

VCSU-MEP Objects

August 7, 2008

Introduction

The guide defines the main VCSU-MEP objects and their uses. The command-line verbs are also given. Most verbs given may only be used by the instructor, unless otherwise noted.

Key

The first line of the verb description shows the command-line form. Any word prefixed by an octothorp (#) means that this item must be the object number of the intended object. The keyword “this” means the object itself, which is usually referred to by name. Most other words are prepositions used in MOO syntax.

The linked objects shows any object that this object might have a link in this object. The Manner of creation is the usual way this item is created. The @create and @chparent commands are standard MOO commands, most of the rest are verbs on one of these objects.

\$agent

The agent is the ancestor of several important agents. An instructor should never need to create a new agent. However there are two descendants of interest: The \$lost agent (#12086) and the \$aimless agent (#12314). Generally the agents may only be modified by the wizard.

Verbs:

show this

Displays the students accepted the agent’s offer to move them. This must be the object number of the agent in question.

\$assign_object

The assign_object object contains the text of an assignment. The enCore distribution of the MOO core already has an \$assignment, which has a completely different purpose, hence the name \$assign_object.

Properties:

goalie

Contains the goalie object number that should deliver the assignment.

text

Contains the plain text of the assignment. Since this is normally displayed without HTML interpretation, no tags should be present.

tqd

The TorqueMOODa script, if one exists.

Verbs:

show this

Displays the text of the assignment.

verify this

Validates that the object is correctly set up.

Linked objects: \$roving_goalie.

Manner of creation: @create.

\$autoroom

The autoroom has two features of some value. It will not allow a student to leave with an object of several types. It will also automatically start certain types of interactive objects. This latter feature is set by the following two properties.

Properties:

Auto_object

The object number of the object to automatically start.

Auto_command

The string which is the command needed to start the above object.

Verbs:

next (or n)

If there is a single code machine (or other object that takes next or n) in the autoroom, this will trigger the next command of this object. This is a convenience for using objects in an autoroom.

Manner of creation: @chparent on a \$lecture_room.

\$codeapplet

The codeapplet is the interface to a Java applet similar to MOOApplet. However, this version of the applet is only for tracing code snippets in a graphical way. Each codeapplet represents one predefined trace of the code. The codeapplet is a descendent of the code_machine. Code_machines were typically placed in workrooms since the internal state of the trace could not be shared when used by multiple students. Since the applet does all of the work a codeapplet may be placed in an autoroom. The codeapplet may be started automatically by the autoroom, but the usual mechanism is for the student to click on the icon.

When a student completes an explanation or trace, the appropriate event is posted on the student's object indicating completion. These events may be used as a requirement for lesson completion. If the trace asks either a question concerning flow or variable values another event stating the number of right and questions asked is also posted on the student event. This event is not yet used in requirements.

Properties:

code_list

A list of strings containing the code to be displayed and traced.

exp_list

A list of strings that explains each line of code. This should be exactly the same length as the code list. This gives a brief description of each line of code. Some lines of code may not be executable, but still need an explanation.

tr_list

A list of strings that represents the trace execution of the code snippet in the code_list. The format of tr_list lines is rather involved and is discussed on its own page later in this document.

web_height

A property of most objects, but important for applets. This is the height of the applet panel. It is a string containing the number of pixels. If it is too large the

HTML_suffix text is separated by excessive white space. If too small scroll bars are required to see the applet.

web_width

A property of most objects, but important for applets. This is the width of the applet panel. It is a string containing the number of pixels. If it is too large the scroll bars are generated for no reason. If too small scroll bars are required to see the applet.

Manner of creation: @create

\$code_machine

The code machine is the text based version of the \$code_applet, which is now the preferred object to use.

\$code_repository

The code repository is the text based version of the \$repository_applet, which is now the preferred object to use.

\$course_info

The course_info object contains the basic information about a course, such as the course name, the object numbers of the students, lessons, etc. The enCore distribution of the MOO core already has a \$course object, which has a completely different purpose, hence the name \$course_info.

Properties:

base

A list of monitor rooms. The monitor room is an ancestor of \$sublesson. Most of the foyers are monitor rooms. Every student that enters a monitor room is examined to determine if they are lost or aimless. Placing the foyer in the base list prevents the student from being declared as lost.

course_name

The name of this course for display purposes. This should be set during course creation.

dispatcher

The object number for the dispatcher for this course. This should only be set during course creation.

gradebook

The object number of the gradebook. This should be set during course creation.

instructor

The instructor object number. This should only be set during creation.

lesson_list

The list of lessons that need to be completed. This should only be set by Append.

open

Indicates that the course is open. Should be set to open at course creation and set to closed by the Close command.

students

A list of student object numbers. This should only be set by the Add command.

survey

The object number of the current survey. #-1 if there is none.

test

The object number of the current test. #-1 if there is none.

