MOO Programming Reference

This section is geared toward people who are at least somewhat comfortable with programming. For the fine points I refer you to the programmer's manual itself. Here, I have tried to present an overview of the MOO programming language in a format that is biased towards ease of reference rather than exhaustive explication. I have provided brief explanations of some points where I think clarification might be helpful.

Data Types

Variables are not of fixed data type; they become the data type of the value assigned to them. Use `typeof(<variable>)` to see what you've got.

`INT`	Integer: `29, 0, -0, 5`
`FLOAT`	Floating point number: `29.0, 0.0, 5.0`
`NUM`	Historical, same as `INT`. (`FLOAT` was a later addition.) `FLOAT`s and `INT`s don't mix and match. Use toint() or tofloat() to force one to be the other. Note that (1 == 1.0) evaluates to false.
`STR`	A quoted string: `"pickles"`. To include the double quote mark itself in a string, precede it with the backslash character: `"Oliver shouts, \"Yow!\""` Strings are case-insensitive: `("carrot" == "CarROT")` evaluates to true. Use `equal("carrot", "CarROT")` if you need to differentiate between the two.
`OBJ`	An object, e.g: `#1234`. In conditional statements, an object by itself evaluates to false. Use `valid(<object>)` instead. Some special objects: • `#-1` or `$nothing`. Not valid, but anything may be moved there. The canonical invalid object. • `#-2` or `$ambiguous_match`. • `#-3` or `$failed_match` • `$garbage` This object and kids of it are valid but not useful. They are all owned by the special system player Hacker. When an object is recycled, it becomes a kid of `$garbage`.

LIST	`{1, 2.5, "a string", {"a", "sublist"}, #321, E_RANGE, 2.5}`
	
	`LIST`s are designated with curly braces (`{}`), may nest to an arbitrary number of levels, and their elements need not be of the same data type. Their elements are preserved in order and may include duplicates (i.e. they are not like mathematical sets).
	
	The @ operator yields the elements of the list as separate elements. Another way to phrase this is that it is the inverse of putting curly braces around some elements. Two canonical uses:
	
	`<some-list> = {@<some-list>, <new_element>};` This is the usual way to add an element onto the end of a list. If `a` is `{1, 2, 3}`, and `b` is 4, then `{@a, b}` gives the four-element list `{1, 2, 3, 4}`, whereas `{a, b}` would give the two-element list `{{1, 2, 3}, 4}`.
	
	`pass(@args);` This causes the identically-named verb on the current object's parent to be run with the same argument list. If you just did `pass(args)`, then the arguments to the verb being called would have an extra set of braces around them, thus making the argument list `{{a, b, c}}`, for example, instead of `{a, b, c}`.
	
	`<element> in <some-list>` will return the (1-based) index of the first instance of `<element>` in `<some-list>` if it is present, `0` otherwise. A typical usage might be:
	
	`if (item.location in {player,`
	` player.location})`
	`  <exprs>;`
	`endif`

ERR	E_NONE	no error
	E_TYPE	type mismatch
	E_DIV	division by zero
	E_PERM	permission denied
	E_PROPNF	property not found
	E_VERBNF	verb not found
	E_VARNF	variable not found
	E_INVIND	invalid indirection
	E_RECMOVE	recursive move
	E_MAXREC	too many verb calls (max recursion)
	E_RANGE	range error (subscript too large, or zero, or negative)
	E_ARGS	incorrect number of arguments
	E_NACC	move refused by destination (i.e. object not acceptable)
	E_INVARG	invalid argument
	E_QUOTA	resource limit exceeded
	E_FLOAT	floating-point arithmetic error

Errors can be raised (yielding a traceback) or caught (then handled or ignored). The following two constructs are used to trap errors and deal with them:

```
`<expr1> ! ANY => <expr2>'
```

See section 4.1.12 of the programmer’s manual. The single quotes are part of the expression, and are specifically back single quote and forward single quote.

```
try
  <exprs>;
except ANY
  <alternate exprs>;
endtry
```

See sections 4.2.7 and 4.2.8 of the programmer’s manual.

Subscripting

Everything is 1-based.

