Coldmud Programmer’s Reference

For Coldmud version 0.10
Revision 1

Greg Hudson

1 Overview

Coldmud is an object-oriented multi-user network database server with a C-like programming language for defining object behavior.

The components of Coldmud's database are *objects*, which together comprise the *object hierarchy*. Objects have *variables*, which determine their state, *methods*, which determine their behavior in response to *messages* from other objects, and *parameters*, which give the names of the variables used by the object's methods.

Coldmud objects also have *parents*, which are other objects. An object's behavior is determined by its *inherited* methods, which it gets from its parents, and its *overridden* methods, which it defines itself. One object, the *root object*, has no parents; all other objects have at least one parent.

Methods determine objects' responses to messages using a C-like language called `C--`. The philosophy of `C--` stresses simplicity, readability, and vertical code organization.

Encapsulation is an important feature of Coldmud's object system. An object cannot directly access another object's data; it must send a message to perform the required action. Moreover, an object cannot directly access the data used by its parents' or its children's methods.

Coldmud uses a disk-based database; only a limited number of objects exist in memory at any time. As long as the bulk of the object database is not concentrated in a few objects, Coldmud's process size should not grow beyond a certain point even if the database gets very large.

This manual is intended as a reference for programmers and administrators. It does not document any object database, and may not be appropriate as a tutorial. The best way to get started learning `C--` is to look at existing code, such as code in a core database, and refer to this manual when you don't understand how a piece of code works.

2 Objects

Objects in Coldmud have four characteristics: a *database reference* (called a *dbref*) which identifies the object, a list of *parent* dbrefs which determine what default behavior the object inherits, *variables* which store information, and *methods* which determine how the object responds to messages. Objects refer to each other using dbrefs, and interact with each other by sending messages. The objects which a Coldmud server has to work with are known collectively as the *object database*.

Coldmud gives special status to two objects, the *system object* and the *root object*. The system object is allowed to call administrative functions and receives `startup` and `heartbeat` messages from the server. The root object is the only object without parents, and is thus an ancestor of every object.

2.1 Database References

Database references are numbers which refer to objects. When an object is created, it is assigned a unique database reference. Database references cannot be changed after an object is created.

In `C--`, you indicate a database reference by preceding a number with a hash mark ('#'). For instance, `#126` refers to the object with the dbref 126.

In general, programmers should not have to be concerned with database references. See Section 3.6.2 [Names], page 20 for information on naming objects.

2.2 Inheritance

In addition to the methods an object defines for itself, objects inherit methods from their *parents*. An object can override a parent's method by defining its own methods with the same name.

The root object has no parents. All other objects have at least one parent, and eventually inherit from the root object. The root object has the dbref `1`.

It is easy to determine how an object with one parent will react to a message; either the object defines its own method for that message, or it uses the inherited method from its parent.

Objects with more than one parent are more complicated. Ancestors of such an object take precedence over each other according to the following rules:

1. Any ancestor object has higher precedence than its own ancestors.
2. The first parent of an object and the first parent's ancestors have higher precedence than the second parent and the second parent's ancestors, and so on, except when this conflicts with the first rule.

Normally, an object's response to a message is determined by checking its own method definitions, and then those of its ancestors in order of decreasing precedence[1]. However, it is possible for a method to disallow overrides. If one or more ancestors of an object define non-overridable methods, then Coldmud will select the non-overridable method defined on the ancestor with the least precedence.

2.3 Messages and Methods

Objects interact with each other by sending *messages*. Messages have two components, a name and a list of arguments. To see how to send a message in a `C--` method, see Section 3.6.10 [Sending Messages], page 29.

Coldmud processes a message by looking on the recipient object and its ancestors for a method with the same name as the message, following the rules given in the previous section. The recipient's method processes the message according to its `C--` instructions, and returns a value to the calling object.

While a message is being processed, the object which received the message is called the *current object*. Only the current object's variables can be accessed or changed by the method which processes the message; to examine or modify another object, the method must send a message to that object.

[1] Because of Coldmud's precedence rules, its multiple-inheritance scheme can be described loosely as "left-to-right, up-to-join". That is, Coldmud performs a depth-first traversal of the ancestors digraph, ordering the parents from left to right, stopping at an ancestor if it will reach the ancestor again later in the traversal.

If a method definition overrides a definition on an ancestor of the current object with lower precedence, the overriding method can *pass* the message to the ancestor; see Section 3.6.11 [Passing Messages], page 30 for details.

Methods can use the functions `this()` and `sender()` to determine the dbrefs of the current object and the object which sent the message to the current object. The functions `definer()` and `caller()` allow methods to determine the dbrefs of the objects which define the current method and the calling method.

Messages to an object are not always sent by another object; they can also come from the server. Coldmud sends `parse`, `connect`, and `disconnect` messages to handler objects of connections, `startup` and `heartbeat` messages to the system object, and `connect` and `failed` messages to receiver objects of pending connections generated by the `connect()` function. When Coldmud sends a message to an object, `sender()` and `caller()` return the integer `0`.

2.4 Variables and Parameters

An object's *parameters* gives the names of the variables its methods have access to. An object's *variables* correspond to parameters on that object or one of its ancestors.

The function `add_parameter()` adds a parameter to an object. Once you have created a parameter on an object, the methods on that object can use it to refer to a variable on that object or any of its descendents, using the `C--` constructs described in Section 3.6.7 [Variables], page 24.

Two objects can have parameters of the same name; a third object which inherits from both of them will contain two separate variables for those parameters. Methods on the first object will refer to one variable, and methods on the second object will refer to the other. Likewise, variables on separate objects are distinct, even though the methods which can access them and the parameter used by the method are the same.

Note that this system does not permit children to access the variables used by their ancestors directly. Objects can build on the abstractions of their parent objects, but not their internal representations.

2.5 The System Object

The system object is a special object for communicating with the Coldmud server. The system object has the dbref `0`.

At startup time, the server sends a `startup` message to the system object, with one argument, a list of strings giving the arguments passed on the command line when the server was invoked.

If you use the `set_heartbeat_freq()` function to enable the heartbeat, then the server will send a `heartbeat` message to the system object periodically.

Methods on the system object can call administrative functions such as `shutdown()` which ordinary objects are not permitted to call. In most databases, the system object will define methods which allow certain authorized objects to access these functions.

3 Defining methods

A method in Coldmud is an object's definition of how to handle a message. Methods are written in a language called `C--`, which resembles `C` at first glance. However, `C--` is a simpler language, and it stresses readability instead of conciseness.

The building blocks of methods are character sequences called *tokens*. From these tokens you can assemble a sequence of instructions to the interpreter called *statements*. You express values in statements using *expressions*.

Methods can return values using a `return` statement. If a method does not explicitly return a value, its return value is the dbref of the current object.

Methods have a limited amount of running time; if a method executes too many instructions before returning, it will abort with a `~ticks` error, which it cannot catch. See Chapter 4 [Errors], page 33

3.1 Example Method

Here is an example of a simple `C--` method. This method might be used in a mud server to handle a `tell_contents` message sent to a container object. Here is the code for the method:

```
arg str;
var obj;

for obj in (contents) {
    // Tell the string to the object.
    (| obj.tell(str) |);
}
```

This method takes one argument, `str`, a string to send to each object in the room. The method uses a local variable `obj` to traverse the list stored in the object variable `contents`. The body of the method is a for loop, which traverses the contents list. For each element in the for list, this method sends a `tell` message with the string. The parenthesis and vertical bar delimiters (‘`(|`’ and ‘`|)`’) surrounding the message expression mark it as a *critical expression*.

Errors which occur inside a critical expression are ignored; thus, errors which occur during the processing of the `tell` messages won't cause the `tell_contents` method to abort.

3.2 Tokens

The building blocks of `C--` methods are character sequences called *tokens*. There are many different types of tokens, ranging from single characters to lines of text. `C--` methods are assembled from sequences of tokens.

The following characters and pairs of characters are tokens:

```
{ } [ ] #[ %[ ( ) (| |) (> <) .. ; , = !
- + * / % == != > >= < <= . || && ? | : @
```

These tokens are used as operators and as punctuation in various types of expressions.

You can use *identifiers* as tokens and as parts of tokens. An identifier is a sequence of alphabetical and numeric characters or underlines beginning with an alphabetical character or an underline. Identifiers in `C--` are case-sensitive, so the identifiers `car` and `CAr` are not equivalent. The following are valid identifiers:

```
we_3_kings
obj
a
```

By themselves, identifiers usually represent variables. However, certain identifiers have special meanings to the parser. These *reserved words* are used in writing certain kinds of statements and expressions. They are:

```
var if else while for switch case default break continue return
catch any with handler pass to in fork atomic non_atomic
```

(`fork`, `atomic`, and `non_atomic` are not actually used by the current version of Coldmud, but are reserved for future use.)

There are several kinds of tokens for denoting literal expressions of various data types. These are *integers*, denoted by a sequence of digits; *strings*, denoted by a sequence of characters enclosed in double quotes; *dbrefs*, denoted by a sharp ('#') and a number; *symbols*, denoted by a single forward quote ('`'') and an identifier or string; and *error codes*, denoted by a tilde ('`~`') and an identifier or string. These literals are described more fully in Section 3.6.1 [Data Types], page 18, along with the data types they represent.

A *name token* is a dollar sign ('`$`') followed by an identifier or string. Names refer to objects indirectly; see Section 3.6.2 [Names], page 20.

Tokens which contain identifiers but begin with a distinguishing character (that is, symbol tokens, error code tokens, and name tokens) relax the restriction that identifiers not begin with an integer. Thus, `$23a` is a valid name token, even though `23a` is not an identifier.

Finally, a *comment token* is any text on a line after a pair of slashes ('`//`'). Comment tokens can be used as statements in methods; such statements are ignored by the interpreter.

3.3 Method Structure

A `C--` method has the following structure:

```
disallow_overrides;
arg arg1, arg2, ..., [rest];
var var1, var2, ...;

statement1
statement2
.
.
.
```

The `disallow_overrides;` declaration indicates a non-overridable method (see Section 2.2 [Inheritance], page 3). Normally, you should omit this declaration.

The `arg` declaration gives a list of argument variables, whose values will correspond to the arguments passed with the message. You may omit the `arg` declaration if the method does not take any arguments. If the final argument variable is given in square brackets (that is, if you specify *rest*), then the method can accept a variable number of argments; *rest* will contain a list of the arguments beyond the ones corresponding to the argument variables. If you do not specify *rest*, then the method can accept only the number of arguments specified by the argument variables, no more.

The `var` declaration tells the compiler that the listed identifiers should refer to local variables. You may omit the `var` declaration if you do not wish to declare any local variables.

The statements *statement1*, *statement2*, ... are the body of the method. They tell the interpreter what the method does.

3.4 Statements

Statements are instructions to the interpreter. Statements can be as simple as comments, or they can be complex loops and conditionals. However, most statement types in `C--` have simple structures. Because statements can include other statements, you can use these simple statement types to build complex directives.

The following sections describe simple statements, which perform simple actions once, conditional statements, which perform different actions depending on the value of an expression, looping statements, which perform an action multiple times, and error-handling statements, which affect how the interpreter handles error conditions.

3.4.1 Simple Statements

The *comment statement* does nothing at all. A comment statement is just a comment token, which is a sequence of two slashes ('//') followed by any characters up to the end of the lines. For instance:

```
// This is a comment.
```

The interpreter ignores comment statements completely; they are for the benefit of human readers. Note that comments in `C--` are actual statements, unlike comments in `C`, which are filtered out by the preprocessor.

The *expression statement* evaluates an expression and discards its value. The syntax of the expression statement is:

expression `;`

The following are expression statements:

```
3;
sender().tell($sys.who());
```

The *assignment statement* evaluates an expression and assigns the value to a variable. The syntax of the assignment statement is:

variable = *expression*;

The interpreter assigns the value of *expression* to *variable*, which is a local variable if it has been declared as one, or an object variable on the current object if it has not been declared. See Section 3.6.7 [Variables], page 24.

The *compound statement* allows you to treat one or more statements as a single statement. The compound statement has the following syntax:

```
{
    statement1
    statement2
    .
    .
    .
}
```

The interpreter executes each of *statement1*, *statement2*, ... in turn.

