop-name*) Function
`none` **set_property_info** (`obj` *object*, `str` *prop-name*, `list` *info*) Function

These two functions get and set (respectively) the owner and permission bits for the property named *prop-name* on the given *object*. If *object* is not valid, then `E_INVARG` is raised. If *object* has no non-built-in property named *prop-name*, then `E_PROPNF` is raised. If the programmer does not have read (write) permission on the property in question, then `property_info()` (`set_property_info()`) raises `E_PERM`. Property info has the following form:

`{`*owner*`,` *perms* `[,` *new-name*`]}`

where *owner* is an object, *perms* is a string containing only characters from the set '`r`', '`w`', and '`c`', and *new-name* is a string; *new-name* is never part of the value returned by `property_info()`, but it may optionally be given as part of the value provided to `set_property_info()`. This list is the kind of value returned by `property_info()` and expected as the third argument to `set_property_info()`; the latter function raises `E_INVARG` if *owner* is not valid, if *perms* contains any illegal characters, or, when *new-name* is given, if *prop-name* is not defined directly on *object* or *new-name* names an existing property defined on *object* or any of its ancestors or descendants.

`none` **add_property** (`obj` *object*, `str` *prop-name*, *value*, `list` *info*) Function

Defines a new property on the given *object*, inherited by all of its descendants; the property is named *prop-name*, its initial value is *value*, and its owner and initial permission bits are given by *info* in the same format as is returned by `property_info()`, described above. If *object* is not valid or *info* does not specify a valid owner and well-formed permission bits or *object* or its ancestors or descendants already defines a property named *prop-name*, then `E_INVARG` is raised. If the programmer does not have write permission on *object* or if the owner specified by *info* is not the programmer and the programmer is not a wizard, then `E_PERM` is raised.

none **delete_property** (obj *object*, str *prop-name*) Function

Removes the property named *prop-name* from the given *object* and all of its descendants. If *object* is not valid, then E_INVARG is raised. If the programmer does not have write permission on *object*, then E_PERM is raised. If *object* does not directly define a property named *prop-name* (as opposed to inheriting one from its parent), then E_PROPNF is raised.

int **is_clear_property** (obj *object*, str *prop-name*) Function
none **clear_property** (obj *object*, str *prop-name*) Function

These two functions test for clear and set to clear, respectively, the property named *prop-name* on the given *object*. If *object* is not valid, then E_INVARG is raised. If *object* has no non-built-in property named *prop-name*, then E_PROPNF is raised. If the programmer does not have read (write) permission on the property in question, then is_clear_property() (clear_property()) raises E_PERM. If a property is clear, then when the value of that property is queried the value of the parent's property of the same name is returned. If the parent's property is clear, then the parent's parent's value is examined, and so on. If *object* is the definer of the property *prop-name*, as opposed to an inheritor of the property, then clear_property() raises E_INVARG.

4.4.3.4 Operations on Verbs

list **verbs** (obj *object*) Function

Returns a list of the names of the verbs defined directly on the given *object*, not inherited from its parent. If *object* is not valid, then E_INVARG is raised. If the programmer does not have read permission on *object*, then E_PERM is raised.

Most of the remaining operations on verbs accept a string containing the verb's name to identify the verb in question. Because verbs can have multiple names and because an object can have multiple verbs with the same name, this practice can lead to difficulties. To most unambiguously refer to a particular verb, one can instead use a positive integer, the index of the verb in the list returned by verbs(), described above.

For example, suppose that verbs(#34) returns this list:

```
{"foo", "bar", "baz", "foo"}
```

Object #34 has two verbs named 'foo' defined on it (this may not be an error, if the two verbs have different command syntaxes). To refer unambiguously to the first one in the list, one uses the integer 1; to refer to the other one, one uses 4.

In the function descriptions below, an argument named *verb-desc* is either a string containing the name of a verb or else a positive integer giving the index of that verb in its defining object's verbs() list.

> For historical reasons, there is also a second, inferior mechanism for referring to verbs with numbers, but its use is strongly discouraged. If the property $server_options.support_numeric_verbname_strings exists with a true value, then functions on verbs will also accept a numeric string (e.g., "4") as a verb descriptor. The decimal integer in the string works more-or-less like the positive integers described above, but with two significant differences:

1. The numeric string is a *zero-based* index into `verbs()`; that is, in the string case, you would use the number one less than what you would use in the positive integer case.
2. When there exists a verb whose actual name looks like a decimal integer, this numeric-string notation is ambiguous; the server will in all cases assume that the reference is to the first verb in the list for which the given string could be a name, either in the normal sense or as a numeric index.

Clearly, this older mechanism is more difficult and risky to use; new code should only be written to use the current mechanism, and old code using numeric strings should be modified not to do so.

`list` **verb_info** (`obj` *object*, `str` *verb-desc*) Function
`none` **set_verb_info** (`obj` *object*, `str` *verb-desc*, `list` *info*) Function

These two functions get and set (respectively) the owner, permission bits, and name(s) for the verb as specified by *verb-desc* on the given *object*. If *object* is not valid, then `E_INVARG` is raised. If *object* does not define a verb as specified by *verb-desc*, then `E_VERBNF` is raised. If the programmer does not have read (write) permission on the verb in question, then `verb_info()` (`set_verb_info()`) raises `E_PERM`. Verb info has the following form:

`{`*owner*`,` *perms*`,` *names*`}`

where *owner* is an object, *perms* is a string containing only characters from the set '`r`', '`w`', '`x`', and '`d`', and *names* is a string. This is the kind of value returned by `verb_info()` and expected as the third argument to `set_verb_info()`. `set_verb_info()` raises `E_INVARG` if *owner* is not valid, if *perms* contains any illegal characters, or if *names* is the empty string or consists entirely of spaces; it raises `E_PERM` if *owner* is not the programmer and the programmer is not a wizard.

`list` **verb_args** (`obj` *object*, `str` *verb-desc*) Function
`none` **set_verb_args** (`obj` *object*, `str` *verb-desc*, `list` *args*) Function

These two functions get and set (respectively) the direct-object, preposition, and indirect-object specifications for the verb as specified by *verb-desc* on the given *object*. If *object* is not valid, then `E_INVARG` is raised. If *object* does not define a verb as specified by *verb-desc*, then `E_VERBNF` is raised. If the programmer does not have read (write) permission on the verb in question, then `verb_args()` (`set_verb_args()`) raises `E_PERM`. Verb args specifications have the following form:

`{`*dobj*`,` *prep*`,` *iobj*`}`

where *dobj* and *iobj* are strings drawn from the set `"this"`, `"none"`, and `"any"`, and *prep* is a string that is either `"none"`, `"any"`, or one of the prepositional phrases listed much earlier in the description of verbs in the first chapter. This is the kind of value returned by `verb_args()` and expected as the third argument to `set_verb_args()`. Note that for `set_verb_args()`, *prep* must be only one of the prepositional phrases, not (as is shown in that table) a set of such phrases separated by '`/`' characters. `set_verb_args` raises `E_INVARG` if any of the *dobj*, *prep*, or *iobj* strings is illegal.

```
verb_args($container, "take")
                      ⇒   {"any", "out of/from inside/from", "this"}
set_verb_args($container, "take", {"any", "from", "this"})
```

`none` **add_verb** (`obj` *object*, `list` *info*, `list` *args*) Function

Defines a new verb on the given *object*. The new verb's owner, permission bits and name(s) are given by *info* in the same format as is returned by `verb_info()`, described above. The new verb's direct-object, preposition, and indirect-object specifications are given by *args* in the same format as is returned by `verb_args`, described above. The new verb initially has the empty program associated with it; this program does nothing but return an unspecified value.