Verbs:

Add #student to this

Adds a new student to the course.

Append #lesson to this

Adds a new lesson to the course.

Awards this

Show the awards given to all students. An award is usually an assignment given by a goalie. The completion of a sublesson generates an event, but the completion of a sublesson that has a goalie generates an award.

Close this

Close the course. A closed course may no longer be modified.

Display this

Displays just the students and lessons in the course.

Goalies this

Shows the lessons and their corresponding goalies.

Grades this

Show the grades, if any have been assigned.

Remove #object from this

The object may be either a lesson or student.

Show this

Shows information on the various students who are enrolled in the course. This includes exhibits visited, number of events, and last goalie.

Showss this

Shows most of the same information as the above, but in comma delimited format. This allows it to be cut and pasted into a spreadsheet easily.

@student #student in this

Display a list of lesson the student has completed.

Verify this

Checks that the course is properly set up.

Linked objects: \$dispatcher, \$gradebook, \$instructor, \$requirements, \$student, \$sublesson, and \$testgen.

Manner of creation: @create_course

This verb has no parameters and prompts for everything that it needs. It will be helpful to know beforehand the object numbers of existing students who will be entered, lesson numbers to add, the dispatcher number and the gradebook number if they exist. These may also be created from within the verb.

If there is an attached gradebook then modifications to students (add or remove) will be propagated there as well.

\$dispatcher

The dispatcher object receives notifications from sublessons on student completion. It contains a list connecting sublessons and goalies. If the notifying sublesson has a goalie that goalie is dispatched to give the student an assignment.

Verbs:

@Add #object to this

Adds a new lesson or goalie to the dispatcher. If object is a lesson it will prompt for the goalie and if object is a goalie it will prompt for the lesson. If the goalie is in the current room it may be specified by name as well as object number.

Clear this

Clear the error property.

Show_error this

Show the errors.

Verify this

Checks that the course is properly set up.

Linked objects: \$roving_goalie and \$sublesson.

Manner of creation: @create_course

\$gradebook

The gradebook object records and displays scores for students. It must be connected to a course_info object to be usable. The student may find his or her grade with the @grade command.

Properties:

grade_type

If this is "standard" then letter grades are computed using the percents in the grade_breaks property.

grade_breaks

A list of lists. The first element of the list is a number that is the cutoff percent. The second element is the letter grade, which is a string. The cutoff percents are in descending order and the last cutoff percent should be zero. The default value is: {{90, "A"}, {80, "B"}, {70, "C"}, {60, "D"}, {0, "F"}}

Verbs:

Add this

Adds a new name to the gradebook. The verb asks for the name and the maximum points allowed for it. Each name must be unique. If the maximum value is zero, then this is a non-graded item. If the maximum is -1, then the maximum is the highest score attained by any student.

Attach this to #course_info

Make this the gradebook for the given course. Most other commands will not work until an attach is accomplished.

Export this

This will write the contents to the console as comma separated values format. It may then be cut there and pasted into a spreadsheet.

Remove this

Remove a test item from the gradebook. The verb will prompt for which item to remove. This cannot be reversed.

Score this

Add or modify scores. The scores may be handled in one of two ways. Either a single score for a single student may be entered or all the scores for a single name are entered at once. The verb prompts for all of these.

Set_percents this

Interactively set the grade_breaks property. It prompts the instructor for the percents and letter grades.

Summary this

Give a summary of student scores.

Verify this

Checks that the course is properly set up.

Linked objects: \$goalie and \$sublesson.

Manner of creation: @create_course

\$home_room

A descendent of office that displays help for the student and contains a @test verb.

Properties:

test_allowed

This is a boolean that determines if a test is allowed. When a test is available it should be set to 1 (true). If the student has left during the test or already finished the test it should be set to 0 (false). This should be set by the system.

Verbs:

@test

Starts a test, provided that a test is possible.

Manner of creation: @create_course or @chparent on a room

\$instructor

The instructor is a descendent of programmer, thus has all the privileges of a programmer. In addition it has some verb definitions that ease a typical instructor's use of VCSU-MEP.

Verbs:

@create_course

Make a new course.

@create_requirement

Make a new requirements object.

@create_students course_info

Make a new student and add them to the course_info object. The course_info object may be an object number or the name if the object is present.

