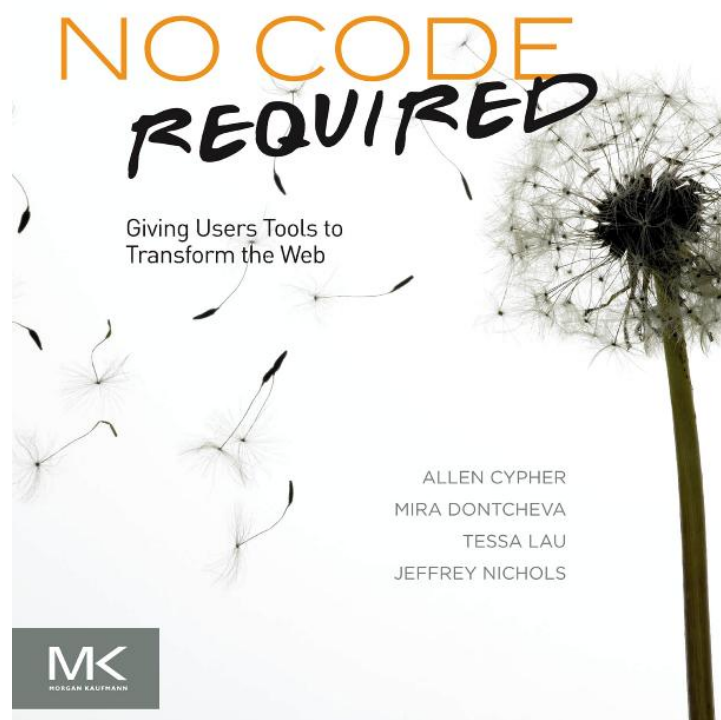


**Provided for non-commercial research and educational use only.  
Not for reproduction, distribution or commercial use.**

This chapter was originally published in the book *No Code Required: Giving Users Tools to Transform the Web*, published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues who know you, and providing a copy to your institution's administrator.



All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

<http://www.elsevier.com/locate/permissionusematerial>

From: Henry Lieberman, Moin Ahmad, Knowing what you're talking about: Natural language programming of a multi-player online game. In: Allen Cypher, Mira Dontcheva, Tessa Lau and Jeffrey Nichols, editors: *No Code Required: Giving Users Tools to Transform the Web*. Burlington: Morgan Kaufmann, 2010, pp. 331-343.

ISBN: 978-0-12-381541-5

© Copyright 2010 Elsevier Inc.

Morgan Kaufmann.

