# Knowing What You're Talking About: Natural Language Programming of a Multi-Player Online Game

**Henry Lieberman**
**and Moin Ahmad**
Media Laboratory
Massachusetts Institute of
Technology
Cambridge, MA 02139 USA
lieber@media.mit.edu


*CHI 2009*, April 4 – 9, 2009, Boston, MA, USA
ACM 978-1-60558-246-7/09/04.

## Abstract
Enabling end users to express programs in natural language would result in a dramatic increase in accessibility. Previous efforts in natural language programming have been hampered by the apparent ambiguity of natural language. We believe a large part of the solution to this problem is *knowing what you're talking about* – introducing enough semantics about the subject matter of the programs to provide sufficient context for understanding.

We present MOOIDE (pronounced "moody"), a natural language programming system for a MOO (an extensible multi-player text-based virtual reality storytelling game). MOODIE incorporates both a state-of-the-art English parser, and a large Commonsense knowledge base to provide background knowledge about everyday objects, people, and activities. End-user programmers can introduce new virtual objects and characters into the simulated world, which can then interact conversationally with (other) end users.

In addition to using semantic context in traditional parsing applications such as anaphora resolution, Commonsense knowledge is used to assure that the

virtual objects and characters act in accordance with Commonsense notions of cause and effect, inheritance of properties, and affordances of verbs. This leads to a more natural dialog.

## Programming in a MOO

Figure 1 illustrates MOOIDE's interface. A MOO [1] is a conversational game modeling a simulated world containing virtual rooms or environments, virtual objects such as tables or flower pots, and virtual characters (played in real-time by humans or controlled by a program). Players of the game may take simulated physical actions, expressed in natural language, or say things to the virtual characters or other human players. Programming consists of introducing new virtual environments, objects, or characters. They then become part of the persistent, shared environment, and can subsequently interact with players. Since the players use natural language to interact with the game, programming new game components themselves in natural language is, well, natural.

Our previous work on the Metafor system ([2], [3]) showed how we could transform natural language descriptions of the properties and behavior of the virtual objects into the syntax of a conventional programming language, Python. We showed how we could recognize linguistic patterns corresponding to typical programming language concepts such as variables, conditionals, and iterations. However, that system did not contain a full MOO runtime environment in which we could dynamically query the states of objects. MOODIE also adds the ability to introduce new Commonsense statements as necessary to model the (necessarily incomplete) simulated environment.

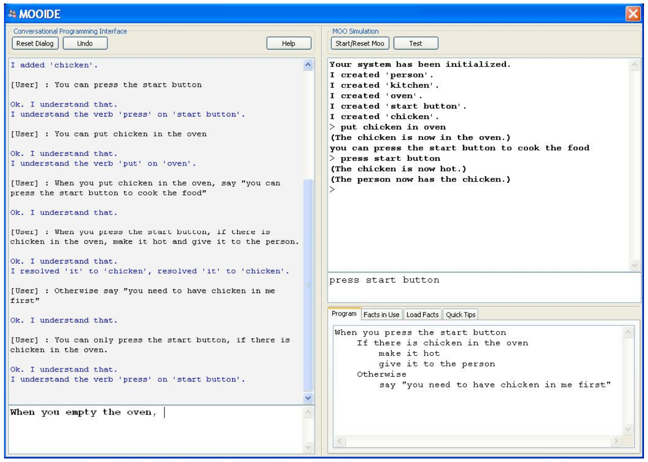


**Figure 1.** MOOIDE's interface. The user is programming the behavior of a microwave oven in the simulated world.

## A dialogue with MOOIDE

Let's look at an example of interaction with MOOIDE. These examples are situated in a common household kitchen where a user is trying to build new virtual kitchen objects and giving them behaviors.

**There is a chicken in the kitchen.**
**There is an oven.**
**You can only cook food in an oven.**
**When you cook food in the oven, if the food is hot, say "The food is already hot."**
**Otherwise make it hot.**

The user builds two objects, a chicken and an oven and teaches the oven to respond to the verb "cook".

MOO syntax then allows any player to use the verb by entering the following text into the MOO:

**`cook chicken in oven`**

In the verb description, the user also describes a decision construct (the If-Else construct) as well as a command to change a property of an object—"make it hot". To disallow cooking of non-food items, he puts a rule saying that only objects of the 'food' class are allowed to be cooked in the oven ("You can only cook food in an oven"). Note this statement is captured as a commonsense fact because it describes generic objects.