@make_sub_lesson #exit

Make the current room into a sublesson. The exit is that exit that leaves the lesson.

This verb gives the ability to set up requirements and all_rooms as well.

make_sub_lesson_exit #exit #lesson

The main exit is fixed by the @make_sub_lesson_exit command. However, if there are two or more exits that leave a lesson this command is used to fix them.

Manner of creation: Only the wizard creates an instructor

If the instructor has a make_initial_room verb, then creating students will use that verb to create an initial room for the student and an exit to a designated room. However, only the wizard can create such a verb.

\$lecture_room

The lecture_room is the standard room created with the @dig command.

Properties:

indices

This is a list of strings that are the keywords that this room discusses. These indices will be rolled into the containing lesson by a verb. Use @set to initialize, eg:

```
@set here.indices to {"polymorphism", "inheritance", "object hierarchy"}
```

quiz

This contains quiz questions over the content of this room. These will only be used if three conditions are satisfied: 1) this room is required by the lesson that it is within 2) the student has not visited this room and 3) the quiz agent offers the student a quiz. The standard question format is used, which is described later in this document.

Verbs:

Verify

Checks that the indices property is properly set up. Called by sublesson:verify.

Manner of creation: @dig

\$lesson

An object originally conceived to be at a higher level than a sublesson. However, the use of the lesson has been discontinued, so use sublesson instead.

\$monitor_room

The monitor_room is the ancestor of sublesson. Its purpose is to monitor the movement of students. It checks whether the lost or aimless agent needs to be summoned.

Manner of creation: @chparent on a \$lecture_room.

\$mooapplet

This is the MOO interface to an external Java applet. The applet needs to be in the encore directory of the server and the related jar files should be there as well. There is a section at the end of this document on MOOApplets. The completion of a MOOApplet may be a requirement for a lesson.

A MOOApplet should be in an \$autoroom so that it may not be carried away by a student. However, multiple students use may use it simultaneously, since each gets a copy. Therefore a \$workroom is not needed to contain it. Placing a MOOApplet in a \$workroom will only restrict access.

Properties:

applet

This is the (case sensitive) name of the applet. It should include the class extension.

This will be inserted into the HTML that the MOO generates. Use @set to initialize,

eg:

```
@set demo.applet to "demo.class"
```

archive

This is the name of the Jar file that contains all the class files of this particular applet. Use @set to initialize, eg:

```
@set demo.archive to "demo.jar"
```

html_prefix

This is a list of strings, that will precede the applet. This may be used to cite the original author of the applet or to give instructions. HTML tags as well as text are

allowed. Use @edit to initialize, eg:

```
@edit demo.html_prefix
```

html_suffix

This is a list of strings, that will follow the applet. This may be used to cite the original author of the applet or any other text. HTML tags as well as text are allowed. Use @edit to initialize, eg:

```
@set demo.html_suffix
```

new_display

If set to 1 the applet will be displayed in a new window or new tab of the browser. If set to 0, it will replace the room display . The default is 1.

params

This is a series of strings that will be passed to the applet. They should have exactly the form that the HTML expects. This will be inserted into the HTML that the MOO generates. Use an editor to produce the lines. A sample file follows:

```
<param name=imagesource
value="jar:http://euler.vcsu.edu/encore/while_loop.jar!/images/">
<param name=endimage value=54>
<param name=backgroundcolor value="0xFFFFFF">
```

web_height

This is an inherited property that determines the height of the item in display. It is a string containing the number of pixels of height.

web_width

This is an inherited property that determines the width of the item in display. It is a string containing the number of pixels of width.

All verbs are used internally only.

Manner of creation: @create

\$presentation

A descendent of \$moopplet designed specifically for displaying a recorded presentation. The presentation consists of slides, with have extention .JPG and sound clips which have extension .wav. These are stored in a jar file which is on the server. The properties of a MOOApplet which are normally set will be set by default. However there are other properties that may also be set. With the exception of jarfile all of these have a good default.