# Knowing what you're talking about

# 17

## Natural language programming of a multi-player online game

Henry Lieberman, Moin Ahmad

Media Laboratory, Massachusetts Institute of Technology

### 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.

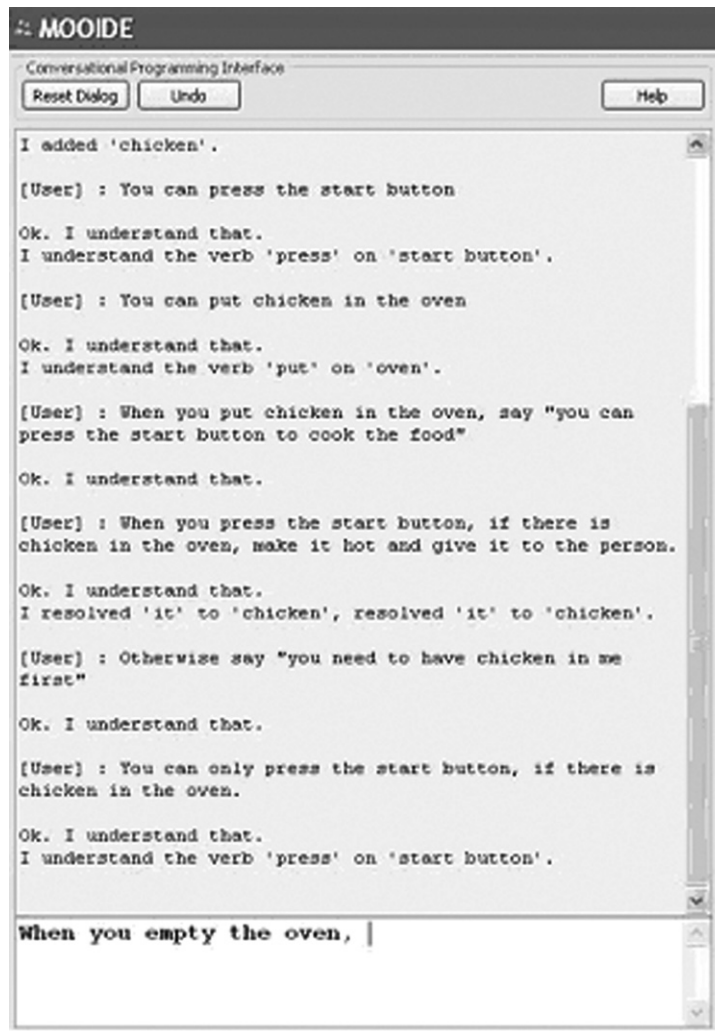
We present MOOIDE (pronounced “moody”), a natural language programming system for a MOO (an extensible multi-player text-based virtual reality storytelling game). MOOIDE incorporates both a state-of-the-art English parser, and a large commonsense knowledge base to provide background knowledge about everyday 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 ensure 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 17.1 illustrates MOOIDE's interface. A MOO (Bruckman & Resnick, 1995) 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.

We choose the MOO programming domain for several reasons. Even though a conventional MOO has a stylized syntax, users conceive of the interaction as typing natural language to the system; an opportunity exists for extending that interaction to handle a wider range of expression.

**FIGURE 17.1**

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

We leverage the ordinary person's understanding of natural language interaction to introduce programming concepts in ways that are analogous to how they are described in language. Interaction in natural language is, well, natural.

Contemporary Web-based MOOs are examples of collaborative, distributed, persistent, end user programmable virtual environments. Programming such environments generalizes to many other Web-based virtual environments, including those where the environments are represented graphically, perhaps in 3D, such as Second Life. Finally, because the characters and objects in the MOO

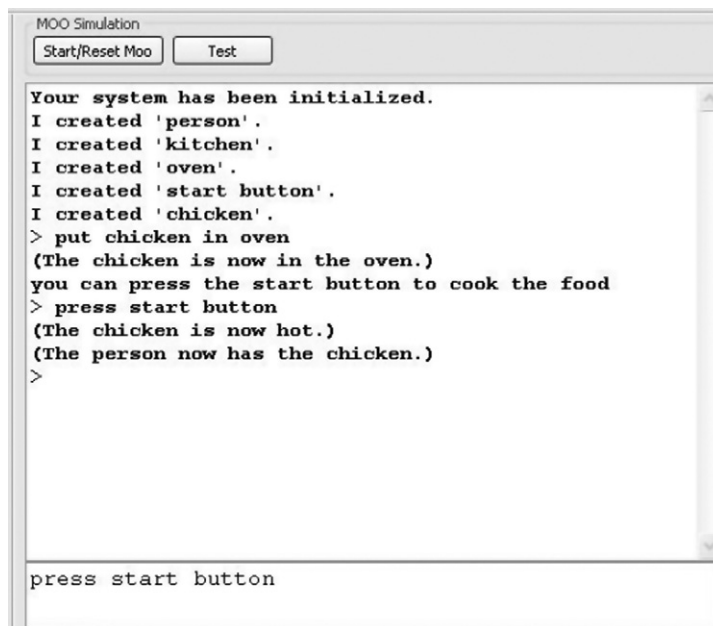
imitate the real world, there is often a good match between knowledge that is useful in the game, and the commonsense knowledge collected in our Open Mind Common Sense knowledge base.