When the user presses the "Test" button on the MOOIDE interface, MOOIDE generates Python code and pushes it into the MOO where the user can test and simulate the world he made. To test the generated world, he enters **`cook chicken in oven`** into the MOO simulation interface. However, because the MOO doesn't know that chicken is a food, the MOO generates an error— **`You can only cook food in an oven`**. This is not what the user expected!

To resolve this error, he then has to add the statement **`Chicken is a kind of food`**. Then he tests the system again using the same verb command. Now, the command succeeds and the MOO prints out **`The chicken is now hot`**. To test the decision construct, the user types **`cook chicken in oven`** into the MOO simulator. This time the MOO prints out **`The food is already hot`**.

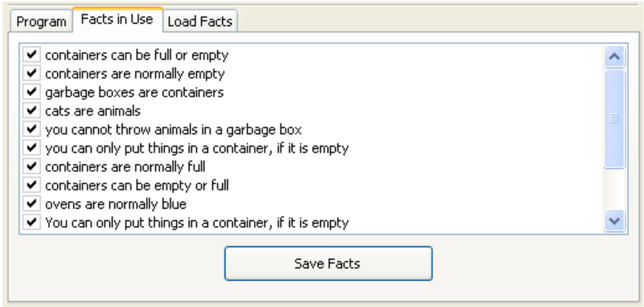


**Figure 2:** Commonsense facts used in the microwave oven example.

## Implementation

MOODIE performs natural language processing with a modified version of the Stanford link parser [6] and the Python NLTK natural language toolkit. As in Metafor, the ConceptNet Commonsense semantic network provides semantics for the simulated objects, including object class hierarchies, and matching the arguments of verbs to the types of objects they can act upon, in a manner similar to Berkeley's FRAMENET. We are incorporating the AnalogySpace inference described in [7] to perform Commonsense reasoning. In aligning the programmed objects and actions with our Commonsense knowledge base, we ignore for the moment, the possibility that the author might want to create "magic ovens" or other kinds of objects that would intentionally violate real-world expectations for literary effect.

*Parsing*

The system uses two different types of parsing- syntactic parsing and frame based parsing. Syntactic parsing works on a tagger that identifies syntactic categories in sentences and that generates parse trees by utilizing a grammar (often a probabilistic context free grammar). For example a sentence can be tagged as:

*You/PRP can/MD put/VB water/NN in/IN a/DT bucket/NN ./.*

From the tag, a hierarchical parse tree that chunks syntactic categories together to form other categories (like noun/verb phrases) can also be generated:

*(ROOT (S (NP (PRP You)) (VP (MD can)*
*(VP (VB put) (NP (NN water))*
*(PP (IN in) (NP (DT a) (NN bucket)))))*
*(. .)))*

Frame based parsing identifies chunks in sentences and makes them arguments of frame variables. For example one might define a frame parse of the above sentence as: *You can put [ARG] in [OBJ]*

The Stanford parser [6] provides good syntactic parsing. We wrote a simple frame based parser for our use. Syntactic parsing allows identification of syntactic artifacts like noun phrases and verb phrases and dependency relationships between them. Frame based parsing allows us to do two things - first it allows us to do chunk extractions that are required for extracting things like object names, messages and verb arguments. Second, frame parsing allows us to identify and classify the input. For example a user input that is of the form "If....otherwise..." would be identified as an "IF_ELSE" construct very typical in programming. The logic of the parsing system is controlled by the dialog manager that facilitates and interprets user interaction. The dialog manager waits for user input.

When the user enters something into the system, it first categorizes the input. It uses three kinds of information to do the categorization: the current context, a frame based classification of current input and the object reference history. The current context broadly keeps track of what is being talked about - the user might be conversing about creating a new verb or adding decision criteria inside an IF construct. The dialog manager also keeps track of object reference history to allow users to use anaphora so that they do not need to fully specify the object in question every time. Using the previous information, the frame based classifier does a broad syntactic classification of the input.