If *object* is not valid, or *info* does not specify a valid owner and well-formed permission bits and verb names, or *args* is not a legitimate syntax specification, then `E_INVARG` is raised. If the programmer does not have write permission on *object* or if the owner specified by *info* is not the programmer and the programmer is not a wizard, then `E_PERM` is raised.

`none` **delete_verb** (`obj` *object*, `str` *verb-desc*) Function

Removes the verb as specified by *verb-desc* from the given *object*. If *object* is not valid, then `E_INVARG` is raised. If the programmer does not have write permission on *object*, then `E_PERM` is raised. If *object* does not define a verb as specified by *verb-desc*, then `E_VERBNF` is raised.

`list` **verb_code** (`obj` *object*, `str` *verb-desc* [, *fully-paren* [, *indent*]]) Function

`list` **set_verb_code** (`obj` *object*, `str` *verb-desc*, `list` *code*) Function

These functions get and set (respectively) the MOO-code program associated with the verb as specified by *verb-desc* on *object*. The program is represented as a list of strings, one for each line of the program; this is the kind of value returned by `verb_code()` and expected as the third argument to `set_verb_code()`. For `verb_code()`, the expressions in the returned code are usually written with the minimum-necessary parenthesization; if *full-paren* is true, then all expressions are fully parenthesized. Also for `verb_code()`, the lines in the returned code are usually not indented at all; if *indent* is true, each line is indented to better show the nesting of statements.

If *object* is not valid, then `E_INVARG` is raised. If *object* does not define a verb as specified by *verb-desc*, then `E_VERBNF` is raised. If the programmer does not have read (write) permission on the verb in question, then `verb_code()` (`set_verb_code()`) raises `E_PERM`. If the programmer is not, in fact. a programmer, then `E_PERM` is raised.

For `set_verb_code()`, the result is a list of strings, the error messages generated by the MOO-code compiler during processing of *code*. If the list is non-empty, then `set_verb_code()` did not install *code*; the program associated with the verb in question is unchanged.

`list` **disassemble** (`obj` *object*, `str` *verb-desc*) Function

Returns a (longish) list of strings giving a listing of the server's internal "compiled" form of the verb as specified by *verb-desc* on *object*. This format is not documented and may indeed change from release to release, but some programmers may nonetheless find the output of `disassemble()` interesting to peruse as a way to gain a deeper appreciation of how the server works.

If *object* is not valid, then `E_INVARG` is raised. If *object* does not define a verb as specified by *verb-desc*, then `E_VERBNF` is raised. If the programmer does not have read permission on the verb in question, then `disassemble()` raises `E_PERM`.

4.4.3.5 Operations on Player Objects

`list` **players** () Function

Returns a list of the object numbers of all player objects in the database.

`int` **is_player** (`obj` *object*) Function

Returns a true value if the given *object* is a player object and a false value otherwise. If *object* is not valid, `E_INVARG` is raised.

`none` **set_player_flag** (`obj` *object*, *value*) Function

Confers or removes the "player object" status of the given *object*, depending upon the truth value of *value*. If *object* is not valid, `E_INVARG` is raised. If the programmer is not a wizard, then `E_PERM` is raised.

If *value* is true, then *object* gains (or keeps) "player object" status: it will be an element of the list returned by `players()`, the expression `is_player(`*object*`)` will return true, and the server will treat a call to `$do_login_command()` that returns *object* as logging in the current connection.

If *value* is false, the *object* loses (or continues to lack) "player object" status: it will not be an element of the list returned by `players()`, the expression `is_player(`*object*`)` will return false, and users cannot connect to *object* by name when they log into the server. In addition, if a user is connected to *object* at the time that it loses "player object" status, then that connection is immediately broken, just as if `boot_player(`*object*`)` had been called (see the description of `boot_player()` below).

4.4.4 Operations on Network Connections

`list` **connected_players** ([*include-all*]) Function

Returns a list of the object numbers of those player objects with currently-active connections. If *include-all* is provided and true, then the list includes the object numbers associated with *all* current connections, including ones that are outbound and/or not yet logged-in.

`int` **connected_seconds** (`obj` *player*) Function
`int` **idle_seconds** (`obj` *player*) Function

These functions return the number of seconds that the currently-active connection to *player* has existed and been idle, respectively. If *player* is not the object number of a player object with a currently-active connection, then `E_INVARG` is raised.

`none` **notify** (`obj` *conn*, `str` *string* [, *no-flush*]) Function

Enqueues *string* for output (on a line by itself) on the connection *conn*. If the programmer is not *conn* or a wizard, then `E_PERM` is raised. If *conn* is not a currently-active connection, then this function does nothing. Output is normally written to connections only between tasks, not during execution.

The server will not queue an arbitrary amount of output for a connection; the `MAX_QUEUED_OUTPUT` compilation option (in '`options.h`') controls the limit. When an attempt is made to enqueue output that would take the server over its limit, it first tries to write as much output as possible to the connection without having to wait for the other end. If that doesn't result in the new output being able to fit in the queue, the server starts throwing away the oldest lines in the queue until the new ouput will fit. The server remembers how many lines of output it has 'flushed' in this way and, when next it can succeed in writing anything to the connection, it first writes a line like `>> Network buffer overflow:` *X* `lines of output to you have been lost <<` where *X* is the number of flushed lines.

If *no-flush* is provided and true, then `notify()` never flushes any output from the queue; instead it immediately returns false. `Notify()` otherwise always returns true.

`int` **buffered_output_length** ([`obj` *conn*]) Function

Returns the number of bytes currently buffered for output to the connection *conn*. If *conn* is not provided, returns the maximum number of bytes that will be buffered up for output on any connection.

`str` **read** ([`obj` *conn* [, *non-blocking*]]) Function

Reads and returns a line of input from the connection *conn* (or, if not provided, from the player that typed the command that initiated the current task). If *non-blocking* is false or not provided, this function suspends the current task, resuming it when there is input available to be read. If *non-blocking* is provided and true, this function never suspends the calling task; if there is no input currently available for input, `read()` simply returns 0 immediately.

If *player* is provided, then the programmer must either be a wizard or the owner of `player`; if `player` is not provided, then `read()` may only be called by a wizard and only in the task that was last spawned by a command from the connection in question. Otherwise, `E_PERM` is raised. If the given `player` is not currently connected and has no pending lines of input, or if the connection is closed while a task is waiting for input but before any lines of input are received, then `read()` raises `E_INVARG`.

The restriction on the use of `read()` without any arguments preserves the following simple invariant: if input is being read from a player, it is for the task started by the last command that player typed. This invariant adds responsibility to the programmer, however. If your program calls another verb before doing a `read()`, then either that verb must not suspend or else you must arrange that no commands will be read from the connection in the meantime. The most straightforward way to do this is to call

```
set_connection_option(player, "hold-input", 1)
```

before any task suspension could happen, then make all of your calls to `read()` and other code that might suspend, and finally call

```
set_connection_option(player, "hold-input", 0)
```

to allow commands once again to be read and interpreted normally.

