COOLMUD Programmer's Manual

For COOLMUD Version 2.0
September 1992

by Rusty Wright (aka "Gus")

(This document is heavily modified from the LambdaMOO manual by Pavel Curtis.)

Introduction

COOLMUD is a network-accessible, multi-user, programmable, interactive system designed for the construction of text-based adventure games, conferencing systems, and other collaborative software.

Participants (usually referred to as *players*) connect to COOLMUD using `telnet` or some other, more specialized, *client* program. Upon connecting, 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 COOLMUD.

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

The job of interpreting commands is shared between two major components in the COOLMUD 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 COOL programming language. The database contains representations of all objects in the virtual reality, including the COOL programs the server executes to give objects their specific behaviors.

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

In the next chapter we'll go over the structure and contents of a COOLMUD database. The following chapter gives a complete description of how the server performs its primary duty: parsing the commands typed by players. Next, we'll examine the syntax and semantics of the COOL programming language. Finally, we'll cover the database conventions assumed by the server.

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

1 The COOLMUD database

In this chapter we'll examine in detail the various kinds of data that can appear in a COOLMUD database and that, therefore, COOL programs can manipulate. In a few places, we'll refer to the `boot` database. This is just one particular COOLMUD database.

1.1 Values

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

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

The only *numbers* that COOL understands are 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 COOL programs, numbers are written just as you see them here, an optional minus sign followed by a sequence of decimal digits. In particular, you may not put commas, periods, or spaces in the middle of large numbers, as we sometimes do in natural languages (e.g., '2,147,483,647').

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."
```

COOL strings may not include special ASCII characters like carriage-return, line-feed, bell, etc.

Objects are the backbone of the COOL database and, as such, deserve a great deal of discussion; the next section is devoted to them. Every object has a number, unique to that object. In programs, we write a reference to a object by putting a hash mark ('`#`') followed by the object's number, like this:

```
#495
```

There is one special object number used for an error value; `#-1`.

COOLMUD allows servers to interconnect, and for objects to move between servers. A *visitor* object is specified just like a local object and is appended with an ampersand '`@`' and the name of the remote server:

```
#23@east
#13@unlucky
```

Errors are, by far, the least frequently used values in COOL. 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 an error message. 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 appropriate 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_DIV`	Division by zero
`E_FOR`	For variable not a list
`E_INTERNAL`	Internal error
`E_INVIND`	Invalid indirection
`E_MAXREC`	Maximum recursion exceeded
`E_MESSAGE`	Message unparseable
`E_METHODNF`	Method not found
`E_NONE`	No error
`E_OBJNF`	Object not found
`E_PERM`	Permission denied
`E_RANGE`	Range error
`E_SERVERDN`	Server down
`E_SERVERNF`	Server not found
`E_STACKOVR`	Stack overflow
`E_STACKUND`	Stack underflow
`E_TIMEOUT`	Timed out
`E_TYPE`	Type mismatch
`E_VARNF`	Variable not found

The final kind of value in COOL programs is *lists*. A list is a sequence of arbitrary COOL values, possibly including other lists. In programs, lists are written with each of the elements 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:

```
{"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 COOL: 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.

1.2 Objects

Objects are, in a sense, the whole point of the COOL programming language. They are used to represent objects in the virtual reality; for example, players, rooms, exits, and other concrete things.

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 `clone()` function described later, is required to bring an object into existence. Symmetrically, once created, an object continues to exist until is explicitly destroyed by the `destroy()` function (also described later).

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

Every object is made of four pieces that together define its behavior; its *parents*, *variables*, *methods*, and *verbs*.

1.2.1 Parents

Except for the root object ('`#1`') all objects have one or more parents. COOLMUD has multiple inheritance, so an object can have more than one "parallel" parent. When an object is created, it is cloned from some other object. The child object inherits all of the methods and variables from the parents of the object it was cloned from. The object it was cloned from isn't its parent, but it has the same parents as that object. After an object is cloned it can have its parents changed by either reprogramming the entire object or by calling the built-in `chparents()` function.

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 `boot` 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 may 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.