You can subscript lists or strings.

`<list-or-string>[<expr1>..<expr2>]` gives slices (sub-list or sub-string). `<expr1>` and `<expr2>` must be in range; `<list-or-string>[$]` is the last element or character.

The following are well-formed:

```
<variable> = <list-or-string>[<expr>];
<variable> = <list-or-string>[<expr1..expr2>];
<list-or-string>[<expr1>] = <expr2>;
<some-list>[<expr1>..<expr2>] = <expr3>;
<some-string>[<expr1>..<expr2>] = <expr4>;
```

Note that in the above example, `<expr3>` must evaluate to a list and `<expr4>` must evaluate to a string.

Accessing Properties and Verbs on Objects

`$<something>` is the same as `#0.<something>`.

You can use parentheses to access property names and verbs dynamically:

```
<object>.(<calculated-property-name>)
<object>:(<calculated-verb-name>)
```

Variables

Variables are local and dynamic, coming into existence when assigned a value. For global variables, define and use a property.

In addition to the data types themselves (which evaluate to integers), the following built-in variables are provided:

`this`	The object on which the currently-running verb is defined.
`player`	The object number of the player who typed in the command.
`caller`	The objnum of the object on which the calling verb is defined, or `player`, if the verb was called from the command line.
`verb`	The name by which the currently-running verb was invoked. Verbs may have aliases.
`args`	The list of arguments with which a subroutine was called or, if a command-line verb, with which the command was invoked.
`argstr`	Everything that was typed in after the verb name on the command line.
`dobj`	The direct object as parsed from the command line.
`dobjstr`	The string from which dobj was matched.

`prepstr`	The string that was parsed as the preposition.
`iobj`	The indirect object as parsed from the command line.
`iobjstr`	The string from which iobj was matched.

Any of these may be reassigned within a verb and their values will persist into the next verb call except that `caller` will change to the current object, and a changed value of `player` will not persist unless the verb's owner is a wizard.

Scattering Assignment

Several variables may be assigned values in a single line, and this is often done to assign incoming arguments to named variables. A typical example might be:

```
    {who, what, ?where = player.location, ?when=time()} = args;
```

See section 4.1.9 of the programmer's manual for a detailed explanation.

Operators

The following operators apply, in order of precedence:

`!` `-`	not arithmetic negation (without a left operand)
`^`	exponentiation

`*`
`/`
`%`
	multiplication
	division
	modulo

`+` `-`	addition (note, + also concatenates two strings) subtraction

`==`
`!=`
`<`
`<=`
`>`
`>=`
`in`
	is equal to (note, easy to confuse with the assignment operator – nasty!)
	is not equal to
	less than
	less than or equal to (note, =< doesn't work)
	greater than
	greater than or equal to (note, => doesn't work)
	element position in a list

`&&`	logical "and"

`||`	logical “or”
`… ? … | …`	the conditional operator. `<expr1> ? <expr2> | <expr3>` is equivalent to: `if (<expr1>) <expr2>; else <expr3>; endif`
`=`	assignment (note, easy to confuse with a test for equality – nasty!)

Assignments may appear within expressions. Use parentheses liberally to avoid mistakes and confusion.

Truth Values

`0`, `-0`, `0.0`, `-0.0`, `""`, `{}`, errors, and objects all evaluate to false. Anything else evaluates to true.

Compound Statements

Use a semicolon after expressions within the body of a compound statement, but not after lines of the compound statement itself, thus:

```
if (<expr1>)
  "This is a comment.";
  <expr2>;
  <expr3>;
elseif (<expr4>)
  <more-exprs>;
elseif (<expr5>)
  <something else entirely>;
else
  "None of the above.";
  <final-exprs>;
endif
```

Looping

```
for <variable> in (<some-list>)
  <exprs>;
endfor

for <index> in [<int1>..<int2>]
  <exprs>;
endfor

while (<condition>)
  <exprs>;
endwhile

break;

break <name>;

continue;

continue <name>;
```

(See section 4.2.5 of the programmer's manual for the fine points of `break` and `continue`.)