`none` **force_input** (`obj` *conn*, `str` *line* [, *at-front*]) Function

Inserts the string *line* as an input task in the queue for the connection *conn*, just as if it had arrived as input over the network. If *at_front* is provided and true, then the new line of input is put at the front of *conn*'s queue, so that it will be the very next line of input processed even if there is already some other input in that queue. Raises `E_INVARG` if *conn* does not specify a current connection and `E_PERM` if the programmer is neither *conn* nor a wizard.

none **flush_input** (obj *conn* [*show-messages*]) Function

Performs the same actions as if the connection *conn*'s defined flush command had been received on that connection, i.e., removes all pending lines of input from *conn*'s queue and, if *show-messages* is provided and true, prints a message to *conn* listing the flushed lines, if any. See the chapter on server assumptions about the database for more information about a connection's defined flush command.

list **output_delimiters** (obj *player*) Function

Returns a list of two strings, the current *output prefix* and *output suffix* for *player*. If *player* does not have an active network connection, then `E_INVARG` is raised. If either string is currently undefined, the value `""` is used instead. See the discussion of the `PREFIX` and `SUFFIX` commands in the next chapter for more information about the output prefix and suffix.

none **boot_player** (obj *player*) Function

Marks for disconnection any currently-active connection to the given *player*. The connection will not actually be closed until the currently-running task returns or suspends, but all MOO functions (such as `notify()`, `connected_players()`, and the like) immediately behave as if the connection no longer exists. If the programmer is not either a wizard or the same as *player*, then `E_PERM` is raised. If there is no currently-active connection to *player*, then this function does nothing.

If there was a currently-active connection, then the following verb call is made when the connection is actually closed:

`$user_disconnected(`*player*`)`

It is not an error if this verb does not exist; the call is simply skipped.

str **connection_name** (obj *player*) Function

Returns a network-specific string identifying the connection being used by the given player. If the programmer is not a wizard and not *player*, then `E_PERM` is raised. If *player* is not currently connected, then `E_INVARG` is raised.

For the TCP/IP networking configurations, for in-bound connections, the string has the form

`"port `*lport*` from `*host*`, port `*port*`"`

where *lport* is the decimal TCP listening port on which the connection arrived, *host* is either the name or decimal TCP address of the host from which the player is connected, and *port* is the decimal TCP port of the connection on that host.

For outbound TCP/IP connections, the string has the form

`"port `*lport*` to `*host*`, port `*port*`"`

where *lport* is the decimal local TCP port number from which the connection originated, *host* is either the name or decimal TCP address of the host to which the connection was opened, and *port* is the decimal TCP port of the connection on that host.

For the System V 'local' networking configuration, the string is the UNIX login name of the connecting user or, if no such name can be found, something of the form

`"User #`*number*`"`

where *number* is a UNIX numeric user ID.

For the other networking configurations, the string is the same for all connections and, thus, useless.

`none` **set_connection_option** (`obj` *conn*, `str` *option*, *value*) Function

Controls a number of optional behaviors associated the connection *conn*. Raises `E_INVARG` if *conn* does not specify a current connection and `E_PERM` if the programmer is neither *conn* nor a wizard. The following values for *option* are currently supported:

`"hold-input"`
: If *value* is true, then input received on *conn* will never be treated as a command; instead, it will remain in the queue until retrieved by a call to `read()`.

`"client-echo"`
: Send the Telnet Protocol '`WONT ECHO`' or '`WILL ECHO`' command, depending on whether *value* is true or false, respectively. For clients that support the Telnet Protocol, this should toggle whether or not the client echoes locally the characters typed by the user. Note that the server itself never echoes input characters under any circumstances. (This option is only available under the TCP/IP networking configurations.)

`"binary"`
: If *value* is true, then both input from and output to *conn* can contain arbitrary bytes. Input from a connection in binary mode is not broken into lines at all; it is delivered to either the read() function or the built-in command parser as *binary strings*, in whatever size chunks come back from the operating system. (See the early section on MOO value types for a description of the binary string representation.) For output to a connection in binary mode, the second argument to 'notify()' must be a binary string; if it is malformed, E_INVARG is raised.

`"flush-command"`
: If *value* is a non-empty string, then it becomes the new *flush* command for this connection, by which the player can flush all queued input that has not yet been processed by the server. If *value* is not a non-empty string, then *conn* is set to have no flush command at all. The default value of this option can be set via the property `$server_options.default_flush_command`; see the chapter on server assumptions about the database for details.

`list` **connection_options** (`obj` *conn*) Function

Returns a list of `{`*name*`,` *value*`}` pairs describing the current settings of all of the allowed options for the connection *conn*. Raises `E_INVARG` if *conn* does not specify a current connection and `E_PERM` if the programmer is neither *conn* nor a wizard.

`value` **connection_option** (`obj` *conn*, `str` *name*) Function

Returns the current setting of the option *name* for the connection *conn*. Raises `E_INVARG` if *conn* does not specify a current connection and `E_PERM` if the programmer is neither *conn* nor a wizard.

`obj` **open_network_connection** (*value*, ...) Function

Establishes a network connection to the place specified by the arguments and more-or-less pretends that a new, normal player connection has been established from there. The new connection, as usual, will not be logged in initially and will have a negative object number associated with it for use with `read()`, `notify()`, and `boot_player()`. This object number is the value returned by this function.

If the programmer is not a wizard or if the `OUTBOUND_NETWORK` compilation option was not used in building the server, then `E_PERM` is raised. If the network connection cannot be made for some reason, then other errors will be returned, depending upon the particular network implementation in use.

For the TCP/IP network implementations (the only ones as of this writing that support outbound connections), there must be two arguments, a string naming a host (possibly using the numeric Internet syntax) and an integer specifying a TCP port. If a connection cannot be made because the host does not exist, the port does not exist, the host is not reachable or refused the connection, `E_INVARG` is raised. If the connection cannot be made for other reasons, including resource limitations, then `E_QUOTA` is raised.

The outbound connection process involves certain steps that can take quite a long time, during which the server is not doing anything else, including responding to user commands and executing MOO tasks. See the chapter on server assumptions about the database for details about how the server limits the amount of time it will wait for these steps to successfully complete.

It is worth mentioning one tricky point concerning the use of this function. Since the server treats the new connection pretty much like any normal player connection, it will naturally try to parse any input from that connection as commands in the usual way. To prevent this treatment, you should use `set_connection_option()` to set the `"hold-input"` option true on the connection.

`value` **listen** (`obj` *object*, *point* [, *print-messages*]) Function

Create a new point at which the server will listen for network connections, just as it does normally. *Object* is the object whose verbs `do_login_command`, `do_command`, `do_out_of_band_command`, `user_connected`, `user_created`, `user_reconnected`, `user_disconnected`, and `user_client_disconnected` will be called at appropriate points, just as these verbs are called on `#0` for normal connections. (See the chapter on server assumptions about the database for the complete story on when these functions are called.) *Point* is a network-configuration-specific parameter describing the listening point. If *print-messages* is provided and true, then the various database-configurable messages (also detailed in the chapter on server assumptions) will be printed on connections received at the new listening point. `Listen()` returns *canon*, a 'canonicalized' version of *point*, with any configuration-specific defaulting or aliasing accounted for.

This raises `E_PERM` if the programmer is not a wizard, `E_INVARG` if *object* is invalid or there is already a listening point described by *point*, and `E_QUOTA` if some network-configuration-specific error occurred.

For the TCP/IP configurations, *point* is a TCP port number on which to listen and *canon* is equal to *point* unless *point* is zero, in which case *canon* is a port number assigned by the operating system.

For the local multi-user configurations, *point* is the UNIX file name to be used as the connection point and *canon* is always equal to *point*.

In the single-user configuration, the can be only one listening point at a time; *point* can be any value at all and *canon* is always zero.

`none` **unlisten** (*canon*) Function

Stop listening for connections on the point described by *canon*, which should be the second element of some element of the list returned by `listeners()`. Raises `E_PERM` if the programmer is not a wizard and `E_INVARG` if there does not exist a listener with that description.

`list` **listeners** () Function

Returns a list describing all existing listening points, including the default one set up automatically by the server when it was started (unless that one has since been destroyed by a call to `unlisten()`). Each element of the list has the following form:

{*object*, *canon*, *print-messages*}

where *object* is the first argument given in the call to `listen()` to create this listening point, *print-messages* is true if the third argument in that call was provided and true, and *canon* was the value returned by that call. (For the initial listening point, *object* is `#0`, *canon* is determined by the command-line arguments or a network-configuration-specific default, and *print-messages* is true.)