1.2.2 Object variables

An object *variable* is a named "slot" in an object that can hold an arbitrary COOL value. An object can have any number of variables, and which are declared to be of a certain type.

Objects appear to have variables corresponding to every variable in its parents' objects. To use the jargon of object-oriented programming, this is a kind of *inheritance*. If some parent object has a variable named `foo`, then it

appears that all of its children and thus its children's children, and so on have that variable. We say it "appears" to have all of its parents' variables because you don't have to declare any variables on a child object that are on its parents; when you ask for the value of any of these variables the COOLMUD server finds the variable on the nearest parent object and returns its value. But when an object changes the value of one of these variables, the object then gets its own permanent copy of the variable, which is then changed. This behavior is typically referred to as "copy-on-write."

An object may also have a new variable defined only on itself (and its descendants). For example, an object representing a rock might have variables indicating its weight, chemical composition, and/or pointiness, depending upon how the rock is used in the virtual reality.

Variables on objects can only be read or modified when there are methods that provide such access to the variables. For example, there are often methods on objects that provide simple "read" access for variables:

```
method name
    return name;
endmethod
```

For modifying variables, methods typically implement some permission check to see if the variable can be modified by the *caller*:

```
method set_name
    if (!(caller in owners))
        raise(E_PERM);
    endif
    name = args[1];
endmethod /* set_name */
```

In the above example the check is quite simple. Since methods implement the permission scheme, access is completely controlled by them. It is important to note that the COOLMUD server provides no "override" on variable access; even the wizards can be excluded access to a variable, which the above piece of code is an example of.

1.2.3 Methods

The other piece making up an object is its *methods*. A method is a named COOL program that is associated with a particular object. Methods are also used to implement commands that a player might type; for example, in the `boot` database, there is a method on all objects representing containers that implements commands of the form 'put *object* in *container*'. COOL methods can also invoke methods defined on objects. Some methods are designed to be used only from within COOL code; they do not correspond to any particular player command at all. Thus, methods in COOL are like the 'procedures' or 'functions' found in other programming languages.

Method variables

Methods can have their own variables. They are untyped and are local to the method; when the method finishes running, its variables cease to exist. Method variables are declared with the `var` declaration.

1.2.4 Verbs

In order for an object's method to be used as a command by players, the method must be "bound" to a *verb*. If a method is not bound to a verb it can't be accessed by players, only by COOL code. In a later section we'll go over how to bind a method to a verb.

When a method is run as a verb, any words following the verb are given to the method as arguments. For example, if object `xyz` has a `look` verb bound to the `look_verb` method, and a player types '`look at xyz with glass`' the `look_verb` method on the `xyz` object will be run with the arguments `at`, `xyz`, `with`, and `glass`. If there is another object in the room or carried by the player, named `glass`, with a a `look` verb, it may also be called, and with the same arguments. Since every object in the room or carried by the player with a `look` verb may get called, each object must check the arguments to see if they were the one the `look` was meant for. When an object's `look` verb determines that it's the object that matches, it returns `0` as its value to tell the server that no further `look` verbs on the other objects need to be called.

An object's `verb` binding can specify different words to invoke the method the verbs are bound to. For example, the words `poke` and `prod` can both be verbs bound to the `poke_verb` method. Then a player could type either '`poke xyz`' or '`prod xyz`'.

Verb bindings can also be set up so that in addition to the verb, another word must be typed as part of the command. Typically the second word is a preposition; for example, '`with`', '`in`', '`to`', '`from`', and so on. This allows you to set up commands like '`put money in jar`' and '`rub lamp with rag`'.

2 The COOL programming language

The COOL programming language is a relatively small and simple object-oriented language designed to be easy to learn for most non-programmers.

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

2.1 Comments

You can include bits of text in your COOL 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 add a comment you use 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 (#0.players)
  "Grendel eats peanut butter!";
  player:tell(x.name, " (", x, ")");
endfor
```

2.2 Expressions

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

```
3 + 4
```

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

2.2.1 Errors

Most kinds of expressions can be used improperly in some way. For example, the expression

```
3 / 0
```

is improper because it tries to divide by zero. In such cases, COOL "raises" an error value (`E_DIV` in this example), which causes the method's code to be aborted and a message to be printed on the player's screen.

2.2.2 Literals

The simplest kind of expression is a literal COOL 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"}
```

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}`'.