Properties:

author

This is the person who created the slide show and presumably the voice on the sound clips. The default is "Curt Hill." This will only show if either html_prefix or html_suffix is empty.

debug

The default behavior of the Presentation is to ignore clicks on the Forward button as long as the sound clip is active. This prevents a student from clicking through the presentation rapidly, merely to gain credit. If the debug parameter is present with any value, then clicking the Forward button will cause the current clip to be ended and the presentation will immediately move forward. This is helpful for checking the jarfile's completeness.

jarfile

This contains the slides and sound clips. There should be the same number of each. The default name for a slide is: SlideN.JPG, where N is an integer starting at 1. There should be no gaps between 1 and the maximum number. The default sound clip should be sN.wav where N is also an integer. If N is less than 10 it will have one leading zero. This is a required parameter.

imageprefix

This is the first so many characters of each slides name. This should only be used if it is not "Slide". It is case sensitive.

imagesuffix

This is the last so many characters of each slides name. This should only be used if it is not ".JPG". It is case sensitive. An image does not have to be a JPEG, Java handles .GIF and other formats as well.

max

The number of the slides or sound clips. It is a string. If max is 12 then there should twelve slides with no gaps. This is a required parameter.

soundprefix

This is the first so many characters of each sound clip's name. This should only be used if it is not "s". It is case sensitive.

soundsuffix

This is the last so many characters of each sound clip's name. This should only be used if it is not ".wav". It is case sensitive. A sound clip does not have to be a Wave file, Java handles .AIFF and other formats as well.

Verbs:

Setup this

A verb to set up the important objects of the presentation.

Manner of creation: @create

\$repository_applet

A descendent of code_applet, except no trace is possible. This is usually used for pieces of code that are too large to trace. The student may still copy or get a line by line explanation.

Manner of creation: @chparent on a \$lecture_room.

\$requirements

The requirements object contains a req property which is a list of lists. This list contains the alternate requirements of this object. Each of the sublists contain either the object numbers of rooms that must be visited or specially formatted strings that indicate the object that must be completed. Most of the work on this object is actually done by other objects.

Properties:

req

A list of lists of requirements. The inner list may contain room numbers or quoted strings that are the object requirements.

lesson

The lesson to which this applies. An object number.

Verbs:

Verify this

Checks that the requirements is properly set up.

Linked objects: \$sublesson.

Manner of creation: @create_course or @create

\$roving_goalie

The roving goalie object is activated by a dispatcher. It then visits the student and gives the student and assignment. The goalie may also give a score to the grade book. The goalie maintains a list of assignments that may be given and rotates through these assignments. The goalie also records the assignments that are made to students. When a goalie enters the room where the student is located it does the following things: displays the prefix message, activates the assignment, displays the suffix message, waits a few seconds and then leaves. The prefix and suffix messages may have general information, while the assign_object has specific details.

Properties:

prefix_msg

Upon entering the room that the student is examining this message is displayed.

suffix_msg

Upon finishing the display of the assignment this message is displayed.

grade_name

The name the gradebook uses to identify this score.

grade_points

The number of points this assignment is worth.

grade_scheme

How the score devalues past the due date. The following values are supported:

0 – No reduction, due date is disregarded.

1 – Divide by two. Late assignments are worth half for any number of days late.

2 – Reduce the score by one point for each day late, but never make it less than 1.

grade_month

The month the assignment is due.

grade_day

The day the assignment is due.

Verbs:

Add #object to this

Adds an assignment object to the list of assignments.

Clear this

Clear the record of assignments made.

Remove #object from this

Removes the assignment object from the list of assignments.

Set_grade this

The program will prompt for all the details of a grade opportunity. You must know the object number of the gradebook and the name of this scoring opportunity. The scoring name must already be present in the gradebook.

Set_due_date this

The program will prompt for month and year

Show this

Shows what assignments have been made to what students.

Verify this

Checks that the goalie is properly set up.

Linked objects: \$assign_object.

Manner of creation: @create

\$student

The student object is the basic player for a class.

Verbs:

call_aimless

Summons the aimless agent.

@course

Displays the lessons of the course, including which have been completed

@grade

Shows scores and grade if the student is a member of a course with a gradebook.

@index

Shows the topical keywords and the lesson that they are discussed within. This only includes lessons that the student has completed.

@progress

Shows all the work that has been done on the MOO.

@showgoal

Shows the next lesson that the student should be working on.

@showgoal N

Shows the Nth goal that the student has in their progress list.

Linked objects: \$course_info, \$gradebook, \$instructor, \$requirements, \$student and \$sublesson.

Manner of creation: @create_course or @create_students

\$sublesson

The sublesson is a form of room. The @quiz and @requirements verbs may be used by students. Originally there was both a \$lesson and \$sublesson, however the \$lesson has since been removed.

Properties:

all_rooms

This contains a list of all rooms present in the lesson. These will be removed from the students history list, once the lesson is completed. This should only be set as a consequence of other commands, such as add_room and @make_sub_lesson

indices

This contains a set of all the indices of rooms present in the all_rooms property. This should be only set by the verify command.

quiz

This contains overview quiz questions for the entire lesson. These will only be used if the quiz generator cannot find five questions from the required, unvisited rooms and the quiz agent offers the student a quiz. The standard question format is used. The programmer should edit this property.