Please note that there is nothing special about the initial listening point created by the server when it starts; you can use `unlisten()` on it just as if it had been created by `listen()`. This can be useful; for example, under one of the TCP/IP configurations, you might start up your server on some obscure port, say 12345, connect to it by yourself for a while, and then open it up to normal users by evaluating the statments

```
unlisten(12345); listen(#0, 7777, 1)
```

4.4.5 Operations Involving Times and Dates

`int` **time** () Function

Returns the current time, represented as the number of seconds that have elapsed since midnight on 1 January 1970, Greenwich Mean Time.

`str` **ctime** ([`int` *time*]) Function

Interprets *time* as a time, using the same representation as given in the description of `time()`, above, and converts it into a 28-character, human-readable string in the following format:

```
Mon Aug 13 19:13:20 1990 PDT
```

If the current day of the month is less than 10, then an extra blank appears between the month and the day:

```
Mon Apr  1 14:10:43 1991 PST
```

If *time* is not provided, then the current time is used.

Note that `ctime()` interprets *time* for the local time zone of the computer on which the MOO server is running.

4.4.6 MOO-Code Evaluation and Task Manipulation

`none` **raise** (*code* [, `str` *message* [, *value*]]) Function

Raises *code* as an error in the same way as other MOO expressions, statements, and functions do. *Message*, which defaults to the value of `tostr(`*code*`)`, and *value*, which defaults to zero, are made available to any `try-except` statements that catch the error. If the error is not caught, then *message* will appear on the first line of the traceback printed to the user.

`value` **call_function** (`str` *func-name*, *arg*, ...) Function

Calls the built-in function named *func-name*, passing the given arguments, and returns whatever that function returns. Raises `E_INVARG` if *func-name* is not recognized as the name of a known built-in function. This allows you to compute the name of the function to call and, in particular, allows you to write a call to a built-in function that may or may not exist in the particular version of the server you're using.

`list` **function_info** ([`str` *name*]) Function

Returns descriptions of the built-in functions available on the server. If *name* is provided, only the description of the function with that name is returned. If *name* is omitted, a list of descriptions is returned, one for each function available on the server. Raised `E_INVARG` if *name* is provided but no function with that name is available on the server.

Each function description is a list of the following form:

{*name*, *min-args*, *max-args*, *types*

where *name* is the name of the built-in function, *min-args* is the minimum number of arguments that must be provided to the function, *max-args* is the maximum number of arguments that can be provided to the function or `-1` if there is no maximum, and *types* is a list of *max-args* integers (or *min-args* if *max-args* is `-1`), each of which represents the type of argument required in the corresponding position. Each type number is as would be returned from the `typeof()` built-in function except that `-1` indicates that any type of value is acceptable and `-2` indicates that either integers or floating-point numbers may be given. For example, here are several entries from the list:

```
{"listdelete", 2, 2, {4, 0}}
{"suspend", 0, 1, {0}}
{"server_log", 1, 2, {2, -1}}
{"max", 1, -1, {-2}}
{"tostr", 0, -1, {}}
```

`Listdelete()` takes exactly 2 arguments, of which the first must be a list (`LIST == 4`) and the second must be an integer (`INT == 0`). `Suspend()` has one optional argument that, if provided, must be an integer. `Server_log()` has one required argument that must be a string (`STR == 2`) and one optional argument that, if provided, may be of any type. `Max()` requires at least one argument but can take any number above that, and the first argument must be either an integer or a floating-point number; the type(s) required for any other arguments can't be determined from this description. Finally, `tostr()` takes any number of arguments at all, but it can't be determined from this description which argument types would be acceptable in which positions.

`list` **eval** (`str` *string*) Function

The MOO-code compiler processes *string* as if it were to be the program associated with some verb and, if no errors are found, that fictional verb is invoked. If the programmer is not, in fact, a programmer, then `E_PERM` is raised. The normal result of calling `eval()` is a two element list. The first element is true if there were no compilation errors and false otherwise. The second element is either the result returned from the fictional verb (if there were no compilation errors) or a list of the compiler's error messages (otherwise).

When the fictional verb is invoked, the various built-in variables have values as shown below:

`player`	the same as in the calling verb
`this`	`#-1`
`caller`	the same as the initial value of `this` in the calling verb
`args`	`{}`
`argstr`	`""`
`verb`	`""`
`dobjstr`	`""`
`dobj`	`#-1`
`prepstr`	`""`
`iobjstr`	`""`
`iobj`	`#-1`

The fictional verb runs with the permissions of the programmer and as if its '`d`' permissions bit were on.

```
eval("return 3 + 4;")   ⇒   {1, 7}
```

`none` **set_task_perms** (`obj` *who*) Function

Changes the permissions with which the currently-executing verb is running to be those of *who*. If the programmer is neither *who* nor a wizard, then `E_PERM` is raised.

> **Note**: This does not change the owner of the currently-running verb, only the permissions of this particular invocation. It is used in verbs owned by wizards to make themselves run with lesser (usually non-wizard) permissions.

`obj` **caller_perms** () Function

Returns the permissions in use by the verb that called the currently-executing verb. If the currently-executing verb was not called by another verb (i.e., it is the first verb called in a command or server task), then `caller_perms()` returns `#-1`.

`int` **ticks_left** () Function

`int` **seconds_left** () Function

These two functions return the number of ticks or seconds (respectively) left to the current task before it will be forcibly terminated. These are useful, for example, in deciding when to call '`suspend()`' to continue a long-lived computation.

`int` **task_id** () Function

Returns the non-zero, non-negative integer identifier for the currently-executing task. Such integers are randomly selected for each task and can therefore safely be used in circumstances where unpredictability is required.

`value` **suspend** ([`int` *seconds*]) Function

Suspends the current task, and resumes it after at least *seconds* seconds. (If *seconds* is not provided, the task is suspended indefinitely; such a task can only be resumed by use of the `resume()` function.) When the task is resumed, it will have a full quota of ticks and seconds. This function is useful for programs that run for a long time or require a lot of ticks. If *seconds* is negative, then `E_INVARG` is raised. `Suspend()` returns zero unless it was resumed via `resume()`, in which case it returns the second argument given to that function.

In some sense, this function forks the 'rest' of the executing task. However, there is a major difference between the use of '`suspend(`*seconds*`)`' and the use of the '`fork (`*seconds*`)`'. The '`fork`' statement creates a new task (a *forked task*) while the currently-running task still goes on to completion, but a `suspend()` suspends the currently-running task (thus making it into a *suspended task*). This difference may be best explained by the following examples, in which one verb calls another:

```
.program   #0:caller_A
#0.prop = 1;
#0:callee_A();
#0.prop = 2;
.

.program   #0:callee_A
fork(5)
  #0.prop = 3;
endfork
.

.program   #0:caller_B
#0.prop = 1;
#0:callee_B();
#0.prop = 2;
.

.program   #0:callee_B
suspend(5);
#0.prop = 3;
.
```

Consider `#0:caller_A`, which calls `#0:callee_A`. Such a task would assign 1 to `#0.prop`, call `#0:callee_A`, fork a new task, return to `#0:caller_A`, and assign 2 to `#0.prop`, ending this task. Five seconds later, if the forked task had not been killed, then it would begin to run; it would assign 3 to `#0.prop` and then stop. So, the final value of `#0.prop` (i.e., the value after more than 5 seconds) would be 3.