COOL also has some constants, which are returned by the `typeof()` built-in function:

```
NUM          OBJ          STR
LIST         ERR
```

Their meanings are as follows:

`NUM`	a number, the type code for numbers
`LIST`	a number, the type code for lists
`STR`	a number, the type code for strings
`OBJ`	a number, the type code for objects
`ERR`	a number, the type code for error values

2.2.3 Variables

As discussed earlier, it is possible to store values in variables on objects; the variables will keep those values forever, or until another value is put there. It's often useful to have a place to put a value for just the duration of the execution of a method; COOL provides method (local) variables for this purpose.

Method variables are named places to hold values; you can get and set the value in a given method variable as many times as you like. Method variables are temporary, though; they only last while a particular method is running; after it finishes, all of the method variables cease to exist and the values are forgotten. The method variables set in one method are not visible to the code of other methods. When a method begins executing, the method variables are initialized to 0.

The name for object and method variables is made up of letters, digits, and the underscore character ('`_`') and cannot 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 COOL, 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; it's value is the value of the named variable.

To change the value stored in a variable, use an *assignment* statement:

`variable = expression`

For example, to change the variable named '`x`' to have the value 17, you would write '`x = 17;`'. An assignment statement changes the value of of the named variable.

COOL also has some predefined pseudo-variables, they are read-only:

```
player      this          caller
args
```

Their values are as follows:

`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 method was found.

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

`args` usually a list, the arguments given to this method. The `parse` method on the player object gets the entire command line typed by the player, it hands it off to `call_verb`, which splits it into words, which is passed as a list of words to a method bound to the verb.

2.2.4 Arithmetic

All of the usual simple operations on numbers are available to COOL 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 % 2      ⇒  1
5 % -2     ⇒  1
-5 % 2     ⇒  -1
-5 % -2    ⇒  -1
-(5 + 2)   ⇒  -7
```

Note that division in COOL 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.

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 (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.

2.2.5 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
"foo" == "Foo"                          ⇒  1
#34 != #34                              ⇒  0
{1, #34, "foo"} == {1, #34, "FoO"}      ⇒  1
E_DIV == E_TYPE                         ⇒  0
3 != "foo"                              ⇒  1
```

Note that comparison of strings is case-insensitive; that is, it does not distinguish between the upper- and lower-case version of letters. To perform a case-sensitive comparison, use the `strcmp` function described later.

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
#34 >= #32       ⇒  1
"foo" <= "Boo"   ⇒  0
```

Note that, as with the equality operators, strings are compared case-insensitively. If the operands to these four comparison operators are of different types, or if they are lists, then `E_TYPE` is raised.

2.2.6 Conditional expressions

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

- all numbers other than zero
- all non-empty strings (i.e., other than '`""`')
- all non-empty lists (i.e., other than '`{}`')
- all non-negative object numbers. (Note that a negative object number doesn't necessarily mean that such an object exists.)

All other values are false:

- zero
- the empty string ('`""`')
- the empty list ('`{}`')
- all positive object numbers
- all error values

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

To negate the truth value of a COOL 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. COOL 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 '`||`' operator works similarly, except that *expression-2* is evaluated only if *expression-1* returns a false value.

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
```

2.2.7 Lists and strings

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}
```

Both strings and lists can be seen as ordered sequences of COOL 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"`'. COOL allows you to refer to the elements of lists and strings by number, 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.

Extracting an Element from a List or String

The indexing expression in COOL 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 a number (the *index*). If either of the expressions returns some other type of value, `E_TYPE` is raised. 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.

```
"fob"[2]              ⇒  "o"
"fob"[1]              ⇒  "f"
{#12, #23, #34}[3]    ⇒  #34
```

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

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

The three expressions are evaluated in order. *Expression-1* must return a list or string (the *sequence*) and the other two expressions must return numbers (the *low* and *high* indices, respectively); otherwise, `E_TYPE` is raised. 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. As the second and third forms show, you can leave off either the low or high index; you'll automatically get 1 if you leave off the low index, and the value of length of the sequence if you leave off the high index.