The *return statement* allows a method to stop exeuting statements and to return a value other than the default return value of `this()`. The return statement can have either of the following two forms:

```
return expression;
return;
```

The interpreter stops executing statements in the method and returns the value of *expression*, or the dbref of the current object if *expression* is not given.

3.4.2 Conditional Statements

There are three types of conditional statements in `C--`. The *if statement* has the following syntax:

```
if (expression)
    statement
```

The interpreter evaluates *expression*. If the value of *expression* is true (see Section 3.6.1 [Data Types], page 18), then the interpreter executes *statement*.

The *if-else statement* is similar to the if statement. The if-else statement has the following syntax:

```
if (expression)
    true-statement
else
    false-statement
```

The interpreter evaluates *expression*, as before. If the value of *expression* is true, then the interpreter executes *true-statement*; otherwise, it executes *false-statement*.

Because the if statement and the if-else statement are similar, they can sometimes be ambiguous. The following code will produce unexpected results:

```
if (a)
    if (b) c;
else
    d;
```

The indentation suggests that the `else` clause should apply to the first `if` clause, but in fact it applies to the more recent one. You can avoid ambiguities like this by using braces to create compound statements out of conditional and looping statements, even if there is only one of them. In this case, you might write:

```
if (a) {
    if (b)
        c;
} else {
    d;
}
```

The third type of conditional statement is the *switch statement*, which allows you to compare one value against a series of other values. The switch statement is the most complicated statement type in C--, so we'll start with an example:

```
switch (val) {
    case 0:
        echo("The value is zero.");
    case 1 .. 10:
        echo("The value is between one and ten inclusive.");
    case 11 .. a:
        echo("The value is between eleven and a inclusive.");
    case "foo", "bar".."baz":
echo("The value is \"foo\" or between \"bar\" and \"baz\"");
    case a .. b, c .. d, 42:
        count = count + 1;
        echo("The value is in the counted area.");
    case ~perm:
        echo("Permission denied while getting the value.");
    default:
        echo("Did not recognize value.");
}
```

This example illustrates all of the capabilities of the switch statement. The expression given by `val` in the example is the *controlling expression*, and is compared against each of the cases inside the switch body. Each case has a value or list of values to compare against. The values can be of any type, and need not be constant expressions. You can also specify ranges, using two dots (‘`..`’) to separate the lower and upper bounds. The keyword `default` specifies an action to perform if no cases were matched by the controlling expression.

Here is a more formal description of the syntax of the switch statement:

```
switch (controlling-expression) {
    case expr-or-range, expr-or-range, ...:
        statements
    case expr-or-range, expr-or-range, ...:
        statements
    .
    .
    .
    default:
        default-statement
}
```

When executing a switch statement, the interpreter scans through the list of cases and compares *controlling-expression* against each of the lists of values in the cases, evaluating the value lists from left to right until there is a match. The lower and upper bounds of ranges must be of the same type and must be either integers or strings, or the interpreter will throw a `~type` error. When the interpreter finds a match, it will execute the statement for that case. The interpreter will not continue checking cases after a match.

If the interpreter does not find a match, it will execute *default-statement*, the statement corresponding to the default case. You do not need to provide a default case if you do not wish to provide a default action.

`C` programmers should note that switch statements in `C--` differ from switch statements in `C` in several respects. Because case values do not have to be constants, they may conflict, in which case the first match will take precedence. Also, there is no fall-through in `C--` switch statements; only the statements corresponding to the matching case will be executed. Because there is no fall-through, the `break` statement does not apply to switch statements. Finally, the default case must be placed last in the list of cases if it is given.

3.4.3 Looping Statements

`C--` provides three kinds of looping statements. These statements allow you to traverse a list or a range of numbers, or to execute a statement as long as a certain condition is true. `C--` also provides two kinds of jump statements which allow you to prematurely exit from a loop or continue onto the next iteration.

The *for-list statement* allows you to traverse a list. It has the following syntax:

```
for variable in (list)
    statement
```

variable must be the name of a local variable, and *list* must be an expression of list type. The interpreter executes *statement* once for each element of *list*. *variable* contains the list element that the interpreter is at. Here is an example of a method using a for-list statement:

```
var a, s;

a = ["foo", "bar", "baz"];
for s in (a)
    .tell(s);
```

You can also iterate over the associations in a dictionary with the for-list statement; see Section 3.6.4 [Dictionaries], page 22.

The *for-range statement* allows you to traverse a range of integers. It has the following syntax:

```
for variable in [lower .. upper]
    statement
```

As before, *variable* must be the name of a local variable. *lower* and *upper* must be expressions of integer type. The interpreter executes *statement* once for each number from *lower* to *upper* inclusive. *variable* contains the number that the interpreter is at. Assigning to *variable* within the loop body will not change the status of the loop; the interpreter remembers what number it is at independently of the loop index. Here is an example of a method using a for-range statement:

```
var i;

for i in [1 .. 10]
    .tell(tostr(i));
```

The *while statement* allows you to execute code as long as a condition expression is true. The while statement has the following syntax:

```
while (expression)
    statement
```

The interpreter continually evaluates *expression* and executes *statement* until the value of *expression* is false, according to the rules in Section 3.6.1 [Data Types], page 18. Here is an example of a method using a while statement:

```
var a;

a = 1;
while (a < 35)
    a = a * 2;
.tell(tostr(a));
```

The *break statement* allows you to exit a loop prematurely. The break statement has the following syntax:

```
break;
```

The interpreter jumps to the end of the innermost for-list, for-range, or while statement.

The *continue statement* allows you to jump to the next iteration of a loop. The continue statement has the following syntax:

```
continue;
```

The interpreter skips the remainder of the loop body and begins another iteration of the innermost for-list, for-range, or while statement.

The break and continue statements do not allow you to break out of two loops at once. `C--` does not provide any general jump mechanism. If you find that you require jumps other than break or continue statements, then you probably need to reorganize your code.

3.5 Error-Handling Statements

The *catch statement* allows you to indicate how errors should be handled in a block. You can use this statement when you anticipate that an error might occur for reasons other than your programming errors. For instance, you may anticipate being denied permission to perform an operation, and use a catch statement to handle this error explicitly. You can also use the catch statement to make sure that cleanup operations are performed if an error occurs.

The catch statement has the following syntax:

```
catch error code, error code, ...
    body-statement
with handler
    handler-statement
```

You can substitute the keyword `any` for the list of errors to indicate that you wish to catch all errors. You can leave out the `with handler` and the handler statement if you do not wish to do anything in the handler.

If an error listed in the error list is thrown inside *body-statement*, the interpreter will execute the handler rather than aborting the method. After the handler is done, control continues after the end of the catch statement, as if *body-statement* had completed with no errors.

Inside the handler, you can use the function `error()` to determine the error code which triggered the handler, the function `error_arg()` to retrieve the error argument specified by `throw()` if one weas given, and the function `traceback()` to retrieve a list of strings describing the propagation of the error down the call stack. You can use the function `rethrow()` to continue propagating an error.

Here is an example of how you might use a catch statement to intelligently handle a `~methodnf` error from the `list_method()` function:

```
catch ~methodnf {
    code = list_method(method_name);
} with handler {
    .tell("There is no method named " + tostr(method_name) + ".");
}
```

Note that critical expressions (see Section 3.6.12 [Error-Handling Expressions], page 30) inside *body-statement* will override the behavior of the catch statement. For more detail on errors in `C--`, see Chapter 4 [Errors], page 33.

3.6 Expressions

You can express values in `C--` methods using *expressions*. The following sections document the various types of expressions in `C--`, as well as the types of values they can have when evaluated.

3.6.1 Data Types

Every piece of data in `C--` has a type. This section documents the data types in `C--`. You can produce values of the simpler data types (integers, strings, dbrefs, symbols, error codes) can be produced with *literal* expressions; you can produce values of the more complex data types (lists, dictionaries, frobs, buffers) with *constructor* expressions. As we go, we will explain how to specify or construct values of each of the data types, and explain what it means for a value of each data type to be true.

Every piece of data in `C--` has a type and a truth value. The type can be one of:

- Integer
- String
- Dbref
- List
- Symbol
- Error type
- Frob
- Dictionary
- Buffer

An *integer* is just a number. Integers can reliably be from -2147483648 to +2147483647. An integer is true if it is not zero. You can denote integer literals in `C--` by writing a sequence of digits, optionally preceded by a '–' or '+' sign.

A *string* is a sequence of printable characters. A string is true if it is not empty. You can denote string literals in `C--` by enclosing a sequence of characters in double quotes ('`"`'). To include a double quote or backslash character in a string literal, precede it by a backslash ('\'). The following are string literals:

```
"foo"
"\"foo\" is a metasyntactic variable."
"The backslash ('\\') is a much-abused character in many languages."
```

A *dbref* is a reference to an object, which may or may not exist. Database references are always true, whether or not they refer to actual objects. You can denote dbref literals in `C--` by preceding a number with a sharp ('#'); for example, `#1` and `#267` are valid dbref literals.

A *list* is a collection of objects of any type. Lists are useful for manipulating several objects at once. A list is true if it is not empty. You can construct a list value by enclosing a comma-separated series of expressions in square brackets; for example, `[1, 2, 3]` and `[1, ["foo", 'bar], #23]` are valid list construction expressions.

A *symbol* is a symbolic value. You can denote symbols in `C--` by preceding an identifier or string with an apostrophe ('''). `'remote`, `'template`, and `'object` are valid symbol literals. Symbols are always true.

An *error code* identifies a type of error condition. Both the `C--` interpreter and `C--` methods use error codes to identify types of errors when they occur. See Chapter 4 [Errors], page 33 for information about how Coldmud handles errors. You can denote error code literals in `C--` by preceding an identifier or string with a tilde ('~'). Error codes are always false.

A *frob* consists of a *class*, the dbref of an object, and a *representation*, a list or dictionary. Frobs are intended to serve as lightweight objects. You can construct a frob value by enclosing expressions for the class and representation in triangular brackets, separated by a comma; for instance, `<#73, [1, 2]>` is a valid frob construction expression. Frobs are always true.

A *dictionary* is a collection of *associations*, each of which has a *key* and a *value*. Dictionaries take up more space than lists, but searching for a data value in a dictionary is generally faster than searching for data values in lists. You can construct a dictionary by enclosing a comma-separated series of associations in square brackets, where each association is a two-element list giving a key and a value, and preceding the whole thing with a hash mark ('#'). For instance, `#[["foo", 3], ["bar", 'baz]]` is a valid dictionary construction expression. A dictionary is true if it is not empty.

a *buffer* is an array of unsigned eight-bit values, intended for network I/O. You can construct a buffer by enclosing a comma-separated series of integer expressions in square brackets, preceded by a percent sign ('%'). For instance, `%[65, 66, 67, 10]` is a valid buffer construction expression. A buffer is true if it is not empty.

You can get the type of a piece of `C--` data using the `type()` function. The `type()` function returns a symbol corresponding to the type of its argument. This symbol can be `'integer`, `'string`, `'dbref`, `'list`, `'symbol`, `'error`, `'frob`, or `'dictionary`.

3.6.2 Names

A *name* is an indirect reference to an object. Coldmud translates names into dbrefs during method interpretation according to the associations made using the administrative function `set_name()`. A *name expression* is simply a name token, an identifier or string following a dollar sign (`$`).

When the interpreter encounters a name expression, it looks up the name to see what dbref it has been assigned by the `set_name()` function. If no such assignment has been made, then the interpreter throws a `~namenf` error; otherwise, the value of the name expression is the dbref assigned to the name by `set_name()`.

You can also use the (non-administrative) function `get_name()` to look up a symbol as a name.

The administrative function `del_name()` deletes the association for a name.

