Introduction to the COOL Programming Language

Stephen F. White

October 28, 1994

1 Datatypes

There are six datatypes in COOL: string (**str**), number (**num**), object (**obj**), list (**list**), error (**err**) and mapping (**map**).

The only compile-time checking done by COOL is to ensure that any instance variables are initialized only from the correct constants. Other than that, all type checking is done at run-time, with the exception `E_TYPE` being raised if an error occurs in an operator or system function.

1.1 String (**str**)

Strings are enclosed in doublequotes (`"`). The following are examples of string constants:

```
"foo"
"The rain in spain.\n"
"They call me \"The Woodmaster\", son."
```

String instance variables are declared as follows:

```
str foo;
str bar, baz;
str zip = "The rain in spain.\n";
```

1.2 Number (**num**)

Numbers are long integers (typically in the range -23̂1 to 23̂1 - 1).

```
num a, n = -1034;
```

1.3 Object ID (**obj**)

Object ID's consist of two parts: the ID, and the servername. The following are values of type OBJ:

```
#5@joemud
#10@fredmud
```

If the object in question is on the local MUD, the server part may be omitted:

```
#7
```

represents object #7 on the local MUD. The value #-1 is a special value, usually meaning "nothing", "nowhere", or some kind of error condition.

1.4 List (**list**)

Lists are heterogenous, ordered collections of other datatypes. They can be manipulated as unordered sets, using the setadd() and setremove() functions, or as ordered lists, using listinsert(), listappend(), listassign(). and listdelete().

Typically lists are used to store things like the contents of a room (a list of OBJ), or a list of methods on an object (a list of STR).

The elements of a list are enclosed by braces, and separated by commas. The following are examples of lists:

`{}`	(the empty list)
`{1, 2, 3}`	(a list of numbers)
`{"abc", "def", "ghi"}`	(a list of strings)
`{ {1, 2}, {3, 4}, {5, 6} }`	(a list of lists)
`{1, "abc", #3}`	(a heterogenous list)

The + and − operators are overloaded for lists to perform the `setadd()` and `setremove()` functions, respectively.

1.5 Error (**err**)

Error values store the result of an operation. The following is a list of the current errors and their meanings:

Error Symbol	Description
E_TYPE	Type mismatch
E_ARGTYPE	Argument type mismatch
E_NARGS	Incorrect number of arguments
E_RANGE	Range error
E_INVIND	Invalid indirection
E_DIV	Division by zero
E_MAXREC	Maximum recursion exceeded
E_METHODNF	Method not found
E_VARNF	Variable not found
E_FOR	For variable not a list
E_SERVERNF	Server not found
E_SERVERDN	Server down
E_OBJNF	Object not found
E_MESSAGE	Message unparseable
E_TIMEOUT	Timed out
E_STACKOVR	Stack overflow
E_STACKUND	Stack underflow
E_PERM	Permission denied
E_INTERNAL	Internal error
E_FILE	File not found
E_TICKS	Task ran out of ticks
E_TERM	Task terminated
E_MAPNF	Mapping not found

1.6 Mapping (**map**)

Mappings are arrays which can be indexed by values of any type. The most common use is indexing by strings, for use as associative arrays. Each element of a mapping consists of two parts: the key, and the data. Indexing of mappings is done in exactly the same way as lists:

```
map a;

a["name"] = "stephen";
a["cpus"] = 11;
```

Now indexing on `name` or `cpus` will return the values assigned:

```
a["name"] => "stephen"
a["cpus"] => 11
a["foobar"] => E_MAPNF
```

1.6.1 Initializing mappings

Mappings may be initialized using a constant mapping expression, which is of the form:

`[` *key1* `=>` *value1*`,` *key2* `=>` *value2*`,` ... `]`

For example, the above assignments could be written as:

```
map a;

a = ["name" => "stephen", "cpus" => 11];
```

Of course, indexing on values of other types is possible also:

```
a = [#3 => "Object Number Three", E_RANGE => "Value
out of Range"];
```

You can even index on other associative arrays, but that's just perverse!

