TinyMUD 2.4 Reference Manual (preliminary version)

James Aspnes
School of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

August 12, 2005

1 Introduction

TinyMUD 2 was written as a successor to the original TinyMUD and its many descendants. Its principal features include a specialized programming language designed to grant enough power to produce interesting effects without granting so much power that a programmer can easily damage or shut down the TinyMUD server, an inheritance hierarchy that allows the easy construction of classes of objects which can be used by non-programmers, and an efficient parser that supports (relatively) complex inputs allowing a command to take multiple objects. One of the main goals of TinyMUD 2 has been to eliminate nearly all game-specific code from the server implementation, and move it instead into the database, coded in the built-in programming language.

This document describe the built-in programming language and the behavior of the TinyMUD 2 parser.

2 Language

This section describes the features of the programming language available in TinyMUD 2.4.

2.1 Types

The data types supported by the language are: objects, booleans, strings, numbers, actions, sets, and times. All variables except those referring to objects are prefixed with a single-character type specifier, as follows:

`x`	object
`?x`	boolean
`$x`	string
`%x`	number (integer)
`&x`	action
`@x`	set; an unordered collection of objects
`~x`	time

Values of type string and type action can be used interchangeably. Values of all types may be used as booleans; they take on the value `?false` if they are null and `?true` otherwise. This feature is primarily useful for detecting "empty" variables. Values of type number are automatically coerced to their decimal expansion when used as a string.

Most language primitives restrict the types of their arguments; these restrictions will be described with each primitive.

2.2 Constants

All constants of type number consist of a contiguous sequence of decimal digits, e.g. `1`, `5567`, etc. There are only two boolean values, represented by the constants `?true` and `?false`.

String constants have a rather complicated syntax. In their simplest form, string constants look much like string constants in many other languages: a sequence of characters delimited by double-quotes, as in `"Every good boy deserves favor."`. The set of characters that may appear in side a string constant is severely restricted— only printing ASCII characters (including space) and the tab character are allowed. By convention the tab character is used to represent newlines within a "multi-line" string, as the newline character is reserved for separating inputs to and outputs from the server process.

String constants may also appear in the form of "giant" strings. These are delimited by matching pairs of square brackets. Other string constants may be nested within a giant string constant. Some examples of giant string constants:

```
[Every good boy deserves favor.]
```

```
[The sign says "ELVIS NEEDS BOATS"]
[set thing.&code to [tell "You lose!" to you]]
```

Double-quotes appearing within a simple string constant may be escaped using a backslash, as in `"The sign says \"ELVIS NEEDS BOATS\""`, which will print as `The sign says "ELVIS NEEDS BOATS"`. Backslashes and square brackets (in giant strings) may also be escaped using backslashes. The effect of putting a backslash before other characters is unpredictable.

2.2.1 Null constants

Two special constants exist for referring to "null" values, which are the default values held by variables of type object, string, or action. These are `nothing` (for objects) and `$null` (for strings and actions.) Null values are used primarily for shadowing inherited variable bindings. The null values of type number and boolean are `0` and `?false`, respectively. It is dangerous to depend on the existence of null values for types time and set.

2.3 Variables

Every object can possess variables of any of the seven types. A variable name by itself refers to a component of `me`. It is possible to refer to variables on other objects using the dot operator,
e.g. `location.owner.pet_dog.%number_of_fleas`. Most variable names may contain only letters, digits, and the underscore character; action names may also contain a single occurence of one of the characters `<` or `>`. Object variables may not start with a digit to prevent confusion with numeric constants. It is recommended that non-action variable names beginning with an underscore be avoided except for "temporary" variables that are only used within a single action invocation.

Variables occuring in a statement return their value. It is possible to set the value of most variables using the `set` statement, e.g.:

```
set dog.%fleas to 5
set location.$description to "Foo"
set ?happy to ?true
set broken_container.&take>from to [tell "You lose!" to you]
```

Variables of type set cannot appear in a set statement (other mechanisms exists for modifying such variables, as described below.) An object can only

set variables on objects it controls. Any object may read any variable on any other object.

Variables may also be cleared using the clear statement, e.g.:

```
clear dog.%fleas
clear location.$description
clear ?happy
clear broken_container.&take>from
clear @tools
```

Clearing a variable has the affect of removing it completely from an object, allowing a default value to be inherited from its ancestors. Variables of any type may be cleared.

It is necessary to have control of an object to modify any of its variables. However, any object can read any variable.

Certain variables have special meaning to the system and thus have additional restrictions on their use:

`me` Holds the object processing an action. May not be set.

`you` Holds the object which initiated an action, i.e. the object corresponding to the user which typed the currently-executing command. May not be set.

`next` Iteration variable inside loops; holds the current object in the set or contents list being iterated over. May not be set, and has no meaning outside of a loop.

`$text` Any unparsed text from the command line. May not be set.

`~time` The current time.

`%id` The unique ID number of an object. ID numbers also exist for strings (e.g. `$name.%id`) and other types. Use of ID numbers is discouraged.

`%count` The number of objects contained in an object's contents set. `%count` may also be used on set variables to count their contents, e.g. `TOP.@connected_players.%count`.