3.6.3 Lists

A *list* is a collection of pieces of data of any type. You can create a list in `C--` using a *list construction expression*. A list construction expression has the following syntax:

[*expression1*, *expression2*, ...]

expression1, *expression2*, ... are the elements of the list. You can also splice a list into a list construction expression; see Section 3.6.13 [Splicing], page 31. The result of a list construction expression is a list containing the values of the expressions. Here are some examples of list construction expressions:

```
[1, 2, 3]
[a + 5, 'string, [3, "b"], tostr(~none)]
[]
```

To retrieve an element of a list, you can use a *list selection expression*, which has the following syntax:

list[*element*]

list must be an expression of list type (or string type; see below) and *element* is an expression of integer type whose value is between one and the length of the list. The value of a list selection expression is the value of the element of *list* numbered by *element*, where the elements of *list* are numbered starting from one. For example, the following method returns `3`:

```
var a;

a = ["foo", 2, 'bar, 3, ~parents];
return a[4];
```

You can use the list selection expression with strings as well as with lists. In this case, `C--` treats the string as if it were a list of one-character strings. The following method returns `"p"`:

```
var a;

a = "Depth";
return a[3];
```

You can also use the list selection expression with dictionaries; we will describe this usage in the next section.

You can look for a piece of data in a list, you can use a *list search expression*, which has the following syntax:

element `in` *list*

The value of this expression is the index of the first occurrance of *element* in *list*, or `0` if *element* does not occur in *list*. The following method returns `[3, 0]`:

```
var a;

a = ['foo, "bar", 4 + 2, #77];
return [6 in a, 'quux in a];
```

As with the list selection expression, you can use the list search expression with strings as well as with lists. In this case, both *element* and *list* must be strings, and the result is the first position at which *element* occurs in *list*, or 0 if it does not occur.

3.6.4 Dictionaries

A *dictionary* is a list of *associations*. Each association contains a *key* and a *value*, both arbitrary `C--` data values. Dictionaries take up more space than lists, but allow fast searches. Only one association in a dictionary may have a given key.

Dictionary construction expressions have the following syntax:

`#[[`*key1*`, `*value1*`], [`*key2*`, `*value2*`], `...`]`

The result of a dictionary construction expression is a dictionary with the keys *key1*, *key2*, ... and the associated values *value1*, *value2*, ... The following are valid dictionary construction expressions:

```
#[["foo", 3], ['bar, 4], [8, [#0, 'startup]]]
#[['type, 'and], ['lhs, 3], ['rhs, 0]]
```

You can search for the value associated with a key in a dictionary using the list selection expression:

dict`[`*key*`]`

If any of the associations in *dict* have the key *key*, the result of this expression is the value associated with that key. If *key* is not the key of any of the associations in *dict*, then the interpreter raises a `~keynf` error.

You can iterate over a dictionary's associations using the for-list statement; at each step of the for-list statement, the loop index will contain a list `[`*key*`, `*value*`]` for the current association.

The four dictionary manipulation functions `dict_keys()`, `dict_contains()`, `dict_add()`, and `dict_del()` allow you to manipulate dictionaries in other ways; see the descriptions of each function for details.

3.6.5 Buffers

A *buffer* is a vector of unsigned eight-bit values, intended primarily for network I/O. Although they are not as convenient to use as strings, they can contain any character value, while strings can contain only printable characters.

You can construct a buffer using a *buffer construction expression*, which has the following syntax:

`%[`*byte1*`, `*byte2*`, `...`]`

The result of a buffer construction expression is a buffer containing the byte values expressed by *byte1*, *byte2*, ..., all of which must be integers. If any of the values of *byte1*, *byte2*, ... cannot fit inside an unsigned eight-bit number, then that byte will contain the lower order eight bits of the specified number.

Apart from operations which apply to all data types, the only language operations which apply to buffers are the functions whose names begin with 'buffer_': `buffer_len()`, `buffer_retrieve()`, `buffer_append()`, `buffer_replace()`, `buffer_add()`, `buffer_truncate()`, `buffer_to_strings()`, and `buffer_from_strings()`. The last two functions allow you to convert between buffers and lists of strings.

Coldmud passes data between the object heirarchy and network connections in terms of buffers; see Section 6.3 [Connections], page 74 for details.

3.6.6 Frobs

A *frob* is a dictionary or list (its *representation*) associated with a dbref (its *class*). Frobs represent a kind of lightweight object, with less overhead than real objects. The encapsulation around frobs is not as strong as the encapsulation around real objects, and in some respects frobs are not as convenient to work with as real objects. In general, objects should be used for long-lasting, complicated, changing entities, while frobs should be used for small pieces of data.

Frob construction expressions have the following syntax:

`<`*class*`, `*representation*`>`

The result of a frob construction expression is a frob of class *class*, which must be a dbref, with a representation *representation*, which must be a list or dictionary. In general, you should only use the frob construction expression to construct frobs of class `this()`, but this is not enforced. The following are valid frob construction expressions:

```
<#76, #[['type, 'true]]>
<#89, ["You cannot go that way."]>
```

Apart from operations which apply to all data types, the only operations supported on frobs are message-passing and the `class()` function. When you send a message to a frob, the interpreter treats it as a message to the frob's class, with the frob's representation inserted as the first argument. The `class()` function returns the class of a frob.

3.6.7 Variables

`C--` provides two kinds of variables, *local variables* and *object variables*.

Local variables allow methods to temporarily store information. To use a local variable in a method, declare the local variable's name (an identifier) at the top of the method in a `var` declaration (see Section 3.3 [Method Structure], page 9). The variable's name is then an expression whose value is the contents of the variable. The variable initially contains the integer `0`.

To set the variable's contents, use an *assignment statement*, which has the following syntax:

variable `=` *value*`;`

This assigns the value *value* to the variable *variable*. The values of a method's local variables are specific to the processing of a particular message call; when the method is invoked another time, all of its local variables begin with the value `0` again.

The following example method returns `6`:

```
var a;

a = 3;
return a + 3;
```

Object variables store information for an indefinite period of time. You refer to object variables through parameters, which are defined on the current method's defining object (*not* the current object). As with local variables, you can retrieve the value of an object variable by writing the name of the parameter which refers to that variable, and you can assign to an object variable using an assignment statement. You can also use the `get_var()` and `set_var()` functions to perform these operations. It is an error of type `~paramnf` to refer to a parameter which does not exist on the current method's defining object. Object variables begin with the value `0`.

3.6.8 Operators

Operators are used to perform simple operations on expression values, such as adding two values together. `C--` provides a variety of operators to perform arithmetic and logical operations on data. Most of these operators fall into two syntactical categories: *unary operators* and *binary operators*.

Unary operators act on a single expression value. In `C--`, all unary operators are single characters which precede expressions. For example, `!a` is the logical negation of the value of the variable `a`.

Binary operators act on two expression values. For example, `a + b` is the sum of the values of the variables `a` and `b`.

Several operators are neither unary nor binary. The message operator (`.`) acts on a value on the left but a message name and an argument list on the right (see Section 3.6.10 [Sending Messages], page 29). The index operator (`[]`) uses two punctuation marks to delimit the beginning and end of the offset (see Section 3.6.3 [Lists], page 20). The conditional operator (`?|`) operates on three data values (see Section 3.6.8.4 [Conditional Operators], page 27).

3.6.8.1 Precedence

It is easy to write an expression whose order of evaluation is unclear. For instance, the expression `a - b + c` could be parsed as `(a - b) + c` or as `a - (b + c)`. To resolve these conflicts, each operator has two properties: *precedence* and *association*.

Precedence determines whether an operator binds more tightly than another operator; for instance, `a + b * c` is equivalent to `a + (b * c)` because multiplication has higher precedence than addition.

Association determines whether operators at the same precedence level associate left to right or right to left; `a - b - c` is equivalent to `(a - b) - c` because subtraction associates left to right.

Here is a list of operators grouped by precedence, in order from highest precedence to lowest:

- `.`
- `[]`
- `!` and unary `-`
- `* / %`
- `+ -`
- `== != > >= < <=`
- `in`
- `&&`
- `||`
- `?|`

All operators associate from left to right except the `&&`, `||`, and `?|` operators, which associate from right to left. You can always use parentheses ('`(`' and '`)`') to specify the order of evaluation explicitly.

3.6.8.2 Arithmetic

`C--` provides seven operators for doing arithmetic, two unary operators and five binary operators. These operators apply primarily to integers, but the addition operator also applies to strings. Using these operators on inappropriate data is an error of type `~type`.

The unary `-` operator takes the negative of an integer value. The expression `-(3 + 4)` has the value `-7`. The unary `+` operator has no effect on its argument, and is provided only for completeness.

The binary operator * performs multiplication on its arguments. The expression (3 * 4) has the value 12. The binary operators / and % perform integer division and modulus, respectively, on their arguments. The expressions (13 / 5) and (13 % 5) have the values 2 and 3 respectively.

The binary operators + and − perform addition and subtraction. The expressions (3 + 4) and (3 − 4) have the values 7 and −1 respectively. The binary + operator can also apply to strings and lists, in which case it performs concatenation; the expression ("foo" + "bar") has the value "foobar", and the expression (["foo", "bar"] + ["baz"]) has the value ["foo", "bar", "baz"].

3.6.8.3 Logic

C−− provides a number of relational and logical operators. These operators are primarily useful for expressing conditions, since they return either true or false, represented by the integers 1 and 0.

The == and != operators test for equality and inequality, respectively, of their arguments. Two pieces of data are equal if they have the same type and contain the same value. Equality of strings is not case-sensitive, but equality of symbols is, so ("foo" == "fOo") is true, but ('car == 'CAr) is false. Lists are equal if all of their elements are equal.

The <, <=, >=, and > operators test whether their left-hand arguments are less than, less than or equal to, greater than or equal to, or greater than their right-hand arguments, respectively. Arguments to these operators must both be of the same type, and must be of integer, string, or dbref type. String comparison is not case-sensitive, so ("fooa" < "fooB") is true, even though the ASCII value of 'a' is greater than that of 'B'.

The unary ! operator tests whether its argument is false, thus acting as a logical not operation.

3.6.8.4 Conditional Operators

The || and && operators act as logical or and and operators, respectively. The expression (a || b) has the value of a if a is true, or the value of b if a is false. The expression (a && b) has the value of a if a is false, and the value of b if a is true. C−−'s logical operators are *short-circuit operators*, meaning that the right-hand argument is not evaluated if the left-hand argument is sufficient to determine the value of the expression. This is important if the right-hand argument has side-effects or could cause an error.

The `?|` operator is a trinary operator, with the following syntax:

condition `?` *true-expr* `|` *false-expr*

The result of this expression is the result of *true-expr* if *condition* is true, or the result of *false-expr* if *condition* is false. See Section 3.6.1 [Data Types], page 18 for the definition of truth for `C--` data.

3.6.9 Calling Functions

A `C--` method can call a built-in function using a *function call expression*, which has the following syntax:

function-name`(`*arg1*`,` *arg2*`,` ...`)`

function-name is an identifier naming the function, and *arg1*, *arg2*, ... are expressions. There do not have to be any arguments, but you must include the parentheses even if there are no arguments. The arguments are evaluated from left to right.

As an example, the `pad()` function pads a string argument with spaces to a certain length. The following expression has the value `"foo   "`:

```
pad("foo", 6);
```

Coldmud's built-in functions allow you to perform string and list operations, modify and retrieve information from the current object, perform administrative tasks, among other things. For descriptions of the available functions, see Chapter 5 [Function Descriptions], page 35.

For various reasons, functions can throw errors. For instance, calling a function with an incorrect number of arguments causes a `~numargs` error, and calling a function with arguments of incorrect type causes a `~type` error. See Chapter 4 [Errors], page 33 for information on how to handle errors.

3.6.10 Sending Messages

If you want to retrieve information from or modify an object other than the current object, you can do so by sending it a message. You can do this with a *message expression*, which has the following syntax:

receiver`.`*message*`(`*arg1*`, `*arg2*`, `*...*`)`

You may omit *receiver*, in which case it is assumed to be the current object. Otherwise, *receiver* must be an expression of dbref or frob type, and *message* an identifier giving the name of the message. *arg1*, *arg2*, ... are the arguments to be sent with the message. The arguments are evaluated from left to right. You must include the parentheses around the argument list, even if there are no arguments. The result of a message expression is the value returned by *receiver*'s method for the message *message*.

If *receiver* is a frob, then the message is sent to the frob's class object, with the frob's representation inserted as the first argument. If *receiver* is a dbref, then the message is sent to the object with that dbref.