## METAFOR

Our previous work on the Metafor system (Liu & Lieberman, 2005a; Mihalcea, Liu, & Lieberman, 2006) 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.

In MOOIDE, like Metafor, we ask users to describe the operation of a desired program in unconstrained natural language, and, as far as we can, translate it into Python code. Roughly, the parser turns nouns into descriptions of data structures (“There is a bar”), verbs into functions (“The bartender can make drinks”), and adjectives into properties (“a vodka martini”). It can also untangle various narrative stances, different points of view from which the situation is described (“When the customer orders a drink that is not on the menu, the bartender says, “I’m sorry, I can’t make that drink”).

However, the previous Metafor system was positioned primarily as a code editor; it did not have a runtime system. The MOOIDE system presented here contains a full MOO runtime environment in which we could dynamically query the states of objects. MOOIDE also adds the ability to introduce new commonsense statements as necessary to model the (necessarily incomplete) simulated environment.



```
MOO Simulation
Start/Reset Moo Test

Your system has been initialized.
I created 'person' .
I created 'kitchen' .
I created 'oven' .
I created 'start button' .
I created 'chicken' .
> put chicken in oven
(The chicken is now in the oven.)
you can press the start button to cook the food
> press start button
(The chicken is now hot.)
(The person now has the chicken.)
>

press start button
```

FIGURE 17.2

MOOIDE’s MOO simulation interface. It shows user interaction with the microwave oven defined in [Figure 17.1](#).

## A DIALOG WITH MOOIDE

Let's look at an example of interaction with MOOIDE in detail, a snapshot of which is shown in the figures above. The following 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 a microwave 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." Any player can subsequently use the verb by entering the following text into the MOO:

**cook chicken in microwave 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 nonfood items, he or she 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 or she made. To test the generated world, he or she enters **cook chicken in oven** into the MOO simulation interface. However, in this case the MOO generates an error – **You can only cook food in an oven**. This is not what the user expected!

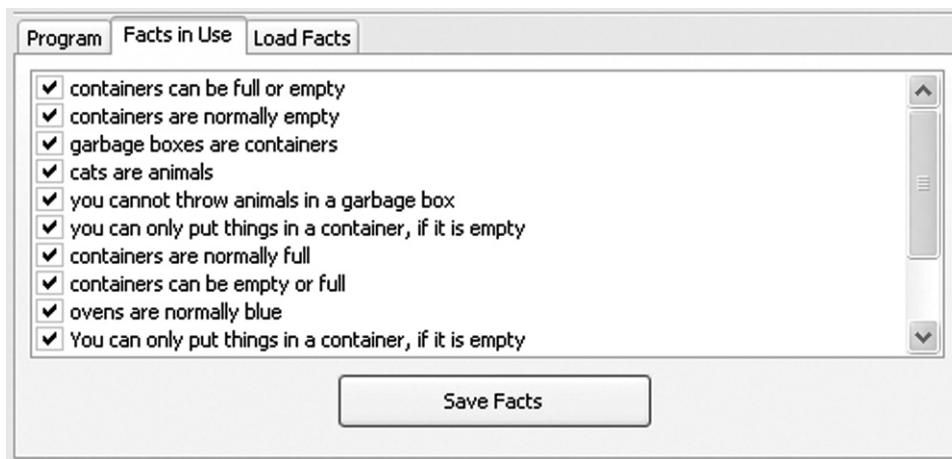


FIGURE 17.3

Commonsense facts used in the microwave oven example.

Of course you should be able to cook chicken; after all, isn't chicken a food? Wait, does the system know that? The user checks the commonsense knowledge base, to see if it knows this fact. The knowledge base is never fully complete, and when it misses deductions that are "obvious" to a person, the cause is often an incompleteness of the knowledge base. In this case, our user is surprised to find this simple fact missing. To resolve this error, he or she simply has to add the statement **Chicken is a kind of food**. Many other variants, some indirect, such as **Chicken is a kind of meat** or **People eat chicken**, could also supply the needed knowledge.