```
"foobar"[2..6]                    ⇒  "oobar"
"foobar"[2..]                     ⇒  "oobar"
"foobar"[3..3]                    ⇒  "o"
"foobar"[..3]                     ⇒  "foo"
"foobar"[17..12]                  ⇒  ""
{"one", "two", "three"}[1..2]     ⇒  {"one", "two"}
{"one", "two", "three"}[3..3]     ⇒  {"three"}
{"one", "two", "three"}[17..12]   ⇒  {}
```

Other operations on lists and strings

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

```
expression-1 in expression-2
```

Expression-2 must return a list or string, otherwise, `E_TYPE` is raised. If the value of *expression-1* is in that list or string, then the index of its first occurrence in the list or string 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
"bit" in "frobitz"                ⇒  4
```

Note that the membership operator is case-insensitive in comparing strings, just like the comparison operators. Note also that since it returns zero only if the given value is not in the given list or string, the `in` expression can be used either as a membership test or as an element or substring locator.

2.2.8 Calling built-in functions and other methods

COOL provides a number of functions for performing a variety of operations; a complete list, giving their names, arguments, and semantics, appears in a separate section later.

The syntax of a call to a built-in 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. Most functions require that certain of the arguments have certain specified types (e.g., the `lengthof()` function requires a list or a string as its argument); `E_TYPE` is raised if any argument has the wrong type.

Object methods can also call other methods, usually using this syntax:

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

or, if there aren't any arguments you can use either of the following 2 forms:

```
expr-0.name()
expr-0.name
```

Expr-0 must return an object number; `E_TYPE` is raised otherwise; if *expr-0* doesn't evaluate to an object value, `E_INVIND` is raised. If the object with that number does not exist, `E_OBJNF` is raised. If this task is too deeply nested in methods calling methods calling methods, then `E_MAXREC` is raised; the limit in COOLMUD at this writing is 50 levels. If neither the object nor any of its ancestors defines a method matching the given name, `E_METHODNF` is raised. Otherwise, if none of these things happens, the named method on the given object is called; the various built-in variables have the following initial values in the called method:

`this`	an object, the value of *expr-0*
`args`	a list, the values of *expr-1*, *expr-2*, etc.
`caller`	an object, the value of `this` in the calling method
`player`	an object, the same value as it had initially in the calling method.

Note that these are really pseudo-variables; they're read-only and you can't assign new values to them.

We said "usually" at the beginning of the previous paragraph because that syntax is used when the *name* follows the rules for allowed variable names. There is also a syntax allowing you to compute the name of the method:

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

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

2.2.9 Parentheses and operator precedence

As shown in a few examples above, COOL 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, COOL 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 bind more tightly 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, COOL would have grouped it as follows:

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

The table below gives the relative precedence of all of the COOL 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
&&
||
=
```

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.

2.3 Statements

Statements are COOL 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.

2.3.1 Simple statements

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

```
;
```

It doesn't do anything at all.

The next simplest statements are also some of the most common, the expression statement and the assignment statement:

```
expression;
var = expression;
```

For the expression statement, the given expression is evaluated and the resulting value is ignored. The typical expression for such statements is the method call. Of course, there's no use for such a statement unless the evaluation of *expression* has some side-effect, such as printing some text on someone's screen, etc. For the assignment statement, the variable gets the new value.

2.3.2 Conditional execution

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

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

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

Sometimes you'll want 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, you'll need 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. COOL 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 is as follows:

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

2.3.3 Iteration

COOL 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 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, you 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 generated. 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 = listappend(evens, n + 1);
endfor
```

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

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

The syntax for performing a set of statements once for each number in a given range is as follows:

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

The two expressions are evaluated and should return numbers; `E_TYPE` is raised otherwise. The *statements* are then executed, once for each integer 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 in question. For example, consider the following statements:

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

The value of the variable `evens` after executing these statements is the same as in the previous example, the list

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

The final kind of loop in COOL 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 statements have precisely the same effect as the loop just shown above:

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

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 numbers, it happens when *expression-1* returns a larger number than *expression-2*. Finally, for the `while` loop, it happens if the expression returns a false value the first time it is evaluated.