If *receiver* is not an object or frob, then the interpreter throws a `~type` error. If *receiver* is a dbref which does not refer to an existing object, or if it is a frob whose class is not an existing object, then the interpreter throws an `~objnf` error. If *receiver* does not have a method defined for *message*, then the interpreter throws a `~methodnf` error. See Chapter 4 [Errors], page 33 for information on how to handle errors.

Here are some examples of message expressions:

```
.tell("I don't see that here.");
$sys.wizards()
loc.tell_contents(args[1]);
```

You can substitute an arbitrary expression for *message*, by enclosing it in parentheses. The syntax for sending an arbitrary message is:

receiver`.(`*message-expression*`)(`*arg1*`, `*arg2*`, `*...*`)`

As before, *receiver* can be omitted, in which case it is assumed to be the current object. *message-expression* must be an expression of symbol type, or the expression will cause a `~type` error. You must include the parentheses around the argument list, even if there are no arguments.

In order to prevent incidents of infinite recursion, Coldmud has a maximum calling depth for messages. This maximum depth is `128` method calls in the current version. If sending a message would exceed the maximum calling depth, the interpreter raises a `~maxdepth` error.

3.6.11 Passing Messages

A method can pass control of a message to a different ancestor of the current object using a *pass expression*, which has the following syntax:

```
pass(arg1, arg2, ...)
```

The arguments are evaluated from left to right. The result of this expression is the result of calling the "next" method with the arguments *arg1*, *arg2*, ... The "next" method is defined as the method which would be called if the current method, and any other methods which have already passed control of the message, were not defined. This method may not be defined on an ancestor of the current object; you should, in general, not depend on a specific method being called when you use a pass expression.

The called method sees the same current object, sender, and caller as the current method.

The pass expression can cause a `~methodnf` errors if the called method does not exist. See Chapter 4 [Errors], page 33 for information on how to handle errors.

3.6.12 Error-Handling Expressions

`C--` provides two kinds of expressions for handling errors in expressions. These are the *critical expression* and the *propagation expression*.

The *critical expression* allows you to ignore errors which occur inside an expression. It has the following syntax:

```
(| expression |)
```

If an error occurs in *expression*, then the interpreter will stop evaluating *expression*, and continue to execute the current method as if it had succeeded evaluating *expression*. The value of *expression* will be the error code for the error condition which occurred.

The *propagation expression* allows you to indicate that any errors which occur inside an expression are the responsibility of the calling routine. It has the following syntax:

```
(> expression <)
```

If an unhandled error occurs in *expression*, then it will propagate to the calling routine as itself, instead of as `~methoderr`.

Critical expressions override the behavior of catch statements, so that errors which occur within critical expressions do not trigger catch error handlers. Propagation expressions do not override critical expressions or catch statements, however; they do not prevent errors from being caught, but only determine how errors propagate if they are not caught.

For more information on how `C--` handles errors, see Chapter 4 [Errors], page 33.

3.6.13 Splicing

Whenever you are writing an argument list or a list constructor expression, you can splice a list value into the sequence of expressions by prepending a list expression with '`@`'. For instance, the following method returns `['foo, 4, 5, 6, 'quux]`:

```
var a;

a = [4, 5, 6];
return ['foo, @a, 'quux];
```

You use the splicing operator to pass a message along with all of its arguments:

```
pass(@args);
```

The splicing operator is not listed in the operator precedence list (see Section 3.6.8.1 [Precedence], page 25). This is because it is meaningless to talk about, say, adding a spliced list to another value. Using the splicing operator never causes an ambiguity.

4 Errors

When something goes wrong in a `C--` method, the interpreter will often throw an error. Methods can also throw their own errors using the `throw()` function. An error condition consists of an error code (the *type* of the error) and a string describing the error. Methods can recognize errors using the error codes; the string appears, along with the error code, in the traceback obtainable through the `traceback()` function inside a catch handler.

When the interpreter throws an error, it checks to see how the current method handles that error type. If the error occured in a critical expression (see Section 3.6.12 [Error-Handling Expressions], page 30), then the interpreter will cease evaluating the critical expression. Processing of the method will continue as if the interpreter had completed evaluation of the critical expression. The value of the critical expression will be the error code associated with the thrown error. In this case, the traceback is not accessible from `traceback()`; in order to get a traceback, you must use a catch statement.

If the error did not occur in a critical expression, but occurred in a catch statement which catches the error code (either because it is a `catch all` statement or because it lists the error code—see Section 3.5 [Error-Handling Statements], page 16), then processing of the method jumps to the error handler, if one was provided, or to the end of the catch statement if not. Inside an error handler, you can use the `error()`, `traceback()`, and `rethrow()` functions to retrieve the error or traceback associated with the error condition, or continue propagating the error.

If the error did not occur in a critical expression or in an appropriate catch statement, then the current method aborts, and the interpreter throws an error in the calling method. Normally, the error thrown in the calling routine will have the error code `~methoderr`, but if the original error occurred in a propagation expression (see Section 3.6.12 [Error-Handling Expressions], page 30, then the error code will be the same as it was for the original error. A propagation expression has no effect on how an error is handled except to cause it to propagate differently to the calling routine.

You can throw your own errors using the `throw()` function. This does not throw an error in the current method; instead, it exits the current method and throws an error in the calling method. Thus a method cannot ignore an error which it threw itself using `throw()`.

There is one case in which a method cannot catch an interpreter-generated error. Methods have a limited amount of time to run, measured in *ticks*. A method will generally only run out of ticks if it gets stuck in an infinite loop. If a method runs out of ticks, then the interpreter will throw a `~ticks` error, which the method cannot catch. This causes the method to abort, which in turn causes the interpreter to throw a `~methoderr` error in the calling routine.

You should use critical expressions when you anticipate that you may be calling a buggy or undefined method, but you do not wish your own method to bomb as a result. For instance, a method which announces a string to every object in a container should probably ignore errors in the methods for each individual object which handle receiving the string. You should be careful that your critical expressions are correct code, however, because you will not immediately notice errors which occur while the interpreter is evaluating them.

You should use catch statements when you wish to handle errors with any kind of sophistication. The catch statement is much more powerful than the critical expression, and is ideal for situations in which fine-grain control over error handling is required.

You should use propagation expressions when your method is an intermediary between an outside object and an internal feature. For instance, a method which checks permissions and calls an object function such as `list_method()` is acting as an intermediary. In this case, the method should throw the same errors as the `list_method()` function, so you should enclose the function call in a propagation expression.

5 Function Descriptions

The function descriptions are arranged by category, and alphabetically within their category. You can use the function index to look up the description of a function by name.

Every function can throw a `~numargs` error if passed the wrong number of arguments, and most can throw a `~type` error if passed arguments of the wrong type. If a function can throw any other type of error, it will be documented in the function's description.

5.1 Data functions

The functions described in this section allow you to perform various operations on data. Operations specific to strings or lists are grouped in their own sections.

class

`class(`*frob*`)`

This function returns the class of a frob. See Section 3.6.6 [Frobs], page 23 for details on frobs.

todbref

`todbref(`*number*`)`

This function converts *number* to a dbref.

Examples:

```
todbref(0)
     ⇒ #0
```

toerr

```
toerr(string)
```

This function converts *string* to an error code. *string* can be any string; it does not have to be a valid identifier.

Examples:

```
toerr("foo")
     ⇒ ~foo
```

toint

```
toint(string)
```

This function converts *string* to an integer. If *string* is not a number, the resulting integer will be `0`.

Examples:

```
toint("67")
     ⇒ 67
toint("foo")
     ⇒ 0
```

toliteral

```
toliteral(data)
```

This function converts any data to a `C--` literal or constructor expression which, when read by the interpreter, will result in the same data object.

Examples:

```
toliteral([3, "foo", 'bar, #60, ~none])
     ⇒ "[3, \"foo\", 'bar, #60, ~none]"
```

```
toliteral(tosym("foo-" + tostr(3 + 4)))
     ⇒ "tosym(\"foo-7\")"
toliteral(<#56, #[['type, 'obj], ['obj, #22]]>)
     ⇒ "<#56, #[['type, 'obj], ['obj, #22]]>"
```

tostr

`tostr(`*data*`)`

This function converts any data to a string representation of it. For strings, the string representation is the string itself; for symbols and error codes, the string representation is the identifier part of the data object converted to a string value; for lists and frobs, the string representations are `"<list>"` and `"<frob>"` respectively; for integers and dbrefs, `tostr()` is equivalent to `toliteral()`.

Examples:

```
tostr(3)
    ⇒ "3"
tostr("foo")
    ⇒ "foo"
tostr(#0)
    ⇒ "#0"
tostr([2, 3, 4])
    ⇒ "<list>"
tostr('foo)
    ⇒ "foo"
tostr(~methodnf)
    ⇒ "methodnf"
tostr(<#66, #[['type, 'obj], ['obj, #40]]>)
    ⇒ "<frob>"
```

tosym

`tosym(`*string*`)`

This function converts *string* into a symbol. *string* can be any string; it does not have to be a valid identifier.

Examples:

```
tosym("foo")
     ⇒ 'foo
```

type

`type(`*data*`)`

This function returns a symbol giving the type of *data*. The return value is one of `'integer`, `'string`, `'dbref`, `'list`, `'symbol`, or `'error`.

Examples:

```
type(5)
     ⇒ 'integer
type([6, 'foo, "bar"])
     ⇒ 'list
type(<#44, #[['type, 'obj], ['obj, #20]]>)
     ⇒ 'frob
```

valid

`valid(`*data*`)`

This function returns `1` if *data* is a dbref and is valid, or `0` otherwise. A dbref is valid if it refers to an object that exists in the database.

Examples:

```
valid(#0)
     ⇒ 1
valid('foo)
     ⇒ 0
```

5.2 String Functions

The functions described in this section allow you to perform operations on strings. All of the functions described in this section are case-insensitive except for `strcmp()` and, if specified, `match_regexp()`.

crypt

```
crypt(string)
crypt(string, salt)
```

This function performs one-way encryption on *string*, using the host system's `crypt()` library routine. If *salt* is specified, then it must be a two-character string; otherwise, a salt is chosen randomly. The return value of `crypt()` is the encrypted string; the first two characters of the encrypted string are the salt used.

The encryption performed by this function has the property that it is very difficult to find a string which will produce a given result; however, a given string and a given salt will always yield the same encrypted string.

Examples:

```
crypt("foo", "ab")
     ⇒ "abQ9KY.KfrYrc"
```

explode

```
explode(string)
explode(string, separator, [want-blanks])
```

This function returns a list of the words in *string*. If the string *separator* is specified, then it is used as the word separator; otherwise a space (`" "`) is used. If the optional argument *want-blanks* is specified and is true, `explode()` will include zero-length words in the returned list; otherwise, it will not.

Examples:

```
explode(" foo  bar baz")
     ⇒ ["foo", "bar", "baz"]
explode("foo:bar:baz", ":")
     ⇒ ["foo", "bar", "baz"]
```

lowercase

`lowercase(`*string*`)`

This function returns the result of changing each uppercase character in *string* to lowercase.

```
lowercase("fOOBAr 23")
     ⇒ "foobar 23"
```

match_begin

`match_begin(`*string*`, `*search*`)`
`match_begin(`*string*`, `*search*`, `*separator*`)`

This function looks for the string *search* at the beginning of each word in *string*. The word separator is given by the string *separator* if it is specified; otherwise, a space (`" "`) is used. The return value of `match_begin()` is `1` if *search* was found at the beginning of a word in *string*, or `0` if not.

Examples:

```
match_begin("foo:bar:baz", "fo", ":")
     ⇒ 1
match_begin("foo bar baz", "ar")
     ⇒ 0
```

match_pattern

`match_pattern(`*pattern*`, `*string*`)`

This function matches the wildcard pattern *pattern* against *string*. A wildcard pattern is a string with asterixes ('*') signifying wildcards. A regular character matches itself, while a wildcard matches any number of arbitrary characters. The return value of `match_pattern()` is a list of the substrings of *string* which matched the wildcards in *pattern*, or `0` if the match fails.

`match_pattern()` allows you to do simple character-based matching. For matching against command formats, however, you should use the `match_template()` function.