Now consider `#0:caller_B`, which calls `#0:callee_B` instead of `#0:callee_A`. This task would assign 1 to `#0.prop`, call `#0:callee_B`, and suspend. Five seconds later, if the suspended task had not been killed, then it would resume; it would assign 3 to `#0.prop`, return to `#0:caller_B`, and assign 2 to `#0.prop`, ending the task. So, the final value of `#0.prop` (i.e., the value after more than 5 seconds) would be 2.

A suspended task, like a forked task, can be described by the `queued_tasks()` function and killed by the `kill_task()` function. Suspending a task does not change its task id. A task can be suspended again and again by successive calls to `suspend()`.

By default, there is no limit to the number of tasks any player may suspend, but such a limit can be imposed from within the database. See the chapter on server assumptions about the database for details.

`none` **resume** (`int` *task-id* [, *value*]) Function

Immediately ends the suspension of the suspended task with the given *task-id*; that task's call to `suspend()` will return *value*, which defaults to zero. `Resume()` raises `E_INVARG` if *task-id* does not specify an existing suspended task and `E_PERM` if the programmer is neither a wizard nor the owner of the specified task.

`list` **queue_info** ([`obj` *player*]) Function

If *player* is omitted, returns a list of object numbers naming all players that currently have active task queues inside the server. If *player* is provided, returns the number of background tasks currently queued for that user. It is guaranteed that `queue_info(`*X*`)` will return zero for any *X* not in the result of `queue_info()`.

`list` **queued_tasks** () Function

Returns information on each of the background tasks (i.e., forked, suspended or reading) owned by the programmer (or, if the programmer is a wizard, all queued tasks). The returned value is a list of lists, each of which encodes certain information about a particular queued task in the following format:

```
{task-id, start-time, x, y,
 programmer, verb-loc, verb-name, line, this}
```

where *task-id* is an integer identifier for this queued task, *start-time* is the time after which this task will begin execution (in `time()` format), *x* and *y* are obsolete values that are no longer interesting, *programmer* is the permissions with which this task will begin execution (and also the player who *owns* this task), *verb-loc* is the object on which the verb that forked this task was defined at the time, *verb-name* is that name of that verb, *line* is the number of the first line of the code in that verb that this task will execute, and *this* is the value of the variable '`this`' in that verb. For reading tasks, *start-time* is `-1`.

The *x* and *y* fields are now obsolete and are retained only for backward-compatibility reasons. They may be reused for new purposes in some future version of the server.

`none` **kill_task** (`int` *task-id*) Function

Removes the task with the given *task-id* from the queue of waiting tasks. If the programmer is not the owner of that task and not a wizard, then `E_PERM` is raised. If there is no task on the queue with the given *task-id*, then `E_INVARG` is raised.

`list` **callers** ([*include-line-numbers*]) Function

Returns information on each of the verbs and built-in functions currently waiting to resume execution in the current task. When one verb or function calls another verb or function, execution of the caller is temporarily suspended, pending the called verb or function returning a value. At any given time, there could be several such pending verbs and functions: the one that called the currently executing verb, the verb or function that called that one, and so on. The result of `callers()` is a list, each element of which gives information about one pending verb or function in the following format:

```
{this, verb-name, programmer, verb-loc, player, line-number}
```

For verbs, *this* is the initial value of the variable '`this`' in that verb, *verb-name* is the name used to invoke that verb, *programmer* is the player with whose permissions that verb is running, *verb-loc* is the object on which that verb is defined, *player* is the initial value of the variable '`player`' in that verb, and *line-number* indicates which line of the verb's code is executing. The *line-number* element is included only if the *include-line-numbers* argument was provided and true.

For functions, *this*, *programmer*, and *verb-loc* are all `#-1`, *verb-name* is the name of the function, and *line-number* is an index used internally to determine the current state of the built-in function. The simplest correct test for a built-in function entry is

```
(VERB-LOC == #-1  &&  PROGRAMMER == #-1  &&  VERB-NAME != "")
```

The first element of the list returned by `callers()` gives information on the verb that called the currently-executing verb, the second element describes the verb that called that one, and so on. The last element of the list describes the first verb called in this task.

`list` **task_stack** (`int` *task-id* [, *include-line-numbers*]) Function

Returns information like that returned by the `callers()` function, but for the suspended task with the given *task-id*; the *include-line-numbers* argument has the same meaning as in `callers()`. Raises `E_INVARG` if *task-id* does not specify an existing suspended task and `E_PERM` if the programmer is neither a wizard nor the owner of the specified task.

4.4.7 Administrative Operations

`str` **server_version** () Function

Returns a string giving the version number of the running MOO server.

`none` **server_log** (`str` *message* [, *is-error*]) Function

The text in *message* is sent to the server log with a distinctive prefix (so that it can be distinguished from server-generated messages). If the programmer is not a wizard, then `E_PERM` is raised. If *is-error* is provided and true, then *message* is marked in the server log as an error.

`obj` **renumber** (`obj` *object*) Function

The object number of the object currently numbered *object* is changed to be the least nonnegative object number not currently in use and the new object number is returned. If *object* is not valid, then `E_INVARG` is raised. If the programmer is not a wizard, then `E_PERM` is raised. If there are no unused nonnegative object numbers less than *object*, then *object* is returned and no changes take place.

The references to *object* in the parent/children and location/contents hierarchies are updated to use the new object number, and any verbs, properties and/or objects owned by *object* are also changed to be owned by the new object number. The latter operation can be quite time consuming if the database is large. No other changes to the database are performed; in particular, no object references in property values or verb code are updated.

This operation is intended for use in making new versions of the LambdaCore database from the then-current LambdaMOO database, and other similar situations. Its use requires great care.

`none` **reset_max_object** () Function

The server's idea of the highest object number ever used is changed to be the highest object number of a currently-existing object, thus allowing reuse of any higher numbers that refer to now-recycled objects. If the programmer is not a wizard, then `E_PERM` is raised.

This operation is intended for use in making new versions of the LambdaCore database from the then-current LambdaMOO database, and other similar situations. Its use requires great care.

`list` **memory_usage** () Function

On some versions of the server, this returns statistics concerning the server consumption of system memory. The result is a list of lists, each in the following format:

`{`*block-size*`,` *nused*`,` *nfree*`}`

where *block-size* is the size in bytes of a particular class of memory fragments, *nused* is the number of such fragments currently in use in the server, and *nfree* is the number of such fragments that have been reserved for use but are currently free.

On servers for which such statistics are not available, `memory_usage()` returns `{}`. The compilation option `USE_GNU_MALLOC` controls whether or not statistics are available; if the option is not provided, statistics are not available.

`none` **dump_database** () Function

Requests that the server checkpoint the database at its next opportunity. It is not normally necessary to call this function; the server automatically checkpoints the database at regular intervals; see the chapter on server assumptions about the database for details. If the programmer is not a wizard, then `E_PERM` is raised.

`int` **db_disk_size** () Function

Returns the total size, in bytes, of the most recent full representation of the database as one or more disk files. Raises `E_QUOTA` if, for some reason, no such on-disk representation is currently available.

`none` **shutdown** ([`str` *message*]) Function

Requests that the server shut itself down at its next opportunity. Before doing so, a notice (incorporating *message*, if provided) is printed to all connected players. If the programmer is not a wizard, then `E_PERM` is raised.

5 Server Commands and Database Assumptions

This chapter describes all of the commands that are built into the server and every property and verb in the database specifically accessed by the server. Aside from what is listed here, no assumptions are made by the server concerning the contents of the database.

5.1 Built-in Commands

As was mentioned in the chapter on command parsing, there are five commands whose interpretation is fixed by the server: `PREFIX`, `OUTPUTPREFIX`, `SUFFIX`, `OUTPUTSUFFIX`, and `.program`. The first four of these are intended for use by programs that connect to the MOO, so-called 'client' programs. The `.program` command is used by programmers to associate a MOO program with a particular verb. The server can, in addition, recognize a sixth special command on any or all connections, the *flush* command.