Inside either of the `for` or `while` iteration loops you can have a `break` or `continue` statement. The `break` statement causes execution of the `for` or `while` loop to end prematurely; execution continues with the first statement

after the `endfor` or `endwhile`. The `continue` statement causes all statements after it in the iteration loop to be skipped and execution continues with the next iteration of the loop. If you have `for` or `while` statements inside of other `for` or `while` statements you can specify which iteration loop should be broken out of by following `break` with a number specifying the loop level, where 1 means the current loop. Likewise, for the `continue` statement you can specify which iteration loop to to skip the rest of by following `continue` with a number specifying the loop level.

2.3.4 Returning a value from a method

The COOL program in a method is just a sequence of statements. Normally, when the method is called, those statements are simply executed in order and then the number 0 is returned as the value of the method-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 method is terminated immediately after evaluating the given *expression*, if any. The method-call expression that started the execution of this method then returns either the value of *expression* or the number 0, if no *expression* was provided.

2.3.5 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 `at` statement is intended for just such situations and has the following syntax:

```
at (expression)
  statements
endat
```

The `at` statement first executes the expression, which must return a number; call that number *n*. It then creates a new COOL *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 `at` statement was executed. The task executing the `at` statement immediately continues execution.

2.3.6 Errors

Statements do not return values, but some kinds of statements can be used improperly and thus generate errors. If such an error is generated in a

method that is not ignoring that particular error, then an error message is printed to the current player and the current command (or task, really) is aborted. If the method is ignoring that error then the error is ignored and the statement that generated it is simply skipped; execution proceeds with the next statement.

(Need to add stuff about `raise` *here as well.)*

2.4 Built-in functions

There are a number of built-in functions available to COOL programmers. Each one is discussed in detail in this section. The presentation is broken into subsections by grouping functions with similar or related uses.

For most functions, the expected types of the arguments are given; if the arguments are not of these types, `E_TYPE` is raised. Some arguments can be of any type; in such cases, no type specification is given for the argument. For most functions, the type of the result of the function is given. Some functions do not return a result; in such cases, the specification `void` is used. Some functions can return a result of any type, for them the specificaton `value` is used.

Most functions take a fixed number of 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.

2.4.1 Passing execution

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 COOL.

Often it is useful for a child object to define a method that *augments* the behavior of a method on its parent object. For example, in the `boot` database, the `DESCRIBED` object (which is an ancestor of most other objects) defines a method called `description` that simply returns the value of `description`; this method 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` method and then appending the sentence onto the end of that description. The function `pass()` is for such situations.

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

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

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

`value pass (`*arg*, ...`)` [Function]
`value pass (`*arg*, ...`) to object` [Function]
`pass` calls the method with the same name on the parent of the object who's method is running. The arguments given to `pass` are the ones given to the called method and the returned value of the called method is returned from the call to `pass`. The initial value of `this` in the called method is the same as in the calling method.

Since COOL provides for multiple inheritence, the second form of the `pass()` call can be used to specify which parent's method to call.

2.4.2 Type-checking and conversion

`num typeof (`*value*`)` [Function]
Takes any COOL value and returns a number representing the type of *value*. The result is the value of one of these built-in constants: `NUM`, `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 more readable than the latter.

`str tostr (`*value*`)` [Function]
Converts the given COOL value into a string and returns it.

```
tostr(17)                ⇒   "17"
tostr(#17)               ⇒   "#17"
tostr("foo")             ⇒   "foo"
tostr({1, 2})            ⇒   "{1, 2}"
tostr(E_PERM)            ⇒   "Permission denied"
```

`num tonum (`*value*`)` [Function]
Converts the given COOL value into a number and returns it. Object numbers are converted into the equivalent numbers, strings are parsed as the decimal encoding of a number, and errors are converted into numbers. `tonum()` raises `E_TYPE` if *value* is a list. If *value* is a string but the string does not contain a syntactically-correct number, then `tonum()` returns 0.

```
tonum(#34)          ⇒   34
tonum("34")         ⇒   34
tonum(" - 34  ")    ⇒   34
tonum(E_TYPE)       ⇒   1
```

Notice that when parsing digits, spaces are ignored.