2 Variables

There are two kinds of variables: method variables and object variables. Method variables are declared in methods, and are temporary. They are created for the execution of the method, and are freed when execution is complete. Object variables are permanent, and are used to store the state of an object (its name, location, etc). Object variables are inherited by instances of an object.

Method variables are sometimes referred to as "local", and object variables as "global".

An object's variables may only be assigned by that object's methods. The only way to retrieve or modify another object's variables is to send a message to that object. For example, the "location" message says, "give me your location", and the "moveto" message says, "move yourself to the following location".

Variables may be fixed type, in that they can store one type of value, or variable, so that they can store any. Assigning the wrong type of value to a fixed-type variable raises `E_TYPE`, a type mismatch.

Currently, object variables can only be fixed-type, and method variables can only be variable type. This may change in the future. Both types of variables may be initialized in their declaration.

2.1 Examples

```
str name;
```

declares a fixed-type string variable called "name".

```
num contents;
```

declares a fixed-type list variable called "contents".

```
var a, b;
```

declares a local variables "a" and "b" which may hold values of any type.

2.2 Built-In Variables

There are a number of tokens in COOL which act like built-in variables. However, they may not be assigned to. The following table lists them.

Variable	Type	Value
`this`	`obj`	Object on which the current thread is acting
`player`	`obj`	Player object which initiated this thead
`caller`	`obj`	Object which called the current method
`args`	`list`	Arguments to the current method

3 Methods

3.1 Declaring Methods

Methods consist of a `method` declaration, variable and exception declarations, statements, and an `endmethod` declaration.

The simplest method of all is the null method:

```
method foo
endmethod
```

Methods describe the action to be taken when an object receives a certain message. If a message is sent to an object for which it does not have a corresponding method, `E_METHODNF` is raised. Methods may be called by any other object, so any permissions checking must be done by the method itself.

```
method add
  return args[1] + args[2];
endmethod
```

3.2 Blocked Methods

Sometimes it is desired that a method should be declared on an object, and non-overrideable by any of its children. The keyword `blocked` may be used in a method declaration to declare such a method.

```
blocked method foo
  ...
endmethod
```

Thus, any children of this object would have the "foo" method, but any "method foo" declared on those children would be ignored.

4 Message Passing

Passing a message from one object to another is the only way in COOLMUD for objects to interact. Objects may not retrieve other objects' data, except by passing a message. Similarly, objects may not set other objects' properties, except by passing a message.

4.1 Syntax

Message passing is accomplished with the '`.`' operator:

```
#3@joemud.foo(1, 2, 3);
```

passes the message 'foo' to object #3 on joemud, with the numeric arguments 1, 2, 3.

```
var a;
a = #3.location();
```

sends the message 'location' to #3 on the local mud, and stores the result in temporary variable a. If there are no arguments, the brackets may be omitted, like so:

```
var a;
a = #3.location;
```

Note: When no arguments are required, it's a good idea to use brackets to indicate the type of message being passed. If the message will change the remote object or acts like a function, use brackets. If the message will only retrieve data, use no brackets.

5 Verbs

Verb declarations are a simple way of attaching player commands to objects. Verb declarations must appear in the declarations section of an object, before any method declarations. A verb declaration is like a template, which reads from left to right. On the left is the pattern to be matched, on the right is the method called if there is a match. For example,

```
verb "look" = look;
```

This defines a mapping between the command "look" and the "look" method on the object. When the object is checked for commands, the command "look" will cause the method named "look" on the object to be called. Any arguments may follow the verb in this example, so the commands "look fred", "look", and "look at the pretty flowers" would all match. If further parsing is required, prepositions may be used (see below). If only a verb is specified, only the first word of the command will be checked; checking the arguments must be performed by the associated method.

5.1 Verb Aliasing

Multiple verbs may be listed in the same declaration, separated by spaces:

```
verb "look examine" = look;
```

These act as aliases, so either "look" or "examine" will call the "look" method.

5.2 Abbreviations

The asterisk character, *, may be used to indicate abbreviations:

```
verb "l*ook" = look;
```

This will match the commands "l", "lo", "loo", and "look".