Examples:

```
match_pattern("*", "foobar")
     ⇒ ["foobar"]
match_pattern("foo * bar * baz", "foo quux bar quuux baz")
     ⇒ ["quux", "quuux"]
match_pattern("foo * bar", "baz")
     ⇒ 0
```

match_regexp

`match_regexp(`*regexp*`, `*string*`, [`*case_matters*`])`

This function matches the regular expression *regexp*, a string,against the string *string*. If *case_matters* is specified and is true, the match is case-sensitive; otherwise, it is case-insensitive. If the match succeeds, `match_regexp()` returns a ten-element list giving the substitutions for the match (see below); otherwise, `match_regexp()` returns 0.

Coldmud uses a regular expression matcher written by Henry Spencer. Its syntax is very similar to the regular expression syntax used by Unix utilities like `ed` or `egrep`. Here is Spencer's description of his regular expression syntax:

> A regular expression is zero or more branches, separated by '|'. It matches anything that matches one of the branches.
>
> A branch is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.
>
> A piece is an atom possibly followed by '*', '+', or '?'. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by '+' matches a sequence of 1 or more matches of the atom. An atom followed by '?' matches a match of the atom, or the null string.

> An atom is a regular expression in parentheses (matching a match for the regular expression), a range (see below), '`.`' (matching any single character), '`^`' (matching the null string at the beginning of the input string), '`$`' (matching the null string at the end of the input string), a '`\`' followed by a single character (matching that character), or a single character with no other significance (matching that character).
>
> A range is a sequence of characters enclosed in '`[]`'. It normally matches any single character from the sequence. If the sequence begins with '`^`', it matches any single character not from the rest of the sequence. If two characters in the sequence are separated by '`-`', this is shorthand for the full list of ASCII characters between them (e.g. `[0-9]` matches any decimal digit). To include a literal '`]`' in the sequence, make it the first character (following a possible '`^`'). To include a literal '`-`', make it the first or last character.

The substitutions are the text in *string* which matches the parenthesized subexpressions in *regexp*. The first substitution is the text in *string* which matches the whole regexp. Thus, a regular expression can contain no more than nine parenthesized subexpressions. Substitutions are returned as two-element lists `[`*start*`,` *len*`]` giving the index of the matching text in *string* and the length of the text. When the substitutions are ambiguous, leftmost '`*`' matches are always as long as possible.

If *regexp* is not a valid regular expression, `match_regexp()` throws a `~regexp` error.

Examples:

```
match_regexp("bar", "fooBAR")
     ⇒ [[4, 3], [0, 0], [0, 0], [0, 0], [0, 0],
     [0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]
match_regexp("^([^ ]+) says, \"(.*)\"$", "Greg says, \"Hello.\"")
     ⇒ [[1, 19], [1, 4], [13, 6], [0, 0], [0, 0],
     [0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]
match_regexp("[0-9]+", " 300 100 200 ")
     ⇒ [[2, 3], [0, 0], [0, 0], [0, 0], [0, 0],
     [0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]
match_regexp("foo", "bar")
     ⇒ 0
match_regexp("foo", "Foo", 1)
     ⇒ 0
```

match_template

`match_template(`*template*`,` *string*`)`

This function matches the template *template* against the command *string*. The return value of `match_template()` is a list of fields resulting from the template match, or `0` if the match fails or if *template* is an invalid template.

A *template* is a sequence of word-patterns and wildcards separated by spaces, with wildcards never occurring more than one at a time. A *word-pattern* is a sequence of words separated by pipe characters ('|'). A word is a sequence of alphanumeric characters, with an optional question mark ('?') indicating the beginning of an allowed partial match. A *wildcard* is either a simple wildcard, represented by an asterix ('*') or a *coupled wildcard*, represented by the three-character sequence '*=*'.

That definition of a template is confusing, so we will now go back and explain each component of a template in more detail.

A *word-pattern* is a list of words separated by pipe characters. A word-pattern matches any of the words contained in it. The word-pattern `"look|examine"` matches either of the words `"look"` or `"examine"`. The word separator for template matching is always a space.

A word can include a question mark ('?') to indicate that partial matches that extend at least as far as the question mark are okay. The word pattern `"look|ex?amine"` matches any of the words `"look"`, `"ex"`, `"exa"`, `"exam"`, `"exami"`, `"examin"`, and `"examine"`.

When a word-pattern successfully matches a word in *string*, it results in a *field*, or string in the returned list. This field contains the word which matched the word-pattern.

A *simple wildcard* is represented by an asterix ('*'). A simple wildcard matches any number of words in *string*. If the wildcard is followed by a word-pattern in *template*, then it can also match a *quoted wildcard match*.

A quoted wildcard match is just like a `C--` string literal: it begins and ends with a double quote ('"'), and can include a literal double quote or backslash by preceding the character with a backslash ('\'). If the simple wildcard is followed by a word-pattern, and the words in *string* that the wildcard would match begin with a double quote, then the match must be a quoted wildcard match or the match fails, even if the match would have succeeded if the words were not treated as a quoted wildcard match. However, if the words that the wildcard would match begin with a backslash followed by a double quote, then the backslash is ignored and the double quote and the text following it are treated as regular words.

The template `"* bar"` matches any of the following strings:

```
foo bar
foo baz bar
"foo bar \\ \" baz" bar
\"foo baz bar
```

Matching against a simple wildcard produces one field, the words that the simple wildcard matched. If the wildcard matches a quoted wildcard match, then the beginning and ending double quotes are stripped out of the resulting field, as well as any backslashes used to escape characters inside the double quotes.

A *coupled wildcard* is represented by the three-character sequence '`*=*`'. It matches any sequence of words containing an equal sign ('='), and results in two fields, the text before the equal sign and the text after it. Any spaces surrounding the equal sign are ignored and do not show up in the resulting fields. The text before the equal sign can be a quoted wildcard match (as before, if it begins with a double quote, then it must be a quoted wildcard match or the match fails, unless the initial double quote is escaped by a backslash). If the coupled wildcard is followed by a word pattern, then the text after the equal sign can also be a quoted wildcard match.

The coupled wildcard is a special feature intended for parsing TinyMUD command formats. If possible, its use should be avoided.

If *template* is invalid, then the match usually fails, although this is not guaranteed.

Examples:

```
match_template("@desc?ribe * as *", "@descr me as foobar")
     ⇒ ["@descr", "me", "as", "foobar"]
match_template("@desc?ribe * as *", "@desc \"as is\" as foobar")
     ⇒ ["@desc", "as is", "as", "foobar"]
match_template("@desc?ribe * as *", "@desc \"as\" is as foobar")
     ⇒ 0
match_template("@desc?ribe * as *", "@desc \\\"as\" is as foobar")
     ⇒ ["@desc", "\"as\" is", "as", "foobar"]
match_template("@desc?ribe *=*", "@descr me =foobar")
     ⇒ ["@descr", "me", "foobar"]
match_template("@desc?ribe *=*", "@desc \"2+2=4\"= an equation")
     ⇒ ["@desc", "2+2=4", "an equation"]
match_template("l?ook|ex?amine *", "look at rose")
     ⇒ ["look", "at rose"]
```

pad

```
pad(string, length)
pad(string, length, filler)
```

This function pads or truncates *string* to the length *length*. If *filler* is specified, then it must be a single-character string to use as the filler character; otherwise, a space is used. If *length* is greater than the length of *string*, then *pad* adds filler characters on the right; however, you can force *pad* to add filler on the left by specifying *length* as a negative number.

Examples:

```
pad("foo", 6)
     ⇒ "foo   "
pad("foobar", 3)
     ⇒ "foo"
pad(tostr(29), -4, "0")
     ⇒ "0029"
```

strcmp

```
strcmp(string1, string2)
```

This function compares *string1* against *string2* and returns zero if they are equal, greater than zero if *string1* is lexically greater than *string2*, and less than zero if *string1* is lexically less than *string2*. The comparison performed by `strcmp()` is case-sensitive.

```
strcmp("Foo", "bar")
     ⇒ -28
strcmp("cashmir", "cashmiR")
     ⇒ 32
strcmp("foo", "foo")
     ⇒ 0
```

strlen

```
strlen(string)
```

This function returns the length of *string*.

Examples:

```
length("foo")
     ⇒ 3
```

strsub

strsub(*string*, *search*, *replace*)

This function returns the result of replacing each occurrence of the string *search* in *string* with the string *replace*.

Examples:

```
strsub("foobar", "bar", "baz")
     ⇒ "foobaz"
```

substr

substr(*string*, *start*)
substr(*string*, *start*, *length*)

This function returns a substring of *string* beginning at the character numbered by *start*. If *length* is specified, then the substring has that length; otherwise, the length is given by (length(*string*) - (*start* - 1)). If *start* is less than 1, if the length of the substring is negative, or if the sum of *start* and *length* is greater than one more than the length of the string, then substr() throws a ~range error.

Examples:

```
substr("foobar", 2, 3)
     ⇒ "oob"
substr("foobar", 3)
     ⇒ "obar"
substr("foobar", 7)
     ⇒ ""
```

uppercase

```
uppercase(string)
```

This function returns the result of changing each lowercase character in *string* to uppercase.

```
uppercase("fOOBAr 23")
     ⇒ "FOOBAR 23"
```

5.3 List Functions

The functions described in this section allow you to perform operations on list. None of these functions actually modify the lists passed as arguments; instead, they return modified lists.

delete

```
delete(list, position)
```

This function returns the result of deleting the element of *list* numbered by the integer *position*. If *position* is less than `1` or is greater than the length of *list*, then `delete()` throws a `~range` error.

Examples:

```
delete([2, 3, 4], 2)
     ⇒ [2, 4]
```

insert

```
insert(list, position, data)
```

This function returns the result of inserting *data* into *list* before the element numbered by the integer *position*. If *position* is less than one or is more than one more than the length of *list*, then `insert()` throws a `~range` error.

Examples:

```
insert([2, 3, 4], 3, 'foo)
     ⇒ [2, 3, 'foo, 4]
insert(["foo", 'bar, ~none], 4, 'baz)
     ⇒ ["foo", 'bar, ~none, 'baz]
```

listlen

`listlen(`*list*`)`

This function returns the length of *list*.

Examples:

```
length([2, "foo", 'bar])
     ⇒ 3
```

replace

`replace(`*list*`, `*position*`, `*data*`)`

This function returns the result of replacing the element of *list* numbered by *position* with *data*. If *position* is less than one or is greater than the length of *list*, then `replace()` throws a `~range` error.

Examples:

```
replace([2, 3, 4], 2, 'foo)
     ⇒ [2, 'foo, 4]
```

setadd

`setadd(`*list*`, `*data*`)`

This function returns the result of adding *data* to the end of *list* if it was not already somewhere in *list*. If *data* was already in *list*, then `setadd()` returns *list* unmodified.

Examples:

```
setadd([2, 3, 4], 'foo)
     ⇒ [2, 3, 4, 'foo]
setadd([2, 3, 4], 3)
     ⇒ [2, 3, 4]
```

setremove

`setremove(`*list*`,` *data*`)`

This function returns the result of removing the first occurrence of *data* from *list* if *data* exists in *list*. If *data* does not exist in *list*, then `setremove()` returns *list* unmodified.

Examples:

```
setremove([2, 3, 4, 'foo], 'foo)
     ⇒ [2, 3, 4]
setremove([2, 3, 4], 5)
     ⇒ [2, 3, 4]
setremove([2, 3, 2, 4], 2)
     ⇒ [3, 2, 4]
```

sublist

`sublist(`*list*`,` *start*`)`
`sublist(`*list*`,` *start*`,` *length*`)`

This function returns a sublist of *list* beginning at the element numbered by *start*. If *length* is specified, then the sublist has that length; otherwise, the length is given by `(length(`*list*`) - (`*start* `- 1))`. If *start* is less than `1`, if the length of the sublist is negative, or if the sum of *start* and *length* is greater than one more than the length of the list, then `sublist()` throws a `~range` error.