Quiz_room

The quiz room for this lesson. This should only be set by the @make_sub_lesson command.

requirements

A list of requirement objects for this lesson.

Verbs:

add_room

Add a room to the all_rooms property.

@quiz

Take the student to the quiz room.

@requirements

Show the requirements of this lesson. May be abbreviated to @req.

@student #student

Allows a programmer to find the requirements for a student.

Verify this

Checks that the course is properly set up.

Linked objects: \$roving_goalie and \$sublesson.

Manner of creation: @make_sub_lesson

\$surveybank

The surveybank is a container for survey questions in the usual format. (This format is described following the list of objects.) Each question will be used.

Properties:

text

An editable list of strings that is the set of questions. The survey question format is used.

Verbs:

Attach this

Attaches the item to a course.

Verify this

Checks that the object is properly set up.

Manner of creation: @create

\$surveyfront

The surveyfront is a GUI front end for a surveybank object. It points at the course. It may be started by a click in the Xpress client.

Properties:

course

The object number of the course to which this test belongs.

Verbs:

Start this

Starts the test or notifies the student that it may not be started. This verb is executed if the student clicks on the icon in the Xpress client.

Verify this

Checks that the object is properly set up.

Manner of creation: @create

Surveys lack some of the features of tests, such as random selection of questions, scoring, right and wrong answers and using multiple survey banks for one survey.

\$testbank

The testbank is a container for test questions in the usual format. (This format is described following the list of objects.) The entire group of questions is randomly selected from in the test generation process. Hence, you should have one such object for each topic. A testgen may select from many testbanks.

Properties:

text

An editable list of strings that is the set of questions. The standard question format is used.

Verbs:

Verify this

Checks that the object is properly set up.

Manner of creation: @create

\$testform

The testform is a container for test questions that cannot be graded by the MOO. The text of these questions is presented “as is” to the student. Only one testform is allowed per testgen object.

Properties:

description

An editable list of strings that is the set of questions. This may be in HTML format and should include the instructions as well as the actual questions.

Manner of creation: @create

\$testfront

The testfront is a GUI front end for a testgen object. It points at the course. It may be started by a click in the Xpress client.

Properties:

course

The object number of the course to which this test belongs.

Verbs:

Start this

Starts the test or notifies the student that it may not be started. This verb is executed if the student clicks on the icon in the Xpress client.

Verify this

Checks that the object is properly set up.

Manner of creation: @create

\$testgen

The testgen object is the main object for the giving of an online test.

Properties:

allowed_time

The number of minutes a student is allowed to take this test.

assess_confidence

A Boolean that indicates that after each question is delivered the student should be asked about their confidence in their answer.

course

The course info object that this test belongs.

form

The testform object that this test will present.

grade_name

The grade item in the gradebook associated with the course info object. The testgen object will record this grade item for this student.

open

A Boolean indicating whether the test may be taken now.

test

A list that describes the testbank items and the number of questions to extract from them.

Verbs:

@close this

Make the test unavailable to the students.

@open this

Make the test available to the students.

@prepare this

Reuses this object to prepare for a new test. It prompts the instructor for the options that it needs to set. The instructor should have present in the room (or know the object numbers) of the course and testbank objects.

@Verify this

Checks that the testgen object is properly set up.

Linked objects: \$course_info and \$testbank.

Manner of creation: @create

\$workroom

The workroom is designed as a room that prevents more than one student in at a time. Certain interactive objects such as code machines and syntax tests do not function concurrently. Therefore such rooms disallow multiple students at a time. It is a descendent of \$autoroom which means that it has other features as well.

Manner of creation: @chparent on a \$lecture_room.

Standard Question Format

Quiz format:

The quiz is a list of strings, similar to the description property and should be edited like description. The question starts with a + in the first character, which identifies the question part. The question may include multiple lines. It is terminated by the first answer line.

A question must have one or more right answers. A right answer line starts with a dash.

The wrong answers must start with a right parenthesis). The answer needs no blanks following that. There should be four or more.

Since the answers are randomized, avoid answers like all of these, none of these, or answers that have to be in a certain position to work.

This question format is used in the quiz property of a sublesson or lecture room, as well as the text property of a testbank.

Survey question format:

The survey questions have a similar format as the above with some exceptions. There are no right or wrong answers on a survey, so there are no answers starting with a dash. The question does not have to have any answers starting with a right parenthesis. If it does these will be numbered starting at 1 and the student's answer will be recorded. If the question has no answers starting in this way, the answer is merely captured.