After the input has been classified according to the previous parameters, the dialog manager parses the input and makes changes to the internal representation of the objects, object states, verbs and programs. Post parsing, the dialog manager can generate three types of dialogs - a confirmation dialog, a clarification dialog or an elaboration dialog. A confirmation dialog simply tells the user what was understood in the input and if everything in the input was parsed correctly. A clarification dialog is when the dialog manager needs to ask the user for clarification on the input. This could be simple 'yes/no' questions, reference resolution conflicts or input reformulation in case the parser cannot fully parse the input. If the parser fails to parse the input correctly, the dialog manager does a rough categorization of the input to identify possible features like noun phrases, verb phrases or programming artifacts. This allows it to generate help messages suggesting to the user to reformulate the input so that its parser can parse the input correctly. For the elaboration dialog, the system lets the user know what it did with the previous input and suggests other kinds of inputs to the user. These could be letting the user know what  commonsense properties were automatically added, suggesting new verbs or requesting the user to define an unknown verb.

*Commonsense reasoning*

An important lesson learned by the natural language community over the years is that language cannot be fully understood unless you have some semantic information – you've got to know what you're talking about.

In our case, Commonsense semantics is provided by Open Mind Common Sense [2], a knowledge base containing more than 800,000 sentences contributed by the general public to an open-source Web site. OCMS provides "ground truth" to disambiguate ambiguous parsings, and constrain underconstrained interpretations. OMCS statements come in as natural language, are processed with tagging and template matching similar to the processes used for interpreting natural language input explained above. The result is ConceptNet, a semantic network organized around about 20 distinguished relations, including IS-A, KIND-OF, USED-FOR, etc. The site is available at **openmind.media.mit.edu**.

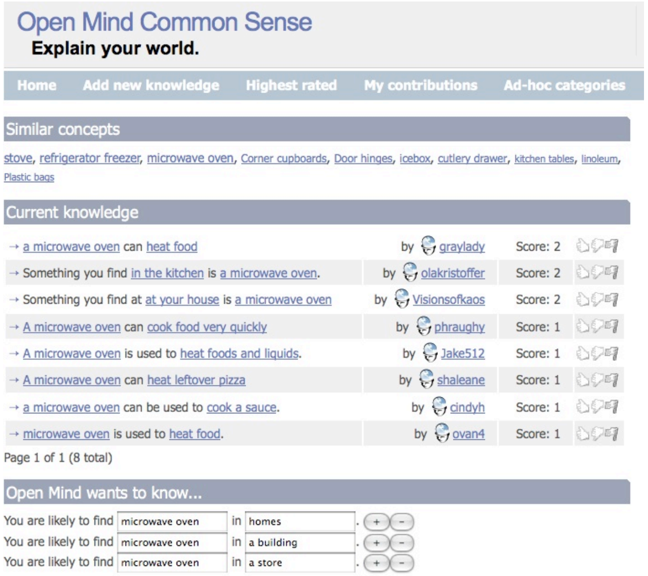


**Figure 3.** What Open Mind knows about microwave ovens.

Commonsense reasoning is used in the following ways. First, it provides an ontology of objects, arranged in object hierarchies. These help anaphora resolution, and understanding intentional descriptions. It helps understand which objects can be the arguments to which verbs. It provides some basic cause-and-effect rules, such as "When an object is eaten, it disappears".

## Understanding language for MOO programming

Key in going from parsing to programming is understanding the programming intent of particular natural language statements. Our recognizer classifies user utterances according to the following speech act categories:

• Object creation, properties, states and relationships.
*"There is a microwave oven on the table. It is empty."*

A simple declarative statement about a previously unknown object is taken as introducing that object into the MOO world. Descriptive statements introduce properties of the object. Background semantics about microwave ovens say that "empty" means "does not contain food" (it might not be literally empty – there may be a turntable inside it).

• Verb definitions.
*"You can put food in the basket".*

Statements about the possibility of taking an action, where that action has not be previously mentioned, are taken as introducing the action, as a possible action a MOO user can take. Here, what it means to "put food". A "basket" is the argument to (object of) that action. Alternative definition styles: "To …, you…", "Baskets are for putting food in", etc.

• Verb argument rules.
*"You can only put bread in the toaster."*

This introduces restrictions on what objects can be used as argument to what verbs. These semantic restrictions are in addition to syntactic restrictions on verb arguments found in many parsers.

• Verb program generation.
*"When you press the button, the microwave turns on."*

Prose that describes sequences of events is taken as describing a procedure for accomplishing the given verb.