The characters up to the asterisk *must* be typed in order for the command to match. Characters after the asterisk are optional. For example,

```
verb "exa*mine" = examine;
```

would match "exa", "exam", etc., but not "ex" or "e".

5.3 Prepositions

In addition the the verb name, a preposition may be supplied:

```
verb "hit" :  "with" = hit;
```

Here, the colon separates the verb and preposition. Only commands containing both the verb and that preposition in them will be matched: “hit joe with sledghammer” and “hit with book” would match; “hit” by itself would not. Multiple prepositions may also be used:

```
verb "hit smack" :  "with using" = hit;
```

Commands matching this template include: “hit using sword” “smack troll with emacs source", “smack with fred”, etc. No abbreviations may be used for prepositions.

Any word or character may be used as a preposition, so emulating TinyMUD is possible:

```
verb "@desc*ribe" :  "=" = describe;
```

This hard-to-read declaration uses "=" as the preposition, so the command

```
@desc me = groovy
```

would match. Note that spaces around the "=" are required, however. A better way to do this would be to modify the “parse” method on the PLAYER object, but that’s beyond the scope of this humble paragraph.

5.4 Inheritance of Verbs

All verbs declared by an object are inherited by instances of that object. For example:

```
object FOO
  verb "hit" = hit;
  method hit
    ...
  endmethod
endobject
object BAR
  parents FOO;
endobject
```

Now both FOO and BAR have "hit" verb available.

It is important to note that verb and method inheritance are separate, and both start from the instances and move up. When a verb matches, the method is passed to the child, even if the verb was declared on the parent. If the child object redefines the method, that method will be used instead. In the example above, if BAR defined a "hit" method, it would be used when BAR was "hit". Redeclaring the verb isn't necessary.

5.5 The Method Part

We've seen how a verb template is set up, but so far no action can be taken because we don't have a method to be called. When writing a method to be used as a verb, certain conditions apply.

5.5.1 Verb Arguments

When a method is called as a verb, the arguments to the verb are passed as strings in the "args" variable, as follows:

	Without Prep	With Prep
`args[1]`	verb	verb
`args[2]`	direct object	direct object
`args[3]`		preposition
`args[4]`		indirect object

The parser itself does no matching of dobj or iobj. The method itself is responsible for ensuring that the arguments specified refer to the correct object. For example, consider a "button" object with the verb declaration:

```
verb "press" = press;
```

This declaration matches "press button", but also "press", "press nancy", etc. In order to make sure the command refers to the button, we must explicitly match args[2], the direct object:

```
method press /* verb */
  if (!this.match(args[2])) /* not this object */
    return 1; /* abort */
  endif
```

```
  ...
endmethod
```

Here we are assuming that the object has a method called "match", which returns 1 if the argument matches the object's name. If the direct object doesn't match the object's name, the verb is exited with numeric value "1". The return value from a verb method has a special meaning. Returning a non-zero value indicates to the parser that no match was found, and the parser should continue to look for verbs on this and other objects. Returning zero means that the match was successful, and no further parsing should be done.

NOTE: It is a good idea to comment the "method" declaration of a method which is being used as a verb, to remind yourself of the special conditions which apply to writing a verb (arguments, return value).

6 Control Flow

6.1 `if` Statement

The `if` statement is the conditional for COOL and whose simplest form is:

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

If *expression* evaluates true, *statements* are executed. Note that both the parentheses around the condition and the `endif` are mandatory, and there is no 'then' after the `if`. `if`s may be nested infinitely.

6.1.1 Conditions

In addition to numeric 1 and 0, the following values may be used in the *expression* above.

Type	Condition for "true"
`NUM`	Non-zero
`STR`	Non-empty
`LIST`	Non-empty
`MAP`	Non-empty
`OBJ`	Positive (ie., not #-1)
`ERR`	(All error values are FALSE)

Note that the objects referenced by `OBJ` values may not actually exist, but as long they are positive and have a valid server id, COOL will treat them as "true".