Then he or she 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**.

---

## PARSING

MOOIDE performs natural language processing with a modified version of the Stanford link parser (Sleator & Temperley, 1991) 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 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.

The system uses two different types of parsing: *syntactic* parsing and *frame-based* parsing. Syntactic parsing works using 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 (e.g., NN for noun, DT for determiner), 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]*".

Syntactic parsing allows identification of 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 into our speech act categories for programming constructs, further explained below. For example a user input that is of the form "If... otherwise..." would be identified as a variant of an "IF\_ELSE" construct very typical in programming.

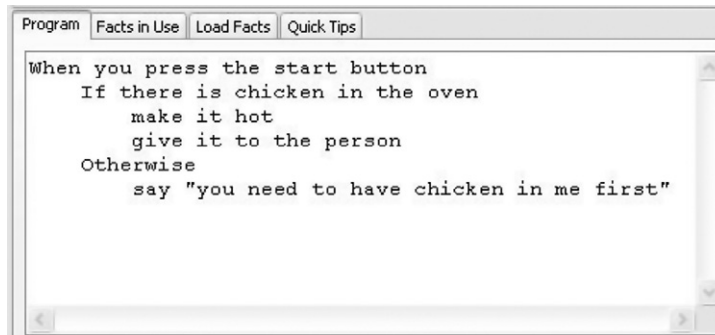


FIGURE 17.4

MOOIDE's English representation of MOO program code.

## DIALOG MANAGER

The logic of the parsing system is controlled by a dialog manager that interprets user interaction. When the user enters something into the system, it uses three kinds of information to categorize the input: 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, 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.

## Open Mind Common Sense

### Explain your world.

Home Add new knowledge Highest rated My contributions Ad-hoc categories

#### Similar concepts

stove, refrigerator freezer, microwave oven, Corner cupboards, Door hinges, icebox, cutlery drawer, kitchen tables, linoleum, Plastic bags

#### Current knowledge

→ a <u>microwave oven</u> can <u>heat food</u>	by  graylady	Score: 2	
→ Something you find in <u>the kitchen</u> is a <u>microwave oven</u> .	by  olakristoffer	Score: 2	
→ Something you find at <u>at your house</u> is a <u>microwave oven</u>	by  Visionsofkaos	Score: 2	
→ A <u>microwave oven</u> can <u>cook food very quickly</u>	by  phraughy	Score: 1	
→ A <u>microwave oven</u> is used to <u>heat foods and liquids</u> .	by  Jake512	Score: 1	
→ A <u>microwave oven</u> can <u>heat leftover pizza</u>	by  shaleane	Score: 1	
→ a <u>microwave oven</u> can be used to <u>cook a sauce</u> .	by  cindyh	Score: 1	
→ <u>microwave oven</u> is used to <u>heat food</u> .	by  ovan4	Score: 1	

Page 1 of 1 (8 total)

#### Open Mind wants to know...

You are likely to find	<input type="text" value="microwave oven"/>	in	<input type="text" value="homes"/>	<input type="button" value="+"/>	<input type="button" value="-"/>
You are likely to find	<input type="text" value="microwave oven"/>	in	<input type="text" value="a building"/>	<input type="button" value="+"/>	<input type="button" value="-"/>
You are likely to find	<input type="text" value="microwave oven"/>	in	<input type="text" value="a store"/>	<input type="button" value="+"/>	<input type="button" value="-"/>

FIGURE 17.5

What Open Mind knows about microwave ovens.

In our case, commonsense semantics is provided by Open Mind Common Sense (OMCS) (Liu & Singh, 2004), a knowledge base containing more than 800,000 sentences contributed by the general public to an open-source Web site. OMCS provides “ground truth” to disambiguate ambiguous parsings, and constrain underconstrained interpretations. OMCS statements come in as natural language and are processed with tagging and template matching similar to the processes used for interpreting natural language input explained earlier. 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 <http://openmind.media.mit.edu>.