MOOApplets

Description:

A MOOApplet is a particular type of Java applet that may send results back to the MOO. Each MOOApplet is a descendent of the class MOOApplet, which is itself a descendent of the standard Java.applet.Applet. Descendants of MOOApplet are used to perform some action that is not easy or possible from the MOO itself. This typically includes visualizations, but may include any program more graphic and interactive than the MOO can easily handle.

Using a MOOApplet requires the creation of a Java applet as described on this pages as well as the creation of the MOO object described earlier.

The MOOApplet ancestor provides two extra methods: PhoneHome() and PhoneHome(String command,String parms).

The PhoneHome() method sends a message back to the MOO indicating that the user successfully completed the exercise or task. The MOOApplet object has a done_html method, which will receive as its parameter the student's object number. It will then post the event to the student's events.

The PhoneHome(String command,String parms) method allows a more customized message to be sent back to the MOO. The command should be the MOO verb on the object. The name of the verb should contain _html or it will not be successfully parsed by the MOO web monitor. The parms string should give additional information. Since this is sent as HTML you should avoid special characters that HTML looks for: slash (/), dot (.), less than (<), greater than (>) and ampersand (&). It is your responsibility to implement the verb on your object. It will receive one parameter, which is the concatenation of the player number (without the leading #) and the parms string.

There are several simple requirements for new applets:

- The applet must be a descendent of MOOApplet
- The applet executes PhoneHome when the user has successfully completed the exercise.
- The applet nicely fits in the right hand pane of the encore web client.
- The applet uses only one jar for its code.
- The applet parameters avoid the three names that are used by the ancestral MOOApplet: objectnum (the MOOApplet itself), player (the student executing the MOOApplet) and command (the verb to use when complete). These may be examined by the applet, but the ancestor cannot signal the MOO if they are used for anything other than the predefined use.

The MOOApplet.java source code is on the project web site.

The intent of the MOOApplet is that any existing simulation could be readily converted to use in the MOO. However there are applets that are parameterized that may be used in multiple locations, such as the Expression Evaluation Animator.

Expression Evaluation Animator Applet

Description:

This applet will animate the evaluation of expressions to make them more understandable. As far as the MOO is concerned it is just another MOOApplet, but the parameters may cause enough difference in effect that there are several of these in various places of the MOO.

The idea is to have a single statement displayed and provide a script that shows how the statement is executed in a step by step fashion. This applet is useful in explaining the priority of operations and side effect operators. The applet and archive parameters should be set as follows:

```
@set c.applet to "ExpressionEvaluationAnimator.jar"  
@set c.archive to "ExpressionEvaluationAnimator.jar"
```

There are two kinds of parameters for this applet. The first defines the statement. The parameter name is "statement" and the value is the single expression that should be executed. This is usually an assignment but does not have to be such. The second kind of parameter is the stepN parameter, where N is an integer starting at zero. Step0 is usually the initialization. Up to 50 steps are allowed and they should start at 0, without leaving any out. The value of the step parameter has three pieces enclosed in parentheses. The first describes what to highlight, possibly giving temporary values. The second is a list of variables and values that they will receive in this step. (This is similar to the form of a code applet or code machine trace line following a question mark.) The third is some commentary that is merely displayed. The three pieces are separated by semicolons and any of them may be of zero length. What follows is an working example:

```
<param name = "statement" value = "val *= ++a * b-- - c++/++d;">  
<param name = "step0" value = "(;val,2,a,2,b,3,c,7,d,2;The initial values before ...)">  
<param name = "step1" value = "(13-16;b,3-;The trailing -- is left of the trailing ...)">  
<param name = "step2" value = "(19-22;c,7+;The trailing ++ has highest remaining ...)">  
<param name = "step3" value = "(7-10;a,3;The leftmost leading ++ has highest ...)">  
<param name = "step4" value = "(23-26;d,3;The rightmost leading ++ has highest ...)">  
<param name = "step5" value = "(7-16:1:9;;The multiply now has highest remaining ...)">  
<param name = "step6" value = "(7-16:1:9,19-27:2:2;The divide now has highest ...)">  
<param name = "step7" value = "(7-26:3:7;;The subtract is now performed, since it ...)">  
<param name = "step8" value = "(0-26:3:7;val,14;The multiply and assign is now ...)">  
<param name = "step9" value = "(0-27;b,2,c,8;Now the deferred increment and ...)">
```

The commentary has been shortened to fit on one line in this document. One step of this example is shown in Figure 1.