Background Tasks

```
fork (<delay-in-seconds>)
  <exprs>;
endfork

fork <variable> (<delay-in-seconds>)
  <exprs>;
endfork
```

In the second example, `<variable>` receives the number (task_id) of the forked task.

Time Management

A task is the execution of a command from start to finish, or the execution of the statements within a `fork/endfork` statement from start to finish. Tasks are identified with numbers, and are allotted a fixed number of ticks and a fixed number of seconds for execution. The LambdaCore default is 30,000 ticks for a foreground task and 15,000 ticks for a background task. Properties to override these numbers may be added to the object `$server_options` by a wizard and inspected (if present) by programmers.

If a task runs out of ticks, it is unceremoniously terminated by the system. If a task is in danger of running out of ticks, a programmer may get a new allotment by suspending the task briefly (note, `suspend ($login.current_lag)` is considered polite). When a task suspends, it obtains an additional allotment of ticks, so it is not uncommon to find or place a `suspend()` statement either right before or inside of a loop.

The utility verb `$command_utils:suspend_if_needed()` might suggest itself, but in fact it uses up a fair number of ticks, itself. Current fashion is to use a line of the form:

```
((ticks_left() < 3000) && suspend($login.current_lag));
```

See also sections 4.4 and 5.2.8 of the programmer's manual.

Argument Specifiers

When defining a verb on an object (with the `@verb` command), you must provide specifiers for the arguments with which the verb will be invoked.

The allowable specifiers are:

- direct object specifiers

  ```
  this
  any
  none
  ```

- prepositions

  ```
  none
  any
  with/using
  at/to
  in front of
  in/inside/into
  on/onto/upon/on top of
  from/from inside/out of
  over
  through
  under/underneath/beneath
  behind
  beside
  for/about
  is
  ```

```
as
of/off of
```

- indirect object specifiers

```
this
any
none
```

Definite and indefinite articles are omitted. When deciding which argument specifiers to use, it is helpful to imagine what a user would actually type when invoking the command, then generalize from that. When writing subroutines that aren't intended to be invoked from the command line, specify the arguments as "`this none this`".

Eval

The `eval` command evaluates a string as MOOcode. Like `say` and `emote`, it can be abbreviated to a single character command, `;`. A second form, `;;` evaluates a sequence of expressions, each terminated by a semicolon (as in a verb). Compound statements don't end in a semicolon. The form using two semicolons prints out 0 as its value; if you want to see results you should include a call to `player:tell();` at the end:

```
;;"Count players who have more than ten aliases"; total = 0;
for dude in (players()) if (length(dude.aliases) > 10) total
= total + 1; endif endfor player:tell("Total:  " +
tostr(total));
```

A second form of `eval`, "#" matches an object by name if it's in your vicinity, and is useful for looking at properties or just quickly finding out the object number of something close by. Property names can be chained:

```
#rock
#rock.moss_list
#yib p
#yib.aliases p
#yib.location.owner.name p
```

The last form, terminated by " `p`" matches the name of a player even if you're not in eir vicinity, so that you don't have to know eir number to look at a (readable) property on em. "#" can also be used with object numbers directly:

```
#58337.location.contents
```

Use `@setenv` to set up some commonly-used variable settings in advance:

```
@setenv me = player; here = player.location;
```

Inspect the result with

```
#me.eval_env
```

See also `help eval` and `help #`.

Ownership and Permissions

Every object has an owner. Every property on every object has an owner, but it doesn't have to be the same as the owner of the object. Every verb on every object has an owner, but it doesn't have to be the same as the owner of the object.

Task permissions are expressed as an object number, that of the player who owns the verb currently being executed.

The function `caller_perms()` returns the task permissions of the calling verb, or `#-1` (an invalid object) if the currently running verb was called from the command line.

Inherited verbs always have the same owner as the owner of the corresponding verb on the parent or ancestor object. They run with that owner's permissions, except that wizard-owned verbs can set the task permissions to another (usually non-wizardly) player.