`%random` Returns a different non-negative random number every time it is used.

`location` The location of an object. This may not be set directly, although it is affected by the `move` statement.

`owner` The owner of an object. May only be set by wizard objects.

`parent` The parent of an object, used to control inheritance. It is not permitted to set the parent of an object to one of its descendants.

`$aliases` A string listing the names with which an object may be named on the command line. The aliases are separated by the pipe character, as in `"George | george | G"`. Case is significant.

`create` Returns a new object every time it is used. May only be used by wizard objects and may not be set.

There are also several boolean variables with special meaning to the system:

`?player` This object is a player. May only be set by wizard objects.

`?builder` Reserved for future expansion. May only be set by wizard objects.

`?programmer` Allows use of syntax from the command line. May only be set by wizard objects.

`?wizard` Objects with the `?wizard` bit set have special privileges, including implicit control over all other objects that do not have their `?admin` bit set. The `?wizard` bit may only be set by objects possessing the `?admin` bit.

`?admin` Gives the object the ability to set `?wizard` bits and execute certain system commands. Cannot be set or reset; under normal conditions, only the TOP object will have this bit set on it.

`?connected` Tells whether a player is connected or not. Automatically maintained by the system; can also be set by admin objects.

`?paranoid` If true, any tell sent to this player will be annotated with the id of its source.

`?open` If true, relaxes some of the restrictions on movement. Can be set by any controlling object.

There are also a large number of action names that have special meaning to the system. As these action names begin with underscores, they cannot be parsed from the command line:

`&_default` Called on the location of the current player if it is impossible to parse the command line; `$text` is set to the entire input.

`&_before` Called on both the location of the current player and on the current player before an action is handled.

`&_after` Called on both the location of the current player and on the current player after an action is handled. If the player moves during the action, it is still the original location which gets the `&_after` action.

`&_invoke` Called on an object if the command line matches one of its aliases.

`&_startup` Called exactly once on every object when the system is started.

`&_connect` Called on a player when the player connects.

`&_disconnect` Called on a player when the player disconnects.

`&_tick` Called when a delay expires.

2.3.1 Inheritance

Variables of all types except set are inherited. If the system cannot find a value for a particular variable on an object, it will try instead to use the value of that variable on the object's parent, and will similarly continue to ascend the chain of ancestors until either a value is found or no more ancestors exist. In the latter case, a null value is returned whose meaning depends on the type of the variable.

The semantics of inheritance create a distinction between the absence of a variable on an object (which allows ancestors' values to be inherited) and the presence of a variable whose value is null. The `clear` statement always results in the former state; if the latter state is preferred it is necessary to explicitly set the variable to the appropriate null constant.

2.3.2 Set variables

Variables of type set are unusual. Unlike variables of other types, they are not inherited. They also may not be set; instead, two additional statements are provided for manipulating set variables. The add/to statement adds an object to a set, as in `add dog to kennel.@inmates`. Similarly, the take/from statement removes an object from a set, as in
`take dog from kennel.@inmates`. The same restrictions are placed on changing a set variable as are placed on changing a variable of any of the other types; the object requesting the change must have control over the object of which the set variable is a component.

2.4 Expressions

The usual simple arithmetic operations on numbers, `+, -, *, /` are supported, with the normal precedence, as is unary minus and the remainder operator `mod` (e.g. `%random mod 3`.) Times may be subtracted to yield a number which counts the seconds between them; a number may also be added to or subtracted from a time to yield a new time, and a time may be taken mod a number to yield a number. This last facility allows times to be displayed in ways other than the default. For example, a clock might contain the following code as part of its look handler:

```
tell (~time mod 43200) / 3600 ":" ~time mod 60 to you
```

This will print out the current hour and minute separated by a colon.

A full set of comparison operators is available: `< > <= >= = !=`. Following C conventions `!=` means "not equals." Values of any type except set may be tested for equality or non-equality; other comparison operators may only be used on numbers or times.

Supported logical operators are `!` (not), `and`, and `or`. The `and` and `or` operators are guaranteed to evaluate their left argument first and to evaluate their right argument only if necessary. Thus, for example, in the statement:

```
move me to you.location and move you to me
```

the second move statement is executed only if the first returns `?true`, i.e. if it succeeds.

2.4.1 Predicates

There are two built-in predicates. `$x matches y` returns `?true` when `$x` is a valid alias for `y`. `@x contains y` returns `?true` when `y` is an element of `@x`. As elsewhere, an object may be substituted for the set, and is then interpreted to mean the object's contents.

2.5 Statements

All statements may also be used in expressions for their boolean value, which indicates whether the statement succeeded or not.

In addition to the set, clear, add/from, and take/to statements already described, the following classes of statements are supported:

`tell` *string-list* `to` *object* Used to send a message to a player. *string-list* may contain any number of expressions of type string, action, time, or number, which are formatted according to their type and concatenated to form the resulting method. No special privileges are required to do a tell.