The first part is a description of the parts of the line to highlight. A line may have zero or more portions highlighted. The highlighting has a different background color. The form of a highlight is:

M-N

where M is the starting position on the line (zero based) and the N is the first character that should not be included.

Following the highlight may be a colon followed by a digit, which indicates a temporary variable that is to be displayed immediately below the highlight. The temporary variables are named temp1, temp2, temp3, etc. so only the digit 1, 2, 3 or whatever is needed. Following that digit is a value, which is the value the temporary currently has. Figure 1 shows the state of the applet after step 5 of the above example was executed.

In the step 5 line, the 7 is the first position to highlight, the 16 is the first to not highlight. This is followed by a one and nine. The one indicates a temporary variable and the 9 is its new value. The enclosing parentheses contain the whole step. Notice that there is no variable assignment other than the temporary. The commentary is shown after the table.

The current state of execution:

```
val *= ++a * b-- - c++/++d;
temp1=9
```

variables:	val	a	b	c	d
current value:	2	3	3-	7+	3
last value:		2	3	7	2

The multiply now has highest remaining precedence. The new value of a is used and the old value of b. The result is stored in a temporary variable until it is needed. Such variables are generated for us by the compiler and we never actually see them.

Figure 1

The second part is a sequence of variable and value pairs, separated by commas. These variables and their values are maintained in the table at the bottom of the applet display. The form of a pair is variable name followed by a comma and the new value. The applet will remember the values, so these need only be entered when a value changes. The pairs are also separated by commas. Step0 usually supplies initialization of variables before the statement began.

The third part of the step is just a text message that explains what has happened. This is displayed below the table of variable values.

When the applet has displayed all steps, it notifies the MOO that the student has completed this exercise.

There is no new object in the MOO, just the \$mooapplet. If your expression needs some declarations, including setting initial values, use the HTML_prefix property to show these. Initial values are also shown in Step zero, but not types.

The web_height property is also important. The applet will always use the web_height property to determine the applet panel dimensions. Since the HTML_suffix appears below the applet panel, if the web_height is too large there will be excessive white space between the panel and the text. The web_height is a string enclosing a number. For this applet a value of "300" is a good place to start.

CodeApplet and Code_Machine Trace Lists

Basic features:

A CodeApplet is a particular type of Java applet. It will perform a predefined trace of a code snippet and then will send results back to the MOO. It is a descendent of the code_machine object and what is here described applies to both.

The code_list property contains the code to be traced. The exp_list property has one line for each line of the code and explains that line of code. The tr_list property contains the information for a predefined trace of the code. There should be one line for each step of the trace, where one step could cover several lines or just a fraction of one line. The basic format of each line is as follows:

line number # description ? variable changes

The following describes these:

- The line number is a numeric value indicating the code_list line that is currently being executed. Since an execution must account for flow of control the tr_list entries are not usually sequential or include every line. This is a required field. When the line is executed the line is highlighted by the codeapplet or displayed by the code_machine with its line number.
- # The octothorp (#) is required to separate the line number from the description. When the line is executed the description is displayed. This may be any explanation that is pertinent to the trace. This description should not include a question mark or many special characters.
- ? The question mark is an optional separator between the description and variable changes. Many executed lines change no variable, in which case the question mark and all following parts are left out.
- The variable changes are pairs of values. Each pair is separated by commas and the pairs themselves are separated by commas. The first item of the pair is the variable name being changed and the second is the new value.

Example:

There are several other options, but before discussing these some examples are in order. Consider the following code snippet from C++. Notice that the line numbers are not part of the code_list, but are applied by the object.

```
1: int w=2,x=3,y=5,z=-3;
2: if(w+x>y) {
3:   w = x + y;
4:   x = y / 2;
5:   y = w - x;
6:   z = z * 2;
7: }
8: else {
9:   w = x - y;
10:  x = w + 3;
11:  y = w - x * 2;
12:  z = w - z * 3;
13: }
14: cout << w << " " << x << " " << y << " " << z << '\n';
```

This code could have the following `tr_list`, which is shown just as it is, except two lines (2 and 11) are longer than can be displayed on one line in this document.

```
1#The variables are initialized.?w,2,x,3,y,5,z,-3
2#The condition is w+x>y. The values of w,x and y make this:
  2+3>5 or 5>5. Since the comparison is greater instead of
  greater than or equal, the condition is false.
9#w is assigned.?w,-2
10#x is assigned.?x,1
11#y is assigned. Precedence forces x*2 to be done first,
  yielding 2, which is subtracted from w giving -4?y,-4
12#z is assigned w - (z * 3).?z,7
13#End of the path.
14#Display all the values: -2 1 -4 7
```