The server also performs special processing on command lines that begin with certain punctuation characters.

This section discusses these built-in pieces of the command-interpretation process.

5.1.1 Command-Output Delimiters

Every MOO network connection has associated with it two strings, the *output prefix* and the *output suffix*. Just before executing a command typed on that connection, the server prints the output prefix, if any, to the player. Similarly, just after finishing the command, the output suffix, if any, is printed to the player. Initially, these strings are not defined, so no extra printing takes place.

The `PREFIX` and `SUFFIX` commands are used to set and clear these strings. They have the following simple syntax:

```
PREFIX  output-prefix
SUFFIX  output-suffix
```

That is, all text after the command name and any following spaces is used as the new value of the appropriate string. If there is no non-blank text after the command string, then the corresponding string is cleared. For compatibility with some general MUD client programs, the server also recognizes `OUTPUTPREFIX` as a synonym for `PREFIX` and `OUTPUTSUFFIX` as a synonym for `SUFFIX`.

These commands are intended for use by programs connected to the MOO, so that they can issue MOO commands and reliably determine the beginning and end of the resulting output. For example, one editor-based client program sends this sequence of commands on occasion:

```
PREFIX >>MOO-Prefix<<
SUFFIX >>MOO-Suffix<<
@list object:verb without numbers
PREFIX
SUFFIX
```

The effect of which, in a LambdaCore-derived database, is to print out the code for the named verb preceded by a line containing only '`>>MOO-Prefix<<`' and followed by a line containing only '`>>MOO-Suffix<<`'. This enables the editor to reliably extract the program text from the MOO output and show it to the user in a separate editor window. There are many other possible uses.

The built-in function `output_delimiters()` can be used by MOO code to find out the output prefix and suffix currently in effect on a particular network connection.

5.1.2 Programming

The `.program` command is a common way for programmers to associate a particular MOO-code program with a particular verb. It has the following syntax:

> `.program` *object*`:`*verb*
> ...*several lines of MOO code*...
> .

That is, after typing the `.program` command, then all lines of input from the player are considered to be a part of the MOO program being defined. This ends as soon as the player types a line containing only a dot (‘`.`’). When that line is received, the accumulated MOO program is checked for proper MOO syntax and, if correct, associated with the named verb.

If, at the time the line containing only a dot is processed, (a) the player is not a programmer, (b) the player does not have write permission on the named verb, or (c) the property `$server_options.protect_set_verb_code` exists and has a true value and the player is not a wizard, then an error message is printed and the named verb's program is not changed.

In the `.program` command, *object* may have one of three forms:

- The name of some object visible to the player. This is exactly like the kind of matching done by the server for the direct and indirect objects of ordinary commands. See the chapter on command parsing for details. Note that the special names ‘`me`’ and ‘`here`’ may be used.
- An object number, in the form `#`*number*.
- A *system property* (that is, a property on `#0`), in the form `$`*name*. In this case, the current value of `#0.`*name* must be a valid object.

5.1.3 Flushing Unprocessed Input

It sometimes happens that a user changes their mind about having typed one or more lines of input and would like to ‘untype’ them before the server actually gets around to processing them. If they react quickly enough, they can type their connection's defined *flush* command; when the server first reads that command from the network, it immediately and completely flushes any as-yet unprocessed input from that user, printing a message to the user describing just which lines of input were discarded, if any.

> *Fine point:* The flush command is handled very early in the server's processing of a line of input, before the line is entered into the task queue for the connection and well before it is parsed into words like other commands. For this reason, it must be typed exactly as it was defined, alone on the line, without quotation marks, and without any spaces before or after it.

When a connection is first accepted by the server, it is given an initial flush command setting taken from the current default. This initial setting can be changed later using the `set_connection_option()` command.

By default, each connection is initially given ‘`.flush`’ as its flush command. If the property `$server_options.default_flush_command` exists, then its value overrides this default. If `$server_options.default_flush_command` is a non-empty string, then that string is the flush command for all new connections; otherwise, new connections are initially given no flush command at all.

5.1.4 Initial Punctuation in Commands

The server interprets command lines that begin with any of the following characters specially:

```
"        :        ;
```

Before processing the command, the initial punctuation character is replaced by the corresponding word below, followed by a space:

```
say      emote    eval
```

For example, the command line

```
"Hello, there.
```

is transformed into

```
say Hello, there.
```

before parsing.

5.2 Server Assumptions About the Database

There are a small number of circumstances under which the server directly and specifically accesses a particular verb or property in the database. This section gives a complete list of such circumstances.

5.2.1 Server Options Set in the Database

Many optional behaviors of the server can be controlled from within the database by creating the property `#0.server_options` (also known as `$server_options`), assigning as its value a valid object number, and then defining various properties on that object. At a number of times, the server checks for whether the property `$server_options` exists and has an object number as its value. If so, then the server looks for a variety of other properties on that `$server_options` object and, if they exist, uses their values to control how the server operates.

The specific properties searched for are each described in the appropriate section below, but here is a brief list of all of the relevant properties for ease of reference:

`bg_seconds`
: The number of seconds allotted to background tasks.

`bg_ticks`
: The number of ticks allotted to background tasks.

`connect_timeout`
: The maximum number of seconds to allow an un-logged-in in-bound connection to remain open.

`default_flush_command`
: The initial setting of each new connection's flush command.

`fg_seconds`
: The number of seconds allotted to foreground tasks.

`fg_ticks`
: The number of ticks allotted to foreground tasks.

`max_stack_depth`
: The maximum number of levels of nested verb calls.

`name_lookup_timeout`
: The maximum number of seconds to wait for a network hostname/address lookup.

`outbound_connect_timeout`
: The maximum number of seconds to wait for an outbound network connection to successfully open.

`protect_`*property*
: Restrict reading of built-in *property* to wizards.

`protect_`*function*
: Restrict use of built-in *function* to wizards.

`support_numeric_verbname_strings`
: Enables use of an obsolete verb-naming mechanism.

5.2.2 Server Messages Set in the Database

There are a number of circumstances under which the server itself generates messages on network connections. Most of these can be customized or even eliminated from within the database. In each such case, a property on `$server_options` is checked at the time the message would be printed. If the property does not exist, a default message is printed. If the property exists and its value is not a string or a list containing strings, then no message is printed at all. Otherwise, the string(s) are printed in place of the default message, one string per line. None of these messages are ever printed on an outbound network connection created by the function `open_network_connection()`.

The following list covers all of the customizable messages, showing for each the name of the relevant property on `$server_options`, the default message, and the circumstances under which the message is printed:

`boot_msg = "*** Disconnected ***"`
: The function `boot_player()` was called on this connection.

`connect_msg = "*** Connected ***"`
: The user object that just logged in on this connection existed before `$do_login_command()` was called.

`create_msg = "*** Created ***"`
: The user object that just logged in on this connection did not exist before `$do_login_command()` was called.

`recycle_msg = "*** Recycled ***"`
: The logged-in user of this connection has been recycled or renumbered (via the renumber() function).

`redirect_from_msg = "*** Redirecting connection to new port ***"`
: The logged-in user of this connection has just logged in on some other connection.

`redirect_to_msg = "*** Redirecting old connection to this port ***"`
: The user who just logged in on this connection was already logged in on some other connection.

`server_full_msg`
: Default:

```
*** Sorry, but the server cannot accept any more connections right now.
*** Please try again later.
```

This connection arrived when the server really couldn't accept any more connections, due to running out of a critical operating system resource.