Examples:

```
sublist([2, 3, 4, 5, 6, 7], 2, 3)
     ⇒ [3, 4, 5]
sublist([2, 3, 4, 5, 6, 7], 3)
     ⇒ [4, 5, 6, 7]
sublist([2, 3, 4, 5, 6, 7], 7)
     ⇒ []
```

union

`union(`*list1*`, `*list2*`)`

This function returns the result of adding each element of *list2* which does not already exist in *list1* to the elements of *list1*. Elements which exist in *list2* more than once will only be added once, but duplicate elements in *list1* will remain.

```
union([2, 3, 4], [4, 5, 4, 6])
     ⇒ [2, 3, 4, 5, 6]
union([2, 2, 4, 5], [4, 5, 6, 6, 7])
     ⇒ [2, 2, 4, 5, 6, 7])
```

5.4 Dictionary Functions

The functions described in this section perform operations on dictionaries. These options do not actually modify the dictionaries they accept as arguments; instead, they return the modified dictionaries as their return values.

dict_add

`dict_add(`*dictionary*`, `*key*`, `*value*`)`

This function returns the result of adding the association `[`*key*`, `*value*`]` to the dictionary *dictionary*. If *key* already exists in *dictionary*, then `dict_add()` replaces the value of that key with *value*.

Examples:

```
dict_add(#[["foo", "bar"]], 3, 'quux)
     ⇒ #[["foo", "bar"], [3, 'quux]]
dict_add(#[["foo", 1], ["bar", 2], ["baz", 3]], "bar", 4)
     ⇒ #[["foo", 1], ["bar", 4], ["baz", 3]]
```

dict_add_elem

`dict_add_elem(`*dictionary*`, `*key*`, `*element*`)`

This function returns the result of adding *element* to the list value *key* in *dictionary*. If *dictionary* has no association with key *key*, then `dict_add_elem()` adds an association `[`*key*`, [`*element*`]]` to *dictionary*. If *dictionary* has an association with key *key*, and the value part of the association is a list, then `dict_add_elem()` adds *element* to the end of that list. If *dictionary* has an association with key *key*, but the value part of the association is not a list, then `dict_add_elem()` throws a `~type` error.

Examples:

```
dict_add_elem(#[["foo", ["bar"]]], "foo", "baz")
     ⇒ #[["foo", ["bar", "baz"]]]
dict_add_elem(#[], "foo", "bar")
     ⇒ #[["foo", ["bar"]]]
```

dict_contains

`dict_contains(`*dictionary*`, `*key*`)`

This function returns 1 if there is an association in *dictionary* with the key *key*, or 0 otherwise.

Examples:

```
dict_contains(#[["foo", "bar"]], "foo")
     ⇒ 1
dict_contains(#[["foo", "bar"]], "bar")
     ⇒ 0
```

dict_del

```
dict_del(dictionary, key)
```

This function returns the result of removing the association in *dictionary* which has the key *key*. If there is no such association, then `dict_del()` raises a `~keynf` exception.

```
dict_del(#[["foo", 1], ["bar", 2]], "foo")
     ⇒ #[["bar", 2]]
```

dict_del_elem

```
dict_del_elem(dictionary, key, element)
```

This function returns the result of removing *element* from the list value of *key* in *dictionary*. If *dictionary* contains no association with key *key*, then `dict_del_elem()` throws a `~keynf` error. If *dictionary* contains an association with key *key*, and it is not a list, then `dict_del_elem()` throws a `~type` error.

Assuming that *dictionary* contains an association with key *key*, and it is a list, `dict_del_elem()` removes the first occurrence of *element* in the list if *element* occurs in the list. If the resulting list is an empty list, then `dict_del_elem()` removes the association from *dictionary* altogether. If *element* does not occur in the list, then `dict_del_elem()` returns *dictionary* unchanged.

Examples:

```
dict_del_elem(#[["foo", ["bar", "baz"]]], "foo", "bar")
     ⇒ #[["foo", ["baz"]]]
dict_del_elem(#[["foo", ["baz"]]], "foo", "baz")
     ⇒ #[]
```

dict_keys

```
dict_keys(dictionary)
```

This function returns a list of the keys of the associations in *dictionary*.

```
dict_keys(#[["foo", 1"], ["bar", 2], ['baz, 3]])
     ⇒ ["foo", "bar", 'baz]
```

5.5 Buffer Functions

The functions described in this section apply to buffers. As with strings and lists, these functions will not actually modify any buffers passed as arguments, but will instead return the results of performing modifications on the buffers.

buffer_add

buffer_add(*buffer*, *byte*)

This function returns the result of adding *byte*, an integer, to the end of *buffer*. If *byte* is not between 0 and 255, the value appended to *buffer* will be the lower-order eight bits of *byte*.

Examples:

```
buffer_add(%[65, 66, 67], 10)
     ⇒ %[65, 66, 67, 10]
```

buffer_append

buffer_append(*buffer1*, *buffer2*)

This function returns the result of appending *buffer2* to the end of *buffer1*.

Examples:

```
buffer_append(%[65, 66, 67, 10], %[67, 66, 65, 10])
     ⇒ %[65, 66, 67, 10, 67, 66, 65, 10]
```

buffer_from_strings

`buffer_from_strings(`*list-of-strings*`, [`*terminator*`])`

This function returns a buffer constructed by appending the strings in *list-of-strings* together, treating the strings as sequences of ASCII values, with each string followed by *terminator*, a buffer. If *terminator* is not specified, then `buffer_from_strings()` uses the buffer `%[13, 10]`, a two-byte buffer containing the ASCII codes for a carriage return and a newline.

Examples:

```
buffer_from_strings(["foo", "bar"])
     ⇒ %[102, 111, 111, 13, 10, 98, 97, 4, 13, 10]
buffer_from_strings(["foo", "bar"], %[10])
     ⇒ %[102, 111, 111, 10, 98, 97, 4, 10]
```

buffer_len

`buffer_len(`*buffer*`)`

This function returns the length of the buffer *buffer*.

Examples:

```
buffer_len(%[65, 66, 67, 10])
     ⇒ 4
```

buffer_replace

`buffer_replace(`*buffer*`, `*pos*`, `*value*`)`

This function returns the result of replacing the byte numbered *pos* in *buffer* with the value *value*, with the first byte of *buffer* is numbered `1`. If *pos* is less than `1`, or greater than the length of *buffer*, then `buffer_replace()` throws a `~range` error.

Examples:

```
buffer_replace(%[65, 66, 67, 10], 3, 15)
    ⇒ %[65, 66, 15, 10]
```

buffer_retrieve

buffer_retrieve(*buffer*, *pos*)

This function returns the byte numbered *pos* in *buffer*, with the first byte of *buffer* is numbered `1`. If *pos* is less than `1`, or greater than the length of *buffer*, then `buffer_replace()` throws a `~range` error.

Examples:

```
buffer_retrieve(%[65, 66, 67, 10], 3)
     ⇒ 67
```

buffer_to_strings

buffer_to_strings(*buffer*, [*separator*])

This function returns a list of strings constructed from *buffer* by splitting it around the bytes given by the buffer *separator*. If *separator* is not specified, `buffer_to_strings()` uses the buffer `%[10]`, a single-byte buffer containing the ASCII code for a newline. `buffer_to_strings()` treats the bytes in *buffer* not accounted for by the separator as ASCII values; unprintable ASCII values are stripped. The last element of the returned list is a buffer containing the bytes in the buffer after the last separator found; no string is included in the returned list for these bytes.

Examples:

```
buffer_to_strings(%[65, 66, 67, 13, 10, 67, 66, 65, 13, 10, 66])
     ⇒ ["ABC", "CBA", %[66]]
buffer_to_strings(%[66, 10, 10, 65, 10])
     ⇒ ["B", "", "A", %[]]
buffer_to_strings(%[65, 66, 67, 13, 10, 67, 66, 65, 10, 66], %[66])
     ⇒ ["A", "CC", "A", %[]]
```

buffer_truncate

`buffer_truncate(`*buffer*`, `*length*`)`

This function returns the result of truncating *buffer* to the length *length*. If *length* is less than `0` or greater than the length of *buffer*, then `buffer_truncate()` throws a `~range` error.

Examples:

```
buffer_truncate(%[65, 66, 67, 10], 2)
     ⇒ %[65, 66]
```

5.6 Method Functions

The functions described in this section return information about the objects involved in the call to the current method: the caller, the definer, the sender, and the current object. If a method defined by object A, processing a message for object B, sends a message to object C, which results in a call to a method defined by object D, then A is the caller, B is the sender, C is the current object, and D is the definer.

caller

`caller()`

This function returns the dbref of the object which defines the method which called the current method, or `0` if the current method was called by the server.

definer

`definer()`

This function returns the dbref of the object which defines the current method.

sender

```
sender()
```

This function returns the dbref of the object which sent the current message, or `0` if the current method was called by the server.

task_id

```
task_id()
```

This function returns the ID of the current task. The task ID is an integer which begins at `0` for the first task and increases by `1` each time a task runs.

this

```
this()
```

This function returns the dbref of the current object.

5.7 Error Functions

The functions described in this section perform operations related to handling error conditions.

error

```
error()
```

This function, which you should only call inside an error handler for a catch statement (see Section 3.5 [Error-Handling Statements], page 16), returns the error code for the error which triggered the error handler. If you are not in an error handler, this function throws an `~error` error.

error_arg

```
error_arg()
```

This function, which you should only call inside an error handler for a catch statement (see Section 3.5 [Error-Handling Statements], page 16), returns the error argument specified in the `throw()` call which caused the error which triggered the error handler, or `0` if the no error argument was specified in the `throw()` caller or if the error was thrown by the interpreter. If you are not in an error handler, this function throws an `~error` error.

error_str

```
error_str()
```

This function, which you should only call inside an error handler for a catch statement (see Section 3.5 [Error-Handling Statements], page 16), returns the string argument specified in the `throw()` call which caused the error which triggered the error handler, or the interpreter's explanation of the error if the error was thrown by the interpreter. If you are not in an error handler, this function throws an `~error` error.

rethrow

`rethrow(`*error-code*`)`

This function, which you should only call inside an error handler for a catch statement (see Section 3.5 [Error-Handling Statements], page 16), continues propagating an error condition. The interpreter will abort the current method and throw an error of type *error-code* in the calling method. If you are not in an error handler, this function throws an `~error` error.

Examples:

```
rethrow(~perm);
```

throw

`throw(`*error-code*`,` *explanation*`,` `[`*arg*`])`

This function throws an error in the method which called the current method. The interpreter will abort the current method and throw an error of type *error-code* in the calling method. The string *explanation* will appear in the traceback. If *arg* is specified, then it can be retrieved in an error handler by `error_arg()`.

Examples:

```
throw(~perm, "Sender is not the system object.");
```

traceback

`traceback()`

This function, which you should only call inside an eror handler for a catch statement (see Section 3.5 [Error-Handling Statements], page 16), returns the traceback for the error which triggered the error handler. If you are not in an error handler, this function throws an `~error` error.

5.8 Communication Functions

The functions described in this section allow you to perform input and output to connections. All of these functions apply to all connections associated with the current object; you can never refer to a particular connection directly using these functions.

disconnect

`disconnect()`

This function closes all connections associated with the current object, and returns the number of connections closed.

echo

`echo(`*buffer*`)`

This function sends the bytes in *buffer* to all connections associated with the current object, and returns `1`.

Examples:

```
echo(%[65, 66, 67, 13, 10])
     ⇒ 1
```

echo_file

`echo_file(`*name*`)`

This function sends the contents of the file named by the string *name* to all connections associated with the current object. The filename *name* must not contain the sequence '`../`'. `echo_file()` returns `1` if *name* exists in the subdirectory `"text"` of the current directory; otherwise, the result of `echo_file()` is a `~file` error.

5.9 Object Functions

The functions described in this section allow you to modify or retrieve information from the current object.

add_parameter

`add_parameter(`*name*`)`

This function adds a parameter named *name* to the current object. *name* must be a symbol. If *name* is already a parameter on the current object, then `add_parameter` throws a `~paramexists` error. Otherwise, `add_parameter()` returns 1.

ancestors

```
ancestors()
```

This function returns a list of the dbrefs of the ancestors of the current object, in the order that they would normally be searched for methods. The dbref of the current object will be the first element of the list.

children

```
children()
```

This function returns a list of the dbrefs of the children of the current object, in no particular order.

compile

`compile(`*code*`,` *name*`)`

This function compiles the list of strings *code* and uses the result as the definition of the method named by the symbol *name*. If there were errors in compiling *code*, then `compile()` returns a list of strings describing the errors; otherwise `compile()` returns an empty list.