`&&` and `||` are the boolean 'and' and 'or' operators. `!` is the 'not' operator. They may be used to form boolean conditions such as:

```
(a == 5 && b != 0)
(player.location == location && !player.dead)
(s1 || !s2)
```

The boolean operators have the same precedence as C. They also "short-circuit" in the same way that that they do in C. The first false condition in an `&&` expression will return a false result, and will cause short-circuit execution of the rest of the statement. The first true condition in a `||` expression will make the result true, and will short-circuit execution of the rest of the expression.

6.1.2 `else` Statement

The `else` construct allows a programmer to specify a set of statements to be executed if the condition in an `if` statement evaluates false.

```
if ( expression )
  statements1
else
  statements2
endif
```

If *expression* evaluates true, *statements1* are executed. Otherwise, *statements2* are executed.

6.1.3 `elseif` Statement

The `elseif` statement may be used to test a series of conditions, without requiring another level of `if/endif` pairs. Its function is mostly cosmetic.

```
if ( expression1 )
  statements1
elseif ( expression2 )
  statements2
else
  statements3
endif
```

If *expression1* is true, *statments1* are executed. Otherwise, if *expression2* is true, *statements2* are executed. Otherwise, *statements3* are executed. For example, the code:

```
if (a == "who")
  c = 1;
else
  if (a == "what")
    c = 2;
  else
    if (a == "where")
      c = 3;
    else
      c = 0;
    endif
  endif
endif
```

could be instead written as:

```
if (a == "who")
  c = 1;
elseif (a == "what")
  c = 2;
elseif (a == "where")
  c = 3;
else
  c = 0;
endif
```

6.2 `for` Statement

The `for` statement allows the programmer to traverse a list of values. It comes in two flavours. The first flavour is for iterating over elements of a given list or string, and the second for iterating of values in a given range. `for` statements may be nested infinitely.

6.2.1 Iterating over a given list or string

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

This construct sets *variable* to each element in *expression* in turn, and executes *statements* for each one. *expression* must be an expression whose value is a list or a string. If not, `E_FOR` is raised.

6.2.2 Iterating over a list

Examples:

```
for a in ( {1, 2, 3} )
  player.tell(tostr(a));
endfor
```

would set `a` to 1, 2, and 3 in turn, and echo the result to the player.

```
for item in ( player.contents )
  player.tell( item.name );
endfor
```

would set `item` to each element of `player.contents` and echo the name of each object to the player (assuming that “contents”, “name” and “tell” methods have been defined on the objects in question).

6.2.3 Iterating over a string

Example:

```
for c in ("abcde")
  ...
endfor
```

would set c to "a", "b", "c", "d" and "e" in turn within the body of the loop.

6.2.4 Iterating over a range of values

This flavour sets a variable to an increasing range of numeric values:

`for` *variable* `in [` *expression1* `..`*expression2* `]`
  statements
`endfor`

This construct sets *variable* to each value in the given range, from *expression1* to *expression2* . Both expressions must be integers, or `E_TYPE` is raised. For example,

```
for n in [1..5]
  ...
endfor
```

would set `n` to 1, 2, 3, 4, and 5. If *expression2* is less than *expression1*, the loop will not execute at all.

6.3 `while` Statement

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

This construct executes *statements* while *expression* evaluates true. For example:

```
var a;
a = 5;
while (a > 0)
  echo(tostr(a));
  a = a - 1;
endwhile
```

6.4 `do/while` Statement

<pre>
do
  statements
while (expression);
</pre>

Same as the while statement, except that *expression* is tested at the end of the loop, instead of the beginning. The loop is thus always executed at least once.

6.5 `return` Statement

<pre>
return [expression] ;
</pre>

The `return` statement returns *expression* to the calling function, and exits the current verb. Returning a value to the command parser does nothing.

6.6 `break` Statement

<pre>
break [num] ;
</pre>

The `break` statement is used to exit from a loop, within the body of the loop. The optional numeric constant indicates how many loops to break out of, in nested loops.

6.7 `continue` Statement

<pre>
continue [num] ;
</pre>

The `continue` statement is used to break out of the current iteration of a loop and start again at the top. The optional numeric constant indicates which loop to restart (the default is 1).