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.** For example, *“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**, such as *“You can put food in the basket.”* Statements about the possibility of taking an action, where that action has not been previously mentioned, are taken as introducing the action, as a possible action a MOO user can take. Here, what it means is 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**, such as *“You can only put bread in the toaster.”* This introduces restrictions on what objects can be used as arguments 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**, like *“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 (Pane & Ratanamahatana, 2001), 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 (Mihalcea, Liu, & Lieberman, 2006), we build up a sophisticated model of how people describe loops in natural 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 whether 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 minutes. 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.

## Experimental method

We first ran users through a familiarization scenario so that they could 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 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 that 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, "Ooh, 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.

## Experimental results

Figure 17.6 summarizes the post-test 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 paying little attention to the demonstration part where we showed them an example of LambdaMOO. One such subject was an undergraduate woman who had tried to learn conventional programming but had given up after spending significant effort learning syntactic nuances. It seems that people 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 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. However, 40% of the subjects encountered limitations of our parser, and criticized MOOIDE for not handling a sufficient range of natural language expression. We investigated the cause for the parser failures. Some of the most serious, such as failures in correctly handling “an” and “the,” were due to problems with the interface to MOOP, the third-party MOO environment. These would be easily rectifiable. Others were due to incompleteness of the grammar or knowledge base, special characters typed, typos, and in a few cases, simply parser bugs. Because parsing per se was not the focus of our work, the experimenter helped users through some cases of what we deemed to be inessential parser failure. The experimenter provided a few examples of the kind of syntax the parser would accept, and all subjects were able to reformulate their particular verb command, without feeling like they were pressured into a formal syntax. Advances in the underlying parsing technology will drive steady improvements in its use in this application.