Examples:

```
compile(["echo(\"foo\");"], 'foo)
     ⇒ []
compile(["echo(\"foo\")"], 'foo)
     ⇒ ["Line 2: parse error"]
```

del_method

`del_method(`*name*`)`

This function removes the method named by the symbol *name* from the current object. `del_method()` returns `1` if there was a method named *name* on the current object; otherwise, it throws a `~methodnf` error.

del_parameter

`del_parameter(`*name*`)`

This function removes the parameter named by the symbol *name* from the current object. `del_parameter()` returns `1` if there was a parameter named *name* on the current object; otherwise, it throws a `~paramnf` error.

find_method

`find_method(`*name*`)`

This function returns the object which contains the method definition which would be used if a message *name* were sent to the current object. If none of the ancestors of the current object define a method *name*, then `find_method()` throws a `~methodnf` exception.

find_method

`find_next_method(`*name*`,` *after*`)`

This function returns the ancestor of the current object which contains the method definition which would be used if a method *name* defined by the object *after* executed an inspecific `pass()`. If no method would be found, then `find_method()` throws a `~methodnf` exception.

get_var

`get_var(`*name*`)`

This function returns the value of the object variable on the current object referred to by the parameter *name* (a symbol) on the current method's defining object. If the current method's defining object does not have a parameter *name*, then `get_var()` throws a `~paramnf` error.

list_method

```
list_method(name)
list_method(name, indentation)
list_method(name, indentation, parenthesization)
```

This function returns a list of strings containing a decompiled version of the method named by the symbol *name*. If the method *name* is not defined on the current object, then `list_method()` results in a `~methodnf` error. Note that *name* must be defined on the current object, not on its ancestors.

The decompiled code may not look exactly like the code that was compiled to produce the method. The only guarantee is that it will compile to a method with the same behavior.

If you specify the integer *indentation*, then the decompiler will use that number of spaces for indenting. The default is 4 spaces. If you specify the integer *parenthesization* and it is non-zero, then the decompiler will put parentheses around every subexpression. Otherwise, the compiler will only use parentheses when needed.

methods

```
methods()
```

This function returns a list of symbols giving the names of the methods defined on the current object.

parameters

```
parameters()
```

This function returns a list of symbols giving the names of the parameters defined on the current object.

parents

`parents()`

This function returns a list of the dbrefs of the parents of the current object.

set_var

`set_var(`*name*`,` *data*`)`

This function assigns the value *data* to the object variable on the current object referred to by the parameter *name* (a symbol) on the current method's defining object. If the current method's defining object does not have a parameter *name*, then `get_var()` throws a `~paramnf` error.

5.10 Administrative Functions

The functions in this section allow you to perform privileged operations. These functions can only be called from the system object. For any other object, the result of any of these functions is an `~perm` error.

binary_dump

`binary_dump()`

This function writes out all modified objects in the object cache to the disk database. This guarantees that the disk database files 'db', '`db.dir`', and '`db.pag`' are consistent.

bind

`bind(`*port*`,` *receiver*`)`

This function instructs the server to begin listening on the port numbered *port*, with the receiver object *receiver*. If the server is already listening on that port, then the receiver object is changed, with no other effect. If the socket cannot be created, then `bind()` throws a `~socket` error. If the server cannot bind to the port *port*, then `bind()` throws a `~bind` error. Otherwise, `bind()` returns 1.

chparents

`chparents(`*dbref*`, `*parents*`)`

This function changes the parents of the object referred to by *dbref* to the list of dbrefs in *parents*. If any of the dbrefs in *parents* do not refer to an existing object, then `chparents()` throws an `~objnf` error. If any of the parents have *dbref* as an ancestor, or are *dbref* themselves, then `chparents()` throws a `~parent` error. If *dbref* refers to the root object, or *parents* is an empty list, then then `chparents()` throws a `~perm` error. Otherwise, `chparents()` returns `1`.

conn_assign

`conn_assign(`*dbref*`)`

This function sets the handler object of the current connection to *dbref*. `conn_assign()` always returns `1`.

Examples:

```
conn_assign(#76)
     ⇒ 1
```

connect

`connect(`*address*`, `*port*`, `*receiver*`)`

This function establishes a connection to the remote Internet host named by *address* (a string giving the IP address of the remote host) at the port *port*.

If *address* is not a valid IP address, then `connect()` throw an `~address` error. If a socket cannot be created for the connection, then `connect()` throws a `~socket` error. Otherwise, `connect()` attemptes to connect to the remote host and returns 1 immediately; it does not wait to see if the connection attempt succeeded.

If the connection succeeds, then the server will send the object *receiver* a `connect` message, with one argument the task ID of the task which called `connect()` (see [task_id], page 57). *receiver* then becomes the handler object for the connection, and receives `parse` messages when lines arrive from the connection, as well as a `disconnect` message when the connection terminates.

If the connection fails, then the object *receiver* will receive a `failed` message with two arguments, the task ID of the task which called `connect()`, and an error code: `~refused` indicates that the connection was refused, `~net` indicates that the network of the remote host could not be reached, `~timeout` indicates that the connection attempt timed out, and `~other` indicates that some other error occurred.

create

`create(`*dbref*`,` *parents*`)`

This function creates an object with the dbref *dbref* and the parents *parents*, which should be a list of dbrefs referring to existing objects. If any of the parent dbrefs do not refer to existing objects, then `create()` throws a `~objnf` error. If an object with the dbref *dbref* already exists, `create()` throws a `~perm` error. Otherwise, `create()` returns *dbref*.

data

`data(`*dbref*`)`

This function retrieves all the variables on the object given by the dbref *dbref*. The return value is a dictionary whose associations are between ancestors of the object and subdictionaries giving the variable values for each ancestor's parameters. Each subdictionary's associations are between parameters (expressed as symbols) of the ancestor in question (as symbols) and the values of the variables corresponding to those parameters. Thus, if the object given by *dbref* defines a variable corresponding to a parameter *parameter* on an ancestor *ancestor*, then `data(`*dbref*`)[`*ancestor*`][`*parameter*`])` is the value of that variable.

The subdictionaries in the dictionary returned by `data()` only contains [*parameter*, *value*] associations for variables which have been assigned values on the object given by *dbref*, not variables which have never been assigned values and which still have the default value `0`. Also, if no variables have been assigned values for any parameters on an ancestor of the object given by *dbref* (as is the case when the ancestor has no parameters), then there will not be an association for that ancestor in the dictionary returned by `data()`. Neither the keys in the dictionary returned by `data()` nor the parameters in the subdictionaries have any particular ordering.

If *dbref* is not the dbref of an object, then `data()` throws an `~objnf` error.

destroy

`destroy(`*dbref*`)`

This function destroys the object given by the dbref *dbref*. If any methods are currently executing on that object, or any methods are executing which are defined by that object, then it will not be destroyed immediately. If no object exists with the dbref *dbref*, then `destroy()` throws a `~objnf` error.

Objects left orphaned by the destruction of their only parent are reparented to the parents of the parent which was destroyed.

`destroy()` throws a `~perm` error if you attempt to destroy the root object or system object, which is not allowed. Otherwise, `destroy()` returns `1`.

log

`log(`*string*`)`

This function sends *string* to the standard error string, prefixed by the date. `log()` always returns `1`.

Examples:

```
log("foo")
     ⇒ 1
```

run_script

```
run_script(name, arguments)
run_script(name, arguments, background-flag)
```

This function executes the script named by string *name*, passing it the arguments in the list *arguments*. Each element of *arguments* must be a string. Coldmud looks for the script named by *name* in the directory '`scripts`' in the current directory. *name* cannot contain a '`../`' sequence; that is, it cannot walk back down the directory hierarchy.

If *background-flag* is specified, it must be an integer. If it is non-zero, then *script* is run in the background. Otherwise, Coldmud waits for the script to execute.

If *background-flag* is not specified, `run_script()` returns the return status of the script, or `-1` if it does not successfully execute the script. If *background-flag* is specified, then `run_script()` might return `-1` if the script is not successfully executed, and it may even return the return status of the script if the script executes extremely quickly, but it will usually return `0`.

set_heartbeat_freq

```
set_heartbeat_freq(seconds)
```

This function sets the frequency of the server heartbeat to *seconds* seconds. If *seconds* is less than or equal to zero, then the heartbeat is turned off; otherwise, the system object will receive a `heartbeat` message approximately every *seconds* seconds.

shutdown

```
shutdown()
```

This function updates the binary database in the same manner as `binary_dump()`, and causes the server to shut down at the next cycle of the main loop.

text_dump

```
text_dump()
```

This function writes a text database dump to the file 'textdump'. text_dump() returns 1 if it is successful, or 0 if it cannot write the text dump. text_dump() uses a temporary file 'textdump.new' to avoid overwriting the old 'textdump' until it has finished; thus, if there is a server or machine crash while the text dump is in progress, the partial text dump will be in 'textdump.new', and the old 'textdump' will be unmodified.

unbind

unbind(*port*)

This function instructs the server to stop listening on the port numbered *port*. If the server was not already listening on that port, then unbind() throws a ~servnf error; otherwise, unbind() returns 1.

5.11 Miscellaneous Functions

The functions in this section perform operations which do not fit into any other category.

abs

abs(*number*)

This function returns the absolute value of *number*.

Examples:

```
abs(-6)
     ⇒ 6
abs(7)
     ⇒ 7
```

ctime

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

This function converts the integer *time* into a string format. If *time* is not specified, then `ctime()` uses the current time.

Examples:

```
ctime(739180536)
    ⇒ "Fri Jun  4 03:55:36 1993\n"
```

max

```
max(value1, value2, ...)
```

This function returns the maximum of its arguments. All of the arguments must be of the same type, and must be integers or strings.

```
max(3, 7, 9, 5)
    ⇒ 9
max("Foo", "aardvark", "bar", "Quux")
    ⇒ "Quux"
```

min

```
min(value1, value2, ...)
```

This function returns the minimum of its arguments. All of the arguments must be of the same type, and must be integers or strings.

```
min(3, 7, 9, 5)
    ⇒ 3
min("Foo", "aardvark", "bar", "Quux")
    ⇒ "aardvark"
```

random

```
random(max)
```

This function returns a random integer between one and *max*.

time

```
time()
```

This function returns the system time in seconds since midnight GMT, January 1, 1970. You can use `ctime()` to turn this number into a string.

version

```
version()
```

This function returns the version number of the Coldmud server as a list of three numbers. The first two numbers are the major and minor release numbers; the third number is a bug-fix release number.

Examples:

```
version()
     ⇒ [0, 10, 0]
```

6 Running and Maintaining Coldmud

This chapter is intended for Coldmud adminstrators. It documents Coldmud's behavior on startup, how Coldmud databases should be maintained, and how Coldmud manages network connections. All of the functions referred to in this chapter are administrative functions, and only work if called by the system object (see Section 2.5 [System Object], page 6).

6.1 Starting the Server

Coldmud has the following usage:

`coldmud` *directory* [*other arguments*]

The first argument specifies the database directory, which can be relative to the current directory. You can specify any number of arguments after *directory*; these will be visible to the `startup` method on the system object.

6.2 Disk Database

Coldmud normally operates using a binary disk-based database, storing only a small number of objects in memory at any given time. This number is usually no more than the product of the cache width and the cache depth (fixed at 7 and 23 in this version).

Coldmud's database is normally stored in binary format in the file '`binary/objects`' (relative to the database directory) and in an ndbm database with the prefix '`binary/index`'. The file '`binary/clean`' exists when the database is consistent. The functions `binary_dump()` and `shutdown()` force binary database consistency.

Because ndbm databases are usually byte-order-dependent, a binary database generated by a Coldmud process on one machine cannot be guaranteed to work with a process on another machine. Binary databases are also heavily version-dependent; small changes in the internal format of an object in a new version of the server will invalidate old binary databases, even if the `C--` language and the conceptual structure of a Coldmud object remain the same in the new version.