7 Built-in Functions

COOLMUD has a number of built-in functions. These functions can be divided into roughly 7 groups: functions for manipulating objects, dealing with players, miscellanous functions, thread functions, list manipulation functions, conversion functions, and wizard functions. The following is a summary of the functions, followed by a more detailed description of each.

Function	Description
`clone()`	Make an instance of `this`
`destroy()`	Destroy `this`
`chparents(`*list*`)`	Change the inheritance of `this`
`lock(`*str*`)`	Place a named lock on `this`
`add_verb(`*verb*`,`*prep*`,`*method*`)`	Add a verb to `this`
`rm_verb(`*str*`)`	Remove a verb from `this`
`rm_method(`*str*`)`	Remove a method from `this`
`rm_var(`*str*`)`	Remove a variable from `this`
`unlock(`*str*`)`	Remove a named lock from `this`
`verbs()`	Return a list of verbs on `this`
`vars()`	Return a list of variables on `this`
`methods()`	Return a list of methods on `this`
`getvar(`*str*`)`	Get a variable on `this`, by name
`setvar(`*str*`,`*value*`)`	Set a variable on `this`, by name
`hasparent(`*obj*`)`	Does `this` have *obj* as a parent?
`find_method(`*str*`)`	Find a method on `this` or ancestor
`spew_method(`*str*`)`	Spew internal stack-machine code
`list_method(`*str*`, ...)`	Decompile the method *str* to source
`decompile()`	Decompile the object `this` to source
`objsize()`	Return the size (in bytes) of `this`
`echo(`*str*`)`	Display *str* to `this` (player)
`echo_file(`*str*`)`	Display a local file to `this` (player)
`quit()`	Disconnect `this` (player)
`program(`[*obj*`,`*method*]`)`	Enter programming mode
`typeof(`*var*`)`	Get the type of a value
`lengthof(`*var*`)`	Get the length of a list or string
`serverof(`*obj*`)`	Get the server # of an *obj* value
`servername(`*obj*`)`	Get the server name of an *obj* value
`servers()`	Return the list of known servers
`explode(`*str* [`,`*sep*]`)`	Break a string into a list of strings
`time()`	Get the current time & date
`ctime(`[*num*]`)`	Convert numeric time to ASCII string
`crypt(`*str* [`,`*salt*]`)`	Encrypt a string, with optional salt
`match(`*template*`,`*str* [`,`*sep*]`)`	Match a string to a template (prefix)
`match_full(`*template*`,`*str* [`,`*sep*]`)`	Match a string to a template (full)
`psub()`	Perform %-variable substitutions
`strsub(`*with*`,`*what*`,`*str*`)`	Perform string substitution
`pad(`*str*`,`*length* [`,`*padchar*]`)`	Pad/truncate a string
`random(`*num*`)`	Return a random number 1 ..*num*
`compile(`*str*`,`[*obj*`,`*method*]`)`	Compile *str* into an object or method

Function	Description
`sleep(`*num*`)`	Pause for *num* seconds
`kill(`*num*`)`	Terminate thread *num*
`ps()`	Get a list of active threads
`setadd(`*list*`,`*value*`)`	Add a value to a set, no duplicates
`setremove(`*list*`,`*value*`)`	Remove a value from a set
`listinsert(`*list*`,`*value* [`,`*pos*]`)`	Insert a value into a list
`listappend(`*list*`,`*value* [`,`*pos*]`)`	Append a value to a list
`listassign(`*list*`,`*value*`,`*pos*`)`	Assign a value to an element of a list
`listdelete(`*list*`,`*pos*`)`	Delete a value from a list
`tonum(`*var*`)`	Convert a value to number type
`toobj(`*var*`)`	Convert a value to object type
`tostr(`*var*`)`	Convert a value to string
`toerr(`*var*`)`	Convert a value to error type
`shutdown()`	Shut down the MUD
`dump()`	Dump the database
`writelog(`*str*`)`	Write *str* to the logfile
`checkmem()`	Show memory usage
`cache_stats()`	Show memory cache statistics

7.1 Object Functions

All functions which perform operations on objects refer to the current object, `this`. There is no way to directly modify or get information about a different object. This is done to ensure that an object's methods will work across inter-mud links (see "distrib"). Also, it serves as a permissions system: to clone an object, for example, you must ask the object to clone itself by sending it a message.