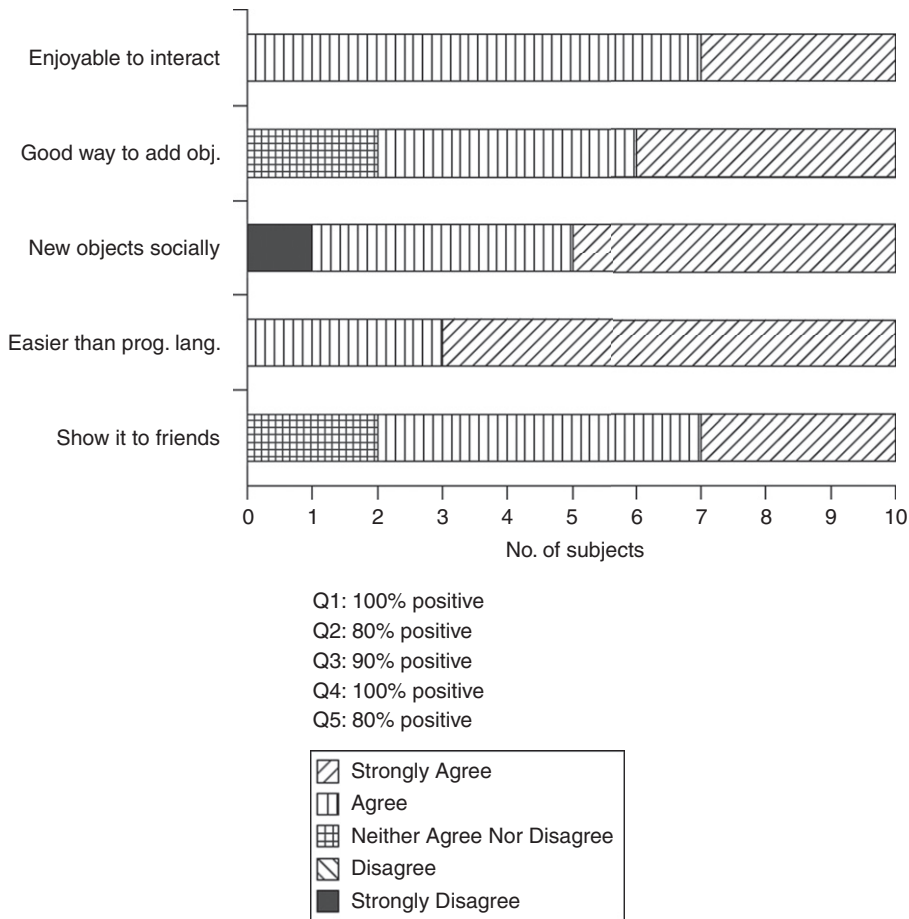


FIGURE 17.6

Results of the evaluation questionnaire.

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, discussed next, came across only once each in the evaluation.

People would often put the event declaration at the end, rather than the beginning, confusing our system. 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 the 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-articulate 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 granularity issue also has been recognized in knowledge acquisition systems, such as Blythe, Kim, Ramachandran, and Gil's EXPECT (2001), because informants are sometimes unsure as to how detailed their explanations need to be. Part of the job of a knowledge acquisition system and/or a human knowledge engineer, is to help the informant determine the appropriate level of granularity.

Unfortunately, there's no one answer to this question. Tutorials, presented examples, and experience with the system can help the user discover the appropriate granularity, through experience of what the system gets right and wrong, and how particular modifications to the system's knowledge affect its performance. A guide is the Felicity Conditions of van Lehn (1984), following Seymour Papert's dictum, "You can only learn something if you almost know it already." The best kinds of commonsense statements for the user to provide are those that are at the level that help it fill in inference gaps, such as the "Chicken is a food" example given earlier. One could certainly say, "Chicken is made of atoms," but that isn't much use if you're trying to figure out how to cook something. Because our commonsense knowledge base can be easily queried interactively, it is easy to discover what it does and does not know. There's some evidence (Maulsby, 1993) that people who interact with machine learning systems do adapt over time to the granularity effective for the system.

The system did not handle object removals at this time, something that is also easily rectified. 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 before 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 (Liu & Lieberman, 2005; Mihalcea, Liu, & Lieberman, 2006), the closest related work is Inform 7, a programming language for a MOO game that does incorporate a parser for a wide variety of English constructs (Sleator & Temperley, 1991). Inform 7 is still in the tradition of "English-like" formal programming languages, a tradition dating back to Cobol. Users of Inform 7 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, Myers, and Ratanamahatana (2001), which considered unconstrained natural language descriptions of programming tasks, but eventually wound up with a graphical programming language of conventional syntactic structure.

**SUMMARY**

Although 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.

**MOOIDE**

Intended users:	All users, focusing on game players, beginning programmers
Domain:	Multi-player online virtual worlds, particularly text MUD/MOO environments
Description:	MOOIDE, pronounced "moody," is a natural language programming environment for creating new characters and new objects in virtual environments. Behavior for characters and objects can be programmed to allow them to interact with game players or each other.
Example:	Create a MOO "room" that simulates a household kitchen, with a refrigerator, stove, etc. A human player could then type, "put chicken in the microwave oven", and tell it to "cook." The microwave oven might complain if you put an inappropriate object in it, such as a cat.
Automation:	Yes.
Mashups:	Not applicable.
Scripting:	Yes.
Natural language:	Yes, all interactions are through natural language.
Recordability:	Yes.
Inferencing:	Yes.
Sharing:	Yes.
Comparison to other systems:	The only other MOO natural language programming system is Inform 7. Inform 7 has no inference capability.
Platform:	Any Python platform, with the MOOP MOO system, the Stanford Natural Language Group parser, and the Divisi commonsense inference toolkit.
Availability:	Available only for sponsor organizations of the MIT Media Lab and selected academic collaborators.