• Imperative commands.
*"Press the button."*

• Decisions.
*"If there is no food in the oven, say 'You are not cooking anything.'"*

Conditionals can be expressed in a variety of forms: IF statements, WHEN statements, etc.

• Iterations, variables, and loops.
*"Make all the objects in the oven hot."*

In [5], user investigations show that explicit descriptions of iterations are rare in natural language program descriptions; people usually express iterations in terms of sets, filters, etc. In [3] we build up a sophisticated model of how people describe loops in nautral language, based on reading a corpus of natural language descriptions of programs expressed in program comments.

## Evaluation

We designed MOOIDE so that it is intuitive for users who have little or no experience in programming to describe objects and behaviors of common objects that they come across in their daily life. To evaluate this, we tested if subjects were able to program a simple scenario using MOOIDE. Our goal is to evaluate whether they can use our interface without getting frustrated, possibly enjoying the interaction while successfully completing a test programming scenario.

Our hypothesis is that subjects will be able to complete a simple natural language programming scenario within 20 min. If most of the users are able to complete the scenario in that amount of time, we would consider it a success. The users should not require more than minimal syntactic nudging from the experimenter.

We first ran users through a familiarization scenario so that they get a sense of how objects and verbs are described in the MOO. Then they were asked to do a couple of test cases in which we helped the subjects through the cases. The experimental scenario consisted of getting subjects to build an interesting candy machine that gives candy only when it is kicked. The experimenter gave the subject a verbal description of of the scenario (the experimenter did not 'read out' the description)

> *You should build a candy machine that works only when you kick it. You have to make this interesting candy machine which has one candy inside it. It also has a lever on it. It runs on magic coins. The candy machine doesn't work when you turn the lever. It says interesting messages when the lever is pulled. So if you're pulling the lever, the machine might say "oooh I malfunctioned" It also says interesting things when magic coins are put in it like "thank you for your money". And finally when you kick the machine, it gives the candy.*

The test scenario was hands-off for the experimenter who sat back and observed the user/MOOIDE interaction. The experimenter only helped if MOOIDE ran into implementation bugs, if people ignored minor syntactic nuances (e.g. comma after a when-clause) and if MOOIDE generated error messages. This was limited to once or twice in the test scenario. Figure 4 summarizes the post-test questionnaire.

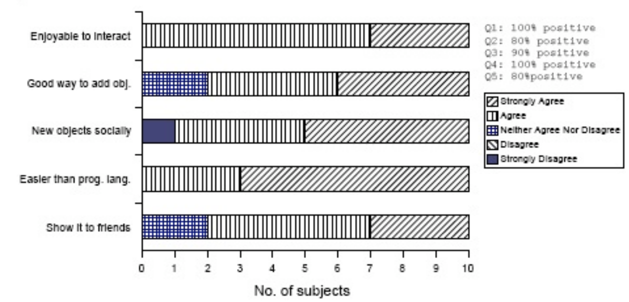


**Figure 4.** Results of evaluation questionnaire.

Overall, we felt that subjects were able to get the two main ideas about programming in the MOOs—describing objects and giving them verb behavior. Some subjects who had never programmed before were visibly excited at seeing the system respond with an output message that they had programmed using MOOIDE while glazing over the demonstration part where we showed them an example of LambdaMOO syntax. One such subject was an undergraduate woman who had tried to learn conventional programming but given up after spending significant amount of effort learning syntactic nuances. It seems that people would want to learn creative tasks like programming, but do not want to learn a programming language. Effectively, people are looking to do something that is interesting to them and that they are able to do that quickly enough with little learning overhead.

In the post evaluation responses, all the subjects strongly felt that programming in MOOIDE was easier than learning a programming language, even though 40% of the subjects mentioned they would like MOOIDE to support a larger variety of syntactic inputs. We feel some requirement of syntax is good, it helps people to learn how to structure procedural information, however they should not be required to put comma delimiters or quotes, that we required in MOOIDE syntax. The system should automatically do that and show it to users. This problem is quite solvable by building a better chunker. One can also use an online parser that parses the input as a person types it into MOOIDE to suggest what kinds of things one might consider typing in after that input.