To `+c` Or To `–c`, That Is The Question

Ownership of an inherited property *depends* on whether the property was initially defined as `+c` or `-c`. If it was defined as `+c` (think "may be *changed* by the owner of the child/descendent object"), then the property is *owned by* the owner of the child/descendent object. If the property was initially defined as `-c` then the property on all children and descendents is owned by the player who defined the property on the parent/ancestor object, *and its value can be changed by verbs running with that player's permissions*. This becomes relevant when one is making a generic object. If the owner of a child or descendent object will need to `@set` or otherwise change the property, then define it as `+c`. This is typically done for messages, and also for other parameters, for example the number of times one must turn the crank before the jack in the box pops out. If, on the other hand, one of the verbs you write on the generic will need to change the value of a property, then it should be defined as -c so that the property on all descendent objects will still be owned by you. Then your verbs, running with your permissions, can change it (for example, the number of times the crank on the jack in the box has been turned so far).

When you make an object strictly for your own use, it really doesn't matter whether the properties are `+c` or `-c`. It becomes an issue when other people make kids of your object. Then if a property that one of your verbs needs and tries to change is mistakenly `+c`, the verb will encounter a permissions error. If you `@chmod` the property to `-c`, then all *new* kids of the object will have that property owned by you, but it isn't changed retroactively for existing kids. If you make a property +c and find out later that it should have been -c, you can change it on all descendents by evaluating the following:

```
;$wiz_utils:set_property_flags(<object>, <property_name>,
<property_flags>)
```

You don't have to be a wizard to use this verb – just the owner of the object. Here's an illustration, supposing that a generic conker is object #1234:

```
;$wiz_utils:set_property_flags(#1234, "thwaps", "r")
```

This would have the effect of making the `.thwaps` property on all descendents of the generic conker readable but neither writable nor changeable (by owners of kid objects), and the property would be owned by the author of the generic conker in all cases (and could be changed by that player's verb(s)).

Hidden Treasures

Some verbs are called automatically, seemingly invisibly. Here are some of them:

`<object>:look_self`	Called when you look at `<object>`
`<object>:description`	Called (if it exists) by `:look_self`
`<object>:tell_contents`	Called (if it exists) by `:look_self`
`<object>:enterfunc`	Called when something is moved to `<object>`
`<object>:exitfunc`	Called when something is removed from `<object>`'s `.contents`.
`<room>:confunc`	Called when someone connects inside a room.
`<room>:disfunc`	Called when someone disconnects inside a room.
`<player>:confunc`	Called when a player connects
`<player>:disfunc`	Called when a player disconnects
`<object>:initialize`	Called when an object is created. Use, for example to initialize parameters on the kid of a generic object.
`<object>:recycle`	Called right before an object is recycled.
`<player-or-room>:huh`	Called if the parser can't find an object with the appropriate verbspec. This is how exits in rooms are invoked without the exit objects' having to be *in* rooms, for example.

A Couple “Tricks of the Trade”

Sending mail messages from within a verb: The relevant verb is `$mail_agent:send_message`. Personally, I always find the help text hard to read, so I am providing this illustrative example, which I hope may be helpful:

```
;$mail_agent:send_message(me, {me},
  "This is the subject heading", {"Line1", "Line2", "",
  "Oooga boooga!"})
```

Creating objects on the fly is fun, and possible if you are not over quota. Here is an example of how it’s done.

```
@verb me:test none none none rd

@program me:test
"Sample verb to demonstrate creating an object on the fly.";
thing1 = `$recycler:_create($thing) ! ANY =>
  $nothing';
if (valid(thing1))
  thing1:set_name("thing1");
  thing1:moveto(player.location);
  "If you create a lot of things, then you need to measure
them as you go to avoid a 'resource limit exceeded' error.";
  $quota_utils:object_bytes(thing1);
  player:tell("You now have something that you didn't have
before!");
else
  player:tell(
    "Couldn't create thing1.  Don't know why.");
endif
.

test
```

To recycle an object (that you own) from within a verb:

```
$recycler:_recycle(<object>);
```

Programming Feature Objects

This is a very brief summary of the steps involved in creating a feature object. It isn’t a tutorial on programming in general, but highlights a couple of quirks associated with programming this particular kind of object.