`obj toobj (`*value*`)` [Function]
Converts the given COOL value into an object number and returns it. The conversions are very similar to those for `tonum()` 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
```

`err toerr` (*value*) [Function]
Converts the given COOL value into an error value and returns that error value.

2.4.3 Operations on strings

`list explode` (*str* ***string*** [, *str* ***string***]) [Function]
Break *string* into a list of strings. By default, explode breaks on spaces; the optional second argument is the character to break on.

`num lengthof` (*str* ***string***) [Function]
Returns the number of characters in *string*. It is also permissible to pass a list to `lengthof()`; see the description in the next section.

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

`str crypt` (*str* ***text*** [, *str* ***salt***]) [Function]
Encrypts the given *text* using the standard UNIX encryption method. If provided, *salt* should be a two-character string used for the extra encryption "salt" in the algorithm. If *salt* is not provided, a random pair of characters is used. The salt used is also returned as the first two characters of the encrypted string.

Aside from the possibly-random selection of the salt, the encryption algorithm is deterministic. You can test whether or not a given string is the same as the one used to produced a given piece of encrypted text; 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, you've got a match.

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

`list match` (*str* ***subject***, *str* ***pattern*** [, ***token***]) [Function]
`list match_full` (*str* ***subject***, *str* ***pattern***, [, ***token***]) [Function]
Looks for *pattern* as a substring of *subject*, where *pattern* must start on a word boundary. Word are separated by spaces, or by *token* if given. Returns 1 if a match was found, 0 if not.

```
match("foo bar baz", "foo")                  ⇒ 1
match("foo bar baz", "f")                    ⇒ 1
match("foo bar baz", "o")                    ⇒ 0
match("large green monster", "green")        ⇒ 1
```

```
match("large green monster", "gre")          ⇒ 1
match("large*green*monster", "monster", "*") ⇒ 1
```

`match_full` is the same as `match`, except that *pattern* must match a full word within *subject*. (Useful for TinyMUD-style exit matching.)

```
match_full("foo bar baz", "foo")              ⇒ 1
match_full("foo bar baz", "f")                ⇒ 0
match_full("out;back;exit;leave", "out", ";") ⇒ 1
match_full("out;back;exit;leave", "ou", ";")  ⇒ 0
```

2.4.4 Operations on lists

`num lengthof` (*list* `list`) [Function]

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

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

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

If *index* is not provided, then `listappend()` adds the *value* at the end of the list and `listinsert()` adds it at the beginning.

```
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}
```

`list listdelete` (*list* `list`, *num* `index`) [Function]

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

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

`list listassign` (*list* `list`, `value`, *num* `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)]`', `E_RANGE` is raised.

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

`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; *list* is treated as a mathematical set. `setadd()` only adds *value* if it is not already an element of *list*. *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}
```

2.4.5 Operations on objects

`obj clone` () [Function]
Clone the current object. A new object is created, whose parent is the current object. Returns the object ID of the new object. If the current object no longer exists (ie., has been destroyed), '`#-1`' is returned.

`void destroy` () [Function]
Destroy the current object. The object itself is responsible for cleaning up any references to itself prior to this call. This might include removing any contained objects, re-parenting or destroying any instances of it, etc.

`void chparents` (*list* ***list***) [Function]

`void call_verb` (*str* ***string***) [Function]
`call_verb` isn't a function, it's a special method; when an object receives the `call_verb` message, the server intercepts it and calls the appropriate verb. The argument should be the command string to be parsed, which is then matched against each verb on the object. If a match is found, the associated method is called, with the parsed results in `args`. (`args[1]` ≡ `verb`, `args[2]` ≡ `dobj`, `args[3]` ≡ `prep`, `args[4]` ≡ `iobj`).