7.1.1 `clone()`

Clone the current object. A new object is created, whose parent is the current object. The new object's init method is called. Return value: The object ID of the new object. If the current object no longer exists (ie., has been destroyed), #-1 is returned.

7.1.2 `destroy()`

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.

7.1.3 `chparents(`*list*`)`

Change the parents of the current object to those specified in *list*. All variables and methods on the object itself remain intact, however any variables or methods it inherited from its old parents parents it may not inherit from the new. *list* must be a non-empty list, and must not cause any loops in the inheritance hierarchy (eg., an object may not have itself or one of its children as a parent). Any children of the current object will also have their inheritance changed by this call, such that the new parents specified in the list will be ancestors of the children as well.

7.1.4 `lock(`*str*`)`

This function is used to lock an object, to prevent another execution thread from modifying the object before the current thread is finished with it (see "locking"). The argument *str* is 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 method terminates.

7.1.5 `add_verb(`*verb*`,`*prep*`,`*method*`)`

Add a verb to the current object. *verb* is the name of the verb to add. *prep* is the preposition, or `""` for none. *method* 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 is to prevent a verb with no preposition masking one with a preposition.

7.1.6 `rm_verb(`*str*`)`

Remove the first verb named *str* from the current object. The argument may also be a string representing the number indexing the verb to be removed (indexed from 0). eg.,

```
rm_verb("3");
```

would remove the 4th verb on the current object.

7.1.7 `rm_method(`*str*`)`

Remove *str* 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 after the method finishes.

7.1.8 `rm_var(`*str*`)`

Remove the variable (property) named *str* from the current object.

7.1.9 `unlock(`*str*`)`

Remove the lock named *str* from the current object. If any execution threads are waiting for this lock to be removed, they will execute.

7.1.10 `verbs()`

Return a list of verbs on the current object. Each element of the list is in turn a 3-element list, consisting of 3 strings: the verb name, the preposition, and the method to call.

7.1.11 `vars()`

Return a list of variables (properties) on the current object. Each element of the list is a string containing the name of the variable.

7.1.12 `methods()`

Return a list of methods on the current object. Each element of the list is a string containing the name of the method.

7.1.13 `getvar(`*str*`)`

Gets the value of the variable named *str* on the current object. Normally, one would just reference the variable in COOL code by name, but `getvar()` allows the use of an arbitrary string to get the value of a variable. Example:

```
getvar ("abc" + "def")
```

would return the value of the variable named `abcdef` on the current object.

7.1.14 `setvar(`*str* `,`*value* `)`

Sets the value of the variable named *str* on the current object to *value* . Again, this would usually be accomplished with assignment operator, but in certain cases (eg., the name of the variable must created at run-time with an expression), this function must be used. If the variable does not exist, it is created. Note that the type of the new variable is determined by *value*, and may not later be changed. Example:

```
setvar ("abc" + "def", 100);
```

would set the value of the variable named `abcdef` on the current object to the numeric value 100. If `abcdef` did not exist, it would be created.

7.1.15 `hasparent(`*obj* `)`

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

7.1.16 `find_method(`*str* `)`

Locates the method named *str* on the current object, if one exists. This activates the same method-searching algorithm as used when actually sending an object a message. Returns the object ID of the object defining the method, or #-1 if none is found. (This was useful in building the `@list` command, for instance).

7.1.17 `spew_method(`*str* `)`

Returns a string containing the internal stack-machine code for method *str* . 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.

7.1.18 `list_method(`*str* [`,` *lineno* [`,` *fullbrackets* [`,` *indent*]]] `)`

Returns a string containing the decompiled code for method *str* . 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 (default on).

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 (default 2).

The string returned contains embedded newlines.

7.1.19 `decompile()`

Decompiles the entire current object back to source. Returns a string, containing embedded newlines containing the source for the object. Variables are shown auto-initialized to their current values. This function can be *very* CPU-intensive, if the object is large.