Some notes on this example:

- Notice that all lines must start with the line number and octothorp. However, there are gaps in the numbering, since some statements are not executed. The code does not have to start execution with line 1 either. Since the if in line 2 has a false condition, the lines from 3 to 8 are skipped.
- Some statements change variables. These include 1, 9, 10, 11 and 12. These must have a question mark followed by the assignments. The other statements do not have the optional question mark, such as 2, 13 and 14.
- The variable changes are just a variable length list of pairs. The first is the variable name, the second the value. Line 9 has just one pair, where a -2 is assigned to the variable w. Line one has four pairs.
- The code machine object or codeapplet will keep track of which variables are currently in use, so question mark is only needed for changes.
- The assignment pairs are just strings, even though the values may look like numbers. No computations are done on these values.

Other features:

There are several additional options that also need to be considered.

The line number may be a single number or it may be a range. This is represented by the first line number, followed by a dash and then the second. Therefore the trace line for line 9 in the above example could be replaced with this:

```
8-9#w is assigned.?w,-2
```

This feature allows multiple lines to be treated as one. Commonly an output statement may extend over several lines and this allows it to be treated easily. Moreover, in advanced example, groups of simple statements may be lumped together.

In block structured languages a variable may be created and later destroyed. There is a mechanism for removing a variable from display. This is to assign it a value of 'd' which is the signal to remove it. Consider the following example from C++.

```
3: int w=7,x=3,y=5,z=-3;
4: if(w>y) {
5:   int a = w+x;
6:   z = 2 * a;
7: }
```

```
8: cout << z;
```

In this example the variable `a` only exists from line 5 through 6. Therefore the trace list might look like this:

```
3#The variables are initialized.?w,7,x,3,y,5,z,-3
4#The condition is w>y. The values of w and y make this:
   7>5, which is true.
5#a is created and assigned.?a,10
6#z is assigned.?z,20
7#Since a is limited to the scope of the if, it is destroyed
   at this point?a,d
8#Display the value of z, which is 20.
```

In most traces an assigned value is a number or simple string. These provide no problems but sometimes a string with a comma may be assigned. Since the applet is looking for commas, the string to be assigned may be enclosed in apostrophes. For example:

```
4#The input buffer ...?buffer,'^ 7, 8'
```

Question features:

Although code machines have always been interactive, they have not always triggered active learning. A student may start the trace and just hit the next button (the next command in a code_machine) until the simulation is done. The student may do this without watching the display or learning a thing. In order to make this a more educational experience these objects may also ask questions during the course of the trace. The two possible questions are: Which line will be executed next? and What value will be assigned to a variable? These are triggered by slight changes in the syntax of the trace list.

Asking the student the next line to be executed is useful for flow of control statements. This is signaled by using two octothorps rather than one on the statement that is questionable? Thus the `##` should be placed on the first line of a then or else sequence rather than on the if. For example with this code list:

```
3: int w=7,x=3,y=5,z=-3;
4: if(w>y) {
5:   int a = w+x;
6:   z = 2 * a;
7: }
8: cout << z;
```

The following trace list may be used:

```
3#The variables are initialized.?w,7,x,3,y,5,z,-3
4#The condition w>y is evaluated.
5##a is created and assigned.?a,10
6#z is assigned.?z,20
7#Since a is limited to the scope of the if, it is destroyed
   at this point?a,d
8#Display the value of z, which is 20.
```

Notice that the description of 4 has been changed. The student will be able to see the current values of `w` and `y`, so the description should not say whether the condition is true or false, but let the student figure that out and where execution will flow next.

When statement 5 is about to be executed, the system will ask them which line number will be executed next. The student must type in a 5 to be correct.

Changing a previous example to:

```
8-9##w is assigned.?w,-2
```

The student could enter either 8 or 9 and be correct.

The mechanism for asking a student about an assigned value requires the student to understand the value assignment mechanism. This might include concepts such as integer arithmetic or precedence. This type of question may be asked by substituting the comma that occurs between a variable name and its assigned value with a semicolon. For example code line:

```
11: x *= ++y - z--;
```

with the following trace line:

```
11#A complicated assignment with side effects?y,2,z;4,x;12
```

Although three assignments are to occur, only two of these will be asked about. First the student will be asked about the new value given to z and second the new value given to x. The student will be able to see the current value for each variable as well as the statement itself.

Notice that the semicolon may only go within the name and value pair. The pairs are still separated by commas.