During the evaluation MOOIDE, as with any complex integrated system, had minor implementation bugs—like output strings would not accept special characters that people might type in. The MOOP simulation environment did not accept articles like “the” and “an” for objects which frustrated a couple of subjects. This is something that is easily rectifiable and we consider the test results to be still valid even though we helped the subjects through these cases (Note: MOOIDE's natural language interface is quite good at handling different types of noun phrases. This issue came up only in the interface to MOOP, the 3rd party MOO environment.) In some cases, certain syntax of verb commands and object creation was not parsed either because of a bug in the grammar specification or it was not handled at all. In such cases, when given an example of a syntax that was parsed, subjects were able to reformulate the particular verb command. It seems that unlike in programming with a computer language in which excessive wording could be considered an overhead, the most common things that people want—words like “the” and “an” and fillers like the word “like” should definitely be parsed in the input. People get frustrated if the system cannot handle these most basic things.

There were some other things that came up in the test scenario that we did not handle and we had to tell people that the system would not handle them. All such cases below came across once each in the evaluation: – People do not necessarily start verb behaviors with event declarations, they would often put the event declaration at the end. So one might say “the food becomes hot, when you put it in the oven” instead of “when you put the food in the oven, it becomes hot”. This is a syntactic fix that requires addition of a few more patterns. – The system does not understand commands like “nothing will come out” or “does not give the person a candy” which describe negating an action. Negation is usually not required to be specified. These statements often correspond to the “pass” statement in Python. In other cases, it could be canceling a default behavior. – One subject overspecified – “if you put a coin in the candy machine, there will be a coin in the candy machine”. This was an example where a person would specify very basic commonsense which we consider to be at the sub-articulatable level, so we do not expect most people to enter these kind of facts. This relates to a larger issue—the kind of expectation the system puts upon its users about the level of detail in the commonsense that they have to provide.

The system did not handle object removals at this time, this is something that is also easily rectifiable. It does not handle chained event descriptions like “when you kick the candy machine, a candy bar comes out” and then “when the candy bar comes out of the candy machine, the person has the candy bar”. Instead one needs to say directly, “when you kick the candy machine, the person has the candy bar”. In preliminary evaluations we were able to identify many syntactic varieties of inputs that people were using and they were incorporated in the design prior to user evaluation. These were things like verb declarations chained with conjunctions e.g. “when you put food in the oven and press the start button, the food becomes hot” or using either “if” or “when” for verb declarations e.g. “if you press the start button, the oven cooks the food”.

## Related Work

Aside from our previous work on Metafor, the closest related work is Inform 7, a programming language for a MOO game which does incorporate a parser for a wide variety of English constructs [4]. Inform 7 is still in the tradition of "English-like" formal programming languages, a tradition dating back to Cobol. Users of Inform reported being bothered by the need to laboriously specify "obvious" commonsense properties of objects. Our approach is to allow pretty much unrestricted natural language input, but be satisfied with only partial parsing if the semantic intent of the interaction can still be accomplished. We were originally inspired by the Natural Programming project of Pane and Myers [5], which considered unconstrained natural language descriptions of programming tasks, but eventually wound up with a graphical programming language of conventional syntactic structure.

## Conclusion

While general natural language programming remains difficult, some semantic representation of the subject matter on which programs are intended to operate makes it a lot easier to understand the intent of the programmer. Perhaps programming is really not so hard, as long as you know what you're talking about.

## Acknowledgements

## Citations

[1] Bruckman, A., Resnick, M. (1995) The MediaMOO Project: Constructionism and Professional Community, Convergence, 1995 1: 94-109

[2] Liu, H., & Lieberman, H. (2005). Metafor: Visualizing stories as code. Proceedings of the 10th International Conference on Intelligent User Interfaces, San Diego, California, USA. 305-307.

[3] Mihalcea, R., Liu, H., & Lieberman, H. (2006). NLP (natural language processing) for NLP (natural language programming). CICLing, 319-330.

[4] Nelson,G. (2006). Natural language, semantics and interactive fiction. http://www.inform-fiction.org/I7Downloads/Documents/WhitePaper.pdf

[5] Pane, J. F., Myers, B. A., & Ratanamahatana, C. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. 54(2), 237-264.

[6] Sleator, C. D., & Temperley, D. (1993). Parsing english with a link grammar. In Third International Workshop on Parsing Technologies.

[7] Speer, Rob, Catherine Havasi and Henry Lieberman, AnalogySpace: Reducing the Dimensionality of Commonsense Knowledge, Conference of the Association for the Advancement of Artificial Intelligence (AAAI-08), Chicago, July 2008.