`void lock` (*str* ***string***) [Function]
This function is used to lock an object, to prevent another execution stream from modifying the object before the current stream is finished with it (see the section on locking). The argument is an arbitrary string, the name of the lock to place on the object. Locks placed by an execution thread remain in effect until a corresponding `unlock()` call, or until the thread terminates.

`void rm_verb` (*str* ***verbname***) [Function]
Removes the first verb named *verbname* from the current object. The argument may also be a string representing the number indexing the verb to be removed (starting at 0). eg., '`rm_verb("3")`' would remove the 4th verb.

`void rm_method` (*str* ***methodname***) [Function]
Removes the indicated method from the current object. Note that COOLMUD has special provision to allow a method to remove itself and continue executing. It won't be actually destroyed until the method finishes.

`void rm_var` (*str* ***variablename***) [Function]
Removes the indicated variable from the current object.

`void unlock` (*str* ***string***) [Function]
Removes the indicated lock from the current object. If any execution threads are waiting for this lock to be removed, they will execute.

`void add_verb` (*str* ***verbname***, *str* ***preposition***, *str* ***methodname***) [Function]
Adds a verb to the current object. The first argument is the name of the verb. The second argument is the preposition, or '`""`' for none. The third argument is the name of the method to call in the current object when the verb gets triggered. The verb is added to the end of the object's verb list, unless a verb with the same name and no preposition exists, in which case it is inserted before that verb. This prevents a verb with no preposition masking one with a preposition.

`void setvar` (*str* ***string***, ***value***) [Function]
Sets a variable, specified in *string*, on the current object to *value*. `E_VARNF` is raised if the variable doesn't exist, and `E_TYPE` is raised if there's a type mismatch (either between an existing variable, or an inherited one).

`list verbs` () [Function]
Returns a list of verbs on the current object. Each element of the list is a 3-element list, consisting of 3 strings: the verb name, the preposition, and the method to call.

`list vars` () [Function]
Returns a list of variables on the current object. Each element of the list is a string containing the name of the variable.

`value getvar` (*str* ***variablename***) [Function]
Gets the value of the indicated variable on the current object. This allows the use of an arbitrary string to get the value of a variable. (eg., '`getvar("abc" + "def")`')

`list methods` () [Function]
Returns a list of methods on the current object. Each element of the list is a string containing the name of the method.

`num hasparent` (*obj* ***object***) [Function]
Returns a positive value if the current object has *object* as a parent. This function looks recursively on all parents of the current object, so it will return 1 if the object has *object* as a parent anywhere in its inheritance tree, and 0 otherwise.

`str spew_method` (*str* ***methodname***) [Function]
Returns a string containing the internal stack-machine code for method *methodname*. This code is pretty unintelligible unless your brain works in RPN. Even then, some instructions are hard to figure out, and there's not much point. Only for the habitually curious.

`str list_method` (*str* ***methodname*** [, *num* ***lineno*** [, *num* ***fullbrackets*** [, *num* ***indent***]]]) [Function]
Returns a string containing the decompiled code for method *methodname*. This works by turning the stack machine code back into readable form. It does automatic indentation, line numbering, and smart bracketing (ie., it will use the minimum number of brackets when decompiling an expression). The three optional arguments are numeric arguments which control the decompilation:

lineno Turns line numbering on and off.

fullbrackets
When on, dumb bracketing will be used in every expression. Default is off, or smart bracketing.

indent The number of spaces to use in indenting the code.

`void echo` (*str* ***string***) [Function]
Display *string* to the current object, a player.

`void quit` () [Function]
Disconnect the current object, a player.

`void program` ([*obj* ***object***, *str* ***methodname***]) [Function]
Enter programming mode. This sets a flag on the player's descriptor such that all input from the player is diverted to a temporary file. When the player enters '`.`', the file is compiled, and then erased. There can either be no arguments, in which case the server expects a series of objects, or two arguments, which should be the object and method to program. In either case, the server currently uses a built-in set of permissions checks to determine whether the player may reprogram that object: either they must be in the object's `owners` list, or in `SYS_OBJ.wizards`.