7.1.20 `objsize()`

Returns the size, on disc, of the current object. This reflects the ideal size, and not the actual amount of memory or disc consumed by the objecct, which are subject to `malloc()` tax, dbm tax, etc.

7.2 Player Functions

These functions also work on `this`, the current object. However, they assume that `this` is a connected player. They will have no effect if on a non-player object, or a player object which isn't connected.

7.2.1 `echo(`*str*`)`

Display *str* to the current object. Does nothing if the current object is not a connected player.

7.2.2 `echo_file(`*str*`)`

Read in the contents of the local file *str*, and echo them, one like at a time, to the current object. The file is located relative to RUNDIR (usu. bin/online) of the server's installation.

7.2.3 `quit()`

Disconnect the current object.

7.2.4 `program(`[*obj*`,`*method*]`)`

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.

7.3 Miscellanous Functions

7.3.1 `typeof(`*var*`)`

Returns a number representing the type of *var* . This value may be checked against the pre-defined constants NUM, OBJ, STR, LIST and ERR.

7.3.2 `lengthof(`*var*`)`

Returns a number representing the length of *var* . *var* must be a string or list expression.

7.3.3 `serverof(`*obj*`)`

Returns a number representing the server ID of *obj*. 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(thingy))
  ...
endif
```

would evaluate to true if thingy is a local object.

7.3.4 `servername(`*obj*`)`

Returns a string representing the server name part of *obj*.

7.3.5 `servers()`

Returns a list corresponding to the system object (#0) at each remote server (eg., `#0@remotemud`, `#0@localmud`, etc).

7.3.6 `explode(`*str* [`,` *sep*]`)`

Break *str* into a list of strings. By default, explode breaks on spaces; the optional second argument is the character to break on.

7.3.7 `time()`

Returns a numeric value representing the current date and time, given in seconds since 12:00 GMT, January 1, 1970.

7.3.8 `ctime(`[*num*]`)`

Returns a string representing the integer *num* as an English date, or the current time if no argument is given.

7.3.9 `crypt(`*str* [`,` *salt*] `)`

Encrypt a string. This function uses UNIX's crypt() routine to encrypt a string. Useful for password checking, etc.

7.3.10 `match(`*template* `,` *str* [`,` *sep*]`)`

This function matches *str* to *template* . *str* should be a 1-word string which is compared against each word in *template* . If *str* matches a substring of any word in *template* , 1 is returned, otherwise 0 is returned. The optional third argument is the separator to use when matching (default is a blank).

7.3.11 `match_full(`*template* `,` *str* [`,` *sep*]`)`

This function matches *str* to *template* , as above, except that *str* must match an entire word in *template* , not just a substring.

7.3.12 `psub(`*str* `)`

This function substitutes the value of the local (method) variable "foo" for each instance of "%foo" or "%foo%" in *str*. Example:

```
foo = "system";
n = #0;
echo(psub("%n is the %foo object."));
```

would result in the output "#0 is the system object".

7.3.13 `strsub(`*with* `,` *what* `,` *str* `)`

Perform string substitution.

7.3.14 `pad(`*str* `,` *length* [`,` *padchar*]`)`

Pad/truncate a string.

7.3.15 `random(`*num* `)`

Return a random number 1 ..*num*

7.3.16 `compile(`*str* `,`[*obj* `,`*method*] `)`

Compile *str* into an object or method

7.4 List Functions

Lists are heterogenous collections of values, which may be treated as ordered lists, or unordered sets (see 'datatypes'). Choosing either the 'set' operations or 'list' operations determines how they are treated.

Since COOL has no concept of pointers (and all arguments are passed by value), none of the list operations can modify lists passed to them. Instead, they return new lists with the indicated modifications performed. So the code:

```
setadd(a, 3);
```

on its own does nothing.

