**Parsers**



Submitted by javelin on Sat, 2002-11-30 19:07

PennMUSH has two parsers: a command parser that executes commands, and an expression parser that evaluates expressions. The expression parser is sometimes referred to by hardcoders as "process\_expression", which is the name of the main hardcode function that implements it. It's also loosely called the function parser, because it's the parser that evaluates mushcode functions (though it also gets run at various times to do other kind of expression parsing that doesn't involve function evaluation)

The parsers are both synchronous -- when passed a command or expression, they parse it, and no other unrelated command or expression can preempt the parser. The effects of an expressions are also synchronous -- when expression evaluation ends, there is nothing left to do with that expression in the future. On the other hand, the effects of commands may be asynchronous -- commands can schedule other commands to run in the future via the queues (see below), and unrelated commands may intervene between a queuing command and the command that it queues.

# Command Execution



Submitted by javelin on Sat, 2002-11-30 19:21

When a player types a command, or when a command is dequeued from the queue to be run, it is passed to the command parser, process\_command(). In the command parser, the first word of the input is mangled a bit -- spaces are squished, a outer layer of curly braces may be stripped, and %-substitutions are evaluated (the expression parser is used to do these things - it's not just for functions!) To see this behavior at work, type: {say } foo

The command parser now attempts to match the input in the following order, and stops as soon as one of these works:

1. The complete input to a local exit
2. The first word (or token in the case of several special symbols like double-quote, colon, etc.) of the input to a standard mushcode command. If this doesn't match, the command parser evaluates the rest of the input (as above, squishing spaces, stripping braces, and evaluating %-substitutions but not functions) and continues.
3. The complete input to an enter alias of an object in the room
4. The complete input to a leave alias of the current container
5. The complete input to $commands on objects in the player's location (including the player)
6. The complete input to $commands on the current container
7. The complete input to $commands on objects in the player's inventory
8. The complete input to exits in the zone master room, if any
9. The complete input to $commands on objects in the zone master room, if any
10. The complete input to $commands on the zone master (non-room) object, if any
11. The complete input to $commands on the player's personal zone, if any
12. The complete input to exits in the master room, if any
13. The complete input to $commands on objects in the master room, if any

Then, usually, the commands' arguments are passed to the function parser to be evaluated. The command itself is then run with the evaluated arguments. There are two exceptions to this behavior: commands that get the /noeval switch receive their arguments without this evaluation step, and the &attrib <obj>=<value> style of attribute setting does not evaluate the <value> when run directly by a player descriptor.

# Functions aren't commands



Submitted by javelin on Sat, 2002-11-30 19:22

This model explains why typing the following results in a Huh? and nothing else:

if(1,pemit(num(me),test))

The command parser calls the expression parser, but without telling it to evaluate functions, so the above is treated as a single command, fails to match, and gets you a Huh?. On the other hand, this:

[if(1,pemit(num(me),test))] pose

may result in both 'test' being pemitted and a Huh?, because the []'s force the expression parser to recurse and parse what's inside them with mandatory function checking. However, the expression still returns a null string, which is not a valid command, yielding Huh?

In PennMUSH 1.8 and later, however, most admins have enabled the WARN\_ON\_MISSING internal command, which generates an error message whenever any command starting with "[" is attempted.

Finally:

[if(1,say[pemit(num(me),test)])] pose

can result in both 'test' being pemitted, and the player saying "pose", because the expression now evaluates to "say", which is a valid command, and pose is taken to be the argument to the say command.

# $commands



Submitted by javelin on Sat, 2002-11-30 19:24

Consider the following attribute, FOOCMD, set on an object:

$foo \*: @pemit %#=%0

What happens when a player (let's say #10) types 'foo A%rB'?

The player's command ('foo A%rB') is passed to process\_command. The input doesn't match a local exit, and 'foo' isn't a standard command. Now the rest of the input is evaluated, resulting in the total input being transformed to 'foo A<newline>B'. This isn't an enter or leave alias, so the command parser now checks to see if the input matches a $command, and it does. The resulting action (@pemit %#=%0) is now queued, along with some information about the context in which it was queued (for example, that the enactor was #10, and the first wildcard matched 'A<newline>B').

Eventually, the queued entry is dequeued, and '@pemit %#=%0' is passed to process\_command (with the context at which the command was run restored). The input doesn't match a local exit, but @pemit is a standard command. Its arguments are evaluated (and in this context, %# evaluates to #10, and %0 evaluates to A<newline>B), and the command is run, which results in the appropriate @pemit.