First, create a kid of the generic feature object:

```
@create $feature named <your-FO-name>
```

Then describe it.

Then program some verbs on it. Note that the verbs have to have the "`x`" permission flag set, so that they can be called from other verbs.

Then add help text. This can be done in either of two different ways. The first is to edit your feature object's `.help_msg` property. You should present each of the verbs on your FO that are intended for public use (as opposed to internal subroutines), give the syntax for the verb's usage, and a brief explanation of what the verb does. The other way is to put the documentation for each verb intended for public use as a set of comments at the top of the verb. The second is the officially preferred method (as per `help $feature`), but both will work.

THEN: In either case you must edit your FO's `.feature_verbs` property. If you put all the documentation in the help_msg property, then type:

```
;<your-FO>:set_feature_verbs({})
```

If you put the documentation for each public-use verb at the top of each verb, then type:

```
;<your-FO>:set_feature_verbs({"<verb1>", "<verb2>", ... ,
"<last-verb>"})
```

If you wish to restrict who may add your feature object, write a custom `:feature_ok` verb on it. This verb should return 0 if for whatever reason the person may not add the feature, or a truth value otherwise. An example of when this might come in handy might be a feature only for use by wizards.

See also `help features` and `help $feature`.

Built-In Functions

See the online help text or the programmer's manual for the specifics of each individual function – here's what's there:

The quintessential object-oriented function:

```
pass()
```

General operations applicable to all values:

```
typeof()                            toobj()
tostr()                             tofloat()
toliteral()                         equal()
toint()                             value_bytes()
tonum()                             value_hash()
```

Operations on Numbers:

```
random()                            max()
min()                               abs()
```

floatstr()
sqrt()
sin()
cos()
tan()
asin()
acos()
atan()
sinh()
cosh()
tanh()
exp()
log()
log10()
ceil()
floor()
trunc()

Operations on Strings:

length()
strsub()
index()
rindex()
strcmp()
decode_binary()
encode_binary()
match()
rmatch()
substitute()
crypt()
string_hash()
binary_hash()

Operations on Lists:

length()
is_member()
listinsert()
listappend()
listdelete()
listset()
setadd()
setremove()

Manipulating Objects:

chparent()
valid()
parent()
children()
object_bytes()
max_object()
move()
properties()
property_info()
set_property_info()
add_property()
delete_property()
is_clear_property()
clear_property()
verbs()
verb_info()
set_verb_info()
verb_args()
set_verb_args()
add_verb()
delete_verb()
verb_code()
set_verb_code()
disassemble()
players()
is_player()
set_player_flag()
connected_players()
connected_seconds()
idle_seconds()
notify()
buffered_output_length()
read()
force_input()

```
flush_input()                     connection_option()
output_delimiters()               open_network_connection()
boot_player()                     listen()
connection_name()                 unlisten()
set_connection_option()           listeners()
connection_options()
```

Operations Involving Times and Dates:

```
time()
ctime()
```

MOO-Code Evaluation and Task Manipulation:

```
raise()                           task_id()
call_function()                   suspend()
function_info()                   resume()
eval()                            queue_info()
set_task_perms()                  queued_tasks()
caller_perms()                    kill_task()
ticks_left()                      callers()
seconds_left()                    task_stack
```

Administrative Operations:

```
server_log()                      dump_database()
renumber()                        db_disk_size()
reset_max_object()                shutdown()
memory_usage()
```

$Utils

Some of the built-in functions are used frequently in everyday programming, some are used rarely, or only by wizards, or both. The MOO also provides a collection of utilities packages. Each utilities package has its own top-level help text, and each verb has more detailed help text. This list is just the $utils packages available in LambdaCore. A reference list of all the verbs on each is provided in Appendix B.

```
$wiz_utils                        $lock_utils
$math_utils, $trig_utils          $list_utils
$set_utils                        $command_utils
$seq_utils                        $code_utils
$gender_utils                     $building_utils
$time_utils                       $string_utils
$match_utils                      $generic_utils
$object_utils                     $quota_utils
```

```
$byte_quota_utils                  $matrix_utils
$object_quota_utils                $convert_utils
```