`timeout_msg = "*** Timed-out waiting for login. ***"`
: This in-bound network connection was idle and un-logged-in for at least `CONNECT_TIMEOUT` seconds (as defined in the file '`options.h`' when the server was compiled).

Fine point: If the network connection in question was received at a listening point (established by the '`listen()`' function) handled by an object *obj* other than `#0`, then system messages for that connection are looked for on *obj*`.server_options`; if that property does not exist, then `$server_options` is used instead.

5.2.3 Checkpointing the Database

The server maintains the entire MOO database in main memory, not on disk. It is therefore necessary for it to dump the database to disk if it is to persist beyond the lifetime of any particular server execution. The server is careful to dump the database just before shutting down, of course, but it is also prudent for it to do so at regular intervals, just in case something untoward happens.

To determine how often to make these *checkpoints* of the database, the server consults the value of `#0.dump_interval`. If it exists and its value is an integer greater than or equal to 60, then it is taken as the number of seconds to wait between checkpoints; otherwise, the server makes a new checkpoint every 3600 seconds (one hour). If the value of `#0.dump_interval` implies that the next checkpoint should be scheduled at a time after 3:14:07 a.m. on Tuesday, January 19, 2038, then the server instead uses the default value of 3600 seconds in the future.

The decision about how long to wait between checkpoints is made again immediately after each one begins. Thus, changes to `#0.dump_interval` will take effect after the next checkpoint happens.

Whenever the server begins to make a checkpoint, it makes the following verb call:

```
$checkpoint_started()
```

When the checkpointing process is complete, the server makes the following verb call:

`$checkpoint_finished(`*success*`)`

where *success* is true if and only if the checkpoint was successfully written on the disk. Checkpointing can fail for a number of reasons, usually due to exhaustion of various operating system resources such as virtual memory or disk space. It is not an error if either of these verbs does not exist; the corresponding call is simply skipped.

5.2.4 Accepting and Initiating Network Connections

When the server first accepts a new, incoming network connection, it is given the low-level network address of computer on the other end. It immediately attempts to convert this address into the human-readable host name that will be entered in the server log and returned by the `connection_name()` function. This conversion can, for the TCP/IP networking configurations, involve a certain amount of communication with remote name servers, which can take quite a long time and/or fail entirely. While the server is doing this conversion, it is not doing anything else at all; in particular, it it not responding to user commands or executing MOO tasks.

By default, the server will wait no more than 5 seconds for such a name lookup to succeed; after that, it behaves as if the conversion had failed, using instead a printable representation of the low-level address. If the property `name_lookup_timeout` exists on `$server_options` and has an integer as its value, that integer is used instead as the timeout interval.

When the `open_network_connection()` function is used, the server must again do a conversion, this time from the host name given as an argument into the low-level address necessary for actually opening the connection. This conversion is subject to the same timeout as in the in-bound case; if the conversion does not succeed before the timeout expires, the connection attempt is aborted and `open_network_connection()` raises `E_QUOTA`.

After a successful conversion, though, the server must still wait for the actual connection to be accepted by the remote computer. As before, this can take a long time during which the server is again doing nothing else. Also as before, the server will by default wait no more than 5 seconds for the connection attempt to succeed; if the timeout expires, `open_network_connection()` again raises `E_QUOTA`. This default timeout interval can also be overridden from within the database, by defining the property `outbound_connect_timeout` on `$server_options` with an integer as its value.

5.2.5 Associating Network Connections with Players

When a network connection is first made to the MOO, it is identified by a unique, negative object number. Such a connection is said to be *un-logged-in* and is not yet associated with any MOO player object.

Each line of input on an un-logged-in connection is first parsed into words in the usual way (see the chapter on command parsing for details) and then these words are passed as the arguments in a call to the verb `$do_login_command()`. For example, the input line

```
connect Munchkin frebblebit
```

would result in the following call being made:

```
$do_login_command("connect", "Munchkin", "frebblebit")
```

In that call, the variable `player` will have as its value the negative object number associated with the appropriate network connection. The functions `notify()` and `boot_player()` can be used with such object numbers to send output to and disconnect un-logged-in connections. Also, the variable `argstr` will have as its value the unparsed command line as received on the network connection.

If `$do_login_command()` returns a valid player object and the connection is still open, then the connection is considered to have *logged into* that player. The server then makes one of the following verbs calls, depending on the player object that was returned:

`$user_created(`*player*`)`
`$user_connected(`*player*`)`
`$user_reconnected(`*player*`)`

The first of these is used if the returned object number is greater than the value returned by the `max_object()` function before `$do_login_command()` was invoked, that is, it is called if the returned object appears to have been freshly created. If this is not the case, then one of the other two verb calls is used. The `$user_connected()` call is used if there was no existing active connection for the returned player object. Otherwise, the `$user_reconnected()` call is used instead.

> *Fine point:* If a user reconnects and the user's old and new connections are on two different listening points being handled by different objects (see the description of the

`listen()` function for more details), then `user_client_disconnected` is called for the old connection and `user_connected` for the new one.

If an in-bound network connection does not successfully log in within a certain period of time, the server will automatically shut down the connection, thereby freeing up the resources associated with maintaining it. Let L be the object handling the listening point on which the connection was received (or `#0` if the connection came in on the initial listening point). To discover the timeout period, the server checks on L`.server_options` or, if it doesn't exist, on `$server_options` for a `connect_timeout` property. If one is found and its value is a positive integer, then that's the number of seconds the server will use for the timeout period. If the `connect_timeout` property exists but its value isn't a positive integer, then there is no timeout at all. If the property doesn't exist, then the default timeout is 300 seconds.

When any network connection (even an un-logged-in or outbound one) is terminated, by either the server or the client, then one of the following two verb calls is made:

`$user_disconnected(`*player*`)`
`$user_client_disconnected(`*player*`)`

The first is used if the disconnection is due to actions taken by the server (e.g., a use of the `boot_player()` function or the un-logged-in timeout described above) and the second if the disconnection was initiated by the client side.

It is not an error if any of these five verbs do not exist; the corresponding call is simply skipped.

> **Note**: Only one network connection can be controlling a given player object at a given time; should a second connection attempt to log in as that player, the first connection is unceremoniously closed (and `$user_reconnected()` called, as described above). This makes it easy to recover from various kinds of network problems that leave connections open but inaccessible.

When the network connection is first established, the null command is automatically entered by the server, resulting in an initial call to `$do_login_command()` with no arguments. This signal can be used by the verb to print out a welcome message, for example.

> **Warning**: If there is no `$do_login_command()` verb defined, then lines of input from un-logged-in connections are simply discarded. Thus, it is *necessary* for any database to include a suitable definition for this verb.

5.2.6 Out-of-Band Commands

It is possible to compile the server with an option defining an *out-of-band prefix* for commands. This is a string that the server will check for at the beginning of every line of input from players, regardless of whether or not those players are logged in and regardless of whether or not reading tasks are waiting for input from those players. If a given line of input begins with the defined out-of-band prefix (leading spaces, if any, are *not* stripped before testing), then it is not treated as a normal command or as input to any reading task. Instead, the line is parsed into a list of words in the usual way and those words are given as the arguments in a call to `$do_out_of_band_command()`. For example, if the out-of-band prefix were defined to be '`#$#`', then the line of input

```
#$# client-type fancy
```

would result in the following call being made in a new server task:

```
$do_out_of_band_command("#$#", "client-type", "fancy")
```

During the call to `$do_out_of_band_command()`, the variable `player` is set to the object number representing the player associated with the connection from which the input line came. Of course, if that connection has not yet logged in, the object number will be negative. Also, the variable `argstr` will have as its value the unparsed input line as received on the network connection.