```
a = setadd(a, 3);
```

on the other hand, adds the element 3 to the list variable `a`, as intended.

7.4.1 `setadd(`*list* `,`*value* `)`

This function adds *value* to *list* , as long as it's not already present. Returns the new list.

7.4.2 `setremove(`*list* `,`*value* `)`

Remove *value* from *list* , anywhere in the list. Returns the new list.

7.4.3 `listinsert(`*list*`,`*value*`[, `*pos*`])`

Insert *value* into *list*. By default, the new element is inserted at the beginning of the list. If the optional numeric argument *pos* is given, the element is inserted before position *pos*. Returns the new list.

7.4.4 `listappend(`*list*`,`*value*`[, `*pos*`])`

Appends *value* to the end of *list*, or after position *pos*, if given. Returns the new list.

7.4.5 `listassign(`*list*`,`*value*`, `*pos*`)`

Replaces element at position *pos* in *list* with *value*. Returns the new list.

7.4.6 `listdelete(`*list*`,`*pos*`)`

Deletes the element at position *pos* in *list*. Returns the new list.

7.5 Conversion Functions

These functions allow values of one datatype to be converted into values of another datatype.

7.5.1 `tonum(`*var*`)`

Convert an obj, string, or error value into a numeric value. Object values are converted by using the object ID portion as the new value. Strings are parsed like the UNIX function atoi(). Error values return the internal ID of the error (which isn't much use except to trivia addicts).

7.5.2 `toobj(`*var*`)`

Convert a num, string, or error value into an object ID value. Numbers are converted by using the number as the object ID portion of the new value, and the local server for the server ID portion. Strings are parsed, the same way COOL itself parses: `#3` or `#3@foomud` syntax. Errors are converted by using the internal ID of the error as the object ID portion, and the local server for the server ID portion.

7.5.3 `tostr(`*var*`)`

Convert a string, number, object, list or error type into a string value. Strings are converted by enclosing them in doublequotes, and escaping any control chars with a backslash (`\n`, `\t`, etc). Numbers and object ID's are simply printed. Lists are evaluated recursively, printing ′′, followed by the list elements, separated by commas, and then ′′. Errors are converted into a string representing the error identifier (`E_TYPE` becomes `"E_TYPE"`).

7.5.4 `toerr(`*var*`)`

Convert a string, number, or object value into an error value. Strings are parsed, the same way COOLMUD parses errors (`"E_TYPE"` becomes `E_TYPE`). Numbers are converted by using the number as the internal error ID. Object values are converted by using the object ID portion as the error ID.

7.6 Task Functions

7.6.1 `sleep(`*num*`)`

Pause for *num* seconds

7.6.2 `kill(`*num*`)`

Terminate thread *num*

7.6.3 `ps()`

Get a list of all active threads on the MUD. Returns a list of lists, in which each element represents a thread in an 11-element list of the form:

```
msgid, age, ticks, player, this, on, caller,
args, methodname, blocked_on, timer
```

msgid The message identifier of the current thread.

age The "depth" of the message in recursive calls.

ticks The number of ticks used by the thread so far.

player The player who typed the command that initiated this thread.

on The object on which the method which is executing is defined.

caller The object which called this method.

args The arguments to this method, when called.

methodname The name of the currently-executing method.

blocked_on A numeric identifier, indicating the state of the currently-executing thread.

timer A numeric value, indicating the time at which the current thread will resume, or expire (depending on state).

7.7 Wizard Functions

All wizard functions check that the object calling them is a wizard by looking in SYS_OBJ.wizards

7.7.1 `shutdown()`

Shut down the MUD. The database is written, remote servers disconnected, and the COOLMUD process terminates.

7.7.2 `dump()`

Dump the database.

7.7.3 `writelog(`*str*`)`

Write *str* to the logfile. The string is prefixed by the current date and time.

7.7.4 `checkmem()`

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."

7.7.5 `cache_stats()`

Returns a string with embedded newlines containing the current statistics of the object-paging cache. Currently the output looks like this:

```
cache stats, last 24 sec.:
  writes: 8           reads: 161
  dbwrites: 0         dbreads: 10
  read hits: 151      active hits: 126
  write hits to dirty cache: 6
  deletes: 0          checks: 18
  resets: 11          syncs: 0         objects: 161
```