Notice that the parsing insures that when the @pemit is queued, %0 in the queue entry's context will be set to the %-substituted input. This is why you can not generally get access to the raw input in your $command (except via %c) -- by the time the actions are dequeued, %-substitions have already happened. Function evaluation has not, which is why a player typing 'foo add(1,1)' will see 'add(1,1)' pemitted back to them. In this situation, a player must type 'foo \\%r' if they want to have '%r' pemitted to them.

**Function evaluation**



Submitted by javelin on Sat, 2002-11-30 19:27

The expression parser is responsible for evaluating functions (as well as doing other kinds of parsing). It's recursive in design, which means that during evaluation, the parser often calls itself to evaluate subportions of an expression.

For example, here's how [add(1,sub(3,2))]foo is evaluated:

[add(1,sub(3,2))]foo

The parser starts with the first character.

It's a [, which causes the parser to recurse,

and turns on mandatory function evaluation --

the next thing we expect to see is a function.

The next evaluation will terminate when it reaches a ].

add(1,sub(3,2))]foo

The level-2 parser starts with the first character

('a') and keeps reading until it comes to a special

character, the '(', which marks the end of a

function's name. It looks to see if 'add' is a valid

function, which it is, and now recurses to parse the

first argument. The next evaluation will terminate

when it reaches a comma, which is the argument separator.

1,sub(3,2))]foo

The level-3 parser starts with the first character

('1') and keeps reading until it comes to a special

character, the comma, which is the terminator it was

expecting. Because no other special characters have

been reached, it returns what it's got so far ('1')

as a literal.

The level-2 parser recurses again to evaluate the

second argument to add():

sub(3,2))]foo

The level-3 parser starts with the first character

('s') and keeps reading until it comes to a special

character, the '('. It checks that 'sub' is a

function, and recurses to parse its first argument.

3,2))]foo

The level-4 parser returns the literal '3'

The level-3 parser recurses again to evaluate the

second argument to sub():

2))]foo

The level-4 parser returns the literal '2'. It

returns because it hits a ')', which denotes the

end of an argument list.

))]foo

The level-3 parser now calls the internal sub()

function, with arguments '3' and '2'. That function

returns '1', and this parser level returns with that

value.

)]foo

The level-2 parser now reads an ')', which terminates

the argument list to add(). It calls the internal add()

function, with arguments '1' and '1'. That function

returns '2'.

]foo

The top-level parser now expects to see a ']', marking the

end of its recursion for mandatory function evaluation.

It gets one, and the overall result of the recursive

parsing was '2'.

foo

The top-level parser continutes to work its way through

the characters. No more special characters are encountered,

so 'foo' is copied into the return buffer. The top-level

parser returns '2foo', and parsing is done.

A player set DEBUG and evaluating the expression above would see very similar output, though DEBUG, for brevity, does not show evaluations at the level of literals: think [add(1,sub(3,2))]foo #7! [add(1,sub(3,2))]foo : #7! add(1,sub(3,2)) : #7! sub(3,2) => 1 #7! add(1,sub(3,2)) => 2 #7! [add(1,sub(3,2))]foo => 2foo 2foo

**Brackets**



Submitted by javelin on Sat, 2002-11-30 19:29

Square brackets signal the expression parser that it should recurse, and must parse whatever it finds for functions. This means that you should use brackets in situations where the parser isn't expecting to parse for functions. Currently, the parser expects to parse for functions in the first word of whatever it's parsing, and nowhere else.

So this:

think add(1,1) foo

returns '2 foo', but this:

think foo add(1,1)

returns 'foo add(1,1)'. If you wanted to get 'foo 2', you'd have to:

think foo [add(1,1)]

Because the first word of what's being parsed by a level of the parser is automatically checked for functions, you don't need to do this:

start [add([sub([mul(3,3)],2)],1)] end

because the sub (and the mul) are the first words of expressions that one of the recursing parsers will evaluate. So you can write the above as:

start [add(sub(mul(3,3),2),1)] end

In addition to saving a few characters, this is more readable. It also saves a couple of extra recursions, which might be important if you're running into the parser's call\_limit. (The impact on code speed is usually negligible, however).

**Commands aren't functions**



Submitted by javelin on Sat, 2002-11-30 19:34

The command parser calls the function parser when evaluating arguments, but the function parser never directly calls the command parser (although side-effect functions can indirectly result in commands being  
run). This insures the synchrony of the function parser, and resembles the way procedural languages like C are parsed. This means that code like the following is wrong (if you meant for it to @pemit Yes to the enactor):

think [if(1,@pemit %#=Yes,@pemit %#=No)]

(By the way, implementing functions like wait(), trigger(), and force() breaks this fundamental paradigm. This is one reason why the PennMUSH developers do not intend to put these functions into standard PennMUSH.)