Out-of-band commands are intended for use by fancy client programs that may generate asynchronous *events* of which the server must be notified. Since the client cannot, in general, know the state of the player's connection (logged-in or not, reading task or not), out-of-band commands provide the only reliable client-to-server communications channel.

5.2.7 The First Tasks Run By the Server

Whenever the server is booted, there are a few tasks it runs right at the beginning, before accepting connections or getting the value of `#0.dump_interval` to schedule the first checkpoint (see below for more information on checkpoint scheduling).

First, the server calls `$user_disconnected()` once for each user who was connected at the time the database file was written; this allows for any cleaning up that's usually done when users disconnect (e.g., moving their player objects back to some 'home' location, etc.).

Next, it checks for the existence of the verb `$server_started()`. If there is such a verb, then the server runs a task invoking that verb with no arguments and with `player` equal to `#-1`. This is useful for carefully scheduling checkpoints and for re-initializing any state that is not properly represented in the database file (e.g., re-opening certain outbound network connections, clearing out certain tables, etc.).

5.2.8 Controlling the Execution of Tasks

As described earlier, in the section describing MOO tasks, the server places limits on the number of seconds for which any task may run continuously and the number of "ticks," or low-level operations, any task may execute in one unbroken period. By default, foreground tasks may use 30,000 ticks and five seconds, and background tasks may use 15,000 ticks and three seconds. These defaults can be overridden from within the database by defining any or all of the following properties on `$server_options` and giving them integer values:

`bg_seconds`
: The number of seconds allotted to background tasks.

`bg_ticks`
: The number of ticks allotted to background tasks.

`fg_seconds`
: The number of seconds allotted to foreground tasks.

`fg_ticks`
: The number of ticks allotted to foreground tasks.

The server ignores the values of `fg_ticks` and `bg_ticks` if they are less than 100 and similarly ignores `fg_seconds` and `bg_seconds` if their values are less than 1. This may help prevent utter disaster should you accidentally give them uselessly-small values.

Recall that command tasks and server tasks are deemed *foreground* tasks, while forked, suspended, and reading tasks are defined as *background* tasks. The settings of these variables take effect only at the beginning of execution or upon resumption of execution after suspending or reading.

The server also places a limit on the number of levels of nested verb calls, raising `E_MAXREC` from a verb-call expression if the limit is exceeded. The limit is 50 levels by default, but this can be increased from within the database by defining the `max_stack_depth` property on `$server_options` and giving it an integer value greater than 50. The maximum stack depth for any task is set at the time that task is created and cannot be changed thereafter. This implies that suspended tasks, even after being saved in and restored from the DB, are not affected by later changes to $server_options.max_stack_depth.

Finally, the server can place a limit on the number of forked or suspended tasks any player can have queued at a given time. Each time a `fork` statement or a call to `suspend()` is executed in some verb, the server checks for a property named `queued_task_limit` on the programmer. If that property exists and its value is a non-negative integer, then that integer is the limit. Otherwise, if `$server_options.queued_task_limit` exists and its value is a non-negative integer, then that's the limit. Otherwise, there is no limit. If the programmer already has a number of queued tasks that is greater than or equal to the limit, `E_QUOTA` is raised instead of either forking or suspending. Reading tasks are affected by the queued-task limit.

5.2.9 Controlling the Handling of Aborted Tasks

The server will abort the execution of tasks for either of two reasons:

1. an error was raised within the task but not caught, or
2. the task exceeded the limits on ticks and/or seconds.

In each case, after aborting the task, the server attempts to call a particular *handler verb* within the database to allow code there to handle this mishap in some appropriate way. If this verb call suspends or returns a true value, then it is considered to have handled the situation completely and no further processing will be done by the server. On the other hand, if the handler verb does not exist, or if the call either returns a false value without suspending or itself is aborted, the server takes matters into its own hands.

First, an error message and a MOO verb-call stack *traceback* are printed to the player who typed the command that created the original aborted task, explaining why the task was aborted and where in the task the problem occurred. Then, if the call to the handler verb was itself aborted, a second error message and traceback are printed, describing that problem as well. Note that if the handler-verb call itself is aborted, no further 'nested' handler calls are made; this policy prevents what might otherwise be quite a vicious little cycle.

The specific handler verb, and the set of arguments it is passed, differs for the two causes of aborted tasks.

If an error is raised and not caught, then the verb-call

`$handle_uncaught_error(`*code*, *msg*, *value*, *traceback*, *formatted*`)`

is made, where *code*, *msg*, *value*, and *traceback* are the values that would have been passed to a handler in a `try`-`except` statement and *formatted* is a list of strings being the lines of error and traceback output that will be printed to the player if `$handle_uncaught_error` returns false without suspending.

If a task runs out of ticks or seconds, then the verb-call

```
$handle_task_timeout(resource, traceback, formatted)
```

is made, where *resource* is the appropriate one of the strings `"ticks"` or `"seconds"`, and *traceback* and *formatted* are as above.

5.2.10 Matching in Command Parsing

In the process of matching the direct and indirect object strings in a command to actual objects, the server uses the value of the `aliases` property, if any, on each object in the contents of the player and the player's location. For complete details, see the chapter on command parsing.

5.2.11 Restricting Access to Built-in Properties and Functions

Whenever verb code attempts to read the value of a built-in property *prop* on any object, the server checks to see if the property `$server_options.protect_`*prop* exists and has a true value. If so, then `E_PERM` is raised if the programmer is not a wizard.

Whenever verb code calls a built-in function *func*`()` and the caller is not the object `#0`, the server checks to see if the property `$server_options.protect_`*func* exists and has a true value. If so, then the server next checks to see if the verb `$bf_`*func*`()` exists; if that verb exists, then the server calls it *instead* of the built-in function, returning or raising whatever that verb returns or raises. If the `$bf_`*func*`()` does not exist and the programmer is not a wizard, then the server immediately raises `E_PERM`, *without* actually calling the function. Otherwise (if the caller is `#0`, if `$server_options.protect_`*func* either doesn't exist or has a false value, or if `$bf_`*func*`()` exists but the programmer is a wizard), then the built-in function is called normally.

5.2.12 Creating and Recycling Objects

Whenever the `create()` function is used to create a new object, that object's `initialize` verb, if any, is called with no arguments. The call is simply skipped if no such verb is defined on the object.

Symmetrically, just before the `recycle()` function actually destroys an object, the object's `recycle` verb, if any, is called with no arguments. Again, the call is simply skipped if no such verb is defined on the object.

Both `create()` and `recycle()` check for the existence of an `ownership_quota` property on the owner of the newly-created or -destroyed object. If such a property exists and its value is an integer, then it is treated as a *quota* on object ownership. Otherwise, the following two paragraphs do not apply.

The `create()` function checks whether or not the quota is positive; if so, it is reduced by one and stored back into the `ownership_quota` property on the owner. If the quota is zero or negative, the quota is considered to be exhausted and `create()` raises `E_QUOTA`.

The `recycle()` function increases the quota by one and stores it back into the `ownership_quota` property on the owner.

5.2.13 Object Movement

During evaluation of a call to the `move()` function, the server can make calls on the `accept` and `enterfunc` verbs defined on the destination of the move and on the `exitfunc` verb defined on the source. The rules and circumstances are somewhat complicated and are given in detail in the description of the `move()` function.

5.2.14 Temporarily Enabling Obsolete Server Features

If the property `$server_options.support_numeric_verbname_strings` exists and has a true value, then the server supports a obsolete mechanism for less ambiguously referring to specific verbs in various built-in functions. For more details, see the discussion given just following the description of the `verbs()` function.

6 Function Index

P

Q

R

S

T

U

V

Table of Contents