`num serverof` (*obj* ***object***) [Function]
Returns a number representing the server ID of *object*. This ID is used internally by the server, and has no meaning except that ID zero is the local MUD. So the statement

```
if (!serverof(obj))
    ...
endif
```

would evaluate to true if *object* is a local object.

`str servername` (*obj* ***object***) [Function]
Returns a string representing the server name part of *object*.

2.4.6 Miscellaneous operations

`num random` (*num* ***n***) [Function]
Returns a random value between 1 and *n*.

`num time` () [Function]
Returns the current time, represented as the number of seconds that have elapsed since midnight on 1 January 1970, Greenwich Mean Time.

2.4.7 System functions

`void shutdown` () [Function]
Shuts down the MUD. The database is written, remote servers disconnected, and the COOLMUD process terminates.

`void dump` () [Function]
Syncs the cache to the database so that the database on disk is current.

`void writelog` (*str* ***string***) [Function]
Writes *string* to the logfile, prepended by a timestamp.

`num checkmem` () [Function]
Returns a string showing the amount of memory dynamically allocated, and how many chunks it was allocated in. If the server was not compiled with `-DCHECKMEM`, this function will return '`"Memory checking disabled."`'

2.5 Syntax for object code

The syntax for the code of an object is as follows:

```
object objectname
    parent declarations
    verb declarations
    variable declarations
    method declarations
endobject
```

The syntax for an object name is the same as for variables, given above.

2.5.1 Parent declarations

The syntax for the parent declarations is as follows:

```
parents parent-1 , ... parent-n ;
```

2.5.2 Verb declarations

To bind a verb to a method you use the `verb` declaration:

```
verb string = method ;
verb string : string = method ;
```

2.5.3 Variable declarations

The syntax for the variable declarations is:

```
vartype var-1 , ... var-N ;
```

Where *vartype* is one of `num`, `str`, `list`, or `obj`. You can have several lines of variable declarations, one for each different type, and you don't have to have variables of the same type all declared on the same line; you can have several variable declaration lines for the same type.

2.5.4 Method declarations

Method declarations look similar to object code:

```
method methodname
    var local variable declarations
    ignore errors
endmethod
```

3 Differences between COOL and MOO

LambdaMOO objects consist of attributes, properties, and verbs. COOLMUD objects consist of variables and methods; there are no attributes.

COOLMUD object variables and methods are similar to LambdaMOO properties and verbs. With LambdaMOO, all properties can be accessed by other objects, as long as the permissions allow it, which they generally do except for special properties that need to be hidden. With LambdaMOO properties have an owner. With COOLMUD, object variables can only be accessed if there is a method that provides acces, otherwise the object variable is inaccessible. COOLMUD object variables don't have an owner, just the owners of the object. With COOLMUD the object variables' methods that provide access to them also completely control any permission scheme.

COOLMUD methods don't have a "debug" bit, methods can `ignore` specific errors if they want to.

With COOLMUD command parsing is much more controlled by the objects. For the sake of example, let's ignore prepositions. When a palyer types a command, some simple matching is done; all objects that have that "verb" defined on them have the method that's bound to that verb called. The method is responsible for checking the arguments to see if they match its object; e.g., `args[2]` is typically the object and `args[1]` is the verb. The method returns 1 to signify that the arguments didn't match for it and for the parser to continue calling methods on other objects. The method returns 0 to specify that it was the desired object and the parser stops calling methods on the rest of the objects.

With COOLMUD, verbs are "bound" to methods. Unless a method is bound to a verb, it can't be accessed by a player. With LambdaMOO there is a "template" specified for the arguments when creating a verb and the template '`this none this`' is typically used to specify a verb that isn't to be accessed as a command typed by a player; that is, the verb will be used as a subroutine. With COOLMUD you simply don't bind the method to a verb if you want it only used as a subroutine.

COOLMUD treats assignments as statements, not expressions. This means that you can't do looping constructs like

```
while ((var = name.method) != someval)
    ...
endwhile
```

4 Setting up a new COOLMUD

(explain format of .cfg file.)

4.1 Interconnecting COOLMUDs

Function Index

V

W

Variable Index

Concept Index

Table of Contents