`move` *object* `to` *destination* Moves an object to a destination (also an object.) Move statements are tightly restricted: the object being moved must be controlled by `me`, in a location controlled by `me`, or be equal to `you`; the destination must be controlled by `me`, have its `?open` bit set, or be equal to `you`. If these conditions are not met the move does not take place and the statement returns `?false`.

`delay` *time* or `delay` *number* Adds an entry to the system *delay queue.* This has the effect of causing the system to deliver a `_tick` method to `me` when either the time has been reached (first form) or the specified number of seconds have elapsed (second form.) An object may have any number of pending delays, but only wizard objects may execute a delay statement inside a `_tick` handler. The delay queue is not saved in checkpoint files and is not preserved across system reboots.

`destroy` *object* Destroys an object, clearing all of its variables (including system variables) and removing it from its location. References to destroyed objects are *not* removed; however, once an object has been destroyed it can not be modified or moved.

2.6 Control structures

The language supports two main control structures, if/then statements and bounded loops over the contents of a set. The if/then statment has the following general form:

`if` *expr* `then` *statements* [`elseif` *expr* `then` *statements*]* [`else` *statements*] `endif`

where the asterisk indicates that any number of elseif clauses may appear, the else clause is optional, and each block of statements can contain any number of individual statements. Here's an example of an if/then statement that one might actually find in a program:

```
if ?locked then
 tell "The door appears to be locked." to you
elseif move you to other_side then
 tell "You go through the door." to you
else
 tell "The door is stuck." to you
endif
```

Note that the `endif` is not optional!

The other control structure is the loop, which is written as follows:

`in` *set* [`matching` *string*] `do` *statements* `end`

Without the optional `matching` clause, executes the statements once for each object in the set with `next` set to the current object. With the `matching` clause, executes the statements only for objects matching the specified string. While this second form is equivalent to a loop surrounding an if/then statement, it is likely to be much more efficient in practice.

Loops may not be nested. The `break` statement, consisting only of that keyword, may be used to exit a loop immediately.

2.7 Exit Statement

The statement `exit` immediately terminates the execution of an action.

2.8 Security

The security system is centered around the notion of *control*. Under most circumstances, one object controls another if both are owned by the same player. However, certain objects have special permissions which affect control as follows:

1. An object with its `?wizard` bit set to `?true` controls any non-admin object.
2. No non-wizard object may control a wizard object, even if they have the same owner.
3. No other object ever controls an admin object.

Control is necessary to modify an object's variables, and is used to determine when a `move` expression may be successfully carried out. Control is not necessary to read an object's variables. In all cases control is tested for the object `me`, which provides the currently-executing code.

3 Command Parsing

Actions come in several varieties. Single-word actions like `&get` or `&look` take at most one object and are handled by either that object, the player, or the surrounding room. Two-word actions like `&get>from` or `&describe<as` require two objects; the object "pointed to" by the `<` or `>` handles the action and the other is passed through unparsed in the variable `$text`. There are also two special actions used by the parser, `_invoke` and `_default`.

When presented with a line of input the command parser considers many possible ways of interpreting it as an action, taking the first successful parse in the following list:

code If the line begins with an at-sign, the remainder of the line is treated as code in the programming language, compiled, and run with `me` set to the current player. The player must have the `?programmer` bit set to use this feature.

verb If verb is a single-word action on the room (first choice) or the current player, invoke it with `$text` set to the null string. Example: *look*.

object If the entire command line matches an alias for an object in the current room or the player's inventory, and that object has an `&_invoke` method, call that method with `$text` set to null. Example: *north*.

verb object If object matches an object in the current room or the player's inventory, and `&verb` is an action on that object, invoke it with `$text` set to null. Example: *get book*.

verb1 object1 verb2 object2 If `&verb1<verb2` is an action on some object matching object1, invoke it with `$text` set to object2 as an unparsed string (the action may, of course, choose to do its own match on object2.) Similarly, if `&verb1>verb2` is an action on some object matching object2, invoke that action with `$text` set to object1. Examples: *whisper Hi there to fred* could invoke a `&whisper>to` action on fred if it has one, and *open door with key* could invoke an `&open<with` action on some object matching "door".

verb text Call `&verb` on room or player with all other words passed through unparsed in `$text`.
Example: *say Good Morning!*

text Call `&_default` on room or player if possible. This behavior is a last resort for the parser, and is used primarily to implement error messages.

If none of the above parses are possible, the system will print an uninformative message and wait for another line of input.

During the execution of an action the variable `me` is set to the object which "receives" it, i.e. the object on which the action is found. `you` is always the current player, and `$text` is set as described above.

One must be careful when writing an action handler to consider all circumstances under which it might be invoked. For example, suppose that an object A (with an alias named "A") has a handler for `&look`. Then it is possible that this handler might be called in any of the following situations:

- If some player contained in A types "look".
- If A is a player who types "look."
- If some player who either holds A or is in the same location as A types "look A".

- If some player contained in A types "look B" and B does not match anything in A or in the player.
- If A is a player who types "look B" and B doesn't match anything.

In most cases where the differences between these cases are critical (as in the last two cases above) it is possible to distinguish them by examining `$text` and the relative positions of `me` and `you`.