Coldmud also supports a text format for databases. The `text_dump()` function stores a text database dump in the file '`textdump`'. Text dumps specify the database in terms of the `C--` language and a few simple directives, so they are compatible with any version of Coldmud which can understand the `C--` code contained in it. Text dumps are also simple enough to be written and edited by humans, so they provide a good mechanism for distribution of core databases.

At startup time, Coldmud looks for the file '`binary/clean`' to determine if a consistent binary database exists and was generated by the same version of Coldmud as the running process. If a clean binary database exists, Coldmud will start up very quickly, pausing only to read in the root object and system object. Otherwise, Coldmud tries to read in a text dump from the file '`textdump`'. You can force the use of a text dump by simple removing the file '`binary/clean`'. Coldmud will fail to start if it cannot find a consistent binary database or a text dump.

6.3 Connections

As a network server, Coldmud has the ability to listen for Internet connections on ports. The `bind()` function instructs Coldmud to listen on a port, with some object acting as a *receiver object* for that port.

Network connections have associated with them a *handler object*. When a connection occurs on a port, Coldmud initially uses the receiver object for the port as the handler object for the connection, and sends a `connect` message to the handler object, with two arguments: a string giving the IP address of the remote host, and an integer giving the port of the connection on the remote host. Because there is no way to distinguish between connections with the same handler object, the `connect` method on the handler object should arrange to have the system object change either the receiver object for the port (using `bind()`) or the handler object for the connection (using `conn_assign()`).

When text arrives from a network connection, Coldmud sends a `parse` message to the handler object for that connection, with the text as a buffer argument. The `parse` method can then use `buffer_to_strings()` to convert the buffer to a list of text lines, if that is the usual form of input.

When a network connection is terminated, Coldmud sends a `disconnect` message to the handler object for that connection.

Coldmud can also make make connections actively, using the `connect()` function. The third argument to `connect()` specifies a receiver object for the new connection; the server sends a `connect` message to the receiver object upon success, or a `failed` message to the receiver object upon failure, as described in [connect], page 65.

Function Index

A

abs 69
add_parameter 60
ancestors 61

B

binary_dump 64
bind 64
buffer_add 53
buffer_append 53
buffer_from_strings 54
buffer_len 54
buffer_replace 54
buffer_retrieve 55
buffer_to_strings 55
buffer_truncate 56

C

caller 56
children 61
chparents 65
class 35
compile 61
conn_assign 65
connect 65
create 66
crypt 39
ctime 70

D

data 66
definer 56
del_method 61
del_parameter 62
delete 47
destroy 67
dict_add 50
dict_add_elem 51
dict_contains 51
dict_del 52
dict_del_elem 52
dict_keys 52
disconnect 59

E

echo 60
echo_file 60
error 57
error_arg 58
error_str 58
explode 39

F

find_method 62

G

get_var 62

I

insert 47

L

list_method 63
listlen 48
log 67
lowercase 40

M

match_begin 40
match_pattern 40
match_regexp 41
match_template 42
max 70
methods 63
min 70

P

pad 45

parameters 63
parents 64
pass 30

R

random 71
replace 48
rethrow 58
run_script 68

S

sender 57
set_heartbeat_freq 68
set_var 64
setadd 48
setremove 49
shutdown 68
strcmp 45
strlen 45
strsub 46
sublist 49
substr 46

T

task_id 57
text_dump 68
this 57
throw 58
time 71
todbref 35
toerr 36
toint 36
toliteral 36
tostr 37
tosym 37
traceback 59
type 38

U

unbind 69
union 50
uppercase 47

V

valid 38
version 71

Concept Index

A

Addition 26
Administrative functions 64
Ambiguous operator expressions 25
Ancestor precedence 3
And operator 27
Argument list splicing 31
Arithmetic expressions 26
Assigning to variables 24
Assignment statement 10
Associating objects with connections 74
Association of operators 25
Associative arrays 22

B

Behavior of objects 3
Binary operators 25
Binding precedence of operators 25
Break statement 14
Buffer data type 18
Buffer functions 53
Buffers 23
Built-in function calls 28
Built-in functions 35
Built-in local variables 24

C

C-- data 18
C-- language 7
Calling functions 28
Calling methods 29
Case-insensitivity of strings 27
Case-sensitivity of identifiers 8
Cases 12
Catch statement 16
Characteristics of objects 3
Checkpoint databases 73
Closed connections 74
Coldmud Overview 1
Combining data into lists 20
Comment statement 10
Communicating with the server 6
Communicating with users 59
Comparing data 27
Components of a method 9
Compound statement 10
Conditional operator 27
Conditional statements 12
Connecting to remote hosts 74
Connections 74
Consistency of databases 73
Constant expressions 18
Continue statement 14
Counting 14
Critical expression 30
Current object 4, 56

D

Data functions 35
Data types 18
Database 73
Database references 3
Dbref data type 18
Dbref names 20
dbrefs 3
Declaring local variables 9
Defining methods 7
Defining object behavior 7
Descriptions of functions 35
Dictionaries 22
Dictionary data type 18
Dictionary functions 50
Disallowing overrides in a method 9
Disk database 73
Division 26

E

Equality of data 27

Error data type 18
Error functions 57
Error handlers 16
Error propagation 30
Error-handling expressions 30
Error-handling statements 16
Errors 33
Event notification 6
Example method 7
Executing statements conditionally 12
Exiting loops prematurely 14
Expression statement 10
Expressions 18

F

Files used by Coldmud 73
Files, sending to users 59
For-list statement 14
For-range statement 14
Frob constructors 23
Frob data type 18
Frobs 23
Function calls 28
Function descriptions 35
Functions to handle errors 57

G

Greater or equal to operator 27
Greater than operator 27

H

Handling error conditions 33
Handling specific errors 16

I

I/O data type for networks 23
If statement 12
If-else statement 12
Ignoring errors 9, 30
Including lists in arguments 31
Indexing lists 20
Inequality of data 27
Infix expressions 25
Information about the current method 56
Inheritance 3
Input and output functions 59
Integer data type 18
Invoking the server 73

L

Less than operator 27
Less than or equal to operator 27
Lightweight objects 23
List comparison 27
List constructors 20
List data type 18
List functions 47
Lists 20
Literal expressions 18
Local variable declarations 9
Local variables 24
Logic operators 27
Looping statements 14

M

Making connections to remote hosts 74
Making lists 20
Managing connections 74
Manipulating data 35
Manipulating dictionaries 50
Manipulating lists 47
Manipulating objects 60
Manipulating strings 39
Mappings 22
Message expressions 29
Messages from the server 4
Messages to objects 4
Messages to the system object 6
Method functions 56
Method structure 9
Methods 7
Miscellaneous functions 69
Modifying default methods 30
Modulo operations 26
Multiple inheritance 3
Multiplication 26

N

Names 20
Negation 26
Network I/O data type 23
New connections 74
Non-overridable methods 3
Non-overridable methods, declaring 9
Not operator 27

O

Object functions 60
Object IDs 3
Object names 20
Object variables 5, 24
Objects 3
Octet vectors 23
Operations on buffers 53
Operations on data 35
Operations on lists 47
Operations on objects 60
Operations on strings 39
Operators 25
Operators for arithmetic 26
Options to the server 73
Or operator 27
Output to a connection 59
Override blocking in a method 9
Overview of Coldmud 1

P

Parameters on objects 5
Parents 3
Parsing text from connections 74
Passing errors as themselves 30
Passing messages 30
Performing arithmetic 26
Precedence of ancestors 3
Precedence of operators 25
Premature loop exits 14
Privileged operations 64
Processing messages 4
Propagation expression 30

R

Raw byte vectors 23
Reading text from connections 74
Receiving text from connections 74
Referring to objects 3
Relational operators 27
Resolving ambiguous expressions 25
Retrieving data from lists 20
Retrieving variable contents 24
Return statement 10
Runnning the server 73

S

Searching lists 20
Sending messages 29
Sending object 56
Sending text to users 59
Server operations 64
Simple statements 10
Slots (variables) on objects 5
Splicing 31
Starting the server 73
Startup event notification 6
Statements 10
Stopping loops 14
Storing information on objects 5
String comparison 27
String data type 18
String functions 39
Structure of methods 9
Subtraction 26
Switch statement 12
Symbol comparison 27
Symbol data type 18
Symbol translation to dbrefs 20
Symbolic names 20
Syntax of methods 9
Syntax of operators 25
System object 6

T

Text database dumps 73
Text files, sending to users 59

Throwing errors 57
Ticks 7
Tokens 8
Tracebacks, getting 57
Translating symbolic names to dbrefs 20
Traversing lists 14
Types of C-- data 18
Types of C-- operators 25

U

Unary operators 25
Unprintable characters 23
User communication 59
Using default methods 30
Using operators in expressions 25
Using variables 24

V

Variable expressions 24
Variables 24
Variables on objects 5

W

While statement 14

Short Contents

1 Overview 1
2 Objects 3
3 Defining methods 7
4 Errors 33
5 Function Descriptions 35
6 Running and Maintaining Coldmud 73
Function Index 77
Concept Index 79

Table of Contents

1 Overview **1**

2 Objects **3**

2.1 Database References 3
2.2 Inheritance 3
2.3 Messages and Methods 4
2.4 Variables and Parameters 5
2.5 The System Object 6

3 Defining methods **7**

3.1 Example Method 7
3.2 Tokens 8
3.3 Method Structure 9
3.4 Statements 10
3.4.1 Simple Statements 10
3.4.2 Conditional Statements 12
3.4.3 Looping Statements 14
3.5 Error-Handling Statements 16
3.6 Expressions 18
3.6.1 Data Types 18
3.6.2 Names 20
3.6.3 Lists 20
3.6.4 Dictionaries 22
3.6.5 Buffers 23
3.6.6 Frobs 23
3.6.7 Variables 24
3.6.8 Operators 25
3.6.8.1 Precedence 25
3.6.8.2 Arithmetic 26
3.6.8.3 Logic 27
3.6.8.4 Conditional Operators 27
3.6.9 Calling Functions 28
3.6.10 Sending Messages 29
3.6.11 Passing Messages 30
3.6.12 Error-Handling Expressions 30
3.6.13 Splicing 31

4 Errors **33**

5 Function Descriptions **35**

5.1 Data functions 35
- class 35
- todbref 35
- toerr 36
- toint 36
- toliteral 36
- tostr 37
- tosym 37
- type 38
- valid 38

5.2 String Functions 39
- crypt 39
- explode 39
- lowercase 40
- match_begin 40
- match_pattern 40
- match_regexp 41
- match_template 42
- pad 45
- strcmp 45
- strlen 45
- strsub 46
- substr 46
- uppercase 47

5.3 List Functions 47
- delete 47
- insert 47
- listlen 48
- replace 48
- setadd 48
- setremove 49
- sublist 49
- union 50

5.4 Dictionary Functions 50
- dict_add 50
- dict_add_elem 51
- dict_contains 51
- dict_del 52
- dict_del_elem 52

dict_keys 52
5.5 Buffer Functions 53
buffer_add 53
buffer_append 53
buffer_from_strings 54
buffer_len 54
buffer_replace 54
buffer_retrieve 55
buffer_to_strings 55
buffer_truncate 56
5.6 Method Functions 56
caller 56
definer 56
sender 57
task_id 57
this 57
5.7 Error Functions 57
error 57
error_arg 58
error_str 58
rethrow 58
throw 58
traceback 59
5.8 Communication Functions 59
disconnect 59
echo 60
echo_file 60
5.9 Object Functions 60
add_parameter 60
ancestors 61
children 61
compile 61
del_method 61
del_parameter 62
find_method 62
find_method 62
get_var 62
list_method 63
methods 63
parameters 63
parents 64
set_var 64
5.10 Administrative Functions 64

binary_dump 64
bind 64
chparents 65
conn_assign 65
connect 65
create 66
data 66
destroy 67
log 67
run_script 68
set_heartbeat_freq 68
shutdown 68
text_dump 68
unbind 69
5.11 Miscellaneous Functions 69
abs 69
ctime 70
max 70
min 70
random 71
time 71
version 71

6 Running and Maintaining Coldmud 73
6.1 Starting the Server 73
6.2 Disk Database 73
6.3 Connections 74

Function Index 77

Concept Index 79